# Exploring C++:

## The Adventure Begins

Jason James



September 7, 2024

# Volume II
# Intermediate toward Advanced

# Brief Contents

# Contents

# List of Tables

# List of Figures

# Preface

The first thing to know is that this book is written in a conversational — even whimsical style at times. I'll not be formal unless the topic really calls for it.

## Reader Background

In this book I assume you've had a course in basic programming. This course would have to entail data types, input/output with the console, branching and looping, functions, separate compilation (your own libraries), basic `class` design, and the use of at least `vectors` for storing lots of data. If you've had some basic file input and output, so much the better! Basically everything you'd find in our first volume. *smile*

I also expect you've had a decent amount of math. Some schools require having finished Calculus I, but I think any kind of calculus would be good enough to get you those advanced abstract thinking skills prized by programmers.

## Styles

There are some color and style conventions used in the book that might be helpful to know up front as well. For instance, different parts of the C++ language are colored differently. You can see a little sample in this chart:

| | | |
|---|---|---|
| short | #include | 12'456 |
| int rand() | return | "Welcome" |
| <iostream> | cout | '\n' |

All code samples are rendered in a little box like so:

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "\n\t\tWelcome to C++!!!\n\n";

    return 0;
}
```

But beware a bit of code with a red border:

```
bad goeth here
```

Those are anti-examples — **NOT** to be emulated!

## Typography

Definitions are given as they are needed but not highlighted in any special way.

> **Sidebars**
>
> There are also sidebars with little extra bits of knowledge that you might be able to live without. But why would you want to do that?!

This highlights that all knowledge is precious — not just that found in a little rounded box! Also, watch for them everywhere — even in footnotes![1]

Also, there are links to online sites/documents. These links look like this one to the awesome website cppreference.com. It is a great place to look up things you've forgotten the details of.[2]

## Exercises

I've provided no exercises in this text. There are many example codes that are complete and run just fine, but no explorations. This is because my teaching website alongside this one (craie-programming.org) is filled with programming exercises. Each is separated into semesters and kind and difficulty rating. The semesters at my school are CSC121 and CSC122. The kinds are labs for focusing 1-3 topics at a time and projects for synthesizing 3-10 topics into a cohesive whole. The difficulty is given as a Level where 1 is relatively easy at the time the material is learned and 7 is pretty darn challenging at the time the material is learned. They are great for practice even if you aren't taking my courses so feel free to try them out!

One further note about the assignment 'prompts' as assigned: they are not perfectly clear and tediously laid out on purpose. Part of learning to program is interacting with the prospective 'customer' of the program/application. Their initial product description is likely to be imperfect and require some amount of clarification with them — perhaps even a few rounds of it in some cases. Students of programming need to get practice with this process as well. And who better to gather requirements clarification from than their instructor!

## Code Availability

As mentioned above, there are numerous code samples gone over in the text. Many are present in the text in full. A few were much longer and are linked to on the companion site (craie-programming.org/OER). The cutoff is about two pages.

I keep this split on purpose even though the codes in the text cannot generally be copy/pasted out — something about fonts for special characters like underscores and quotes — because I feel it is important to the student's memory to actually type much code for themselves at least in the first two semesters of a typical program of study. This is part of the classical 'muscle memory' tradition found in numerous studies like math, martial arts, etc.

All example codes are under the same copyright as this book — see below.

## Self-Study

In any significant study of material, there comes a time for self-study. And programming is no exception! Here that takes the form of realizing you have an unanswered question or concern with a topic and

---

[1]Like this one, but not actually this one. . .

[2]It seems to be where the standards committee for C++ members hang out and help maintain the wiki, so it must have the latest and greatest info, right?

making a test program to clear that up or deepen your knowledge. Such programs are typically 10 or so lines long, but can be pages in later topics.[3]

It is a good idea to make such programs regularly and document them well with comments, good variable names and the like, etc. Always make the effort to make your code readable and understandable to whomever may come by it later — even if that someone is just you. Don't underestimate the worth of a good test program in reminding oneself the ways of a feature!

## Viewing

I recommend a continuous scroll to keep the flow going from page to page. But it shouldn't look bad in one-page or 2-up modes, either.

Also, in that vein, since this book was produced to be a PDF and not in print, there is no provided index. You've got built-in search, so hit that Control / command and F !

Finally, make sure you check regularly for updates as this is an online document and therefore subject to anytime fixes, additions, or clarifications.

## Coming Soon!

There are many sections currently marked "Coming Soon!". Most of these will be filled in by the end of the Summer 2024 term. Appendices will come along as time permits after the regular chapter sections are done, however.

## Copyright and License

## Acknowledgements

Here are a very few of the folks to whom I owe a great debt and whose wisdom and service and support helped me make this book you see on your screen:

- my partner Tammy

- my boys Kyle and Caleb

- my Mother

- my coworker Carl Molyneaux who copy-edited this work several times

- my students whose struggles led me to refine my craft and come to the point of writing this work[4]

---

[3]Of course, by then you'll be accustomed to writing such longer programs and it won't seem as daunting as right now. *smile*

[4]Not that I'm perfect and know all about my topic or my students. I'm just saying that without coming to understand them with respect to both topical issues and economic issues in learning, I wouldn't have reached the place where I needed to write this book and release it as a free educational resource.

# Part IV
# Storage

# Chapter 8

# C-Style Arrays and Strings

Before there were `vectors` and the `array class`, there were C-style arrays.[1]  These are a statically-sized[2] storage mechanism like the `array class`. In fact, the `array class` is a thin veil over the top of a C-style array.

A special subset of the C-style arrays is known as null-terminated strings or C-strings. We'll also be talking about these in this chapter.

What we'll eventually find is that the `vector` and `string class`es are built from these static structures using the techniques of the next chapter (9 on dynamic memory). In fact, by the end of this book, we would be able to rebuild these `class`es from scratch if we wanted.[3]

## 8.1   Basics of Arrays

### 8.1.1   Versus vectors

As mentioned in brief above, arrays are static memory objects. That is, they are sized at compile-time by the programmer and never get to change in size throughout the entire program run. The only way to

---

[1]For more on everyday storage containers like the `vector` and `array class`es, please see the first volume of this series Programming Basics.

[2]That is sized at compile time rather than at run time. These things are stored on the function call stack instead of in dynamic memory.

[3]But we don't unless we are writing a new version of the standard library or writing them for a new language we are developing. Using the standard library features gives us confidence in their long history of sturdy testing and usage. Don't reinvent the wheel for everyday programs — only for special cases.

change the size, in fact, is to change it in the source code and rebuild the program all over again.

This is drastically different from the `vector` `class` which was able to grow as needed by the program. Sometimes this difference can be a hindrance. But if we keep it in mind as we code, we can keep on top of the difficulties and keep it working.

Do we work with arrays in new designs? NO! Not unless the system we are working on (our target platform) has memory limitations that preclude the use of `vector`s. These embedded or subset compiler environments just don't have the hardware or OS support necessary for us to use the `vector` `class` in our program.

We will, however, use arrays a lot when dealing with legacy libraries that only provide an array interface to their functionality. This alone necessitates that we learn to use arrays and use them well!

So let's get started!

### 8.1.2   Declaration

To declare a basic array, we would have something like this in our program:

```
const short MAX_ARR{25};
double arr[MAX_ARR];
```

Again, this constant size can impact us throughout the program, so we make it an actual constant rather than a literal to make it easier to update when we change our mind and decide the current value is too small or too large.[4],[5]

Also note that you'll need a separate variable to track the number of used positions within your array, say:

```
short used_arr{0};
```

This tracking variable should be the same type as you chose for your maximum size/capacity constant above. The type you chose can be any of the integer types: `short`, `long`, `unsigned short`, `unsigned long`, or `size_t`. (If you don't care about platform independence — aka portability — you can even use `int` or `unsigned int`.)

What's `size_t`? Well, that's one of those `typedef`s (or perhaps a `using` alias) that the library set as a particular `unsigned` integer type that was ideal on the target platform for any array size information or array positions.[6] That probably makes it the best choice for our purposes here, then.

Which library was that again? It is available from almost any standard library, actually. But if you don't have anything else included already, just bring in `cstddef` as a minimal library.

(If you really don't like the name, you can even make a `typedef` or `using` alias for this type if you want (like the `vector` did for you when it chose whatever `unsigned` integer type and called it `size_type`).

So we would probably redo our earlier declaration as:

```
const size_t MAX_ARR{25};
double arr[MAX_ARR];        // the array decl need not change
size_t used_arr{0};
```

---

[4]The size is an integer-type, of course, since we don't have partial positions in the array available.
[5]Changing the size can happen often during early requirements gathering and then again as a result of user feedback between versions/releases.
[6]If it can hold the size, after all, it can hold any value from 0 up to that size, right?

### 8.1.3   Initialization

Initialization of array elements is at the programmer's discretion — sort of... If you begin to initialize an array:

```
const size_t MAX_ARR{25};
double arr[MAX_ARR] = { 42, 4.2, .42 };
size_t arr_used{3};                        // init'd three values
```

and stop short of the actual number of elements, the remaining positions will be default constructed (0 bit patterns for the built-in types[7]). Also note that the use tracking variable has been set to the number of initializers!

Although you can, you should not leave the declared size `const` out even when providing a list of initializers. This leaves you without a known upper bound for loops or other error-checking scenarios. The infamous `sizeof(arr)/sizeof(double)` — dividing the bytes for the array by the bytes for the base type — will not work outside the original declaration context/scope of the program.[8]

### 8.1.4   A Brief Example

A simple example of using this syntax is names for `enum`eration values:

```
enum           STOP_LIGHT                              {  RED,    GREEN,    YELLOW,
                                                          MAX_STOP_LIGHT };

const string STOP_LIGHT_NAMES[MAX_STOP_LIGHT] = { "RED", "GREEN", "YELLOW" };
```

Here, the `enum`eration `const`ants have their names stored in a `const`ant array so that later we can communicate to the user in words instead of numeric values:

```
STOP_LIGHT light_state{RED};
cout << "The light is " << light_state << ".\n";
cout << "The light is " << STOP_LIGHT_NAMES[light_state] << ".\n";
```

The first `cout` prints the somewhat cryptic message `The light is 0.` But with the helper `string` array, the second `cout` shows the much more palatable `The light is RED.` instead.

We've simply used the `STOP_LIGHT` `const`ant as a subscript into the `const`ant array of `strings` to select the proper name for that `STOP_LIGHT`'s value. But what a difference!

While this might be an actual place to use an empty set of square brackets to make the compiler count the elements for us, I decided to be careful and put the extra `enum`eration value in there. This has the one disadvantage of making `switch`es on the `STOP_LIGHT` type ask for a `case` involving the extra `const`ant.

### 8.1.5   Lots of Loops

Since the array is a built-in type, there are no methods to support your programming. There are a few scattered library functions, but we'll not need them here. (Later we'll talk about the C-string library, but that's a very special subset of arrays...)

---

[7]Zeros for numeric types, `false` for `bool`, and `'\0'` for `char`.
[8]The reasoning for this failure will follow in the next chapter, but there is an example of it not working in the section just below — section 8.1.6.1.

#### 8.1.5.1  General

In general, that leaves us writing a lot of loops — mostly `for` loops!

```cpp
for ( size_t c{0}; c != used_arr; ++c )
{
    // do something with arr[c]
}
```

#### 8.1.5.2  If Full

Or we could also use a range-based `for` loop:[9]

```cpp
for ( auto x : arr )
{
    // do something with x
}
```

Or if we wanted to change the elements in the array:

```cpp
for ( auto & x : arr )
{
    // do something with x -- even:
    x = new_value;
}
```

. . . if we had a full array, anyway. If it isn't completely filled, we shouldn't do the range-based version because there aren't mechanisms like with the `vector` to end at the position of the used tracker variable.

#### 8.1.5.3  Input Is Special

Initial input loops would be `do` or `while` depending on your interface/druthers, of course. In addition to making sure your user isn't done with their data, they should also check to make sure that you do not overrun the maximum capacity of the array during input:

```cpp
cin >> an_element;
while ( used_arr != MAX_ARR &&
        !cin.fail() )
{
    arr[used_arr] = an_element;
    ++used_arr;
    cin >> an_element;          // makes them enter in 1 extra
                                // piece of data beyond MAX_ARR!
}
```

This version may make the user enter an extra piece of data when they totally fill the array, but at least the loop head keeps things safe — we are just being a little rude. We'll fix that up in an upcoming example.

---

[9]See the previous volume for more on this loop as used with `vector`s and the `array` `class`.

## 8.1.6 Passing Arrays to Functions

When you go to pass an array to a function, take note that they are automatically passed as a form of reference — the function gets to change the contents of the array by default. To avoid this, simply add the keyword `const` to the formal argument's base type.

Also, note that the sizing brackets of a formal argument are typically left empty. This is because the compiler will not pay attention to their contents, anyway. This allows a function accepting an array to be most generic and accept actual arrays of any length — given the correct base type.

But, to keep the function within the sane bounds of the array — not going past the currently filled in elements, that is — we'll [almost] always pass to the function an extra parameter to signify the number of used elements within the array.

For using the elements of the array in the function, the function head might look like this:

```
void print_arr(const double arr[], size_t max);
```

Or, for changing the elements of the array, the function head might look like this:

```
void read_arr(double arr[], size_t max, size_t & entered);
```

Note how this input function might also change the number of elements and so takes a reference to a `size_t` to track that number of elements for the caller.

### 8.1.6.1 Note on sizeof

The note above about not using `sizeof` to determine the number of elements in an array comes into play once an array has been passed outside its declaring context — to another function, for instance. Here we note that only one element is printed on a modern 64-bit system:

```
void print_arr(const double arr[])
{
    size_t len = sizeof(arr)/sizeof(double);
    for ( size_t i{0}; i+1 < len; ++i )
    {
        cout << arr[i] << ' ';
    }
    cout << arr[len-1] << '\n';
    return;
}
```

(Remember that red-framed examples are anti-examples and **NOT** to be followed but avoided!!!)

Since the size of a `double` here is typically 8 bytes and so is the size of what the function receives for an array, we just get 1 for `len`. What do I mean "what the function receives for an array"? You'll have to wait until the next chapter (section 9.1.4) for that!

### 8.1.6.2 Array Subrange Processing

If your function processes a sub-range or contiguous sub-sequence within an array, you'll need two boundaries:

```
double sum_arr(const double arr[], size_t from, size_t to);
```

Most array programmers take the `to` parameter to be inclusive, but the style with C++ `vectors` is to have it be exclusive. I'll let you decide what's best for you (and your function's callers). Just make it clear in your function's documentation!!!

### 8.1.7   A Comparative Example

I have an application handy that uses one-dimensional storage to collect and average the user's height measurements. I've got one version that uses a `vector` and another version that uses a C-style array. I strongly encourage you to download these and load them side-by-side in your favorite difference checker.

If you don't have a difference checker, you can find many and their relative qualities on this page at Wikipedia. Such tools allow one to see the commonalities and differences between two pieces of code with ease and can be both a learning tool and a design tool of great use![10]

## 8.2   Basics of C-Strings

A C-string is a specially-treated array of characters. Instead of just holding data characters, it also holds a terminator character which is guaranteed to not be a part of the user's data. (This can be guaranteed because this character is not able to be typed at a standard keyboard.)

The terminator is placed immediately following the user's actual data characters in the array. Hence us calling it the terminator — it terminates or ends the user's data.

Because of this special terminator character, we won't have to pass a size/length variable to functions which process a C-string! The downside is we'll be able to store one less character than would otherwise be indicated by the array's declared size. *shrug* Small price to pay!

What is the terminator character? ASCII 0, of course: `'\0'`. (Also called the null character. Don't confuse this with the `nullptr` parameter to `time()`, however! We'll talk more about that constant in the next chapter with respect to pointers.)

### 8.2.1   C-String Initialization

The array initialization syntax of a comma-separated, curly-brace enclosed list of values can be shortened to a C-string literal. That is, we can do this:

```
const char prog_title[MAX_TITLE] = "Heights Averager";
```

instead of this:

```
const char prog_title[MAX_TITLE] = { 'H', 'e', 'i',
                                     'g', 'h', 't',
                                     's', ' ', 'A',
                                     'v', 'e', 'r',
                                     'a', 'g', 'e',
                                     'r' };
```

But this is assuming `MAX_TITLE` is at least 17. If it were 16, the first declaration would fail saying that the initializer was too long — even a string literal has the invisible null terminator implicitly present[11] — whereas the second declaration would 'work', but not produce a C-string constant! It would instead just be a `const` array of `char` ending in `'r'` — not `'\0'` — and therefore not a C-string.

---

[10]For more on difference checking, see the previous volume in the function design section.
[11]Some would call it a C-string literal to emphasize this. . .

### 8.2.2   C-String Library Functions

To help process C-strings in typical ways, the `cstring` library provides the following functions:[12]

| Function | Notes |
| --- | --- |
| `strcpy(d, s)` | copies s into d — destroying d's prior value |
| `strcat(d, s)` | copies s onto the end of d |
| `strcmp(s1, s2)` | compares s1 to s2 lexicographically, returning an integer analogous to that returned by `string::compare` |

So, `strcpy` is like assigning the source C-string to the destination C-string (d = s), but we cannot assign arrays!

And `strcat` is like concatenating the source C-string onto the destination C-string (d += s), but that would be assigning arrays, too!

`strcmp`, of course, returns something like this:

| I want to know if... | So I code... |
| --- | --- |
| s1 < s2 | `strcmp(s1, s2) < 0` |
| s1 == s2 | `strcmp(s1, s2) == 0` |
| s1 > s2 | `strcmp(s1, s2) > 0` |

You can even do multi-combinations, of course, like if you wanted to know that the first C-string was less than or equal to the second (s1 <= s2), you could code to check `strcmp`'s result against 0 with <= like this:

```
if ( strcmp(s1, s2) <= 0 )
{
    // s1's contents comes before or is equal to s2's contents
}
```

for instance.

And, just like with `string::compare`, these comparisons are not truly alphabetical when the data are, they are ASCII-betical! That is, not only do `strcmp("hello", "helix")` and `strcmp("playing", "play")` return a positive value, so does: `strcmp("apple", "Apple")`![13]

#### 8.2.2.1   Protecting from Overrun

Because the `strcpy` and `strcat` functions don't know the declared size of the destination array, they cannot effectively protect that array from overrun by the source array! To do so, the `cstring` library also provides functions that protect these actions from overrunning the destination array. You might suspect overloading to be involved here, but remember that these are old C routines and they didn't and still don't have overloading in C.

So, we had to have different names for these protection functions. Since they were taking a numeric bound on the number of characters that could be copied, the letter n was added to the names. Not at the end, though — in the middle:

| Function | Notes |
| --- | --- |
| `strncpy(d, s, n)` | copies at most n chars from s into d |
| `strncat(d, s, n)` | copies at most n chars from s onto the end of d |
| `strncmp(s1, s2, n)` | compares at most n chars of s1 to those in s2 to determine their lexicographic order |

---

[12]There are many more, we'll discuss them as they become relevant.
[13]Recall that the lowercase letters come after the uppercase letters in the ASCII table.

Whenever the declared size of the destination array is known, these functions should be used preferentially over the others. They are inherently safer by far!

Unfortunately, there is a caveat. When the standard was first interpreted, some folks said it meant to copy the $n^{th}$ character and stop immediately — sometimes leaving the destination not a C-string at all! Others demanded that a null character be stored in the $n^{th}$ spot to make sure the destination was a C-string after all. Eventually the POSIX organization, which proposes standards to fix holes or shortcomings in other standards agreed with the latter folk and so, on a POSIX-compliant library implementation, you'll always end up with a C-string.

But finding out if your current library is POSIX-compliant can be no small task! Perhaps a safety net is in order? Toward this end, many programmers will simply follow up any `strncpy` or `strncat` call with this assignment:

```
d[MAX_D-1] = '\0';
```

Here we cap off the destination array with a null character at its last physical position. This makes the result a C-string no matter what came before it!

For example:

```
const size_t MAX_S{12}, MAX_D{6};
char s[MAX_S] = "long string", d[MAX_D] = "short";
strncpy(d, s, MAX_D-1);
d[MAX_D-1] = '\0';
```

This would end up with `d` holding only `"long "`.

With this in mind, what parameter do we pass for `n` for these two functions? Well, for `strncpy`, we usually pass `MAX_D-1` as the upper bound. This avoids copying that last character since we are going to overwrite it anyway.

But for `strncat` we need to know how many characters are already in the destination in order to find out how much room is left there! Since we designed against having to track the logical length of the C-strings with null termination, we are kinda out of luck here.

We'll have to use a helper function: `strlen`. It takes a single C-string and counts and returns the number of data characters — those preceding the null terminator — therein. This is an expensive operation and not to be called upon lightly! Beware!

Now, in the third parameter to `strncat` we can place: `MAX_D-strlen(d)-1`. This accounts for the maximum size and the current logical size and leaves room for the null terminator we'll add at the end of the array upon return.

### 8.2.2.1.1   Upper Bounded Compare?

But what is `strncmp` for? There are no buffer overrun issues there unless one of the arrays isn't really a C-string, right? True, but it is also useful to build a power-user friendly, command-driven interface.

What's a power user? That's someone who memorizes all the keyboard shortcuts for their apps so that they can keep their hands on the keyboard 95% of the time to maximize efficiency. Many programmers naturally turn into power users during their years of using clunky GUI interfaces so closely during development processes.

What's a command-driven interface? It's like you took the menu from a console program and replaced it with typed commands instead. Just like implementing your own shell — like the terminal many of you compile and run in on your Unix/Linux box.

Why? Well, it is easier for many console applications to avoid the menu and its typical 10-item-or-less rule and just implement a 'help' command that the user can type at any time to remind them of the various commands available. Then they can type 'help command' — listing a particular command, of course — to get more detailed help on that command.

Anyway, I spent many glorious hours in undergrad transferring files. This was a common time-wasting activity before the Web and when you didn't have enough money to buy real games. We'd — yes, I was not alone — download music, art, freeware games, and the like from anonymous FTP sites. And with the command-line `ftp` app, we'd type commands like 'get' and 'quit' to retrieve files from various FTP servers and to end the program when we were done.

I saved many minutes of those hours — commonly put toward visiting another site, of course — by typing 'qui' for the quit command. This was possible because 'qui' was a significant prefix for the quit command. Why wasn't it just 'q'? Or 'qu'? Well, those would have been potentially confused with the 'quote' command for sending special instructions to the remote site. But 'qui' was just enough! *chuckle* Ah... Memories...

#### 8.2.2.2 Case-Insensitive Comparison?

While certain systems have their own pet C-string compare functions that are not ASCII-betical, these are not part of the C++ standard. On those systems you might find `stricmp` (on Windows) or `strcasecmp` (on Unix/Linux). But I won't recommend them here as they are not portable. I'd recommend you use the boost library's `isequals` function or roll your own!

### 8.2.3 Output of C-Strings

`cout` tries to interpret any `char` array that is inserted (`<<`) into it as a C-string — whether it has a null-terminator or not! Beware!!!

Oh, not convinced? Well, try it. You'll find that a plain `char` array with no null terminator displays not only its contents but also an arrangement of garbage values that might include smiley faces, musical notes, line drops, tabs, and other such things. This is because without the null terminator, `cout` doesn't know when to stop. Luckily there are many places in memory where there are 8 zero bits in a row, so it is bound to hit one ...eventually!

### 8.2.4 Input of C-Strings

`cin`'s extraction operator (`>>`) knows to store a null-terminator at the end of a `char` array so that it becomes a C-string after extraction. However, we don't just want to do `cin >> str`, since this leaves the maximum size of the C-string's array in question and us vulnerable to overrun!

Instead, always prefix an extraction attempt with a `setw` call like so:

```
cin >> setw(MAX_S) >> s;
```

This let's `cin` know how long the array for the proposed C-string `s` has been allocated to be and it will stop one `char` short of this maximum to store the null-character. It is still space-separated, though, so you may not reach `MAX_S-1` `char`s of data. (Recall that the `setw` manipulator is in the `iomanip` library.)

#### 8.2.4.1 C-String Input with Embedded Spacing

`cin.getline` allows for the input of C-strings which contain spacing, too:

```
cin.getline(s, MAX_S);
```

There is another variant with a third parameter to specify the input stop `char` so you aren't stuck with `'\n'`-style lines.

In addition to not playing well with prior extractions — which leave behind newlines with abandon, there is an extra caveat with `cin`'s C-string `getline`. When it fails to reach the `'\n'` — or whatever stop `char` you specified — it will tell `cin` that he has `failed`! To fix this, you may want to encapsulate your use of `cin`'s `getline` in a function like so:

```cpp
const bool CLEAR_BUFFER_AFTER_FAIL{true},
           LEAVE_BUFFER_ALONE{false};

inline bool get_line(char s[], size_t MAX_S,
                     char stop = '\n',
                     bool clear = CLEAR_BUFFER_AFTER_FAIL)
{
    bool failed;
    cout.flush();               // or cout << flush;
    if ( cin.peek() == stop )
    {
        cin.ignore();
    }
    cin.getline(s, MAX_S, stop);
    failed = cin.fail();
    if ( failed && clear )
    {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), stop);
    }
    return failed;
}
```

Here I've parameterized the C-string, its size, the desired stopping character, and a `bool` to indicate that the buffer should be emptied upon `failure`. Either way, the `failure` is returned from the function. (The `bool` also has two constants for the caller to use to avoid the magic of `bool` literals. Always a good idea!)

The function takes care of the common `getline` — C-string and `string` `class` alike — problem of a left-behind stop character in the buffer by `peek`ing for it and `ignore`'ing it if present. This requires a `flush` of `cout` to ensure any waiting prompt is displayed since `peek` doesn't actually read and so many libraries won't force the display of prompts for its call.

Then the function takes care of a `fail` during the `getline` by recording it and possibly `clear`ing it as well.

This handy function can be put in an input library and just `#include`d to help out in any program!

### 8.2.5  Standard C-String Processing Loop

For any other processing that you need to do, write your own loop! Yea! C-string processing loops by-and-large look like this:

```cpp
size_t i{0};
while ( s[i] != '\0' )
{
    // do something with s[i]
```

```
        ++i;
}
```

Do **NOT EVER, EVER, EVER** let anyone catch you doing this:

```cpp
for ( size_t i{0}; i != strlen(s); ++i )
{
    // use s[i] some way...
}
```

This will re-process your entire C-string once for every character within it. That is, if your C-string were $n$ characters long, then $(n + 1)^2$ `char`s will be processed! A simple 99-character C-string, then, would end up processing `10'000` characters!

## 8.3   Arrays as class Members

Let's explore putting arrays into a `class` as member data by using example code. Here is a new take on the `Student` `class` from the `vector` chapter of the last volume:

```cpp
class Student
{
public:

    static const size_t MAX_GRADES{50};
    static const size_t MAX_NAME{95};

private:

    double grades[MAX_GRADES];
    size_t entered_grades;

    char name[MAX_NAME];

public:
    // more stuff to come
};
```

Here we have a plain array of `double` for storing the student's grades and a C-string array to store their name. Of course, only the plain array needs a used positions tracker.

Let's now look at the constructor patterns for these array members. Here is the default constructor:

```cpp
Student(void)
    : entered_grades{0}
{
    name[0] = '\0';
}
```

In much legacy code, you'll find that array members are left out of the member initialization list. This is because in C++98 and C++03 such usage was illegal. In a plain array, it is only necessary to initialize the used tracker to 0. This is because later code won't use the elements beyond that point, anyway. For the C-string member, we need to make sure the first character of the array (position 0) is a null character to keep it a C-string and not just a plain `char` array.

In C++11 and beyond, this can be done in the member initialization list as so:

```cpp
Student(void)
    : grades{0.0},
      entered_grades{0},
      name{'\0'}
{
}
```

The copy constructor just copies all data from the other object without error checks since the other object is of our `class` type and has been error-checked its whole life. (We can trust this guy!) For C++03 and before, this would look like so:

```cpp
Student(const Student & g)
    : entered_grades{g.entered_grades}
{
    strcpy(name, g.name);
    for ( size_t s{0}; s < get_num_grades(); ++s )
    {
        grades[s] = g.grades[s];
    }
}
```

For a modern implementation, it would sadly look exactly the same. Only if we use the `vector`, `array`, or `string` `class`es can we copy the elements from an old version of ourselves. Arrays cannot be copied or assigned.

Another constructor would typically not take values for the `grades` member and just get a value for the `name`:

```cpp
Student(const char new_name[])
    : Student{}
{
    set_name(new_name);
}
```

Here we delegate to the default constructor for cleanup and then mutate the `name` member to avoid overrun concerns. (This new data came from outside the `class`, after all! We can't just trust it to be safe.)

Accessors for the `grades` array are much like those for a `vector` member of a `class`. You ask which one the caller is interested in and return that. You also have a meta-accessor to return the count of used elements in the array member.

```cpp
size_t get_num_grades(void) const
{
    return entered_grades;
}

double get_grade(size_t which) const
{
    return which < get_num_grades() ? grades[which] : -42;
}
```

I've used -42 as the error indicator because I like 42 and teachers might want to use -1 thru -5 — or whatever — as flag values for special circumstances.

The mutator for the `grades` member takes a parameter telling it which grade to change and another telling it the new value:

```cpp
bool set_grade(size_t which, double score)
{
    bool okay{false};
    if ( which < get_num_grades() )
    {
        grades[which] = score;   // check domain of score?
        okay = true;
    }
    return okay;
}
```

Although we should check the domain of the new value, I've eschewed it here to focus on the array management.

But we also need a mutator that adds new elements to the `grades` array like so:

```cpp
bool add_grade(double score)
{
    bool okay = false;
    if ( entered_grades < MAX_GRADES )
    {
        grades[entered_grades] = score;
        ++entered_grades;
        okay = true;
    }
    return okay;
}
```

Again, error-checking the score itself is left to the interested reader.

As to the C-string member (`name`), it also needs care to be taken due to its underlying array-based storage.

Let's start with an accessor for C-string member. They give us an array in which to store a copy of our member. If they give us the declared length, we can protect the copy from overrun. If not, we just copy blithely away...

Why they wouldn't just use our above constant to declare their array may at first elude you, but there are situations where they might want their array for this data to be smaller or larger than ours. They may be retrieving the student's name to print in a table, for instance. Then they'd need only enough to fill the column rather than the entire 94 possible characters of the name! Or they might be storing it into a larger area and wanting to later concatenate into that area more text. You never can tell what the other programmer is going to want to do with the data you give them!

```cpp
void get_name(char copy[], size_t len = 0) const
{
    if ( len <= 1 )                  // should 1 be here or nowhere?
    {
        strcpy(copy, name);
```

```cpp
    }
    else // len > 1
    {
        strncpy(copy, name, len-1);
        copy[len-1] = '\0';
    }
    return;
}
```

I left you with one question: Should 1 be considered a degenerate array length or not? A one-length array isn't a useful C-string as it only holds the null character and no data. I left it with the 0 branch and used `strcpy`. You can move it to the `strncpy` branch if you like, but it'll do nothing useful there!

In a mutator for a C-string member always protect yourself from overrun!

```cpp
bool set_name(const char orig[])
{
    strncpy(name, orig, MAX_NAME-1);
    name[MAX_NAME-1] = '\0';
    return true;
}
```

Some people consider that they should we return `false` when the caller's C-string was too long. But that would require a call to `strlen` — doubling our processing time! I say, "No way!"

Here is the code all in one go and with a fairly complete driver on the book's website. The driver only omits testing for the copy and name-based constructors.

## 8.4   2D Arrays

Multidimensional arrays are. . . well, possible!

### 8.4.1   Declaration

In simplest terms, you declare a two-dimensional array like so:

```cpp
const size_t MAX_ROWS{50},
             MAX_COLS{100};
double array[MAX_ROWS][MAX_COLS];
size_t used_rows, used_cols;
used_rows = used_cols = 0;
```

We still use the `size_t` for declared size info, of course. The base type of the array is up to you and your application. I've chosen `double` here rather arbitrarily. The rows maximum comes first followed by the columns maximum — both in square brackets. The dimensions themselves are up to you and your application. I'm using 50 and 100 to show that it need not be square.

We have used trackers for both dimensions because the user won't necessarily use the full extent of either dimension we allot for them. These start out at 0, of course.

### 8.4.2   Initialization

You can also use the initialization syntax we used before for two-dimensional arrays. Just nest the lists for each row inside an outer list for the number of rows.

```
double array[MAX_ROWS][MAX_COLS] = { { 4.2, 42 },
                                     { 420, 42e2 }
                                   };
```

Each row can be `MAX_ROWS` elements long. If it is shorter, the remaining elements are default constructed as with one-dimensional arrays. There can be at most `MAX_COLS` rows. If there are fewer, all left over rows are default constructed in their entirety. (Here, too, I'd've initialized the used trackers to 2 in each direction instead of 0. *smile*)

### 8.4.3   Arrays of C-Strings

An array of C-strings is also 2D:

```
const size_t MAX_CHARS_PER_LINE{80};
typedef char Line[MAX_CHARS_PER_LINE];

const size_t MAX_LINES{60};
Line page[MAX_LINES];
size_t used_lines{0};
```

This makes the overall structure of `page` be as if declared by:

```
char page[MAX_LINES][MAX_CHARS_PER_LINE];
```

But the use of the `typedef` makes the declaration and sub-array extraction more convenient.

### 8.4.4   Sub-Arrays

As you may well guess from this example, you may subscript a two-dimensional array by fewer than its full indices. Doing so gives you a sub-array of the remaining dimensions. That is:

```
page[i][j]
```

refers to a single `char` within the 2D structure. Whereas:

```
page[i]
```

refers to one particular `Line` of the above 2D structure. And:

```
page
```

refers to the entire 2D structure.

### 8.4.5   Passing to Functions

Passing a multidimensional array to a function is a major pain (unless you are packaging the dimensions in `typedef`s). The first dimension (left-most in declaration) can be left empty just as with a 1D array (the compiler will ignore any value provided there for the formal argument). The second dimension **\*\*MUST\*\*** be filled in with the proper declared constant!!!

Returning to the `double` array of earlier, we might use these functions:

```cpp
void print_elem(const double elem);
void print_row(const double row[], size_t max);
void print_2D(const double arr[][MAX_COLS], size_t max_rows, size_t max_cols);
```

We could then call these functions as:

```cpp
print_elem(data[r][c]);
print_row(data[r], used_cols);
print_2D(data, used_rows, used_cols);
```

### 8.4.5.1  But Why?

The reason for this is that a two-dimensional array is actually stored as a long linearized space with address jumps to the start of each row based on the number of columns. (Did I mention that the length of a row is the same as the number of columns? And vice-versa... *shiver*)

That is, we like to think of 2D memory as looking like this somewhere in the RAM:

```
        MAX_COLS
        0    1    2
M   +----+----+----+
A 0|    |    |    |
X   +----+----+----+
_ 1|    |    |    |
R   +----+----+----+
O 2|    |    |    |
W   +----+----+----+
S 3|    |    |    |
    +----+----+----+
```

But it really looks like this:

```
          M         A         X         _         R         O         W         S
   M A X _ C O L S    M A X _ C O L S    M A X _ C O L S    M A X _ C O L S
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
 0,0  0,1  0,2  0,3  1,0  1,1  1,2  1,3  2,0  2,1  2,2  2,3  3,0  3,1  3,2  3,3
```

We'll talk more about exactly how this happens when we discuss dynamic multi-dimensional arrays, but for now, suffice it to say that the mapping from our natural 2D coordinates of `array[r][c]` would be `array[r*MAX_COLS+c]`. That is, for each row desired, we pass through `MAX_COLS` entries in the array and then we move `c` more entries past that position.

### 8.4.6  An Example

On the website you can find an example of a 2D program using an array of C-strings to store the user's name — one 'word' per row.[14] The goal of the program is quite simple: read in the user's name, shuffle the words, repeat it back.

---

[14]Don't forget to download this program and load it into VS Code or some other capable programmer's editor alongside this document.

On lines 29-37 we input the user's name. We start with a `flush` of `cout` because many implementations don't do this when `peek`ing `cin`. We `peek` to find the newline that ends the input to stop the loop, of course. The `while` head also protects us from running out of space by checking for the maximum number of rows.

Line 34 does the core reading. It uses `setw` to protect the input name component from overrun. It also updates the number of components that have been read. This merging of the `++` into the subscript is frowned upon in modern development as too ugly and confusing. But past generations sought this kind of thing as elegant coding. So even though the current trend is away from this style, you'll find it all over the place in the wild. Don't be afraid! It works just fine and merely needs you to be versed in the differences between pre- and post- increment. We'll go over those in a later chapter on operator overloading.

The comment on lines 35 and 36 mentions that we don't really check that the input component was complete. If the input was followed by something that wasn't a space, then an incomplete word was read and more of it is still in the buffer. We might should warn the user about this, but right now we are just looping around and gathering that trailing bit as its own component separate from its beginning.

Lines 38-42 let the user know if they had more name components than we could handle. Then line 43 cleans up the buffer either way.

The `if` that follows makes sure at least one name was entered. If so we display their name, shuffle it about, and reprint it with a quirky message about the shuffled form being 'cooler'. (The else just prints a smarmy message about them having no name at all.)

Looking below, we find a `swap` function for use by the `shuffle` function. It is well documented and laments not being the everything we'd ever want in a C-string swapper. As it says, we will get around to making an improved one when we discuss `template`s later in the book.

Next is the `shuffle` function itself. As mentioned, it `swap`s C-strings with any that precede them in the array — possibly including themselves! This method is perfectly sound and gives statistically good shufflings of the data. This despite your long years of shuffling cards and shaking dice until they are frayed, scarred, and ruined.

The `display` function is nothing amazing. We've been there and done that before. It just points out that the variable name lowercase l is horrible and should never be used by anyone — ever!

### 8.4.7   Arrays Beyond 2D

All of this extrapolates directly to 3D and beyond. You add each new dimension before the previous ones like so:

```
double ThreeD[MAX_PLANES][MAX_ROWS][MAX_COLS];
```

Here we've added a number of planes (like a 2D sheet) in front of the dimensions for each of these 2D structures. Thus, we'll end up with `MAX_PLANES` sheets of size `MAX_ROWS` by `MAX_COLS` each.

The worst of it is the linearization of the subscripting. Here, accessing position `ThreeD[p][r][c]` would be done with `ThreeD[p*MAX_ROWS*MAX_COLS+r*MAX_COLS+c]`. This can be simplified with Horner's method to have fewer multiplies: `ThreeD[(p*MAX_ROWS+r)*MAX_COLS+c]`.[15]

As can be inferred, all dimensions after the first must be filled in for passing 3D and higher dimensional structures to functions.

---

[15]Can you believe it? A whole method named after you for noticing to factor! *shakes head*

## 8.5  Wrap Up

In summation, we've covered a **LOT** of information in this chapter! We learned how to store data in a statically-sized array. We used this technique and the special nature of the ASCII 0 value to make handling strings of text feasible. We learned how to treat these storage types as members of a `class`. And we learned how to chain them together to make multidimensional storage structures.

I hope this chapter end finds you well and not struggling. If you have any troubles, please see your instructor or a qualified tutor for help! Don't just search the Internet. People are helpful there, but often too helpful. They'll teach you things you aren't prepared for and even give bad advice at times. If you must search, make sure you corroborate any advice with several sources and don't just trust the first blog or other posting you find on a subject.

# Chapter 9

# Memory Management

Memory is managed by the operating system (as are all system resources). Normally, a program is given a fixed amount of memory for its function execution 'stack'. Each time a function is called, its local variables as well as its arguments are carved from a new 'activation record' on this stack. When the function returns, its record is reclaimed so that the space can be used by the next function to be called.[1]

However, your machine generally has quite a bit of RAM left over after it, system drivers, and user applications are loaded. This extra RAM is known (historically) as the 'heap'.[2] If you detect that you need extra memory as your program is running, you can request some of this heap memory from the OS. If it can spare it, you will be given a 'pointer' into that memory. If not, the OS will generally return to you the `nullptr` address.[3]

But before we get into that, let's learn more about these things called pointers.

## 9.1 Pointers

A pointer is a variable that holds the address of some location in memory. That is, the contents of a pointer variable is an indication of where in the memory of the program another piece of information is located. This is typically an offset within the program's memory area in a modern system. This allows indirect access to other memory locations by 'following' the pointer to its destination or target.[4] Diagrammatically, we show pointers like this:

---

[1]This is why it sometimes seems that a function retains values in its local variables if it is called many times in sequence with no intervening calls to other functions.

[2]No, I don't know why. And neither does anyone else recall, it would seem.

[3]Remember our argument to the `time` function for finding the number of seconds since the epoch? Here we'll find out why that was the case!

[4]Some might say pointee by way of employer/employee, but this isn't to everyone's taste.

```
   pointer                               memory
                                         location
                                         +----+
     +---+                               |    |
     | --|------------------------------>|    |
     +---+                               |    |
                                         +----+
```
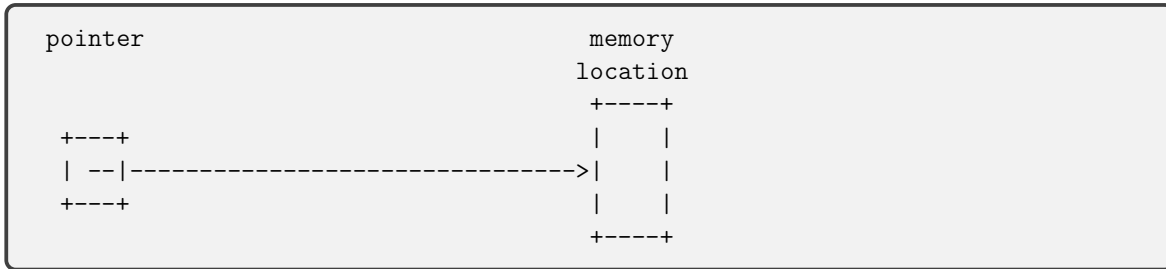
It's just an arrow from the pointer's memory box to the memory destination.

In the past, this allowed programmers to pass such an address to a function for processing without having to make a copy of the original memory area. In C++, we have references to do that sort of thing. So now we typically use pointers only for dynamic memory handling, but on rare occasions we'll have to interface to older/legacy routines that use pointers for 'referring' to original memory.

### 9.1.1  Versus References

We'll be discussing the differences between pointers and references throughout the chapter, but let's just start off that aspect of the discussion by saying that references are a layer of syntactic sugar on top of pointers. But since the compiler manages all the underlying details, we have to — get to — think high-level about them. That makes it much easier to work with concepts like changing another block of memory than using pointers directly as you'll soon find out.

### 9.1.2  Declaration

When declaring a pointer, you must tell the compiler the type of data to which you intend to point. To distinguish a pointer type from a normal type (as used for a variable or constant declaration), our C ancestors used the star/asterisk ($*$) symbol as a modifier for the [base] type. So, to declare a pointer (we'll cleverly call it `ptr`) to a `short` integer, for instance, we would declare this in our code:

```
short * ptr;
```

#### 9.1.2.1  But isn't the Asterisk..?

The spacing around the $*$ symbol here is optional, as usual. This leads to many different styles of spacing and warring camps of programmers. . . again, as usual. The problem with the spacing is that some programmers are lead to believe that the $*$ symbol is completely independent of the type and the identifier. However, given its role as a type modifier, it seems more logical (and, indeed is true) that it is actually associated with the data type . . . which, in turn, defines the very nature of the identifier.

The other point (sorry. . . couldn't avoid it. . . *snicker*) at work here is the fact that the $*$ symbol used to 'modify the data type', doesn't truly *stick* to the type it modifies. Instead, it correlates to the identifier! This leads to programmers creating declarations such as:

```
type* p, q;
```

And then they think that both `p` and `q` are going to be pointers — since the $*$ was modifying the type. Instead, they've declared `p` as a pointer to the correct type and `q` as a plain old variable of that type! To make both identifiers be pointers, you would have to declare them like this:

```
type *p, *q;
```

So, the nature of the thing makes it seem that the ∗ should reside with the type, but the syntax of the thing requires that the ∗ reside with the identifier! Most perplexing! Our champion of choice here is the trusty typedefinition:

```
typedef type * TypePtr;
TypePtr p, q;
```

Now both p and q are correctly declared as pointers to the desired type! The typedefinition makes the ∗ stick to the data type it is modifying as nature intended — language syntax rules be damned! All hail the typedefinition! typedefinition! typedefinition![5]

### 9.1.2.2  Terminology

As mentioned above, when dealing with a pointer, you must also be careful of your vocabulary! Many a fine algorithm has been delayed or outright doomed by one programmer loosely discussing the 'value of the pointer' and another misinterpreting the meaning.

With a pointer, the value is technically the address of some other memory location. Note its basic definition: a pointer is a variable which holds the address of another memory location. Therefore, when we discuss matters involving pointers, we should always clearly define terms amongst ourselves beforehand. A reasonable convention is to use terms like 'target' or 'destination' for the memory location to which the pointer points and then leave the terminology about the pointer itself alone. Then a discussion of the 'value of the pointer' is understood by all involved to mean the pointer's contained address. And the phrases 'value of the target' or 'target's value' would as simply be clear in meaning.

Of course, we could always then take the address of a pointer — and store that in another pointer . . . perhaps that's a topic best left for another section, 'eh?

### 9.1.2.3  To Point, but to where?

What value should we place in a pointer variable to indicate that it does not yet point to a valid address? We long ago created the symbolic constant NULL to represent this idea.

In C, the NULL identifier is, oddly enough, a C-style constant (aka a #define macro with no parameters) which is [typically] equal to the value 0. For years it was a matter of fervid debate in the C++ world as to which is the more proper to use — the named constant or the literal.

Well, technically, the debate is truly over whether the literal or a C++-style constant would be better. But the way the standard, all technical reports from the standards committee, and the stern opinions of Bjarn himself are worded, there seemed to be no real way to define such a beast. . . sadly.

But it appears all those nay-sayers were wrong! Continued efforts produced such a constant. It is named nullptr — note there is no underscore! In compilers supporting this (those supporting C++11), it would be highly preferred to NULL.

So, then, when creating a pointer that you aren't otherwise immediately initializing, always set it to nullptr:

```
double * p = nullptr;
```

### 9.1.2.4  Basic Pointer Operations

There are two basic operators to support pointer actions: & and ∗. The former is used in legacy coding situations to take the address of a variable. This address can then be stored in a pointer:

---

[5]For those interested in a using alias instead, simply do 'using TypePtr = type *;' in your code instead.

```
type object;
type *pointer{&object};
```

This at first glance appears to be the same `&` used for references, but it is an actual unary[6] operator. The `&` used for references is simple syntax to show a variable refers to another memory location in a special way. Context should make it clear which is being used.

The 'latter', then, is the unary `*` operator. It is used to follow a pointer to its destination and use the value stored there:

```
cout << *pointer;
```

This would print the value of the `object` from above, for instance. Again, there is much confusion amongst new programmers between this operation and the syntax used to declare a pointer in the first place. I'd apologize, but it is more the fault of Kernighan and Ritchie — the designers of the C language.[7]

I call this operator the 'follow' operator as it follows the pointer to the destination. But its real name is 'dereference' because of its C heritage of being used to indirectly access other memory — referenced memory. You can use either term, but don't get pointers and references confused, please! They are syntactically and semantically very different ways to access other memory!

#### 9.1.2.4.1  More Depth

For instance, just right off the bat, a reference must be initialized to a valid memory location when created but a pointer can defer this proper initialization until later with either none or a `nullptr` initialization.

In terms of semantics and syntax, the reference is just another name for its original memory location and requires nothing more than the `&` in its declaration for use. For pointers, the initial `*` is important to note what it is, but this is followed up on by loads of dereferencing `*` operations to use the value at the target or no `*` to use the address of the target itself.

There are other differences, but we'll get to them as they become of use.

### 9.1.3  Passing to Functions

Passing a pointer to a function is fairly simple, but certain aspects of it can be confusing to novice programmers.

The basic syntax is:

```
ret func(....type * parg....)
```

Here the `ret` stands in place of the return type and the `....` hold the place of any other arguments this function might take.

The argument `parg` will be a pointer to the specified `type` within the function.

A caller, then, must supply an actual pointer when they call the function:

```
type *pcaller;

func(....pcaller....)
```

---

[6]Having one operand or value to operate on.
[7]The `&` debacle is technically Bjarne's fault.

So, as you can see, the call syntax isn't terribly difficult, either. (Note there is no $*$ on the actual argument!)

But what's happening in memory is a bit disconcerting to programmers new to pointers:

```
pcaller
   +----+                                   +--------+
   |  --|---------------------------------->|  data  |
   +----+                                   +--------+
                                                ^
                                               / \
                                                |
                                                |
                                                |
parg                                            |
   +----+                                       |
   |  --|---------------------------------------+
   +----+
```
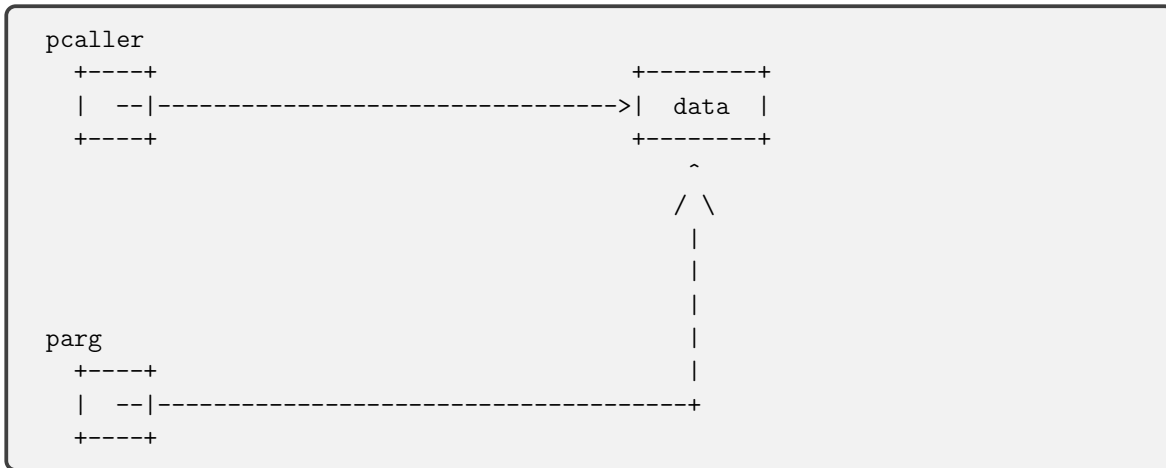
Here we can see that the argument pointer (`parg`) is a copy of the original caller's pointer (`pcaller`) and points to the same memory location. We normally frown upon having two pointers point to the same block of memory, but in this situation, it is almost impossible to avoid.

Because `parg` is a value argument — a copy of its actual argument, it cannot be used to change the actual argument itself. But since it points to the same memory block as the original pointer, it can be used to affect a change to the data there:

```
ret func(....type * parg....)
{
    *parg = new_value;
}
```

To avoid this kind of thing — protecting the data pointed to as well as the original pointer — we can apply `const` to the `type` of the formal argument:

```
ret func(....const type * parg....)
{
    //*parg = new_value;    // not allowed

    cout << *parg;          // still okay
}
```

This can be handy in a situation where you don't want to change the caller's data but just look at it like the earlier `print_arr` or `sum_arr` functions from section 8.1.6.

In a little while (section 9.2), we'll face the need to change where an original pointer points within a function to which it was passed. Since we can't do it with the above syntax, is there a way? Of course! All we need is to refer to the pointer like so:

```
ret func(....type * & parg_ref....)
```

This kind of thing needs to be read from right to left: `parg_ref` is a reference to a pointer to a `type` memory location.[8]

---

[8]In fact, we can read any pointer declaration from right to left and make it always consistent.

Now `parg_ref` doesn't point to the same place as the caller's pointer as a separate pointer itself. Now it refers directly to the caller's pointer like so:

```
pcaller
  +----+                                           +--------+
  |  --|------------------------------->|  data  |
  +----+                                           +--------+
   /\
   ||
   ++=====++
           ||
           ||
parg_ref===++
```
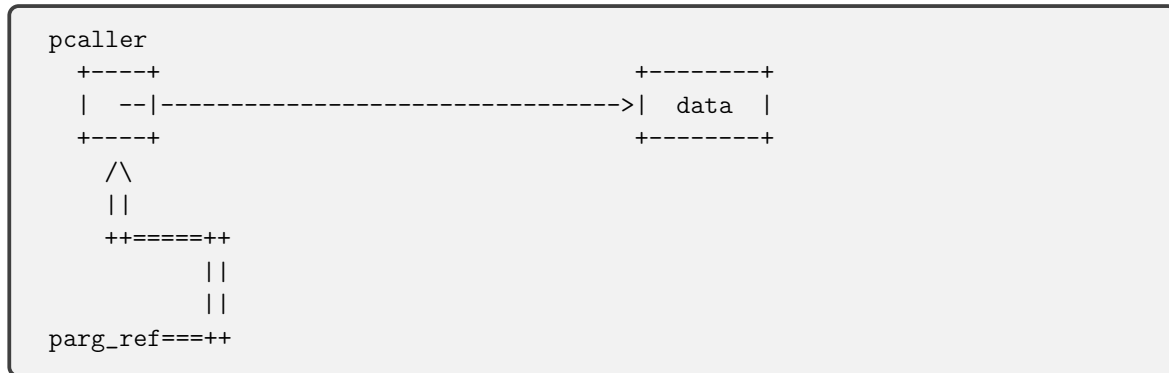
Here the single-line arrow is for a pointer as usual and the double-line arrow is for a reference.

Now, if we could come up with a new address to store in the pointer, we could do something like this:

```
ret func(....type * & parg_ref....)
{
    parg_ref = new_address;    // would make pcaller itself point to
                               // a new destination
}
```

### 9.1.3.1   C-Style Referencing

To be honest, there is another way to effect this task. It was the basic purpose of pointers in our ancestor language C, in fact. It's why the unary * operator is called *dereference* officially.

A C programmer — not us, so don't emulate this! — with a need to change the caller's memory would take its address like so:

```
void add_n(long *x, long n)
{
    *x += n;
    return;
}
```

And then the caller would supply a variable's address using the unary & operator — for taking an address — like so:

```
add_n(&caller_var, 5);
```

They didn't have references, you see, and had to do all caller memory changes by pointer/address. Even their `swap` functions would look like this:

```
void swap(type * x, type * y)
{
    type t{*x};
    *x = *y;
    *y = t;
    return;
}
```

They were sad times... and still are for C programmers! Be thankful Bjarne added the reference syntax to hide all this pointer nonsense!

### 9.1.3.2  Pointers to class Objects

If a pointer is pointing to a `class` base type, we can get to the object's member data (if in a `class` scope) and functions (if `public` or in a `class` scope) via the pointer. This can be done in one of two ways. The first is a bit tricky due to a precedence battle between ∗ for following a pointer and . for member access. It looks like this:

```
(*p).member
```

The seemingly extra parentheses are necessary because without them, the . operator would try to access a member of the pointer itself!

Due to the tedious typing of this, our C ancestors had the forethought to add a new operator to access members of things like `class`es and `struct`s via a pointer: the -> operator. That's a dash and greater than right up next to one another. It works like this:

```
p->member
```

This not only avoids the typing nightmare of using . with pointers, but also reminds us of the diagrams of pointers with their arrows from pointer to destination!

## 9.1.4  Arrays Revisited

If a pointer points to a C-style array, you are allowed to use the subscript operator on it like so:

```
p[0] = new_value;
```

Or for simple access if the base type of the pointer is `const`:

```
cout << p[0];
```

As for function arguments, such a pointer to an array may be passed by the same syntax as above or by the alternative syntax:

```
ret func(....type parg[]....)
```

The watchful reader will note that this is the same syntax we've been using to pass actual arrays to functions as opposed to passing a pointer to an array to a function. This is no coincidence!

As it turns out, when you declare an array like so:

```
type arr[CONST_SIZE];
```

This allocates enough space for `CONST_SIZE` items of the desired `type` on the stack as well as a pointer to the first element of this space:

```
arr
   +----+                                     +==+==+=    =+==+
   |  --|------------------------------------>|  |  |  ... |  |
   +----+                                     +==+==+=    =+==+
```
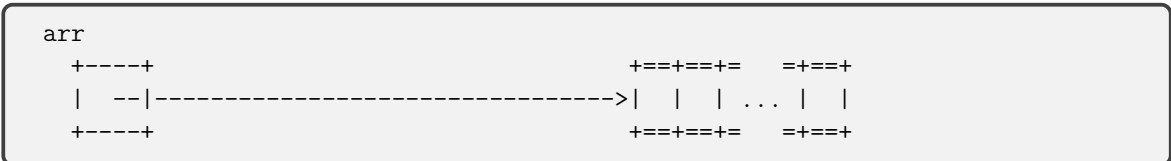
So we've been using pointers for a while and just didn't know it!

Is there any difference between this pointer and one we declare with the $*$ syntax?  A slight bit, actually.  This pointer is a `const`ant pointer.  That is, it will never be able to point to anywhere but its initial destination.  This is as opposed to a pointer to a `const`ant value which cannot change the contents of the destination memory but can point to another block of memory if it needed to.

### 9.1.4.1  Four Types of Constant

What would it look like if we wanted to declare a `const`ant pointer? Something like this:

```
type * const ptr{&object};        // ptr will always point to object
```

(Here I've used `object` in a more general sense of any memory location rather than a variable of a `class` type.)

This must be read, as usual, from right to left for full comprehension: "`ptr` is a `const`ant pointer to a `type` object".

Note that it must be initialized right away like any other `const`ant.

But since we have two ends to any pointer and each can be either `const` or not, we actually have four types of `const`-ness!  (See the grid?  It's a 2x2. . . )

These are:

| Declaration(s) | Meaning |
|---|---|
| `type * ptr1;` | both the pointer's destination and the value at that address can be altered |
| `const type * ptr2a;`<br>`type const * ptr2b;` | the pointer's destination can be altered, but the value at that address can **NOT** be altered |
| `type * const ptr3{&object};` | the pointer's destination can **NOT** be altered, but the value at that address can be altered |
| `const type * const ptr4a{&object};`<br>`type const * const ptr4b{ptr2a};` | neither the pointer's destination nor the value at that address can be altered |

Some people balk at the first row there, but it really is a type of `const`-ness — nothing is `const` or a lack of `const`-ness.

I'm guessing the first row's utility is understood by now, but what of the others'?  The ability to mark a base-type `const`ant has been discussed earlier, but it basically protects the values inside the array from being changed on an array pointer.  For simple pointers to single memory locations, it does the same: protects the target from change.

The marking of the pointer itself as `const`ant is a bit awkward in usage.  The compiler does a lot of things to pointers for us automatically like making an array pointer `const` and keeping it that way.  Or like degrading an array pointer to a normal pointer that is a copy of the original target's address when passing an array to a function.  This protects the original array from being reset to a new target address within the function.  But we can do some of this on our own whenever we want to keep a pointer from changing to a new target, just put a `const` after its $*$ in the declaration.

## 9.1.5  Pointer Math

So what happens internally when we say `arr[p]` in our program? Well, the compiler takes that and makes it into this interesting bit of code: `*(arr + p)`. Here we are adding a number of element positions to a pointer and then following the result.

What is meant by this addition? Technically, it adds p*`sizeof`(type) — where `type` is the base type of the array — to the address the array is located.

But this need for `sizeof` is entirely internal to the compiler. We only need to add a number of elements and the rest is automatic. We normally just use the subscript operator, but we can use the add/follow syntax if we so choose:

```cpp
void print_arr(const double arr[], const size_t count,
               const short wide, const bool newline)
{
    for ( size_t i{0}; i < count; ++i )
    {
        cout << setw(wide) << *(arr + i);
    }
    if ( newline )
    {
        cout << endl;
    }
    return;
}
```

This is, of course, unnatural and lots more typing. So most of us don't do that exactly. However, this addition of positions to pointers to get new pointers is quite a minefield of new ideas! Let's explore.

For instance, since we can use + here, we might equally expect to be able to use the += and ++ shorthand operators, as well. This is entirely true! ++ can even be used on either side.

Another way to write the above function, for instance, would be this:

```cpp
void print_arr(const double arr[], const size_t count,
               const short wide, const bool newline)
{
    size_t i;
    const double * parr;
    for ( parr = arr, i = 0; i < count; ++i, ++parr )
    {
        cout << setw(wide) << *parr;
    }
    if ( newline )
    {
        cout << endl;
    }
    return;
}
```

Here we have a local pointer to `double`s that starts pointed to the same location as `arr` and then advances one position at a time as the loop progresses. We've used the comma operator to put two initializations and two updates into this `for` loop head. Some would argue this was bad style. Others see it as efficient use of programmer time and code space.

Either way, we've removed the clunkier parentheses and addition and replaced it with a nice simple follow and separate increment.

We can even take it a step further. Since adding a position to a pointer gives a new pointer, what must subtracting two pointers yield? That's right! A number of positions! So this version:

```cpp
void print_arr(const double arr[], const size_t count,
               const short wide, const bool newline)
{
    const double * parr;
    for ( parr = arr; (parr - arr) < count; ++parr )
    {
        cout << setw(wide) << *parr;
    }
    if ( newline )
    {
        cout << endl;
    }
    return;
}
```

removes our need to track the index any longer. We still need the `count` to know when to stop, but the `i` variable is no more!

Of course, why didn't we just say `parr < arr+count` in the test? Well, that gets into memory issues with some systems — what is a lesser address, for instance? We could use `!=`, however, and keep things above-board.

Further, we can use this pointer arithmetic to good ends in situations like this:

```cpp
swap(arr+max, arr+cur);
```

Here we are using a C-style `swap` function which takes pointers instead of references — use what you have handy! Note that we just add the offsets to the array pointer instead of addressing an indexed element like so: `&(arr[max])` That would be quite the mess!

Pop quiz: What type of sorting routine would swap a maximum element with a current element, anyway?

### 9.1.6   More From C-Strings

Now that we know the special relationship between pointers and arrays, we can revisit the `cstring` library and find more helpful functions!

Let's start with `strchr` and `strstr`. These take a C-string to search through and either a `char` or another C-string to try to locate within the first argument. The result is either a pointer to the location found or `nullptr` if not found.

Also note that `strcpy`, `strcat`, and so on take pointers as their arguments and so we can do things like this:

```cpp
char src[MAX_S] = "James, Jason",
     dest[MAX_D] = "Welcome xxxxx!";
size_t offsetA = 8, offsetB = 7;
strncpy(dest + offsetA, src + offsetB, 5);
// no '\0' assignment since 5 overwrites our xxxxx
```

Here we are copying from a certain offset into the source C-string instead of its beginning and storing the data not over the entire destination, but after a certain offset into it. So above, `dest` should now equal `"Welcome Jason!"`.

Finally, there is the infamous `strtok` function. Some people fear it and some people revere it. Just be careful and read its documentation carefully as well. You can find out more about it on any Unix-style box by typing `man strtok` at the terminal prompt. Or you can look this up on the Web to find similar information.

In addition to these `cstring` functions, we can also update our end-of-string loop like so:

```cpp
char * p{str};          // where str is a C-string
while ( *p != '\0' )
{
    // act on *p
    ++p;
}
```

Now we've done it with no `[]` and no index variable. Awesome!

### 9.1.7   Weirdness!

Since you can add a number of element positions to a pointer, it stands to reason that you can also subtract them as well. So it does make sense at times to do something like: `*(p - 4)` or even `p[-4]`. But you must be careful to make sure `p` here is not the original address of the array! Make sure it is at least 4 positions away from that address, in fact. If not, you are accessing memory that isn't yours and this is dangerous!

In other disturbing subscripting news, addition is commutative. This means that the following transformation sequence is viable:

```cpp
arr[i] <==> *(arr+i) <==> *(i+arr) <==> i[arr]
```

Not all compilers allow it, but when it works, it is terrible fun as a prank on the new hire.

Finally, make sure you document carefully whether your function's pointer argument should be a single object or an array. There is no way to know what the caller has actually sent you, after all, so making sure they know what you wanted is ever-so-important!

### 9.1.8   Iterators

`iterator`s are to `vector`s and `string` `class`es what pointers are to arrays and C-strings ... sorta:

| `class`es | **Built-Ins** |
|---|---|
| vector/string | array/C-string |
| at/[]/size_type | []/size_t |
| iterator | pointer |

#### 9.1.8.1   Essential Usage

##### 9.1.8.1.1   Declaration, Initialization, and Access

An `iterator` holds a reference to the data at a certain position within a `string` or `vector`. To declare one, you use this syntax:

```cpp
vector<double>::iterator itr;
```

Here we have an `iterator` that can hold a reference to a `double` within a `vector` of `double`s. To make it actually hold such a reference, we must assign it:

```
vector<double> vec;

itr = vec.begin();
```

Here we've made it 'iterate' the first (0th) element of the `vector` `vec`.[9]  Now, how do we access that `double`?!  Use the unary ∗ operator (dereference...remember?):

```
cout << *itr;    // print element itr iterates

*itr = 4.2;      // store 4.2 in element itr iterates
```

The question remains, does `itr` iterate anything?  Hunh?  We assigned it — and even printed it, but does that position exist within the `vector`?  Oh...  Let's check:

```
if ( itr != vec.end() )
{
    cout << *itr;    // print element itr iterates
    *itr = 4.2;      // store 4.2 in element itr iterates
}
```

Much better!  Now we know it exists before we store into it or use it — not doing so would be a definite no-no.  What's that?  What's `.end()` there?  That returns an `iterator` to one past the last element in the container.  So you can use it to detect invalid positions within an `iterator`.  This is used a lot as a flag from functions to indicate they couldn't do something.  It is also used as the end of an iterated range of positions — as usual, a range of values in C++ has the last position excluded and the first included.

### 9.1.8.2  Intermediate Usage

#### 9.1.8.2.1  iterator Math

All the pointer math we learned earlier work in `iterator` versions:

```
iterator + size_type
size_type + iterator
iter2 - iter1 (--> size_type)
iterator += size_type
iterator - size_type
iterator -= size_type
iterator++, ++iterator
iterator--, --iterator
```

Just make sure that the `size_type` you are adding/subtracting is legitimate for the current `vector`/`string`'s `size()`!

#### 9.1.8.2.2  Passing iterators to Functions

If you pass an `iterator` to a function, you do **NOT** need to also pass the `vector` into which it iterates in order to affect that `vector`'s elements!!!

That's right.  Remember that an `iterator` *holds* a REFERENCE to the `vector`'s element and so you can alter a `vector` element without even having the `vector` at hand!

---

[9]This used to be done with raw pointers but is now almost always coded as a `class` that overloads the proper `operator`s to make it function just like a pointer with additional sanity checks.  For more on `operator` overloading, see chapter 11.

```cpp
void swap(vector<double>::iterator a,
          vector<double>::iterator b)
{
    double c{*a};
    *a = *b;
    *b = c;
    return;
}
```

Now we can call this function with:

```cpp
swap(itr, itr+1);
```

To `swap` two adjacent elements in a `vector`.

Pop quiz: What type of sorting routine would swap two neighboring elements, anyway?

### 9.1.8.2.3   The Need to NOT Change

If a `vector` is `const`ant, you cannot use an `iterator` into it. Instead you need to use a `const_iterator` — which holds a `const`ant reference to the `vector`'s element. (You can also use a `const_iterator` on a regular vector when you just don't want to change the contents of any `vector` or `string` accidentally.)

The usage of `const_iterator` is identical to `iterator` except that you cannot store into the dereferenced value.

This all works because the `const` keyword is useful to overload `class` functions in addition to their argument lists. What? Yeah. A `vector` or `string` which is a `const`ant will always call the `const` version of the `begin` function and we'll receive a `const_iterator` as a result. That way you cannot get a plain `iterator` into a `const`ant vector or string — no accidental changes!

But what about if I just want to look at the elements in a non-`const` vector or `string`? Well, through the magic of inheritance (which we'll discuss later in the book), the `iterator` type is upwardly compatible with the `const_iterator` `class`. (This is somewhat like the way you can store a `double` value into a `long double` but not the other way around.)

### 9.1.8.3   Odds and Ends

With `vector`s, you get the additional benefits of access to:

```cpp
.insert(ip, val)
.insert(ip, ipsb, ipse)
.erase(ipb, ipe)
.erase(ip)
.assign(ipsb, ipse)
```

Where `ip` is an `iterator` position, an 's' indicates the 'source' vector and 'b' and 'e' represent 'begin' and 'end', respectively. These pairs are treated as `begin` and `end` `iterator`s on a range and so the end `iterator` is not placed into the sequence — ends are **ALWAYS** exclusive.

(There are iterator versions of these functions for the `string` `class` as well, but we already had such functionality with `size_type`s, so it isn't that big of a deal. *smile* Oh, but `string` also has `iterator` versions of its `replace` functions!)

So instead of coding a loop for insertion or removal like we did in the first volume, we can simply code:

```
vec.insert(vec.begin()+4, 4.2);
```

This would insert the value 4.2 in front of position 4 in the `vector`. The `iterator` in question doesn't have to be directly related to `.begin()`, of course. It could have come from some other source. In fact, most of these functions give you a new `iterator`! They return an `iterator` to the first element in the affected region: `insert` returns an `iterator` to the just inserted element or first thereof and `erase` returns an `iterator` to the first element following those erased or `.end()` if nothing follows. Only `assign` returns nothing.

There are also constructors for both `string` and `vector` `class`es which accept pairs of `iterators` delimiting a range from which to initialize the new sequence.

In fact, the mechanism by which these `iterators` are passed to the constructors is so powerful that it can accept anything that acts like an `iterator` — even an old pointer! This finally gave us a way to initialize a `vector` without a horrid sequence of `push_backs` before we had initialization syntax for vectors added in C++11:

```
const ____ arr[SIZE] = { a, b, c, ..., d };
                 // arr              arr+SIZE


vector<____> vec1(arr, arr+SIZE);


// was used in older code instead of the now more convenient:
vector<____> vec2 = { a, b, c, ..., d };
```

### 9.1.8.4   Invalidation

When is an `iterator` not really an `iterator`? Sounds like a trick question, but I'm being quite serious!

Note that `iterators` into a `vector`/`string` which are at or after an `erased` position are considered 'invalidated' — they should not be dereferenced . . . upon pain of death!

```
itr2 ---------------------------+
                                |
                               \ /
                                `
+----+----+----+----+----+----+----+----+
| aa | bb | cc | dd | ee | ff | gg | hh |    vec [8]
+----+----+----+----+----+----+----+----+
               ^
              / \
               |
itr ------------+
```

Where the [8] is signifying that `vec` contains 8 elements.

Now let's `erase` two elements starting with `itr`:

```
vec.erase(itr, itr + 2);
```

Even though the item gg is still in the `vector`, I am not allowed to dereference `itr2` because it and `itr` and any other `iterators` which iterated at/beyond `itr` are considered invalidated by the erasure!

Furthermore, if I were to accidentally dereference `itr2`, I would be erroneously using memory I no longer have legitimate access to. (It may still contain a gg, but it won't be the one from *inside* the

vector!)

Look at the `vector` after the `erase`:

```
itr2 ---------------------------+
                                |
                               \ /
                                `

+----+----+----+----+----+----+----+----+
| aa | bb | cc | ff | gg | hh | gg | hh |    vec [6]
+----+----+----+----+----+----+----+----+
                ^
               / \
                |
itr ------------+
```
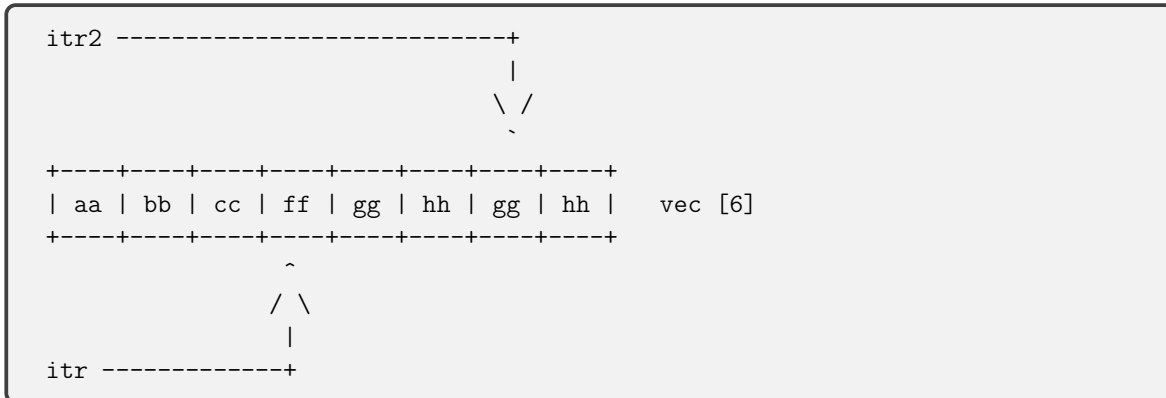
Notice that `dd` and `ee` are gone and `vec` is now only 6 elements long — but seems to be physically 8 positions in length! That's because there was no need to physically remove the memory locations after shifting the data down to cover up the `erased` elements. The `vector` merely remembers that those last two positions are not part of the data anymore.

Unfortunately, no one told the `iterators` that certain positions were no longer valid! Here we've only created two `iterators` into the `vector`, but a typical application may have tens or more `iterators` into a `vector` when an `erase`-ure occurs! It is somewhere between impractical to impossible for the `vector` to inform all `iterators` of position invalidation.

Still, why are the `iterators` invalid? They both still iterate positions in the physical `vector`, right? Well, yes, but they are no longer iterating the data the programmer expected them to iterate. `itr` no longer iterates `dd` because it is no longer part of the `vector`. And `itr2` no longer iterates `gg` because the `vector`'s actual data `gg` is two positions left of `itr2`'s position. Notice what happens to `itr2` when I perform the following assignment:

```
vec[4] = "qq";
```

See? Oh, sorry. Here:

```
itr2 ---------------------------+
                                |
                               \ /
                                `

+----+----+----+----+----+----+----+----+
| aa | bb | cc | ff | qq | hh | gg | hh |    vec [6]
+----+----+----+----+----+----+----+----+
                ^
               / \
                |
itr ------------+
```
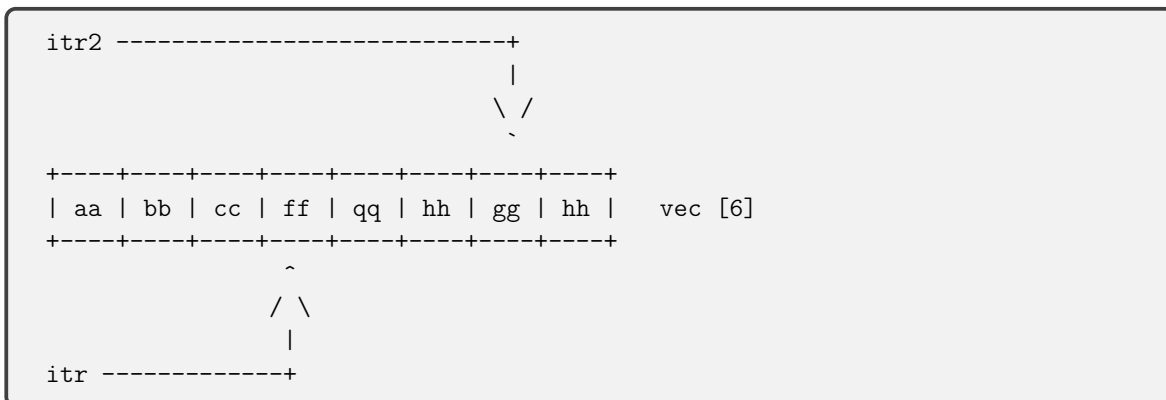
What? You still don't see it? Look where `itr2` iterates: `gg`. But there is no longer any `gg` in the `vector`! Now it is more obvious why `itr2` is considered invalidated, right?

Still not sure about `itr`, eh? Well, all I can say is that it was expected to have been iterating `dd` and we no longer have such a beast! It currently iterates `ff` — whomever that is!

Essentially, `itr2` is lost to us, but we can return `itr` to a valid state by using the result from `erase`:

```
itr = vec.erase(itr, itr + 2);      // erase 2 elements starting with itr
                                     // itr updated to the first remaining element
                                     // after the erasure
```

Most programmers, though simply re-establish the position for an invalidated `iterator` the way they did in the first place. This might be by some search technique or relative positioning or . . .

While this same issue can happen to pointers, no one seems to care. Just FYI. . .

#### 9.1.8.5 For the Adventurous

But I had a nice algorithm that used `size_types` to process my `string` (or `vector`) in reverse. How can I do that with `iterators`?

Well, there are also `reverse_iterators`:

```
vector<double>::reverse_iterator ritr;
vector<double>::const_reverse_iterator ritr2;
```

These are similar to *forward* `iterators` in behavior — only in reverse! For instance, you can initialize them with the `rbegin` function and check them for validity with the `rend` function.

```
ritr = vec.rbegin();
if ( ritr != vec.rend() )
{
    cout << *ritr;
    *ritr = 4.2;
}
```

The most confounding thing about them to most beginners is that when you increment them — `++ritr` — they move to a previous element. Remember, though, they are iterating in reverse!

```
for ( ritr = vec.rbegin(); ritr-vec.rbegin() != vec.size(); ++ritr )
{
    // use *ritr -- it's safe!  (I promise...)
}
```

This loop will go through all the elements of the `vector vec` from the last to the first.

Likewise, decrementing them will move to the following element.

## 9.2 Dynamic Memory

As mentioned above, sometimes you realize you need more memory than you've planned on having. This is usually done on purpose. Because traditional array declarations can waste significant amounts of memory for many runs and not have enough memory for others we will opt to have either a small amount of memory in the heap or even no memory with a `nullptr` initialization and then to ask the OS for more memory later. The asking is called allocation. Let's delve deeper. . .

### 9.2.1 Allocation and Deallocation

The way that you would request such extra memory is with the operators `new` and `new[]`. `new` is used to request (aka allocate) memory for a single object whereas `new[]` is used to allocate memory for an array of objects.

When calling `new`, you can supply constructor arguments so that the newly created object will be initialized appropriately. If you don't, it will be default constructed.

`new[]`, however, always default constructs **ALL** of your array's element objects. (There just isn't the syntax for placing a custom constructor call. . . sorry.)

For example:

```
double * p{new double},          // p{new double(4.2)} to initialize
       * q{new double [10]};      // ten elements all default constructed
```

Note that the `10` passed to the `[]` of `new[]` here can be any integer expression — not just literals or constants like with static[10] array allocation!

### 9.2.1.1 try/catch vs nothrow/nullptr

I said above that the OS **traditionally** returned a `nullptr` to indicate it couldn't find the memory requested in a `new` or `new[]` operation. But that isn't the case in C++ any longer. Now the default behavior of these operators is to `throw` an `exception` at you when they can't get the memory from the OS.

Thus, we can let this `exception` go and let it crash the program, but that seems extreme given that it won't be hard to protect the rest of the program from missing memory. So, what I recommend to avoid this potential crash is one of two approaches.

#### 9.2.1.1.1 try/catch

The first is a bit bulky but works fine. This is to use a `try`/`catch` around any `new` or `new[]` operations you do and set the pointer to `nullptr` in the `catch`:

```
try
{
    p = new _____ [len];
}
catch ( bad_alloc /*const& prob*/ ) // catch the exception in variable..?
{
    p = nullptr;  // reinstate old behavior
    // cout << prob.what();  // returns a string describing the problem
    // throw prob;           // rethrow so that others will be in the know?
}
```

Here we try to allocate an array to pointer p that is `len` long. When we `catch` `bad_alloc`, we know not enough memory was available and we reset the pointer to `nullptr`. `catch`ing the `bad_alloc` `exception` will require *#include*'ing the library `new`. It is in the `std` `namespace` as well.

Other options at this juncture are, of course, to `catch` the `exception` with a variable — preferably by `const&` to avoid copies. Then we can print a message for the user about the issue using the `what` method all `exception`s have.

We can also choose to re-`throw` the `exception`. To do this without `catch`ing by name, just use the statement `throw`; in the `catch` block.[11]

---

[10]Static here refers to arrays declared as local variables which appear on the stack and are set up at compile-time rather than dynamically during the run of the program.

[11]For more on exceptions see section 12.2.

#### 9.2.1.1.2   nothrow/nullptr

The second method is to use overloaded versions of `new` and `new[]` that won't `throw` an `exception` in the first place. There is one extra argument to this overload and it is always the object `nothrow`:

```
p = new(nothrow) _____ [len];
q = new(nothrow) _____;          // q = new(nothrow) _____{value};
```

This object is found in the `new` library if you have trouble using it. It is in the `std` `namespace` as well.

This overload automatically puts the pointer to `nullptr` when the OS can't get the requested amount of memory for us.

#### 9.2.1.2   Use of Dynamic Memory

Whichever way you get back to the original `nullptr`-for-failure behavior, you now have a way to protect the rest of your code from trying to use a pointer that couldn't get the needed memory: an `if` check.

Below I've used the `nothrow` overloads of `new` and `new[]`, but a `try`/`catch` would work out just fine as well.

```
double * p{new(nothrow) double},
       * q{new(nothrow) double [10]};

if ( p != nullptr )
{
    *p = 42;
}
if ( q != nullptr )
{
    *q = 4.2;
    q[3] = 2.4;
}
```

We just surround any usage of the dynamically allocated pointer by an `if` that checks for `nullptr`. Here we are checking for the purpose of storing new data in the dynamic memory. But next we'll check again for access/reading of the stored data:

```
if ( p != nullptr )
{
    cout << *p << '\n';    // p[0]
}
if ( q != nullptr )
{
    cout << q[0] << '\t' << *(q+3) << '\n';
}
```

Note that usage of memory in a program is typically done in helper functions and those functions aren't naturally synchronized with the allocation step. Some other programmer on the team may call them out of order and really mess up a program run if we don't put these `nullptr` checks around all usages of the dynamic memory.

### 9.2.1.3   Cleaning Up with delete

To release the memory back to the OS when you are done, use either the `delete` or `delete[]` operator as appropriate for the original allocation. That is, `delete` when you allocated a single object with `new` or `delete[]` when you allocated an array of objects with `new[]`. Doing so will give the OS back control of that heap space and it is no longer allowed for you to use it in any way.

```
delete p;
delete [] q;
```

No special care has to be taken if the pointer in question holds the value `nullptr`. This was the case long ago in very early C++ compilers, but that has long since been taken out as a 'feature'. We can now simply `delete` a `nullptr` with abandon!

### 9.2.1.4   nullptr Assignment After delete

Even though the OS will get back control of the released memory, **NEITHER** the OS **NOR** the `delete` operators will alter your pointer's content. That is, you'd still have the dynamic address and could technically, if not legally, access it!!!

In addition, the data on the heap you are pointing to won't be altered in any way. If you are working in a highly secure arena, you might consider overwriting it with some masking values as well.

So, if the program will continue to run for a bit — or you are just extra careful — you should really assign the `nullptr` address to your deallocated pointer. This will avoid using that released memory after the fact by accident.

#### 9.2.1.4.1   Dynamic const Pointers

What, some of you might be thinking, about const pointers? Can they be dynamically allocated?

Let's see:

```
double * const z{new double [12]};

if ( z != nullptr )
{
    *z = 3.4;
    cout << z[0] << '\n';
}
```

So far so good. We got the space, we checked it and used it. What's left? Oh, yeah, deallocation! Let's try that:

```
delete [] z;
```

It looks okay at first glance — the pointer has been released to the OS, after all. But then we realize that we can't do `nullptr` assignment because z is a constant pointer! That means we can't protect ourselves from reusing this memory during the rest of the program. So unless this happens inside a particular block scope — like a function or loop body — we probably don't want to do this — ever!

## 9.2.2   Reallocation of a Dynamic Array

When you have a dynamically allocated array of objects and realize that you are going to need *more* space, you'll need to re-allocate the array.

The process is simple enough, but highly prone to error! Be careful!!!

The pseudocode is as so:

```
-- attempt to allocate new space
-- if successful,
    -- copy old data over to new space
    -- deallocate old space
    -- re-point pointer
    -- nullptr-out temporary pointer
    -- update length/size counter(s)
```

The only step in this process that can move is the last one that updates any length or size information related to the dynamic memory. Typically we carry a `size_t` alongside any dynamically allocated array to remember the upper bound on the usable space. We also often keep another `size_t` to track how many spaces within the array we've filled in.

This process is analogous to what we did with static arrays back in the day (section 8.1.2), but the upper bound is now a variable instead of a constant.

What might this look like in code, though? Here is a possible implementation:

```cpp
bool reallocate(double * & p,          // pointer, we hope to change its target!
                size_t & physical_size, // allocated size -- may update
                size_t & logical_size,  // number of used locations -- may update
                ssize_t more_space)     // how many more elements they need/want
{                                       // -- may be negative for shrinking
    double * t{new(nothrow) double [physical_size + more_space]};
    bool okay{t != nullptr};
    if ( okay )
    {
        okay = true;
        physical_size += more_space;
        logical_size = min(logical_size, physical_size);
        for ( size_t i{0}; i != logical_size; ++i )
        {
            t[i] = p[i];
        }
        delete [] p;
        p = t;
        //t = nullptr;  // not necessary since t is about to disappear!
    }
    return okay;
}
```

Here I've moved the size/counter updates to before the copying loop to provide the new upper bound for that loop in a possibly changed number of actual elements. The reason we've used a minimum finding function to change the logical size (actual number of elements) is that we might have shrunk the array instead of making it bigger!

How can this be?! Well, for the `more_space` parameter, I used not a `size_t` but an `ssize_t`. This is the POSIX data type that is the `signed` counterpart to the `unsigned size_t`. Although not a part of C++ itself, this type is often available and fills a much-needed gap in the type system.

If you are wary to use a non-standard type, however, you can make your own like so:

```cpp
using Ssize_t = make_signed_t<size_t>;
```

The `make_signed_t` tool is located in the `type_traits` library.

### 9.2.2.1  Chunks vs Multipliers

To decide how many more elements to allocate, one of two schemes is typically chosen:

- common sized chunks (arithmetic growth)

- common multiplier (geometric growth)

With a chunk sizing you'll typically determine the actual size of the chunks by statistical analysis of a program's expected data. This is easily realized with the above implementation by using the chunk size as the amount of `more_space` requested.

But do you have to rewrite the above code to use the common multiplier technique? Of course not! Just use `(multiplier-1)*physical_size` as the amount of `more_space` requested.

The most common multiplier in use is 2. There was much research in the last decade of the $20^{th}$ century indicating this as the most efficient multiplier to use in programs due to memory patterns and typical OS allocation schemes.

If using the multiplier scheme and also allowing your caller to set an initial size for the array, provide some mechanism to adjust their requested size into a power of your multiplier. This will make things go more smoothly should you decide to allow both shrinkage and growth in your system. In fact, it is so important, I'm just going to tell it straight out! It is a mildly clever bit of math involving logarithms, so I know you could come up with it, but why not share, right? The next higher power of two above $x$ — or $x$ if it already is one — comes out to be:

$$2^{\lceil \log_2 x \rceil}$$

Special care should be taken, of course, when $x = 0$, since logarithms don't really work there...

#### 9.2.2.1.1  Amortization

Only the multiplier scheme will amortize the cost of reallocation. This cost comes from both the actual allocation of more memory from the OS — a small cost — and the copying of the elements from the old space to the new space — a HUGE cost!

But what is amortization again? That's a term that comes from accounting circles and means basically to average out the cost of something over time. For instance, a company wanting to buy a computer might be hesitant to outlay thousands of dollars for a top-of-the-line model. But if they see evidence that these models can last for many years without much in the way of maintenance costs, they might be more accepting of the idea.

Here we are averaging out the cost of the element copies over the time they are not needed by doubling the space we have on hand. If we had 8 elements on hand, for instance, and grew to 16 slots, we would incur a copy time of '8'. But now we won't have to grow for 8 more additions of data so those 8 copies average away.

#### 9.2.2.2  Growth vs Shrinkage

Most new programmers are okay with the idea of getting more memory from the OS for dynamic arrays that have grown beyond their original bounds. That doesn't sound too far-fetched. But shrinking space? What's that about?!

Well, some programs don't run for just a few minutes or hours. Some programs run for days or weeks at a time on large servers. Such applications may see a cyclic need for more memory during peak times. Then, during off hours, their memory needs will wane.

If they keep the extra memory they aren't needing any more, they are not playing well with all the other programs on the system! Other programs might be using more memory during the same time that

this one doesn't need as much. In an extreme case, other programs might be failing to do their work because they can't get enough memory to run!

To avoid these problems and just play nicely with other apps, we set our sights on shrinking memory when it is no longer necessary. This would involve sending in a `more_space` parameter to the above function that was negative, of course.

#### 9.2.2.2.1    Pessimistic Shrinkage

But if we just shrink a chunk or half each time we can, we are in for a possible race condition. This could happen, for instance, when the program just dropped an item, but is about to add an item right afterward. If this happened over and over at some juncture, we'd be stuck shrinking and growing back and forth. Not very efficient after all!

To avoid this race of allocations and deallocations, we typically use pessimistic shrinkage. In this scheme we don't drop back our memory usage a single step until we've got two steps worth of memory unused.

For instance, in a doubling scheme, we'd wait until we were at a quarter of our allocated size used before shrinking by half.

### 9.2.3    Dynamic class Members

Often it is important to have members of a `class` placed on the heap. (Perhaps it is an array we want to grow/shrink!) This is possible but takes some care. Let's explore. . .

#### 9.2.3.1    Where the Parts Go

To place dynamic memory management in a `class`, we merely need to identify the three portions of the process and place them correctly into the `class` structure.

The first portion of the dynamic memory process is allocation. This should fairly obviously be placed in the constructor — or some helper function called from there. It might also happen in a mutator. This makes the helper idea a real win-win.

Second comes use of the dynamically allocated memory. This can be placed in any and all `class` functions which need to use the memory — but don't forget to put `nullptr` checks in **EVERY SINGLE ONE OF THEM**!!!

Last would be deallocation. Where should that go? There seems to be no answer. *snap* Guess we can't do this, eh?

Of course we can! But we'll need a new kind of method! Let's call it a deconstructor! No. . . that would make sense! Okay, okay. Then the destructor. Much better. Now it's an homage to a freaky — albeit really cool — 80s movie.[12]

#### 9.2.3.2    Destructors

The name of the destructor shall be formed similarly to that of the constructors: from the name of the `class`. However, to distinguish it from the constructors, we'll place a tilde (~) in front of it. (Since it is going to be used for memory deallocation, we can maybe think of it as waving bye-bye to our old memory. *chuckle*)

(Just so you know, the tilde in the destructor's name has **ABSOLUTELY NOTHING** to do with the tilde operator. We'll talk about that operator when we discuss files, flags, and operator overloading later. But it is **COMPLETELY UNRELATED** to dynamic memory management!!!)

---

[12]Not sure if this line is repeated in the newer remakes of the movie, but I'll never forget when the self-proclaimed god tells the crew to "choose the form of the destructor!"

For the following examples, presume the following `class` and dynamic members:

```cpp
class DynC
{
    double * p;     // p points to a dynamic array of data
    size_t max,     // allocated maximum elements
           cur;     // currently used elements
public:
    DynC(void)
        : p(nullptr), max(0), cur(0)
    {
    }

    DynC(size_t MAX)
        : p(new(nothrow) double[MAX]),  // find next 2^n past MAX if growing
          max(MAX), cur(0)
    {
        max = p == nullptr ? 0 : max;
    }

    ~DynC(void)     // destructor (also dtor)
    {
        delete [] p;
        // p = nullptr;
        // max = cur = 0;
    }
};
```

This `class` — DynC because it is a `class` with a dynamic member — is just a demonstrator model and doesn't contain all the features we'll want in a typical situation like this in live code. But it will serve its purpose to show us what methods are needed beyond the destructor and why each is necessary.

Here we see that the dynamic array is defaulted to just the `nullptr` value — no allocation at all. Then in the constructor that takes a size, we try to allocate that amount of space and reset the `max` member if we are unsuccessful. Finally, the destructor frees the space for the dynamic member. It could reset the contents of all the members then, but it won't be necessary as the destructor is being called only when an object is leaving scope.

In fact, they will be automatically called in a manner analogous to constructors. Destructors are called when objects fall out of scope — the ends of their lifetimes. Objects' destructors are, in fact, called in the reverse order of their constructors.

For example, let's say we had a scope with two objects of our `class` DynC:

```cpp
{
    DynC A(90), B(1000);
}
```

At the close curly brace there, both objects will be destructed automatically just as they were constructed automatically when we declared them. But, since we declared A before B, it was constructed first. That means that A will be destructed last:

```cpp
{
    DynC A(90), B(1000);  // construct A then construct B
```

```
} // destruct B then destruct A
```

This at first seems like just odd trivia, but it comes into play in the discussions that follow.

### 9.2.3.3  Copy Constructor

Beware copying objects one unto another. During these times, your `class` is in particular danger of leaving dangling pointers laying about the system and/or causing future double-deallocations of dynamic memory!

And remember that this is no small matter, since copying happens so frequently in C++. Not only things like this:

```
DynC obj1;
DynC obj2{obj1};  // copy constructed
```

But also situations like these:

```
DynC func(DynC arg);
```

Here, the `arg` will be copy constructed from the actual argument in the call and the `return` value will be copy constructed from the expression on the `return` statement!

Let's say we revise the above situation to look like this:

```
{
    DynC A(90);  // construct A
    DynC B(A);   // copy construct B from A
}
```

The default copy constructor just takes the value of `A`'s pointer member and copies it directly into `B`'s pointer member and so we might end up with this kind of memory structure:

```
    +=============+
    |   +----+    |
A   |   |   -|----|--------------------------+
    |   +----+    |                         \|/
    +=============+                   +---------+
                                      |  DATA   |
    +=============+                   +---------+
    |   +----+    |                         /|\
B   |   |   -|----|--------------------------+
    |   +----+    |
    +=============+
```

When the B object destructs and releases the `DATA` to the operating system, all is fine:

```
{
    DynC A(90);  // construct A
    DynC B(A);   // copy construct B from A
} // destruct B and then A
```

But when the `A` object later destructs itself and tries to release the **SAME MEMORY** to the operating system, the best case is confusion. The worst case might be actual damage within the memory subsystem of the OS. *shiver*

This simple problem — double-deletion of a resource — assumes that the older object exists in the same scope as the new object and they die relatively close together.

Look rather at this situation:

```
...f(DynC B);


{
    DynC A(90);      // construct A
    f(A);            // B is copy constructed from A
    // A is still here, but B was destructed
    // as the function returned; A's memory is
    // no longer valid!
} // now A is finally destructed!
```

As you can see, if the new object is a function argument and the old object will go on living after the new object exits scope — when the function returns — you face various memory violation errors as the old object accesses memory your program no longer owns!

To fix the problems caused by the compiler-supplied copy constructor, we can do something like this:

```
DynC::DynC(const DynC & dc)
    : p(new(nothrow) double[dc.max]), max(dc.max), cur(dc.cur)
{
    if ( p != nullptr )
    {
        for ( size_t i{0}; i != cur; ++i )
        {
            p[i] = dc.p[i];
        }
    }
    else
    {
        cur = max = 0;
    }
}
```

Now the newly constructed object will have its own dynamic memory area and its own copy of the data from the old object. When the new object is destructed, it will only affect its own dynamic memory — not the older object's.

This will alleviate the double-free and dangling pointer issues the compiler-supplied copy constructor gave us when combined with a proper destructor.

### 9.2.3.3.1 Another Approach

The above approach says to reset the counter members to $0$ to reflect their pointer being `nullptr` for consistency of the object and so later code won't be harmed when other member functions try to use any of them to safeguard pointer target usage. This sometimes is accompanied by a function to access the object's state called a validator or such. The purpose of this would be to let the caller know that the object is currently in possession of valid memory and not a `nullptr` which would prove useless to them.

Some prefer not to have a validator at all and just know all objects are in a valid state in the program. Or at least have a more 'in-your-face' way to know an object has ended up invalid. This would involve the use of exceptions. You would `throw` an exception in the `else` above and the code that tried to create the object would have to `try` to `catch` it. The main reason I don't like this here is that copy

constructors are called all over the place — value arguments to name one of the most difficult. We can't really `catch` while passing an argument to a function. We'd have to `try` all function calls that pass their arguments by value and then `catch` the exception. Do you know how many programmers pass by value instead of `const&` around here? So many!!!

Anyway, if you want to use this technique, read more about it in section .

### 9.2.3.4 operator=

The pictorial representation above also goes for a poorly defined assignment `operator` (aka the one provided by the compiler). That is like here:

```
DynC obj1, obj2;

obj2 = obj1;        // obj2 is assigned to be a copy of obj1
```

Not when being constructed, but sometime later in the code when one is assigned a new value with the single equal `operator` (=).

When the objects later destruct, the same problems may occur. So we might expect our assignment `operator` to look something like this:

```
void DynC::operator = (const DynC & dc)
{
    p = new(nothrow) double[dc.max];  // make a new me?
    if ( p != nullptr )
    {
        max = dc.max;   // yea!  now...
        cur = dc.cur;   //          ...to copy...
        for ( size_t i{0}; i != cur; ++i )  //  ...over their stuff!
        {
            p[i] = dc.p[i];
        }
    }
    else
    {
        cur = max = 0;  // oh no!!  *snaps fingers*
    }
    return;
}
```

Such an `operator` function will be called whenever the compiler sees the `operator` itself in action. In more detail than above:

```
DynC obj1, obj2;

obj1 = obj2;   // calls our operator= function; effectively like this:
               //    obj1.operator=(obj2);
```

So the object on the left of the `operator` is used as the calling object for the member function call. (More on that when we overload other `operator`s later.)

But, the threat is bigger with the assignment `operator` because it is replacing an object that currently exists with a copy of another object. If we aren't careful, we can lose our old identity and leave a pointer

allocated but lose its address — a memory leak! (Wasn't that what the destructor was protecting us from in the first place?)

Before assignment:

```
    +=============+
    |    +----+    |
 A  |    |   -|----|-------------------------+
    |    +----+    |                        \|/
    +=============+               +---------+      +-----------+
                                  |  DATA   |      | OLD DATA  |
    +=============+               +---------+      +-----------+
    |    +----+    |                                    /|\
 B  |    |   -|----|--------------------------------------+
    |    +----+    |
    +=============+
```
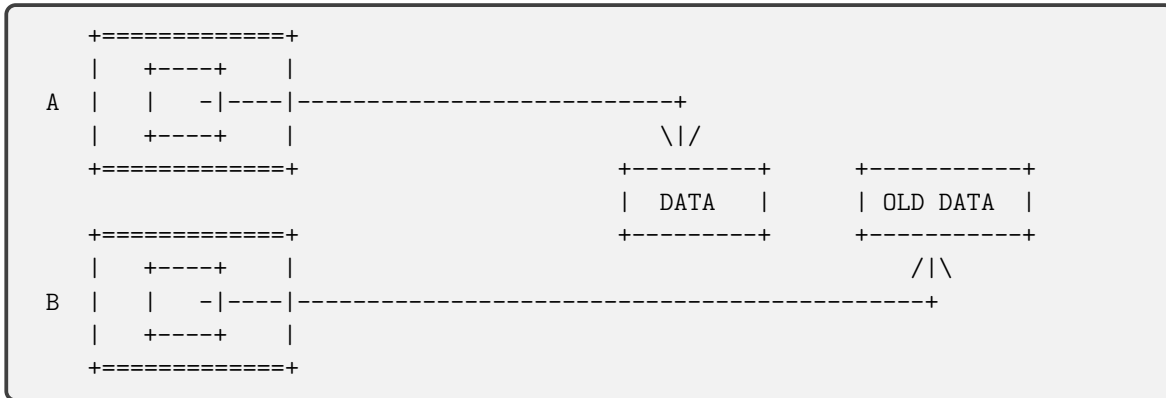
Then, after we execute `B = A;` we'll have:

```
    +=============+
    |    +----+    |
 A  |    |   -|----|-------------------------+
    |    +----+    |                        \|/
    +=============+               +---------+      +-----------+
                                  |  DATA   |      | OLD DATA  |
    +=============+               +---------+      +-----------+
    |    +----+    |                               \ +---------+
 B  |    |   -|----|--------------------------------.|  DATA   |
    |    +----+    |                               / +---------+
    +=============+
```
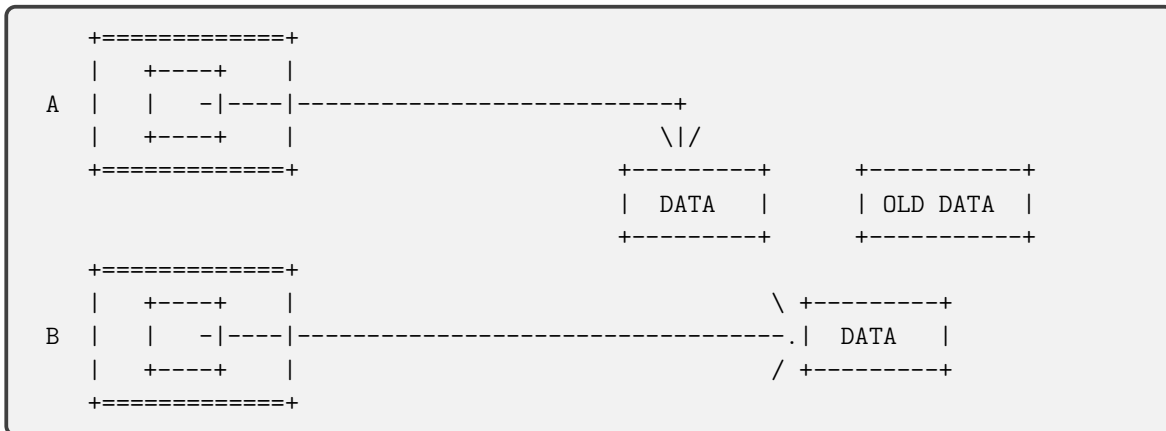
Notice how `OLD DATA` is still allocated but no-one is pointing to it! (That's a memory leak, all right. Icky, hunh?) To fix this, we'd need to precede our new request for storage by a deallocation like so:

```cpp
void DynC::operator=(const DynC & dc)
{
    delete [] p;  // throw away the old me!
    p = new(nothrow) double[dc.max];  // make a new me?
    if ( p != nullptr )
    {
```

That'll clean up the old crap before we allocate the new crap...er, data. (*grin*)

However, to top this all off, we have ancestral problems left from the C language. Did you know that you could do this:

```
B = A = C;
```

Probably.[13] It is a multi-assignment and quite convenient when you want to initialize several variables to the same value (all 0 or all `false`, for instance). The problem is that all operators are treated equally and so you can change the precedence of them via parenthesization.

No, really — even assignment.

---

[13]We did it in the copy constructor above, for instance. Didn't even make you bat an eye, did it?

For instance, you can do this:

```
(B = A) = C;
```

This sets B to be a copy of A and then immediately obliterates that copy to make B be a copy of C. Stupid but time consuming. *shrug*

Maybe this makes more sense:

```
(B = A) += C;
```

Here we make B be a copy of A and then update it by adding information from C.  This saves us several keystrokes and a line of code. Again: *shrug*

But we cannot play favorites!  All `operator`s just `return` a result which is then used in the next operation by standard precedence rules — including their override via parentheses!

And on a related note, it is important to protect the programmer who codes this:

```
A = A;
```

Hunh? Not probably literally, but maybe via `?:` factoring. We did this above in the constructor from a given `MAX` to reset `max` when there was no memory allocated.

If we are deleting our old `CRAP` before copying the new guy's stuff, this kind of thing would be nasty because the old guy and the new guy are the **SAME** guy!!! Once we `delete` the old `CRAP`, we'd have no `CRAP` to copy!?!

To solve both of these issues, we'll need a helper. We need to know the identity of the calling object — the one from the left side of the = operation. It's name is left with the caller. But it still resides at a memory location. So, C++ decided to provide us with the address of the calling object.

In fact, it has been doing so since the very beginning. For all of our `class` member functions, C++ has given us an invisible, implicit parameter: the address of the calling object. We've never discussed it for two reasons: we didn't know what addresses and pointers were and we didn't care. Now we both know what pointers are and care about the identity of the calling object in more than a "the compiler will automatically use members of the calling object in a member function" kind-of way.

### 9.2.3.4.1  A Pointer Named this

There is, of course, a keyword used to access the calling object's address. That keyword is one of the most poorly chosen identifiers we've ever seen. And we've seen some doozies: `cstdlib`, `log/log10`, `npos`, … I could go on, but I think you get the idea.

So what is the keyword for the calling object's address? It is `this`. What? No, that was it: `this`. `this` is the keyword. The word '`this`' is the keyword. I swear it!

It is so hard to tell you about this keyword because it is a pronoun and people always think I'm talking about something else when I really mean '`this`'!

Anyway… How do we use '`this`' to solve the problems of multi-assignment and self-assignment? All we have to do is check the address of the argument object against that of the calling object to make sure they are not the same object. (We couldn't just check if they are equal objects because we want to know they are physically the same object — not that they have equivalent content.)

In our sample `DynC` `class`' `operator=`, we could do this:

```
if ( this != &dc )
```

(Sorry for the double 'this' there. *sigh* I hate this keyword. *augh*)

Place the entire rest of the function — except the `return`, of course — inside this `if` structure to fix self-assignment.

And multi-assignment? That one is slightly trickier. There we need to `return` a reference to the calling object from the `operator=` — just like the built-in assignment operation does. But we have an address — a pointer — not a reference.

Well, having a pointer, we can follow it to the original object. Then we can just reference that! Like so:

```cpp
DynC & DynC::operator=(....)
{
    // ...
    return *this;
}
```

Ta-da! So, putting all this together we get:

```cpp
DynC & DynC::operator = (const DynC & dc)
{
    if ( &dc != this )  // not copying myself, right?
    {
        delete [] p;  // throw away the old me!

        p = new(nothrow) double[dc.max];  // make a new me?
        if ( p != nullptr )
        {
            max = dc.max;   // yea!  now...
            cur = dc.cur;   //          ...to copy...
            for ( size_t i{0}; i != cur; ++i )  //  ...over their stuff!
            {
                p[i] = dc.p[i];
            }
        }
        else
        {
            cur = max = 0;  // crap!!  *snaps fingers*
        }
    }
    return *this;   // _this_ automatically points to the calling object
}
```

And just like that we've fixed another double-`delete`, another dangling pointer, and another memory leak!

### 9.2.3.5   The Big Three

That completes what are known as the **Big Three**. The destructor was necessary to prevent memory leaks from dynamic members. And its addition necessitated the addition of a copy constructor and an `operator` assignment. Since every time you have a dynamic `class` member you need these three functions added to the `class`, we have named them the **Big Three**.

#### 9.2.3.6  Debugging with Hex Addresses

On the support website you can find this complete program with the DynC class and a short driver main. This program not only tests the DynC methods to make sure they are working, but those methods print the addresses of objects involved in their operations along the way.

If you sit the source code on one side of your screen and a terminal running the program on the other, you can follow along back and forth which addresses must coincide with which objects and this will help you build your intuition about construction and destruction order and the like.

The thing about addresses and printing is they come out in hexadecimal — base 16 numbers. This is disconcerting at first, but you can become accustomed to it. Don't worry so much about their values, just be able to tell one of them from the others so you can pinpoint which object did what.

### 9.2.4  2D Dynamic Arrays

One thing that comes up from time to time is multidimensional storage. We've explored this in terms of nested vector and array classes in the previous volume[14] and with static arrays earlier in this volume (section 8.1.2).

But what about dynamic arrays? Do we ever take those multidimensional? Of course! Let's explore this starting with two-dimensional structures.

There are two basic approaches to 2D arrays on the heap: the physically accurate and the mapped. Some applications call for one or the other, so we'll explore both here.

#### 9.2.4.1  Physically Accurate

This approach is the most 'obvious'. That is, it will give us the simple use of two subscript operations to access the data when we are done allocating it — just like a normal statically allocated 2D array. The allocation won't be as clean and simple as other approaches, however, so we may want to give those some serious consideration later.

It uses an array of pointers where each element points to an array of data. Overall this effects a 2D structure as can be seen in the diagram below. See how I've arranged the rows of data to sort of stack on top of one another and you could line them up to form a nice grid if they were movable.[15]

```
 arr1
  +---+
  | , |
  +-|-+     +---+---+---+---+
    |   +->|   |   |   |   |
    |   |   +---+---+---+---+
    |   |
    |   |       +---+---+---+---+
    |   | +-->|   |   |   |   |
   \ /  | |     +---+---+---+---+
    .   | |
  +---+ | |       +---+---+---+---+
  | , |-+ |   +->|   |   |   |   |
  +---+   | |   +---+---+---+---+
  | , |---+ |
  +---+     |
```

---
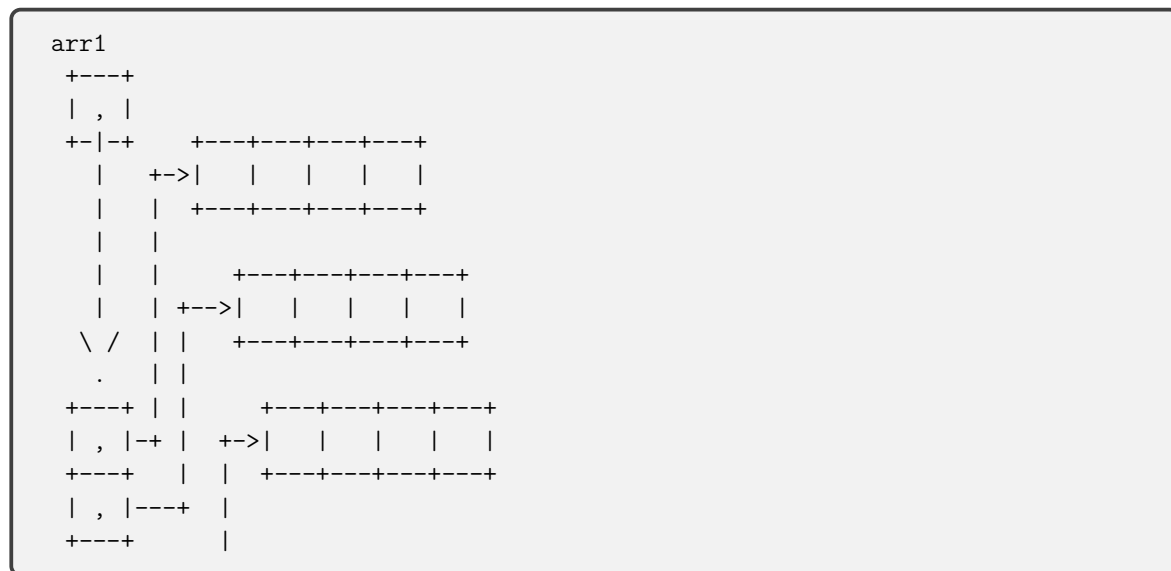
[14]See chapter 6 there.
[15]Of course since this is all done with dynamic allocation, they aren't really even stacked up like that — they are all over the heap anywhere the OS found space. But it makes for a nice diagram, no?

```
  | , |------+
 +---+              +---+---+---+---+
  | , |---------->|   |   |   |   |   |
 +---+              +---+---+---+---+
  | , |--+
 +---+   |   +---+---+---+---+
         +->|   |   |   |   |
             +---+---+---+---+
```
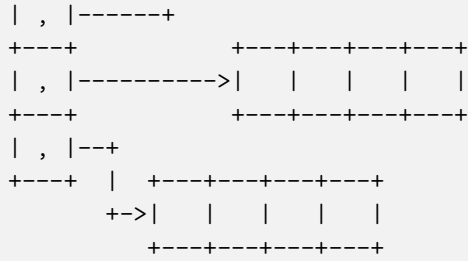
Here the commas are the pointers and the arrows show where they point as usual. The `arr1` is the initial pointer that we have to start our 'journey' through the structure. If we subscript it, we pick out a single pointer from the column array to the left. This element is another pointer. Thus we can subscript it as well to get to one of the data items it points to.

So just one pointer gets us to all the data with two subscript operations as we would have done with static 2D arrays. That's pretty nice and a cozy place to work.

But how do we describe this in code? Well, we'd have to set everything up, of course — nothing comes for free in the dynamic world!

```cpp
// ROWS and COLS are known size_t values
size_t num_rows, r;
type **arr1{new(nothrow) type* [ROWS]};
if ( arr1 != nullptr )
{
    r = 0;
    arr1[r] = new(nothrow) type [COLS];
    while ( arr[r] != nullptr &&  // we succeeded in last allocation and
            r+1 < ROWS )          // there is room for the next allocation
    {
        arr1[++r] = new(nothrow) type [COLS]; // allocate into the next spot
    }
    num_rows = r+1;  // adjust for 0-start
}
else
{
    num_rows = 0;    // we got no space at all!
}
```

Here we've done something quite new. `arr1` is a double pointer! This is because the things it points to aren't real data but are themselves pointers. So `arr1` points to pointers that in turn point to data.

That also means that when we allocate space for the column array on the left that its base type isn't the data type we are interested in but "pointer to data type". This is reflected in the `new[]` allocation for `arr1` itself.

Note how the allocation stops when a row fails to allocate or when we've allocated all the rows. But we keep track of how many rows were successfully allocated for later with the `num_rows` helper variable.

Once allocated, we can use this 2D array normally:

```cpp
for ( size_t r{0}; r < num_rows; ++r )
{
    for ( size_t c{0}; c < COLS; ++c )
    {
```

```
            // use arr1[r][c] like normal
        }
    }
```

See how we use two subscripts just like we would a statically allocated 2D array? So nice!

But, care must be taken when we're done to avoid leaving allocated but inaccessible fragments of memory:[16]

```
for ( size_t r{0}; r < num_rows; ++r )  // delete each row
{
    delete [] arr1[r];
    arr1[r] = nullptr;                 // (not absolutely necessary)
}
delete [] arr1;                        // then delete the outer array
arr1 = nullptr;
```

Here we've made sure that we release the rows of data first before releasing the column array of row pointers. This order is **very** important! Remember that once deallocated, a pointer will retain its old address but we have lost the rights to that memory. The system can reallocate it at any moment. So we must free up the row spaces before freeing up the row pointers!

### 9.2.4.2   Row Mapped

The second allocation scheme follows the compiler's own method of laying out the 2D grid in a long, linearized space row-by-row. That is, we take each row of the array and line it up right after the previous row in the memory so that the second abuts the first and so on. Doing so is going to be less 2D-friendly in the usage phase in that we won't be able to use two subscript operations, but has advantages for both the setup and tear-down phases of the plan. It might be worth a look if it can simplify those processes.

Let's look at this in more detail. As we discussed earlier (section 8.4.5.1), we like to think of 2D memory as being — well, two-dimensional. We like to think of it looking something like this diagram below in the RAM:

```
         MAX_COLS
        0    1    2
 M   +----+----+----+
 A 0|    |    |    |
 X   +----+----+----+
 _ 1|    |    |    |
 R   +----+----+----+
 O 2|    |    |    |
 W   +----+----+----+
 S 3|    |    |    |
     +----+----+----+
```

But in reality it really looks like the below diagram:

```
         M        A        X        _        R        O        W        S
   M A X _ C O L S    M A X _ C O L S    M A X _ C O L S    M A X _ C O L S
 +----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

---
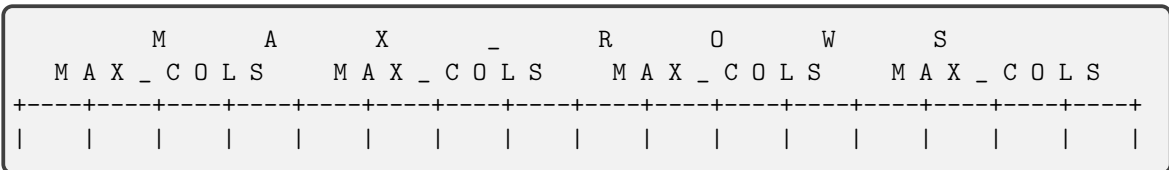
[16]AKA memory leaks.

```
+---+---+---+---+----+----+----+----+----+----+----+----+----+----+----+----+
 0,0  0,1  0,2  0,3  1,0  1,1  1,2  1,3  2,0  2,1  2,2  2,3  3,0  3,1  3,2  3,3
```

And the compiler uses this mapping from our normal double subscripts to 1D positions:

```
arr2D[r][c]  ==  arr1D[r*COLS+c]
```

So we can do the same thing!

The allocation would look like so:

```cpp
type * arr2{nullptr};
arr2 = new(nothrow) type [ROWS*COLS];
```

Note that there is no chance to have lost desired rows — we get the whole array or nothing. There is also just a single pointer * so that's nice.

And use it:

```cpp
for ( size_t r{0}; r < ROWS; ++r )
{
    for ( size_t c{0}; c < COLS; ++c )
    {
        // use arr2[r*COLS+c]
    }
}
```

And the deallocation is simple, too:

```cpp
delete [] arr2;
arr2 = nullptr;
```

What's the problem here? It isn't a tedious setup and take-down. That's all been streamlined! It's the mapping in the middle. It just isn't natural. *pout*

There are two basic ways to hide it away. The first is an `inline` helper function like this one:

```cpp
inline type & map2D(type * arr, size_t row, size_t col, size_t COLS)
{
    return arr[row*COLS+col];
}
```

But this is just a bit icky — it still doesn't hide enough details. Look at it in action, after all:

```cpp
for ( size_t r{0}; r < ROWS; ++r )
{
    for ( size_t c{0}; c < COLS; ++c )
    {
        // use map2D(arr2,r,c,COLS)
    }
}
```

I'd almost go so far as to say it made it worse!

If you are going to use this allocation approach, the coolest way would have to be to embed the array in a `class` and overload `operator()` to do your 2D indexing.

Why `operator()`? Why not overload `operator[]`? Well, the subscript `operator` in C++ only takes one argument — the position along the single dimension to access, presumably. And we need to access along two dimensions.

So `operator()` to the rescue! It is the parentheses used to call a function. It doesn't modify precedence or anything like that. It is normally placed after a function's name to surround any parameters that function might need to run. Since functions can be designed to take any number of arguments, this `operator` can take an arbitrary number of operands — things to operate on. We need it to take two for our dimension positions. Perfect!

It would look something like this:

```cpp
class Arr2D
{
    type * arr2;
    size_t ROWS, COLS;
    void alloc(void)
    {
        arr2 = new(nothrow) type [ROWS*COLS];
        return;
    }
public:
    Arr2D(void) : arr2(nullptr), ROWS(0), COLS(0) { }
    Arr2D(size_t R, size_t C) : arr2(nullptr), ROWS(R), COLS(C)
    {
        alloc();
    }
        // copy constructor left to reader!!!
        // operator= left to reader!!!
    ~Arr2D(void)
    {
        delete [] arr2;
    }
    type & operator() (size_t r, size_t c)
    {
        return arr2[r*COLS+c];
    }
        // *ALL* error checking left to reader!!!
};
```

As you can see, we take in the two dimensional positions and do the mapping inside the `operator()` and `return` that array position by reference so the caller can either change or look at the data at their discretion. If we want to provide similar access for `Arr2D` objects that have been passed as `const&`, we can even overload it a second time:

```cpp
type operator() (size_t r, size_t c) const
{
    return arr2[r*COLS+c];
}
```

All we changed was to mark the function `const` so it would work for a constant `Arr2D` object and `return`ed by value instead of by reference. That C++ allows these two function overloads to sit side-by-side in the same `class` is pretty neat and useful at times like these!

Now you can do something like this:

```
Arr2D ar(ROWS,COLS);

for ( size_t r{0}; r < ROWS; ++r )
{
    for ( size_t c{0}; c < COLS; ++c )
    {
        // use ar(r,c)
    }
}
```

Still not completely normal, but you've hidden the allocation details, the deallocation details, and, of course, the mapping details inside the `class`.

Another possibility would be to use a helper position `class`. For this approach we do overload `operator[]` but instead of passing it a `size_t`, we pass it some kind of `class` or `struct` that holds two values at once. These, then, are mapped to the row and column by our formula and the result `return`ed.

For simplicity I'm going to use the `pair` construct provided by the standard library `utility` and explored in the previous volume.[17] This might look like so:

```
class Arr2D
{
    type * arr2;
    size_t ROWS, COLS;
public:
        // other aspects of the class unchanged
    type & operator[] (const pair<size_t,size_t> & p)
    {
        return arr2[p.first*COLS+p.second];
    }
    type operator[] (const pair<size_t,size_t> & p) const
    {
        return arr2[p.first*COLS+p.second];
    }

        // *ALL* error checking left to reader!!!
};
```

Now we can use this `operator` pair[18] like so:

```
Arr2D ar(ROWS,COLS);

for ( size_t r{0}; r < ROWS; ++r )
{
    for ( size_t c{0}; c < COLS; ++c )
    {
        // use ar[{r,c}] (C++17) (or ar[make_pair(r,c)] on an older library)
    }
}
```

This doesn't look too bad, either. It's a little more typing than our `operator()` approach above — especially pre-C++17, but still not too bad.

---

[17]See the section of chapter 4 on `return`ing a pair of results from a function.
[18]Pardon the pun. . .

### 9.2.4.3 A Step Back

There are several important points to consider here about each method:

- The physically accurate method can also be hidden in a `class` using `operator()` or the `operator[]` with a `pair` argument, but this loses your natural use of two `operator[]` abutting for indexing in two dimensions.

- The physically accurate method can support 'ragged' arrays: those whose rows are not all the same length. In some applications this can be important.

- The row-mapped method might seem slower because you have to multiply and add to reach the correct address, but consider that a normal (statically declared) 2D array does the exact same thing — only the compiler does the multiply and add for you.[19]

- The physically accurate method might seem faster because it uses our normal [] [] notation for access, but consider that it no longer is normal and that you are really doing two dereference operations. Unless the whole array is in the processor's cache, this could take quite a many milliseconds to follow pointers to arbitrary places in the heap.

### 9.2.4.4 Merging the Two Methods

But, for those of you out for the ultimate dynamic 2D experience — the best of both worlds as it were — you can try out this variation. Note well, though, you'll have to be willing to pay the costs of both worlds, too. . .

We start by making the one-dimensional row mapping as in the latter technique. But in addition to that, we make an array of pointers as in the former approach. This time the pointers in the array will aim at the beginning of each row within the linearized space. Diagrammatically it looks something like this:

```
 0,0  0,1  0,2  0,3  1,0  1,1  1,2  1,3  2,0  2,1  2,2  2,3  3,0  3,1  3,2  3,3
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
  '                   '                   '                   '
 / \                 / \                 / \                 / \
  |                   |                   |                   |
  |                   |                   |                   |
  |     +-------------+                   |                   |
  |     |   +---------------------------+ |                   |
  |     |   |   +-------------------------------------------+ |
  |     |   |   |                                             +----------------------+
  |     |   |   |
+---+---+---+---+
| , | , | , | , |
+---+---+---+---+
```
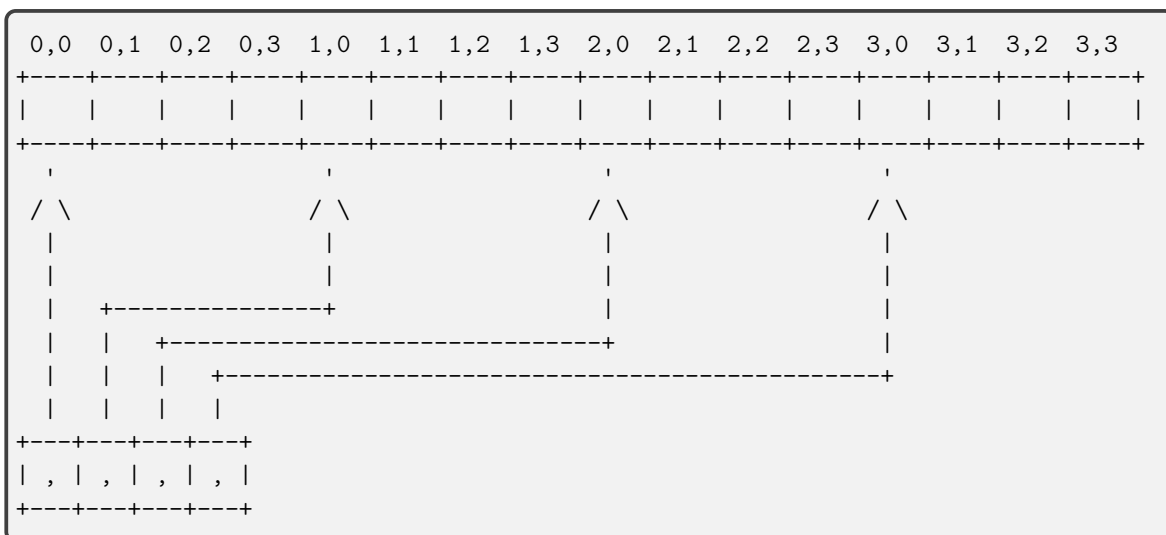
How will this help us? Well, it gets us the single allocation for all the data space as with the mapped method. It also gets us double subscripted access like the physically accurate method.

What? How? Look at the 'column' array: it holds pointers to locations with data. And we'll have a pointer to it as it is an array. So we'll select a pointer with a first subscript and then select an element with a second subscript!

Let's get to the code:

---

[19]Also, there are methods to speed up these calculations by caching the current pointer and simply retrieving the next element.

```cpp
// declaration
type * data{nullptr},
     ** arr2D{nullptr};

// allocation
data = new(nothrow) type [ROWS*COLS];    // allocate linearized space
if ( data != nullptr )
{
    arr2D = new(nothrow) type* [ROWS];   // allocate row pointers
}
if ( arr2D != nullptr )
{
    for ( size_t r{0}; r != ROWS; ++r )    // aim row pointers
    {
        arr2D[r] = data + r*COLS;
    }
}

// access
for ( size_t r{0}; r < ROWS; ++r )
{
    for ( size_t c{0}; c < COLS; ++c )
    {
        if ( arr2D != nullptr )
        {
            // use arr2D[r][c]
        }
        else
        {
            // use data[r*COLS+c]
        }
    }
}

// deallocation
delete [] arr2D;    // no need to delete rows individually!
arr2D = nullptr;
delete [] data;
data = nullptr;
```

Although we have the potential of failing to allocate the row-start pointers array separately from the `data`'s allocation, this method can give nicer access patterns for general use. You can even place the conditional based on `arr2D`'s successful allocation in your new `class` access method to take advantage of the `[] []` ability vs a mapping calculation!

### 9.2.4.5   More than Two Dimensions

Both methods can be easily extended to multiple dimensions. The first way naturally extends as you'd expect (add a $*$, add another loop for allocation, etc.).

The second way's mapping formula for 3D is:

```cpp
arr3D[p*ROWS*COLS+r*COLS+c]
```

Here p is the plane or cross-section of the 3D structure you want. I'll let you continue the math to higher dimensions.

Note that you can factor out the multiply by COLS from the first two terms to speed things up a bit. This is known as Horner's method.

It might also be worth noting that you can use a 'dimensional' array to store supplementary information in several ways to make access faster. (This is fodder for a much later discussion, however.)

### 9.2.5    main Arguments

So far our main function has always taken no parameters. Some of us even put the keyword void in its parentheses to point this out more explicitly. But any command run from the command prompt (and even many of those run by double-clicking an icon on the desktop) can take information right then from the user without having to use cout/cin to discuss the matter.

#### 9.2.5.1    Command-Line Arguments

Look at the compiler you might use at a typical *nix prompt,[20] for instance. You tell it which file(s) to compile and what options should be enforced during this compilation, right? Sometimes you even tell it what to name the final executable file.

In C++, all of this information is prepared by the part of the OS known as the shell and given to the main function in the form of a 2D dynamic array of C-strings.[21]

How does this look from the program's side of things? Well, the head of the main changes to look like this for starters:

```
int main(int argc, char *argv[])
```

The first is the count of arguments — sortof — and the second is the values of the arguments as C-strings — more or less.

argc is actually 1 more than the number of arguments because, for some reason, the first 'argument' is given as the name of the program.

Thus, argv[0] holds the name used on the command-line to invoke your program. Some programs use this to invoke different behaviors. For instance, gzip and gunzip are often the same executable but the code looks at its argv[0] value to see if its goal should be compression or decompression during this execution.

The rest of the argv entries are the actual arguments typed by the user when they ran the program. They are typically taken apart at spacing — tabs or spaces.

Here is a basic program that takes and reprints its command-line arguments:

```
int main(int argc, char *argv[])
{
    cout << "Received " << argc-1 << " arguments to the "
         << argv[0] << " program.\n";
    cout << "\nArguments:\n";
    for ( int a{1}; a != argc; ++a )
    {
        cout << '\t' << argv[a] << '\n';
    }
```

---

[20]Or a cmd prompt for you Windows folk.
[21]The dynamic part is done in the shell and not our responsibility to clean up, luckily!

```
        return 0;
}
```

If we run this with a command line like so:

```
$ ./prog.out a 1 long 345 apples -2 short
Received 7 arguments to the ./prog.out program.

Arguments:
        a
        1
        long
        345
        apples
        -2
        short
$ _
```

But, on the other hand, if we put quotes around some of the arguments, we see something more like this:

```
$ ./prog.out "a 1" long '345 apples -2' short
Received 4 arguments to the ./prog.out program.

Arguments:
        a 1
        long
        345 apples -2
        short
$ _
```

Note how the shell groups things within matching quotes together to form a single C-string argument for the main function to process.

#### 9.2.5.1.1    Patterns on the Command-Line

Also of importance is the way the user types parameters to your program. The standard for doing options is to use either a single dash (-) or slash (/) followed by one or more single-letter options — so-called short options. These are often case-sensitive so that z and Z indicate different options. A more verbose style is to use a doubled prefix character (like -- for instance) before a whole-word option — so-called long options.

If a command-line option requires a filename or number, it will typically follow a short option immediately or sometimes in the next argv slot. But if a long option requires such information, it will immediately follow an immediate = — within the same argv slot.

Note the following *nix commands' options:

```
ls --help
man -h

grep -h
grep --help
```

```
less --help
```

These all give help information about running the command as well as a little idea of what the command does. With `grep`, the `-h` gives a tiny amount of help and the `--help` gives a lot of help.

Let's look at a sample main looking for short options. It takes options to process a number, a Boolean value, and a filename.

```cpp
int main(int argc, char *argv[])
{
    cout << "Received " << argc-1 << " arguments to the "
         << argv[0] << " program.\n";
    cout << "\nArguments:\n";
    for ( int a{1}; a != argc; ++a )
    {
        if ( argv[a][0] == '-' )   // starts with a dash
        {
            bool flag;
            switch ( argv[a][1] )  // char after the dash
            {
                case 'n':
                    cout << "\t\tGot a number!\n";
                    if ( isdigit(argv[a][2]) )
                    {
                        cout << "\t\t\tRead '" << stod(argv[a]+2) << "'\n";
                    }
                    else if ( a+1 != argc )
                    {
                        cout << "\t\t\tRead '" << stod(argv[a+1]) << "'\n";
                        a++;  // skip past arg 'parameter'
                    }
                    else
                    {
                        cout << "\t\tParameter -n needs a number following "
                                "it!\n";
                    }
                    break;
                case 'f':
                    if ( a+1 != argc )
                    {
                        cout << "\t\tGot a file name!\n";
                        cout << "\t\t\tRead '" << argv[a+1] << "'\n";
                        a++;  // skip past arg 'parameter'
                    }
                    else
                    {
                        cout << "\t\tParameter -f needs a filename following "
                                "it!\n";
                    }
                    break;
                case 'b':
                    if ( a+1 != argc )
```

```cpp
                        {
                            cout << "\t\tGot a flag!\n";
                            flag = argv[a+1][0] == 't';
                            cout << "\t\t\tRead '" << boolalpha << flag << "'\n";
                            a++;  // skip past arg 'parameter'
                        }
                        else
                        {
                            cout << "\t\tParameter -b needs a Boolean following "
                                    "it!\n";
                        }
                        break;
                    default:
                        cout << "\t\tI don't understand short option '" << argv[a]
                             << "'.\n";
                        break;
                }
            }
            else
            {
                cout << "\tPlain argument:  " << argv[a] << '\n';
            }
        }
        return 0;
    }
```

As you can see, it looks at the first character of the C-string parameter for a dash and, if found, checks the next character for any of its known short option letters. If it doesn't know the option, it reports as much. And if it doesn't start with a dash, it is simply repeated back out as a plain argument. Here is a downloadable copy so you can play with it on your own.

### 9.2.5.2　The Environment

There is another way that programs sometimes get information from users: the environment. This memory area is managed by the shell and a copy of it is sent to any main asking for such. It contains a list of variables and their values separated by equal signs. The variables traditionally have no spaces, but the values can have such.

This parameter is represented as a 2D dynamic array of C-strings as well. But it isn't counted by an integer argument like argv. It is instead ended by a nullptr in the last slot.[22] It looks like so:

```cpp
    char *env[]
```

and comes right after argv in the main arguments list.

A typical way to process the environment is to use a char double pointer and walk it from C-string to C-string looking for the variable we have interest in. When we find it, we can take the value into our program and make adjustments in configuring this run.

```cpp
    int main(int argc, char *argv[], char *env[])
    {
        cout << "Received " << argc-1 << " arguments to the "
```

---

[22]Except for this one system which ended with an empty C-string! Most annoying debugging session — EVER!

```
            << argv[0] << " program.\n";
    cout << "\nArguments:\n";
    for ( int a{1}; a != argc; ++a )
    {
        cout << '\t' << argv[a] << '\n';
    }

    cout << "Processing environment:\n";
    char ** e{env};
    while ( *e != nullptr &&  // not a nullptr AND
            **e != '\0' )     // not an empty C-string
    {
        char * var{strtok(*e, "=")},      // stop at first = for variable
             * value{strtok(nullptr, "")}; // go to end for value
        cout << var << " is '"
             << ( value == nullptr ? "!!EMPTY!!"   // no value set!
                                   : value )       // regular value
             << "'\n";

        ++e;
    }

    return 0;
}
```

This code can be downloaded from the website so you can play around with it more easily.

Here we look at each pointer with a single dereference to see if it is `nullptr` yet. If it isn't we process that line using the earlier hinted at `strtok` from the `cstring` library.[23]

Once the environment variable is processed, we ++ to the next slot and see if we are done yet.

So what kinds of things can come in from environment variables? All kinds! They can be numeric, string, Boolean, etc. Anything the user can send in a command-line argument, they can send in an environment variable.

So why bother? Well, environment variables can be set once and forgotten. Command-line options must be retyped on each run of the program.

### 9.2.5.2.1   Setting Environment Variables

So how does the user set environment variables? That's a bit of a sticky wicket! It varies drastically from system to system. I'll describe here how to do it in *nix and Windows how to set them temporarily. Sadly, setting them permanently in all systems is beyond the scope of this book.

**\*nix:**   In a *nix terminal, type the following command to set an environment variable for the current terminal session:

```
export ENV_VAR=value
```

This will make that variable available to the system environment until this terminal is closed. Thus the variable will be available to all programs you run from this prompt.

---

[23]This was in section 9.1.6 on pointer goodness from the `cstring` library. Please see some form of online documentation for more on this function. Perhaps at the wonderful site cppreference.com?

**Windows:**   In a Windows `cmd` prompt, type the following command to set an environment for the current command prompt duration:

```
set ENV_VAR=value
```

This will make that variable available to the system environment until this `cmd` window is closed. Thus the variable will be available to all programs you run from this prompt.

## 9.3   Wrap Up

In this chapter we've learned standard C++ tools to manage memory during the run of a program. This included a fairly detailed look at pointers and indirect access to data as well as a hefty examination of dynamic memory allocation, use, and deallocation. We ended with a foray into passing information right into a `main` function from the command line which, it turns out, uses memory managed by the OS on our behalf and in the same shape as we learned to use on our own.

# Chapter 10

# File Streams In Depth

In the previous volume we covered file stream processing at a shallow but effective level. This time around, we're pulling out all the stops and diving deep into the topic!

Some of this might feel a little repetitive to the coverage in the last volume. But we do want a complete coverage in one place, so.

## 10.1 Concepts

So what are files and what are they used for? Well, they store information for some program to process, typically. They are just bits stored on the disk/drive in their primal form, but at a high level, they represent information to us.

We've used files for years, of course. Word files, Excel files, video files, audio files, etc. Most recently, we've been working with C++ source files and their executable forms after compilation.

This should imply to us that there are at least two types of files out there: binary and text. This is quite true! We'll be focusing on the text files in this text, but there will be a brief introduction to binary file processing in a later section (H.5).

Since we are focused on text files, that means we can create sample data files for our programs to process in the same environment that we've been using for C++ source files! Any plain text editor will do, but if you are used to an environment, keep using it!

Let's start our discussions with input since that is what we mostly do with files to start — read in immense amounts of data to process!

## 10.2  Input

Input from a text file is almost just like using `cin`. It differs in just three respects, really:

- You need to connect your program to the disk file before use and disconnect when finished. This is done automatically for us with `cin` and its source — the keyboard.

- You do not need to prompt the user when reading from a disk file. The file has all its data ready to be read and doesn't actually look to the screen for prompts anyway. It will just confuse the user if you do so, so let's not.

- There may be elements in a file stream that we need to skip over to get to data. These will be treated similarly to how we skipped [optional] notation in the previous volume with a little fancy `peeking`. This time we'll be mostly skipping comments the user has left to themselves.

But let's start with a simple file format without comments to make things as simple as possible. Let's say we wanted to read in a sequence of space-separated numbers from a file. Data still needs to be separated by spacing for even a file stream in text mode to read it properly.

### 10.2.1  Connecting to Files

The basics of connecting to a file are having the name of the file as the user sees it and then calling the proper method. To get the user's file name, we need to have a `string` of some kind. I'll actually use a `string` `class` variable for convenience so we don't have to worry about overflow on input of the name:

```
string fname;
cout << "What is the name of your data file?  ";
getline(cin, fname);
```

Make sure to use `getline` to read file names since most users like to put spaces inside their files and/or folders these days!

Once we've got the name, we need to call the proper method. But of what `class` and which method?

Since we are trying to input from this file, we'll use the type `ifstream` — short for Input File STREAM. This `class` is found in the `fstream` library with all File STREAM codes.

The proper method is called `open`. You just need to pass it the file name `string` from earlier:

```
ifstream number_file;
number_file.open(fname);
```

And that should do it — assuming the user typed the right file name and possibly folder path. The file name the user types must include not only the name, but also the extension and path of the file. If the file is in the current directory/folder, the path can be skipped. But if the file is located in another place, they can use a relative or absolute path to indicate this to us.

A relative path would start with the subfolder name if the file is under the current folder or with `../` if the file is located above the current folder. An absolute path would start with either a drive letter and colon (on Windows) or a `/` if on a *nix machine.

Once the file is `open` — connected to our program via the `ifstream` variable we made earlier, we can read from it with any techniques we used with `cin`! We can use `>>` for basic reading or `getline` for `string` data or we could `peek` to make decisions about what to do next or we could `ignore` to skip past data, etc.

## 10.2.2   eof Loops

The tricky bit is that we don't typically have just one piece of data in a file or even a known number of data in a file. It is usually an arbitrarily long sequence of data we are expected to process. With numeric data on the console we did this via calling `fail` to determine when the user was done entering real data. But in files, there is a special marker at the end of all files we can use to determine when the sequence is done!

This marker is known as `eof` — End Of File. We call the `eof` method to determine if we've reached it. But that terminology was a little sloppy. It doesn't return `true` until the program has tried to use the marker as data. We can read the last piece of data next to it successfully and it still won't go off! So we need to be careful when formulating an `eof` loop.

This format borrowed from our `fail` on console experience will work much of the time:

```cpp
double number;
number_file >> number;
while ( ! number_file.eof() )
{
    // process the number
    number_file >> number;
}
```

But it isn't as good as we can do! We can tweak it a bit to make it work on any input file! It turns out there are two ways to make it work so generally: a simple tweak and a more object-oriented approach.

### 10.2.2.1   Tweaking eof

To tweak it, we just prime the `eof` condition with something other than reading data.[1] The problem with the file variants that don't work with the above form is all about spacing.

I believe this stems from the POSIX demand that all text files now end in a newline character (before the `eof` marker). But whatever the case, the tweak to fix things is to extract a contiguous sequence of whitespace as priming:

```cpp
double number;
number_file >> ws;
while ( ! number_file.eof() )
{
    number_file >> number;
    // process the number
    number_file >> ws;
}
```

This makes sure that stray whitespace after the last piece of data or in an otherwise empty file won't cause us problems.

The one difference is that if you are reading with `getline` instead of extraction as above, you should not prime with `ws` as it would eat some of the whitespace you were trying to preserve with `getline`! Instead, prime with a `peek` operation. One of the odd things about `peek` is that even though it doesn't

---

[1]For more on the idea of 'priming' a loop, see the previous volume.

actually read anything it will trigger `eof` if we see the marker at the current location. This would, then, look like so:

```
string data;
file.peek();
while ( ! file.eof() )
{
    getline(file, data);
    // process the data
    file.peek();
}
```

Had to change the file variable and data variable names, of course, but you get the idea. . .

#### 10.2.2.2  Object-Oriented eof

Another approach that works despite the rampage of spacing in modern files is also more object-oriented, so that's nice. *smile*

It two facts to work:

- all reading methodologies return the file variable they are working on as their result

- a file variable evaluated in a `bool` context reports `true` when the file is in a good state and `false` when the file is not

So we merely use the extraction inside the `while` head like so:

```
double number;
while ( number_file >> number )
{
    // process the number
}
```

And all is well! This not only makes the loop more object-oriented, but makes it smaller, too!

For the `getline` version, it looks like this:

```
string data;
while ( getline(number_file, data) )
{
    // process the data
}
```

Again, both `>>` and `getline` return the stream used for input as their result. And this stream in the `bool` context of the `while` head are `true` when the file is happy and `false` when the file has had any sort of problem — like an `eof` encounter.

You'll find other forms of `eof` loop all over the web. They do not work as well as these two forms! Please always use a tweaked or OO `eof` loop in C++ programming!

### 10.2.3   Disconnecting from Files

One thing that escapes many folk first learning about file processing is then need to disconnect their file from the program after it is done being used. This need comes in several flavors depending on the work the file has been doing, but it is always a need!

We use the `close` method to do this job and it is quite simple. But there is one thing about it that is a little odd: it either leaves the file in the original state before closing (`eof`, `fail`, etc.) or in a new but still erroneous state.

The latter is because some library implementers want to signal something to themselves upon reuse of this file variable. Not good practice, but not forbidden by the standard, either. The former is just laziness. So, to be sure we can reuse this variable later in the program, we should also `clear` it **after** we `close` it:

```
number_file.close();
number_file.clear();
```

#### 10.2.3.1  Why close Files?

All files use system resources called file handles. These resources are given to any attempted file open for use in processing the file through the OS. But these resources tend to be finite — albeit very large — in number on a system. So if we don't give them back by `close`'ing a file variable, we can run the user's system out of them! (This wouldn't just affect us, either, but all programs running on the system!)

Another reason is worse for output files. They store their 'displayed' information in a buffer just like `cout`. The difference is that, while `cout`'s buffer was around 2 kilobytes, that of a file stream is typically 16 kilobytes or more! That's a lot of text that would get lost if the file didn't `close` properly!

To compensate for these things, the destructor of the stream `class`es has been made to call `close` for us. But it still doesn't alleviate the problem when we want to reuse a file variable later in the program to access another file and the file variable hasn't been destroyed and recreated in some inner scope like a function or loop body.

### 10.2.4  A Complete Example

Here is a complete example of the above concepts:

```cpp
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    ifstream data_file;
    string fname;      // might also include path; must include extension if any
    double num;

    cout << "What is the name of the file?  ";
    getline(cin, fname);
    data_file.open(fname);

    data_file >> ws;
    while ( ! data_file.eof() )
    {
        data_file >> num;
        cout << "number read '" << num << "'\n";
        data_file >> ws;
```

```
    }

    data_file.close();    // not necessary here, but in a larger program
    data_file.clear();    // with a file used in a loop it could be

    return 0;
}
```

And here is a sample data file for use with it:

```
42 92.3 42 13 7 12 14 9 8 2 3 5 -.52 0 -11 12
```

Just some space-separated data along a line. But remember that any spacing will do. You could add blank lines or just newlines here and there and tabs are fine as well.

## 10.3   Output

The second most popular use of files in programming is outputting to them. We have to save the user's data somewhere for future sessions, after all!

This is much simpler than input, too. You just `open` a connection to a disk space and output to it. Don't forget to `close` the connection when you are done.

What type of variable do we use? Well, `ofstream`, of course! That's for Output File STREAM — just in case. *smile*

And, once `open`, you can do anything to it that you've ever done to `cout`. You can insert to it (`<<`), you can `flush` it, you can format it with various manipulators or methods, etc.[2]

### 10.3.1   Errors on Output

Although you can check for output errors on file streams, this is almost as fruitless as checking for output errors on `cout`. It is usually something catastrophic that has happened and cannot be recovered.

Might as well just go blissfully along writing to a dead file variable and being silently ignored until you are done as to try to stop early and report the problem to the user. Well, maybe if you were processing lots of data it might be worth the effort.

### 10.3.2   A Complete Example

Here is a complete example of the above concepts:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main( void )
{
    ofstream file;
    string fname;       // might also include path; must include extension if any
```

---

[2]More on formatting later in section H.1.

```cpp
        cout << "What should the file's name be?  ";
        getline(cin, fname);
        file.open(fname);

        file << "Hello File World!\n";

        if ( ! file )  // or file.fail() or ! file.good()
        {
            cerr << "\n\aError writing to '" << fname << "'!!!\n\n";
            file.clear();
        }

        file.close();    // not necessary here, but in a larger program
        file.clear();    // with a file used in a loop it could be

        return 0;
    }
```

Again, you don't have to check for errors on an output file,[3] but if you did, there's how to do it. You can use the `!` operator directly on the stream which causes the usual evaluation of it in a `bool` context. Or you can ask the stream if it has `failed` or is `! good`. Once you know it had problems, you would, of course, `clear` it and print some sort of message for the user to know.

## 10.4  Opening Issues

Thus far we've just `opened` a file connection and let fly with activity on that file. But what if the file can't connect? Can this be a problem? Yes!

At its simplest, the user may misspell the file's name or get the path to the file wrong. At the worst, files may exist on external drives and those might be detached during an `open` operation causing a `failure`.

Files exist in folders/directories and these have permissions or allowed actions by different users on the system. If we try to `open` an output file in a folder that is unwritable, this will `fail`. If we try to `open` an input file from an unreadable directory, it will `fail`. Files also have permissions on them and opening an unreadable input file will cause a `failure` as well.

All of these things are fixable, so let's look at ways to do so.

### 10.4.1  Input Opening

To fix problems with `opening` an input file, we just need to write code like this:

```
prompt and read name
try to open file
while ( it failed )
{
    close file
    clear file
    print error message
    prompt and read name
    try to open file
```

---

[3] Things that could happen include running out of drive space, disk crashes, etc.

```
    }
```

Note the `close`/`clear` pair. The `close` ensures that the previously allocated resource from the OS is released before we request a new one. The `clear` gets the stream variable over its 'funk' after the connection from its previous 'file' `failed`. (It also `clear`s the state that some library implementers set after a `close` operation for bookkeeping. We're about to reuse this stream so we need it happy!)

How, then, do we test if the file `failed` to `open`? We can use the `fail` method, of course:

```
while ( file_var.fail() )
```

Or we could be more general — expecting the worst — and check that the stream is not happy:

```
while ( ! file_var.good() )
```

Or we could be more object-oriented and make the file variable act as a `bool` for a moment:

```
while ( ! file_var )
```

This is probably the best, but the choice is really yours.[4]

### 10.4.2  Output Opening

In addition to the problems we noted above, when an output file is connected, the physical disk file is effectively erased or emptied and you are placed at the beginning. In other words, opening an existing file for output destroys it.

If you open a file with gigabytes of important information — your only copy of it in the world! — as an output file, it will all be gone — forever!

Obviously, opening such files should be done with care. Luckily, input files simply won't open when a file doesn't exist! So we can open an input connection to the desired file and, if that fails, re-open it for output with a clear conscience. If the input connection succeeds, however, maybe we should try a different name...

In effect:

```
abort = okay = false;
do
{
    prompt and read name
    open for input
    if ( success )
    {
        eek!  overwrite or append or give up or new name?
        abort = gave up;
        okay = overwrite or append;
    }
    else
    {
        yea!  it's safe!
        okay = true;
```

---

[4]There is also the `is_open` method, but these others are more traditional and direct. `is_open` is normally reserved for situations where we aren't sure we've opened a stream at all yet.

```
        }
        close input connection (and clear!)
    } while ( ! okay && ! abort );
    if ( okay )
    {
        open for output (or appending)
        use...
        close (don't forget to clear!)
    }
    else
    {
        sorry...
    }
```

So we get a filename, `open` the file as if we wanted to input from it (as an `ifstream`), and see if that worked. If it did, we have an existing file![5] We report the problem to the user and ask what they want to do.

A complete list of possibilities is: get a new name, overwrite the file after all, append to the file (see below), or just give up at this time. (The last is useful if they've tried this in a menu option that can just be ditched and go back to the menu, for instance.)

If they choose to give up, we set `abort` to `true` but if they choose to overwrite or append it'll be `okay`.

If the input connection didn't open successfully, we are safe to `open` this file for output as it doesn't yet exist. So it's gonna be `okay`.

This "open-as-input" loop continues while the user isn't `aborting` and we still aren't `okay` to proceed.

So how do we `open` for appending? Well, you can `open` a file in a few different modes. This tells the file to act slightly differently while it is connected to the program with this stream. Append mode is different from output mode in that it doesn't erase the file but rather adds new output to the end of the current content.

To `open` in a different mode, use the [optional] second argument to `open` to change the mode:

```
file_var.open(fname, ios_base::app);
```

This would `open` the `file_var` (an `ofstream`) for output at the end of the file's current content — not erasing but adding to the file.

The other two major modes are `ios_base::in` and `ios_base::out`. These are the default for their specific file types and rarely need to be stated explicitly.


## 10.5  Passing to Functions

Just because we have a new tool doesn't mean we need to use it all in the main function! We should also learn to pass streams about between functions.

There are two things to be aware of when passing a stream to a function. Both involve the type used in the transfer.

---

[5]Input files will fail to open when the file requested doesn't already exist.

### 10.5.1   To Copy or Not?

The first is that streams must always be passed by reference. To see why, imagine the structure of the stream variable and how it communicates to its disk file. Internally, the stream has a buffer and a current position within that buffer where processing is active. This is the get position for an input stream or the put position for an output stream.

If a stream were passed to a function by value, a copy would be made of all of this internal structure. The function's local copy would then process along in the stream and at the least the buffer position would be updated. But if it were an output stream, the buffer content could also be updated. And if the position reaches the end of the buffer it is either refilled for input or `flush`ed for output! Lots of actions are happening here — some with more permanent effects than others!

But, when the function `return`ed, the local copy would be destroyed and we'd be back to the original stream. The exact original stream. It would be in the same buffer state — position and all — that it was before the call! Now processing would pick up from there. If it was for input, we'd reprocess the same data the function did already. If it was for output, we'd start overwriting what the function created for us!

Passing a stream by value would be quite the mess! So, to avoid that and force us to use reference on stream parameters, one of two things is done in their `class`es. In older implementations (pre-C++11), the copy constructors would be made `private` and implemented broken so that they didn't really copy anything. This was effective to a point, but C++11 upped the ante a bit. It introduced a new use for the `delete` keyword. Now the stream `class`es have their copy constructors set equal to `delete` like so:

```
ifstream(const ifstream & if) = delete;
```

This tells the compiler that there will be no copy constructor for this `class` — ever! Needless to say, this is terribly powerful and should be used with caution.

### 10.5.2   A Stream by Any Other Type...

The second thing to keep in mind when passing a stream to a function is how similar the streams are to one another. Not `ofstream`s and `ifstream`s. But `ifstream`s and whatever `class` `cin` happens to be of. They share a plethora of operations, right? In fact, you could almost drop one in where the other was and the code would keep working just fine.

The makers of these `class`es noted this and took advantage of it during their design. Using a technique we'll learn later for ourselves called inheritance, they connected these `class`es together in a sort of family. The input streams all come from a common ancestor `class` as do all the output streams. These two ancestral `class` types can be used to refer to either descendant due to these inheritance relationships! (Again, more on inheritance in chapter 13.)

So, we like to take advantage of this by making our code as general as possible and allowing for the possibility that we are talking to either a file or a console stream — not caring which it is! This is pretty easy to do for output situations. We like to label our output nicely in either a report file or an on-screen report, right? But we do need to watch ourselves a little for input. We don't want to prompt for file input, after all![6]

Okay, okay, don't keep us in suspense any longer! What're these familial types?! Oh, right. They are `istream` and `ostream`. So, if you are making a function to do output, you would pass the stream to use as an `ostream&` type or, for an input function, you'd use the `istream&` type.

### 10.5.3   Two Caveats

This new power comes with two little issues. Neither is truly bad or good, but has aspects of both.

---

[6]For a more advanced way to handle this input conundrum see appendix section H.2.

#### 10.5.3.1   When You Have to Be a File

Sometimes a file just has to be a file. That is, consoles don't share all the same operations as the file streams, after all. The two things they don't do — or rather do automatically and can't be changed — is to `open` and `close`. If you are doing either of these things to a file stream within a function, it must be passed as full-on `ifstream&` or `ofstream&`. It can't be helped.

#### 10.5.3.2   Defaulting a Stream

Two things come into conflux here. We've long-since had the ability to make some arguments default to initial values when the caller failed or forgot to pass them to the function. We've also long-since had references which couldn't default unless they were `const&`. But now we have these nice pure reference streams and we'd probably like to default those to the console so we didn't have to type `cout` and `cin` even more often!

As luck would have it, we can! The reason we couldn't default a pure reference before was because we had no global objects to default them to. You'd have needed such a global to refer to so that the function could see it from anywhere, after all. But now we have the global console streams! You can default an `ostream&` to `cout` or an `istream&` to `cin` with ease.

### 10.5.4   An Old Example Revisited

Let's take a practical example by looking at an old friend: `display_money`. We haven't seen this function in quite some time, so if you need a refresher, you can look back to the original discussion in the last volume.

Here we've got a function to display monetary values with lots of configuration options:

```cpp
const bool UNIT_IN_FRONT{true},
           UNIT_IN_BACK{false};

    inline
void display_money(double amount, char unit = '$',
                   bool unit_front = UNIT_IN_FRONT,
                   streamsize prec = 2, ostream & out = cout)
{
    ios_base::fmtflags old_flags{out.setf(ios_base::showpoint|ios_base::fixed)};
    streamsize old_prec{out.precision(prec)};
    if ( unit_front )
    {
        out << unit;
    }
    out << amount;
    if ( !unit_front )
    {
        out << unit;
    }
    out.flags(old_flags);
    out.precision(old_prec);
    return;
}
```

Note three things of interest here:

- We've defaulted the stream to `cout`. This allows the stream to be specified but it can be left off to display to the console. Care should be taken when having so many default arguments on a function to make sure the ones least likely to change are placed further toward the end of the list, of course. But here there was no clear winner. So I guessed. *shrug*

- We've merged two `setf` calls into one by using a bitwise manipulator: `|`. This technique is covered in more detail in appendix G.

- We've chosen a most horrible name for our stream parameter: `out`. It couldn't be worse, in fact! Why? Well, imagine that you've accidentally typed one of these instances of `out` as `cout`. How likely is that to be noticed? It is a one-character change. It is a word you are used to seeing in output situations. Yeah...not likely at all. *sigh* We'd better try better! Try things like `ostrm` or even just `strm` could work depending on context.

## 10.6 Moving About in a Stream

Sometimes it becomes evident that it is necessary to reprocess some or all of an input file's data.[7] In these situations, never `close` the file and `open` it back up again! This is terribly wasteful of both your time and that of the user. It takes a good deal of communication with the OS, you see, which never goes quickly or smoothly.

Instead, just seek out a new place to get information from in the file. What? That's an odd way to say it? Well, yes, but it matches the syntax chosen by the standards committee. You see, they've called the relevant function `seekg` which is short for 'seek where to move the get position' — effectively.

As mentioned earlier, all input streams have a position from which they are currently getting information — the get position. The beginning of the file is, of course, position 0.[8] So to reprocess all of the file, you could just do something like this:

```
infile.seekg(0);
```

Where `infile` is the name of your `ifstream` variable.[9] (The `seekg` function is available for `cin`, but it doesn't really work, so...)

To reprocess only part of the stream, we'll need a little help. A method called `tellg` will report to you a representation of this position — tell you where the file is getting information — in terms of how many bytes it is past the beginning of the file. This `return` is of the data type `streampos`. This is an odd data type that is restricted to `unsigned` integral values except for the occasion that the library itself needs to store -1 in it for book keeping. It is typically a `class` type to make this happen as smoothly as possible. But for many purposes it works as an `unsigned` integer type.

Once you get a `tellg` result, you can pass it to `seekg` later to return to that position in the file for reprocessing:

```
streampos posit;

posit = infile.tellg();

// ...

infile.seekg(posit);
```

But that's not all! We can also `seekg` to positions not just from `tellg` or 0, but also relative to places other than the beginning of the file. There is a second argument to `seekg` that allows this. This other overload of the function takes a `streamoff`-typed offset and a relative position indicator. A `streamoff`

---

[7]Or we are directed to by the user through some menu choice or command.

[8]You're getting used to this 0-based positioning now, aren't you? *grin*

[9]On some older systems — pre-C++11 — you may have to `clear` the stream before seeking. Or, of course, if your system isn't quite perfect on standards compliance. Check the behavior before going full-scale with it, as always.

value is like a `streampos` except it is `signed`. The the standard even provides that any `streampos` can be converted to a `streamoff`. (The reverse is obviously not possible all the time.)

So what are these 'relative position indicators'? Well, there are three of them: `ios_base::beg`, `ios_base::cur`, and `ios_base::end`. The first is the beginning of the file and is therefore just like the one-argument `seekg`. The second is the current position — aka from where `tellg` would report. Positions relative to `cur` can be either positive, zero, or negative.

But before you go off trying to back up by the number of characters you just read from the stream — a `char` being typically one byte — consider this next bit. There is one `char` that is of variable size depending on its originating OS: the newline. This wee beasty is one byte on Unix-like systems (including MacOS and ChromeOS) but two bytes on Windows and derivatives![10]

This can obviously cause trouble when trying to move around in text files and you can't just rely on seeking to the number of bytes based on the length of the last `string` you read. Even if you remembered that `getline` tosses the newline it processed from the stream, how many bytes was that? If you are developing a dedicated app that'll never run on any other OS, you can get by with hard-coding the 1 or 2 additional bytes. But if you are trying to be portable, that's not possible.

That's where `tellg` also comes in handy! It tells you where you've been so you can back up that way and not have to rely on `ios_base::cur`-relative seeking. I know it feels odd to learn a new tool and then be told not to use it, but it really is more trouble than it is worth — at least in a text file. If you want to use `cur` seeking to good effect, try binary files like in Appendix H.5.

Well, what about that last relative position indicator: `ios_base::end`? That one can come in handy from time to time. The important thing to remember here is that offsets can be only 0 or negative. Zero? Yep. The end-of-file marker is considered 0 relative to the `end` of the file.

### 10.6.1 Be Careful!

Just make sure you never go too far relative to a certain position or in the wrong direction from a certain position. There are typically no safeguards in place to keep you from going before the `beginning` or past the `end` of a file! If you try to go negative from the `beginning` or positive from the `end`, it is liable to 'work'. This could corrupt things on your disk in the worst cases, so be very careful!

### 10.6.2 Full Disclosure

Another way to deal with those Windows line-ending bytes is to convert them to 1-byte variants upon transfer of the file. This is technically allowed for by the ASCII transfer mode in an FTP-style agent, but I haven't seen it properly implemented in years. Most systems default to binary transfer mode and leave the newlines intact.

You can also convert the line endings after transfer more manually. No, don't load it into an editor and retype them all! Just run them through a handy conversion tool like `dos2unix` which is found on or is available for most Unix-like systems. Such a tool is run from the terminal or command or shell prompt — depending on who you talk to. *smile*

### 10.6.3 A Full Example

Perhaps an example is in order. The following program calculates the size in bytes of a file and also converts it to a more human-friendly form for good measure.

```
#include <iostream>
#include <fstream>
```

---

[10]It's even been coded as 3 bytes on at least one OS! Luckily it is now defunct — I've heard. So unless you are processing some older data files, you should be safe from that one.

```cpp
using namespace std;

// assumes the input stream is open
inline streampos file_size(istream & input)
{
    streampos len;
    streampos pos{input.tellg()};  // remember where the caller was

    input.seekg(0, ios_base::end);  // move to last byte
    len = input.tellg();            // number of bytes before here
                                    // the eof marker isn't included
                                    // in the byte size of the file!

    input.seekg(pos);               // return to original position

    return len;                     // return bytes (except eof)
}

// had to convert streampos to streamoff for division
// that's one of the operations streampos doesn't overload
inline streamoff mkb(streamoff size, short & mag)
{
    mag = 0;
    while ( size > 1024 )
    {
        ++mag;              // one more magnitude
        size /= 1024;       // divide out k's
    }
    return size;
}

// what's the prefix for this magnitude?
inline string mkb_pref(short mag)
{
    return mag==0?"":       // nothing
           mag==1?"k":      // kilo
           mag==2?"m":      // mega
           mag==3?"g":      // giga
                  "t";      // tera
           // ...we're already 2 past our name so who cares...
}

int main(void)
{
    ifstream file;
    string fname;
    streamoff mkb_len;
    short mag;

    cout << "what's the name of the file?  ";
    getline(cin, fname);
    file.open(fname);
```

```
    while ( ! file )
    {
        file.close();
        file.clear();
        cerr << "\n\aCould not open file '" << fname << "'!\n\n";
        cout << "what's the name of the file?  ";
        getline(cin, fname);
        file.open(fname);
    }

    file.close();
    file.clear();

    cout << "\nYour file is " << (mkb_len=file_size(file))
            << " bytes long.\n\n";
    mkb_len = mkb(mkb_len, mag);
    cout << "(That's " << mkb_len << mkb_pref(mag) << "b.)\n\n";

    return 0;
}
```

Here we see that we record the position of the file before seeking the `end` so that the caller doesn't have to time their request for the file's size to a certain point in their code. That is, they don't have to record the get position and back up themselves after we've moved things around. Then we move to the `end` and ask where we are. This is the position of the end-of-file marker and — since it is 0-based — it is also the number of bytes that preceed the marker! Luckily we don't even need to add 1 since the eof marker doesn't count as part of the file's size by convention. Once we back up to their original position we `return` the length we recorded before.

The rest of the program just converts the bytes to a more human-readable form with typical metric prefixes. Just note that in file sizes[11] a kilo is 1024 rather than 1000 as in science class.

## 10.7   Layout of Data in a File

The first thing to remember is that the program itself determines the layout or order of data in the file. The user must comply and is not in charge here! When you write the program to read the data from the file in a certain way, that is the way the data **must** be layed out from then on.

So what ways are there to arrange data in a file? Well, there are two basic ways: sequential and block.

In a sequential layout, we put the data one after another with some sort of spacing between until the end of the file. This works for simple files, but not so well for complex ones — or even moderately complex ones!

In a block layout, the data are grouped together in pairs or triples or however many items are needed to fully describe a single entity in the program — a single `class` object, for instance. This has applications in a wide variety of situations as you might expect.

Let's look at some examples of each...

---

[11]And memory sizes too! But not, oddly enough, in disk sizes...

### 10.7.1  Sequential Layout

In a purely sequential layout, we might have a set of numbers to do statistics on listed one after another in a file. Or we might have some names for contacts or products on separate lines of a file. Or…

If the data is more complex, a pure sequential file is a right pain. Let's look, for instance, at a simple list of 2D points. The natural way to list them would be interleaved $x$ and $y$ coordinates. But that would be in blocks, technically. To make it purely sequential, you'd have to list all the $x$ values first and then all the $y$ values.

How do you know where one type of value is done and we are starting the second? We could count as we read them all in and split the list in half in a second pass or we could mandate that the user enter the number of points at the top of the file before the $x$ coordinates start. Either is messy and to be avoided.

#### 10.7.1.1  Common Misconception

A common mistake by programmers new to data file management is thinking that the way they have their test file layed out is the one and only right way. This can lead to trouble when a user sends in a file that is having issues and the layout is even slightly different. The programmer can have a knee-jerk reaction to think the slight difference is the cause of the problem when it has nothing whatsoever to do with it in actuality. For instance, let's say the programmer had a test file with these data in it:

```
42 69
13 43
-3 5
-23 62
```

This file is processing just fine but the user sends in this rearrangement of the file and says there is a problem:

```
42 69 13
43 -3 5
-23 62
```

The programmer could immediately think the slight rearrangement of the data is the issue, but it probably isn't. Let's look at it from a buffer perspective. Here is the first file:

```
+-+-+-+-+-+--+-+-+-+-+-+--+-+-+-+-+--+-+-+-+-+-+
|4|2| |6|9|\n|1|3| |4|3|\n|-|3| |5|\n|-|2|3| |6|2|
+-+-+-+-+-+--+-+-+-+-+-+--+-+-+-+-+--+-+-+-+-+-+
```

And here is the second file:

```
+-+-+-+-+-+-+-+-+--+-+-+-+-+-+-+-+--+-+-+-+-+-+
|4|2| |6|9| |1|3|\n|4|3| |-|3| |5|\n|-|2|3| |6|2|
+-+-+-+-+-+-+-+-+--+-+-+-+-+-+-+-+--+-+-+-+-+-+
```

The only difference is that some of the spacing has changed from space to newline and vice-versa. That is: there is no difference here! Something else must be going wrong for the user and not this file. After all, the only thing >> cares about is the separation by whitespace — not what kind it is.

In fact, many other arrangements of this data will all have the same results:

```
42
69
13
43
-3
5
-23
62
```

Or:

```
42 69 13 43 -3 5 -23 62
```

Or even:

```
42 69                         13
                  43
-3
 5
                  -23              62
```

All of these work the same as the first two! (And it wouldn't matter if there were tabs in them or anything else that was considered spacing...)

### 10.7.2  Block Layout

Let's start the block layout exploration with a simple example of why this is even important. Remember the problem with the *x* and *y* coordinates earlier? What if the data was much more complex? What if we had information about students in a little database like so:

```
5
Joe Sally Hanesh Vong Ella
Soph Fresh Fresh Soph Jr
4 11 9 10 10
Valley Lane
Mountain View
Mountain View
Valley Lane
River Trail
```

Note that the leading 5 tells us how many entities are described in the file. This is followed by the students' names, their class standings, their GPAs (11-point scale), and their branches. Branch names have spaces in them and so are on separate lines.

This is clearly a mess and not readable by the human user trying to make manual adjustments to the data. By simply blocking the data into groups (sometimes called records and the individual data in them fields), we can make it much nicer:

```
Joe Soph 4
Valley Lane
Sally Fresh 11
Mountain View
Hanesh Fresh 9
```

```
Mountain View
Vong Soph 10
Valley Lane
Ella Jr 10
River Trail
```

The user could even keep things more spaced out for further readability enhancement if they so chose:

```
Joe
Soph
4
Valley Lane
Sally
Fresh
11
Mountain View
Hanesh
Fresh
9
Mountain View
Vong
Soph
10
Valley Lane
Ella
Jr
10
River Trail
```

And, noting that the student's name will be extracted (>>) from the file, we could even allow more blank space before it to help separate the blocks from one another:

```
Joe
Soph
4
Valley Lane

Sally
Fresh
11
Mountain View

Hanesh
Fresh
9
Mountain View

Vong
Soph
10
Valley Lane

Ella
```

```
Jr
10
River Trail
```

How would we tackle reading in something like this with the data all mixed together like that? Quite simply with a standard EOF loop:

```cpp
file.peek();
while ( !file.eof() )
{
    object.read(file);
    // do something with object just read
    file.peek();
}
```

Here `object` is an object of the `class` we've written to represent the 'student' concept from the program. We call this `class`' `read` method and pass it the file stream to read from (as an `istream&`, of course). Then we can put the `object` in a `vector` or process it immediately or...whatever we need to do with it!

What does the `read` method look like? It is pretty simple as well:

```cpp
bool Student::read(istream & istrm)
{
    string name, branch, Class;
    double gpa;
    istrm >> name >> Class >> gpa;
    getline(istrm, branch);
    return istrm && set_name(name) && set_class(Class) &&
            set_gpa(gpa) && set_branch(branch);
}
```

Like always, read into local variables and call the mutators. I like to `return` a `bool` to let the caller know it went well or not, even if they don't care.

### 10.7.3   Labeling Data

With JSON and XML being such popular ways to transport data these days, it's easy to forget that labeling data has been around for many, many years. So, let's start with the basics and do some name-value pairs or label-value pairs for our data files.

Taking the student database above as an example, we might see the following in a data file:

```
name = Joe
class = Soph
gpa = 4
branch = Valley Lane
name = Sally
class = Fresh
gpa = 11
branch = Mountain View
```

Here each piece of data is stored alongside a labeling string of characters like "name" or "class". This lets the user see immediately what the value is in context of the database's idea/schema. The labels

and values are separated by some conveniently chosen symbol like an equal sign or colon. I've chosen to not have the labels need quotes around them like in a JSON file. Nor the values, for that matter. The separator is enough to tell one side of the line from the other and so begins our adventure into parsing (reading) such data.

### 10.7.3.1  First Try

We could try to read this data like so:

```
file >> setw(MAX_LABEL) >> label >> sep
     >> setw(MAX_NAME) >> name;
```

Here `label` and `name` are C-strings with the implied maximum allocations and `sep` is a `char` to hold the equal sign.

But this attempt doesn't account for several things. Firstly, the user might re-order their information and, as long as it is within the block, this should be fine by us:

```
name = Joe
gpa = 4
class = Soph
branch = Valley Lane
class = Fresh
gpa = 11
name = Sally
branch = Mountain View
```

Note that the same data is present as before, but the blocks have been internally shuffled. The assumption here is that your program should be able to place the correct data into the correct variables (of the `class` object) by the context of the labels. That is a major assumption! Are we ready for it? I say let's GO!

### 10.7.3.2  Another Go

Let's pause after reading the `label` part and see what it is before we deal with the value part:

```
const char known_labels[MAX_KNOWN_LABELS][MAX_LABEL] = { "name",
                                                         "gpa",
                                                         "class",
                                                         "branch" };
file >> setw(MAX_LABEL) >> label >> sep;
L = 0;
while ( L < MAX_KNOWN_LABELS && strcmp(label,known_labels[L]) != 0 )
{
    L++;
}
```

Here I've made a simple array of C-strings with all the labels we are supposed to know about in this program. Then, after reading the `label` and separator, we look for the `label` read amongst the labels known. At the end of the loop, the variable L will either be the index of the matched label or it will be the constant `MAX_KNOWN_LABELS`. (Both of these are `size_t` type variables, of course.)

After this simple linear search we can use a `switch` to find out which label was seen and take an appropriate action:

```
switch ( L )
{
    case 0:
        file >> setw(MAX_NAME) >> name;
        break;
    case 1:
        file >> gpa;
        break;
    case 2:
        file >> setw(MAX_CLASS) >> year;
        break;
    case 3:
        file.getline(branch, MAX_BRANCH);
        break;
    default:
        // nothing -- ignore unknown labels?
}
```

Easy-peasy! It seems perfect. Almost too perfect...

### 10.7.3.3   A Third Tack

But, if we are labeling data so that it may be easily read, it might also be easy to edit/change outside our program. Users don't type as carefully as our program reads, of coruse. The user might end up with something like this:

```
name=Joe
Class=Soph
gpa=4
branch= Valley Lane
name =Sally
 class = Fresh
GPA =  11
branch  =Mountain View
```

And now our program can no longer read this!

We can write a case-insensitive C-string compare function to dodge the case issues this person is facing in the labels. This kind of thing was done for the `string` `class` in the first book of this series, for instance.

But what about all the spacing issues? Having the separator up against the label like that is disastrous, for instance.

Luckily the labeled data is one item per line of the file. So, we can read the whole line at once and then do some C-string processing — which we are getting pretty good at — to break up the pieces within our program:

```
// get line of data
file.getline(labeled_line, MAX_LINE);

// find the separator
sep_at = search(labeled_line, '=');
```

```
// copy the label over -- watch for the maximum size!
lcap = min(sep_at,MAX_LABEL)-1;
strncpy(label, labeled_line, lcap);
label[lcap] = '\0';
// we'll throw out any extra text before the separator...

// copy rest into value string:
value_index = 0;
do
{
    sep_at++;
    value[value_index++] = labeled_line[sep_at];
} while ( value_index != MAX_VALUE_LEN &&
            labeled_line[sep_at] != '\0' );
```

Here `search` is a function to look [linearly] through a C-string for a particular character. `min` is found in the standard library `algorithm` if you don't want to code it for yourself.

### 10.7.3.3.1 Another Approach

If we are willing to use pointers, things can be done more efficiently:

```
char * sep_at;  // make sep_at a pointer rather than a size_t index
file.getline(labeled_line, MAX_LINE);
sep_at = strchr(labeled_line, '=');  // use C-string library to search
*sep_at = '\0';                      // split string logically in two
strncpy(label, labeled_line, MAX_LABEL-1);
label[MAX_LABEL-1] = '\0';
strncpy(value, sep_at+1, MAX_VALUE_LEN-1);  // use pointer to second half
value[MAX_VALUE_LEN-1] = '\0';
```

We have to use the pointer to the separator for a couple of tasks here that at first seem unusual. First we overwrite that position with a null character. This makes a single `char` array into two C-strings essentially. Since we have the pointer, though, we have the locations of both.

Then we use that address (`sep_at+1`) as the starting point to copy the value from and into the `value` C-string.

### 10.7.3.3.2 Back to Task

But we also now have an issue looking up the label in the array of known labels – skipping that spacing that may/may not have been on either side of the separator. It is now in our `label` C-string and it won't prove the same as the labels in the known array.

We could make our case-insensitive C-string comparison function able to skip spaces — that's a reasonable option to include on such a function, after all. Or we could make a general space-removal function to help in lots of places this might become an issue. I guess I really telegraphed that one, eh?

This can be done with C-strings with a bit of effort:

```
// count number of leading spaces
lead_space = 0;
while ( isspace(str[lead_space]) )
{
```

```
        lead_space++;
}

// shift data over
moving = 0;
while ( str[moving+lead_space] != '\0' )
{
    str[moving] = str[moving+lead_space];
    moving++;
}
str[moving] = '\0';

// remove trailing spaces
while ( moving != 0 && isspace(str[moving-1]) )
{
    moving--;
    str[moving] = '\0';  // overwrite each with a null character
}
```

When making this into a function, make sure you give the caller the option to remove/strip spaces from either the left end, the right end, or both of the C-string. This is easy to do with an enumeration. And even cooler if you use some bit manipulation techniques.

(Of course, if you're willing to use pointers, the shifting loop could be done with `strcpy` instead.)

Now call this function on both your `label` and your `value` C-strings.

### 10.7.3.3.3 Finishing Up

With all this in place, the `class`' input function probably looks something like this:

```
bool Class::read(istream & strm)
{
    // known label array
    // enum to parallel array and clarify switch below
    strm.peek();
    while ( !strm.eof() && !end_of_block )
    {
        // read line
        // split line at separator
        // search for label -- case-insensitively
        // switch to translate and store value based on label
        //          -- all values are strings and we need numbers as numbers
        //             and bools as bools, for instance
        strm.peek();
    }
    if ( seen_any )
    {
        strm.clear();
    }
    return seen_any;
}
```

Wait, what are `end_of_block` and `seen_any`? Those are there to track two other conditions we didn't mention yet. First, since the block elements can now be rearranged, it is a little tricky to tell when

we've read all the elements for a block. So we have to keep track somehow and I'm flagging that all of them have been read with a `bool` flag for our loop. Second, it is best to handle the situation of a block with missing data. `seen_any` is a `bool` to track that any of our elements were read properly and thus we have a partial block of data we can report to the caller.

How do we detect/track these two situations? There are numerous ways, of course. For simplicity we could use a set of `bool`s for a short set of known labels we were tracking. But for extensibility, we'd probably use an array paralleling the known labels array itself. Then we can use a simple loop to accumulate the state of `end_of_block` and `seen_any` after each line has been processed.[12]

One other issue for the `end_of_block` situation ties in with the partial block situation. This is when the partial block isn't the last block in the file. How do we detect this and what do we do about it?

Well, we can use our `bool` array to realize we've seen a label already to detect that we've read too far and that this duplicate label is not ours but the next block's data. But how to fix that we've already read that line and how does the next block get back that data? Hmm... Aha! `seekg`/`tellg`! We'll do a `tellg` before each line is read and we'll then `seekg` back to that location when the duplicate label is detected. We will also mark `end_of_block` as `true` forcibly so we don't forget and try to read even more!

Three more thoughts: a) how to translate the numbers, b) what to do about encountered problems, and c) what to do with skipped elements in a partial block. The first we can do with either functions like `stod`, `stol`, etc. from the `string` library or with their underlying core functions `strtod`, `strtol`, etc. The underlying functions take C-string's as pointers and allow for more configurability in the translation like from what base the numbers are to be assumed and so on. Or we could use the tools from the next section on `stringstream`s (section 10.8).

For the second we can either ignore offending lines without separators or with unknown labels. Or we can print error messages. Or we can `throw` `exception`s as discussed in chapter 12.

For the third we can just default initialize the object before the EOF loop begins so that everything will be in a clean state if not overwritten during processing.

### 10.7.3.3.4   But What About

What about the main program? They just need to call the `read` method in their `while` head in an object-oriented fashion:

```
while ( object.read() )
{
    // process object
}
```

When this loop ends, there will be no more blocks in the file and thus no more data to process. If the `read` method `return`s `true` it has read at least a partial block and it is safe to process it.

### 10.7.4   Mixing Layouts

So far our designs have been pretty simplistic. But is that as complex as it gets or can get? Of course not! Even in a pretty basic program we can have things like nested data! In `class` terms, we could have a `vector` inside a `class`. In data file terms, we'd have a sequence inside a block.

These nested sequences could be numeric or non-numeric in general terms. Let's look at each situation in turn.

---

[12]Some would advocate a counter, but this method gives us finer information of what has or hasn't been seen and we can make more careful reporting at the end of the function if necessary. It will also prove helpful in a minute so wait for it!

### 10.7.4.1  Numeric Sequences

Let's say we had a situation like this, for instance:

```
name
other data here
numeric sequence
```

It is important to note that the numeric sequence is last and that the first thing in the block is a name of some kind. This let's us detect the end of the numeric sequence by either an EOF or the `failure` that will happen when we read for a number and hit the next block's name. Overall we might code this:

```
peek
while ( !eof )
{
    read name
    read other data
    read first of sequence
    while ( good )   // file will either eof or fail at end of sequence
    {
        process sequence item
        read next in sequence
    }
    clear  // forget failure if not done
    peek   // reprime eof test
}
```

Another way this could go down is that the data could be labeled. This is actually advantageous in that we no longer have to worry about the name coming next in a roundabout read — there will be a label or EOF after the numeric sequence! The big question here is: should the sequence still be on a line beside its label or should it be allowed to wrap to the next line or further?

After all, if we just extract the numbers (>>), they'll be able to cross lines with their list. To prevent this is almost excessive — it is their data, after all. If we wanted to, though, we can just `peek` for a newline `char`acter, of course, so it isn't really that bad on our end.

One solution is to introduce syntax for a continuation mark. The C & C++ languages do this with a \ character, for instance. This is implemented fairly crazily, though, in that the wrapped line is immediately concatenated with the original line without indention removal.[13] We could look for such a marker as easily as a newline and know that processing of the sequence should keep going until the next newline.

### 10.7.4.2  Non-Numeric Sequences

If you have a sequence of non-numeric data inside your block, more care is warranted. Since numbers read into `strings` just fine, we need to either `peek` for a digit (`isdigit`) before each extraction (>>). We also need to know in general where the end of this sequence is because it might otherwise subsume the next element of data!

Labeling the data can help here as long as the labels themselves are not allowed as list members. We can also use a special flag value that can't be a list member to terminate each sequence. Or we could fall back on the old "put the count at the start of the list" method we used back in the pure sequence file that contained both *x* and *y* coordinates.

Whatever you do here, it is going to be tricky so be careful.

---

[13]This just drives me crazy, perhaps, because I like to make my code look readable and neat.

### 10.7.5  Comments in Data Files

Often times data files are commented with what are knonw as meta-data. This is typically information that isn't abotu the file contents so much as it is about those who have processed the file — who edited it, when, why, what did they change, etc. Such information can be important when the department's sales figures are all wrong — then we'll know who to blame. *grin*

Your program won't need to read and understand these comments anymore than the compielr doesn't understand your comments in your C++ program. However, you will have to process past them to get to the next bit of data — much like when we did nice input notation with our user back in the previous volume.

First we must pick some sort of syntax that will denote a comment. One of the most common symbols is the pound sign (`'#'`). But others are also used: `'*'` in FORTRAN, `';'` in ini (initialization) files, `'!'` in Access data bases, etc. And many use not just a single character but a short sequence of them to denote a comment: `"rem"` for both ini files and BASIC, `"//"` for C++ or Java, etc. We'll just pick a single character to represent our comment mark for simplicity.[14]

There are also multiple styles of comments: a) whole-line, b) end-of-line, and c) block.Whole-line comments are pretty much a subset of end-of-line comments that just don't allow for there to be other information before the comment mark on the line — most likely whitespace, but nothing else.

An end-of-line comment, though, can appear at any point on the file line and extends to the end of that line. This is the most common allowed comment style and isn't too hard — especially in a labeled format.

Block comments are pretty tricky in that they can span either several lines of the file or just a smaller part of a line that isn't necessarily to the end of that line. There is also the issue of whether you allow them to nest inside of one another. Although this can be done, not even the C++ language allows block comments to be nested! This might be a goal for later or just some extra credit when you have more time on your hands. *smile*

Lastly, should comments be relegated to just the top of the data file or should they be allowed at any point in the file. This is a big deal and can lead to both trouble and efficiency. The trouble is that user's will place comments all over the place whether you remind them of the top-only requirement or not.

The efficiency potential is that if you have a bunch of comments at the top of the file and need to reprocess the file later, you can remember where the comments ended with a `tellg` and seek back to that point later instead of to 0 and rescan the comments all over again!

Overall, I recommend not limiting the user to comments at the top because it is relatively easy — especially in a labeled process — to strip comments from an input before processing a line for data. In fact, if you follow the idea from labeled line processing and combine it with the techniques of the next section, you can strip end-of-line comments from any file with ease and even take care of separating blank lines at the same time!

## 10.8  string Streams

Our last necessary topic for streams is from the `sstream` library. It is a combination of streams and `strings` to allow you to trun a `string` into a stream — essentially.

"Why would this be useful?" you may ask. Well, we've been focused so far on either console interaction or storing/retrieving data from permanent storage on a drive of some sort. But much of our interfaces with the world are through other means. And those deal muchly with `string` (or C-string) data.

For instance, GUI[15] front ends do lots of `string` work with labels for buttons, windows, and messages for the user. And when the user enters info into an input text field, that comes to your program as a

---

[14]But a sequence is doable in labeled line processing for little to no extra cost!

[15]Graphical User Interface

string — even if it has a number in it!

Also, network communication is almost all done with `strings` of information instead of raw numbers and such. Being able to turn our regular data into a `string` form is essential to such transmission. And, at the other end of it, we'll need to pull the data back out of the received `string`.

Let's start with putting the data into a `string` and then we'll work on getting it back out again.

### 10.8.1   Output to strings

To 'output to a `string`' we have to create an `ostringstream`[16], output to it, and then ask for the resulting `string`. So if we wanted to put the number 42 into a `string` we could do this:

```
ostringstream oss;
oss << 42;
string s{oss.str()};
```

Now the `string` s has the value 42 in it. This will work with variables of all types, of course. But further, the `ostringstream` `class` is compatible with `ostream` just like the console stream `cout` is. So anything you could do to an output file stream or `cout` you can do to an `ostringstream` as well! You can even pass an `ostringstream` to an `ostream&` parameter to get it worked on in a custom function and then get back the `string` from its `str` method afterwards.

But the main thing about that is that we can take the attributes of `cout` or some other `ostream`-compatible source and copy them into our `ostringstream` so it will output to its `string` just like they would have. Take this function, for instance:

```
inline string num_to_str(double num, ostream & out = cout)
{
    ostringstream oss;
    oss.imbue(out.getloc());
    oss.flags(out.flags());
    oss.precision(out.precision());
    oss << num;
    return oss.str();
}
```

Here we take the `flags` and `precision` from the given `ostream` and set them into the `ostringstream` so that the given number is formatted like it would have been on the given stream. This is important so that we don't use default formatting when the caller has set up special treatments.

But if they have an `ostream`, why are they turning this data into a `string`? Well, like we said, the data might be bound for a GUI or network interface instead of `cout` or a file. But they want formatting to be consistent throughout their program, perhaps. (Not necessarily a thing for a network, but definitely necessary for a GUI!)

Should we take the `fill` and `width` as well? That's debatable. Some situations call for it and others don't. I think it will be fine here for an individual value to turn it into a `string` and let the output of that set these formats. But if it were going out to a GUI, we might need to do it ourselves.

Wait, what about that `imbue` stuff? That is fancy verbiage for, "tell me how you format thousands-group separators and decimal points in the local customs."[17] `getloc` will return the current `locale` information from the `ostream` we are given. This tells that local customs are a ',' and a '.' or vice versa or whatever they may be. And we then `imbue` that information into our `ostringstream` to make

---

[16]Lovely name, isn't it?
[17]You realize that these two things change around the world in different countries, right?

it act the same way as local custom dictates. We wouldn't want to start giving a European customer American looking measurements, now would we?

And, to do other data types is similar but doesn't need `precision` set up because only floating-point types need that.

You don't have to use this as a separate function, of course. But having a set of routines in a library isn't a bad idea, either.

### 10.8.2   Input from strings

Similar to output, 'input from a `string`' is accomplished with an `istringstream` object. You either construct the `istringstream` with the `string` or set it with the `str` method. (If setting a new `string` after you've already processed another, make sure you `clear` the `istringstream` before calling `str`!) In other words:

```cpp
istringstream iss("42");
short num;
iss >> num;
```

You can also `imbue` the `locale` from an `istream` like `cin` to get local customs on decimal point and thousands group separators as with `ostringstreams`. There are also some `flags` that are used in input, so you might want to set those, too. `precision` doesn't really get used during input, though, so you can skip that. *smile*

```cpp
inline void str_to_num(const string & str, double & num,
                       istream & istrm = cin)
{
    istringstream iss(str);
    iss.imbue(istrm.getloc());
    iss.flags(istrm.flags());
    iss >> num;
    return;
}
```

The main problem we might have here is that if the caller wants to take more than a single value from the same `string`, this function won't work. See here:

```cpp
string s{"4.2 -8.5"};
double a, b;
str_to_num(s, a);
str_to_num(s, b);
cout << a << ", " << b << '\n';
```

This will print 4.2 twice instead of 4.2 and -8.5. The reason is that we pass the same `string` to the function twice, but the local `istringstream` is therefore constructed twice and starts each time from the beginning of the given `string`. If we could remove the first value from the `string` before the second call, we could make progress. But that would depend on application specifics as to how the data are separated in the `string`.

Perhaps separate functions/calls for parsing a single `string` aren't the best idea? But, still, could we make it work? The problem is that the functions are using separate `istringstreams` each time. If we could get the functions to share a single `istringstream`, things would work much more smoothly.

We could put the `istringstream` in the global area of the program... *eww* That would be horrible! We **NEVER** use global variables!

We could put them together in a `class`. Hmm...that has potential! Let's try it:

```cpp
class Extractor
{
    istringstream iss;
public:
    Extractor(const string & str, istream & istrm)
        : iss(str)
    {
        iss.imbue(istrm.getloc());
        iss.flags(istrm.flags());
    }
    void get_num(double & num)
    {
        iss >> num;
        return;
    }

    // more overloads of get_num for other types of data
    // (maybe even a get_char or get_str or get_bool for
    // those types of data, too!)

    // this would allow a class object to be changed from
    // the original string to a new string -- rather than
    // creating a new Extractor object
    void reset(const string & str)
    {
        iss.clear();
        iss.str(str);
        return;
    }

    // could have a format changing function, too, that takes
    // a new istream to get formatting from
};
```

Now we can use a single `Extractor` object to get all the data from a single `string`:

```cpp
string s{"4.2 -8.5"};
double a, b;
Extractor xtr(s);
xtr.get_num(a);
xtr.get_num(b);
cout << a << ", " << b << '\n';
```

This now results in 4.2 and -8.5 being displayed! (The space between the values is eaten by the `>>` on the `istringstream` as usual.)

### 10.8.3 A Practical Example

Here, then is a more practical example using a simple `class` for rolling a set of dice for gaming:[18]

```cpp
class DieRoll
{
    short count{1}, sides{6}, adjust{0};
    bool only_pos(short & to, short from)
    {
        bool okay{from > 0};
        if ( okay )
        {
            to = from;
        }
        return okay;
    }

public:
    DieRoll(short c, short s, short a)
    {
        set_count(c);
        set_sides(s);
        set_adjust(a);
    }
    long roll(void) const;
    short get_count(void) const
    {
        return count;
    }
    short get_sides(void) const
    {
        return sides;
    }
    short get_adjust(void) const
    {
        return adjust;
    }
    bool set_sides(short s)
    {
        return only_pos(sides, s);
    }
    bool set_count(short c)
    {
        return only_pos(count, c);
    }
    bool set_adjust(short a)
    {
        adjust = a;
        return true;
    }

    void output(ostream & ostr) const;
```

---

[18] Check out these pages at Wikipedia for more on dice generally and gaming dice notation particularly.

```cpp
        bool input(istream & istr);
};


long DieRoll::roll(void) const
{
    long tot{0L};
    for ( short d{1}; d <= count; d++ )
    {
        tot += 1L + rand() % sides;
    }
    return tot + adjust;
}
```

And let's focus on the implementations of its `input` and `output` methods more carefully. Let's do output first as that is easier:

```cpp
void DieRoll::output(ostream & ostr) const
{
    ostringstream oss;
    oss.imbue(out.getloc());
    // could also copy flags, precision, fill, etc. from ostr to oss
    //      oss.flags(ostr.flags());
    //      oss.precision(ostr.precision());
    //      ...
    // (but we have integers here and are using the ostream to do our
    // width/fill settings)
    if ( get_count() != 1 )
    {
        oss << count;
    }
    oss << 'd' << sides;
    if ( adjust != 0 )
    {
        oss.setf(ios_base::showpos);  // doesn't affect ostr so we
                                      // don't have to save/reset
        oss << adjust;
    }
    ostr << oss.str();
    return;
}
```

Note that this collects the entire `DieRoll`'s data together in a single `string` before it is output. This will make sure the entire object is in one `width` setting together instead of just the `count` or the `'d'` being in the `width` alone. This can make or break a formatting setup and is a great side-benefit of the string stream tool!

Now for the `input` method:

```cpp
bool DieRoll::input(istream & istr)
{
    short t;
    string line;
    char pm;
```

```
    istringstream iss;
    iss.imbue(istr.getloc());   // imbue with locale information from stream
    getline(istr, line, 'd');   // get count and take out 'd'
    if ( !istr.fail() )         // getline would fail if no 'd' were ever found
    {
        if ( line.empty() )     // no count listed
        {
            set_count(1);
        }
        else                    // get count from string
        {
            iss.str(line);
            iss >> t;
            set_count(t);
        }
        istr >> line;           // get rest of text from stream
        iss.clear();  // if we went through the else, eof got set and needs
                      // to be cleared before we can reset the buffer string
        iss.str(line);
        iss >> t;               // get number of sides -- mandatory
                                //    (might cause fail or eof)
        set_sides(t);           // will be garbage if fail/eof,
                                //    setter checks
        iss >> pm;              // get +/- for adjust
        if ( !iss.good() )      // adjust is optional
        {
            set_adjust(0);
        }
        else                    // it was there!
        {
            iss >> t;
            if ( pm == '-' )    // it should be negative
            {
                t = -t;
            }
            set_adjust(t);
        }
    }
    return istr;                // return in bool context reports good
}
```

Here I've mixed some `istringstream` with some `istream` to make sure we've hit all the features properly. Particularly I wanted to make sure we showed how to `clear` an `istringstream` before calling `str` to set its `string` in the face of possible EOF or `failure`.

A more general `input` method might read from a `string` instead of a `istream`. This would prove useful in a GUI or network program, for instance. But having `input` come from a stream is more traditional and avoids a temporary buffer copy (see below).

### 10.8.4   Caveats and Tips

Other than the obvious uses listed above, we also noted that you can use an `ostringstream` to collect an entire object's data into one `string` so that it fits into a single `width` specification on an `ostream`.

We also saw that an `istringstream` could be useful to help an `istream` with more complex line

inputs. (But it isn't always necessary here if you are careful with your `istream` you'll usually find a way to just work with it directly.)

But don't take this as an 'in' to read an entire file into a `string` and then start parsing that! Such a duplication of the entire stream buffer or even a whole disk file into a `string` is not only RAM and heap intensive, but really, really **SLOW**, too.

## 10.9   Wrap Up

In this chapter we've explored stream input and output more deeply than we had in the first volume. We briefly reviewed the basics of reading and writing files, but quickly dove deeper for a look at several aspects of the topic. We discussed what can go wrong during a file open attempt and how to deal with the most basic problems. We spoke of how to pass files to functions and why it is the way it is. We moved around in the stream and tracked our location therein. Then it was time for a deep look at how to arrange data in a file. And finally we looked at treating `string`s as if they were streams making it easier to connect our C++ code to alternative front-end systems.

That's it for this chapter, but if you want more on file topics, be sure to check out Appendix H as well!

# Part V
# C++ Tools

# Chapter 11

# operator Overloading

Some students ask at this time, "What exactly **is** operator overloading?" Well, it allows us to make our class types and some other types interact with the standard C++ operators just like the buit-in types do. This has the effect of removing much of the need for dotting method calls and also makes possible some advanced techniques of generic programming (see Chapter 14).

## 11.0.1 An Example

For instance, we can take a class whose objects currently have to input and output like so:

```
cout << "Please enter data:  ";
object.input();

cout << "You entered ";
object.output();
cout << ".\n";
```

And let them interact with like normal, built-in types instead:

```
cout << "Please enter data:  ";
cin >> object;


cout << "You entered " << object << ".\n";
```

Isn't that **sweet**?!

## 11.1  All the operators

Rather than try to maintain my own list of operators for this ever-changing language,[1] I've decided to link you to a few lists maintained by experts and enthusiasts alike.

- This chart at CppReference — where the C++ standards committee hangs out a lot — talks to both precedence and associativity. It's only 'failing' is that it lacks entries for the casting operators like `static_cast`. These are described on separate pages that you can find linked from the 'functional cast' link in the above table.

- At wikipedia we find two useful charts. One with general precedence and associativity — which includes the casting operators — and another that talks to the actual overload-ability of the operators. (Tip: if the entry in the prototype column is a dash, you can't overload that operator. See below for more about this issue.)

It remains, then, to find out what exactly precedence and associativity are. Precedence is simply the order of operations when they are mixed together — which comes before others. For instance, multiplication has a higher precedence than addition.

Associativity is whether an operator processes its operands from left-to-right or rather from right-to-left. For instance, addition works from left-to-right but assignment works from right-to-left. This rarely affects us in daily programming, but can at times and is historically placed in this chart together with precedence.

## 11.2  Rules of Overloading operators

There are three sets of rules involving the overloading of operators in C++. They tell you things you can never do, things you have to do, and things you should do.[2]

### 11.2.1  Thou Shalt Not

You can never, ever:

- overload `?:`, `::`, `.`, `.*`, or `sizeof`. This restriction on `::` affects both its unary and binary forms — for global `namespace` access and access of items within a `namespace` or `class` scope.

- create your own operators. You are limited to those operators defined in the language already. (See the charts above for the current list.)

- change the arity of operators. The 'arity' is the number of operands an operator needs. This can be unary, binary, or ternary in C++. But the ternary operator is already off limits by the rule above. This rule affects all operators except () — the function call operator — which has an arbitrary number of operands and so can be set to anything you need for the job!

- change the precedence of operators. operator precedence is hard-coded into the language and cannot be changed.

---

[1]New operators were just added in C++20!
[2]You technically don't have to, but you really, really should!

- change the associativity of operators. operator associativity is hard-coded into the language and cannot be changed.

- overload operators for the built-in types. In particular, students often want to make % suddenly work for floating-point types. This cannot be done.

### 11.2.2  Thou Shalt Always

You must always:

- overload =, ->, (), and [] as member functions. These cannot be coded as non-member or friend functions.[3]

- overload anything with a non-target type as the left-hand operand as a non-member function. By 'target type' here I mean the class or other type that you are trying to overload the operator for. This rule will help in situations like teaching your type to display itself on an output stream, for instance. In such operations, the stream comes on the left side of the operator and the object to be displayed comes on the right.

### 11.2.3  Thou Should Always

You should always:

- make the overloaded operator's meaning as close to that of how it relates to the built-in types as possible. This will avoid confusion by those using your class when trying out your great new operators.

- at least make your 'clever' overloads — those not conforming to the above rule — as clear and obvious as possible.

- overload for built-in compatibility whenever possible or, of course, necessary. That is, if your class would naturally interact with whole numbers in the real world, consider overloading operators for it to interact with long or the like in C++.

Again, you don't have to follow these last three 'rules', but they are **strongly** recommended!

## 11.3  Patterns for Unary and Binary

While the wikipedia article on overload-ability mentioned above does have prototypes of each operator in both member and non-member forms where appropriate, I'd like to talk a little about it here as well.

operators come in two basic arities as far as overloading goes: unary and binary. I'll show the general patterns for overloading in each arity in either member form or non-member form.

### 11.3.1  Unary operators

First, what operators are unary? Well, there are, of course, logical not (!) and arithmetic negation/opposite (-). But there are others that we don't think about or haven't yet seen as well: unary plus (+), ones complement (~), dereference (∗), and address of (&). In addition, although arrow for member access from a pointer (->) is a binary operator, it is overloaded in unary style. I'm also going to leave increment and decrement to their own section (11.5) as they are a bit more complex.

---

[3]If you don't know what a friend is in terms of C++ class designs, consider yourself lucky! Okay, actually it isn't that bad, but friend functions or classes can directly manipulate the private portions of objects of a class which grants such friendship. Kinda defeats the whole 'encapsulation' thing! It can allow for speed unless you've inlined everything already. But it leads to more accidents than should be tolerated. I recommend never making friends for your classes.

I'll show a general example of overloading for the operator for arithmetic negation, but the others would be the same overall. Let's start with the member function form. Let's say you were defining the function non-inline as well. The declaration in the class — let's call it OOver — would look like so:

```
class OOver
{
    OOver operator-(void) const;
};
```

Note that the function name is the keyword operator followed by the operator's symbol. Very unusual and not normal, but still intuitive. There can even be space between the symbol and the keyword if you like that:

```
class OOver
{
    OOver operator - (void) const;
};
```

Note also that there is no argument since there will be a calling object to negate.

Further, we mark this operator const to avoid changing that calling object. We want to send back a new object that represents the calling object's negation — not change the calling object itself! This is just the way arithmetic works on numbers and variables in math and built-in type evaluations and we like to keep it that way.[4]

And its definition would then look something like this:

```
OOver OOver::operator-(void) const
{
    OOver t(*this);
    // arithmetically negate t
    return t;
}
```

To call this operator, you would just need to have code like this:

```
OOver x;

x = -x;
```

But, in some compilers, you might receive an error message in a different form. I've seen compilers show problem reports like this x.operator-() rather than -x when the operator is one that has been overloaded by the programmer. This is odd at first glance and freaks out those new to operator overloading. I just wanted you to be forewarned in case your compiler was of this ilk.[5]

If, on the other hand, you wanted the operator to be a non-member — perhaps even a friend, you'd code the declaration like so:

---

[4]This style has been discussed before and is known — for obvious reasons — as an arithmetic style of function design.

[5]We'll also seen at some point I'm sure that you can use this syntax to call an operator function from inside a member function definition. As with any member function call, the original calling object is passed along to the newly called member function as its calling object. This precludes the need to use *this to get the calling object as many on the internet seem to believe in doing.

```
class OOver
{
    friend OOver operator-(const OOver & o);
};
```

Here we note that the `operator` function does need an argument because we are defining it as a non-member. Also, this declaration would go outside the `class` definition if we weren't making it a `friend`, as usual.

Also, we make the argument a `const` reference for speed and to protect it from change. This keeps that arithmetic style as we did for the member function version above.

And the definition would look something like this:

```
OOver operator-(const OOver & o)
{
    OOver t(o);
    // arithmetically negate t
    return t;
}
```

And the call would look exactly the same as above. Unless there were a crazy error report. Then it might look something like this: `operator-(x)`. Note that now x is an argument to the `operator` function instead of its calling object.

### 11.3.2   Binary operators

The binary `operator`s are numerous and include every one of them besides the above and `?:` for deciding between two values. But to be complete, here is a list of the ones that will concern us:

```
+    -    *    /    %                               // arithmetic
&    |    ^    <<   >>                              // bit operations
&&   ||   <    <=   >    >=   ==   !=               // bool operations
=    +=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>= // assignments
[]                                                 // subscript
,                                                  // comma
```

They are in no particular top-to-bottom order or even left-to-right order. I just wanted to group them by general function category. You'll probably note that the function call `operator` — () — is missing. It has arbitrary arity as it can have as many or few arguments to the function as needed. I'll treat it specially in a separate section (11.10). Likewise, I'll treat the left and right shift `operator`s (<< and >>) separately due to their special treatment in C++ as regards output and input (11.4).

Further, let me just blanketedly say that all the `bool` operations should not change their single operand and should result in a `bool` value. Also all the assignments should follow the style of their base `operator` (=) as we learned in the section on overloading that `operator` before (9.2.3.4) and `return` the left-hand operand by pure reference.

And, lastly, the comma `operator`'s normal purpose is to separate two expressions where only one should normally exist. This is sometimes handy in a `for` loop head to have multiple index variables initialized and updated at once.

Again, let's take a particular `operator` as an example. I'll take addition (+) as my focus. As a `class` member, it would be declared like so:

```
class OOver
{
    OOver operator+(const OOver & p) const;
};
```

And its definition would look something like this:

```
OOver OOver::operator+(const OOver & p) const
{
    OOver t(*this);
    // add p to t
    return t;
}
```

Then, as a non-member, its declaration would look like this:

```
class OOver
{
    friend OOver operator+(const OOver & o,
                           const OOver & p);
};
```

Again, if not a `friend`, you'd make that outside the `class` definition.

Either way, the definition might look like so:

```
OOver operator+(const OOver & o,
                const OOver & p)
{
    OOver t(o);
    // add p to t
    return t;
}
```

No matter how you design it, it would be used like so:

```
OOver x, z;

z = z + x;
```

And those strange error messages might look like `z.operator(x)` for a member version or `operator(z,x)` for a non-member version. Note how the left-hand operand is either the calling object of the `operator` function or its first argument. This is always the case as we mentioned when overloading `operator=` (9.2.3.4) back in the dynamic memory section (9.2).

## 11.4   Stream operators

Let's say we had a very basic `class` for working with rational numbers (fractions — improper or otherwise). It would have two getters and a single setter — since the numerator and denominator might cancel with one another we must mutate them together, after all.

Imagining such a `class`, let's override `operator`s for its input and output with streams. Their prototypes would look like so:

```
istream & operator>>(istream & istr, Rational & r);
ostream & operator<<(ostream & ostr, const Rational & r);
```

Here we see that the streams themselves act as the left-hand operand and that, therefore, we have to overload these operators as non-member functions as per the rule mentioned above.

Also note that the Rational argument to the extraction operator (>>) is by pure reference so we can change it but the one to the insertion operator (<<) is a const& for speed and safety.

A main program — or any function, really — could then call them as in:

```
Rational r1, r2, r3;

cout << "Enter three [improper] fractions:  ";
cin >> r1 >> r2 >> r3;

cout << "Read:  " << r1 << ", " << r2 << ", " << r3 << '\n';
```

Note how we can chain these together naturally. This is because we have returned the original streams from the functions to act as the next operated on object:



See how the input of r1 results in cin which is then used in the subsequent input of r2 and so on until the final resultant cin is just laid aside due to the semi-colon ending the statement.

And what do the functions' definitions look like? We would definitely have operator>> look like so:

```
istream & operator>>(istream & istr, Rational & r)
{
    long n, d;
    char t;
    istr >> n >> t >> d;

    if ( !istr || !r.set(n,d) )
    {
        istr.clear(ios_base::failbit);
    }

    return istr;
}
```

Here we call the setter after locally reading in the data format needed for the class.[6] Then we use the success of the stream itself combined with the success of the setter call to decide whether to set the stream to a failed state or not.

But the operator<< should look like this, typically:

---

[6]The char t is to read in the slash character that is typically used to separate a numerator and denominator in horizontal representations.

```
ostream & operator<<(ostream & ostr, const Rational & r)
{
    return ostr << r.get_numer() << '/' << r.get_denom();
}
```

Once again we use the getters to display the parts of the `class` object onto the stream in proper format/notation. But we also use that last copy of the stream as our `return` value! Instead of having two separate statements where one sets the stream aside and the next picks it up to `return` it, why not just `return` it straight away?

## 11.5   Increment/Decrement operators

As mentioned before, the increment and decrement `operator`s are a bit different. The reason for this difference is that they are actually two `operator`s in one. Remember that you can place the ++ or -- on either the left or right of the operand. In fact, these two positions have different behaviors — it isn't just for looks!

When you pre-increment, you change the value of the operand to be one more than it was before and the result of the operation in the surrounding context is the new value. Thus:

```
short x = 4, y;

y = ++x;
```

will result in both `x` and `y` being 5. Pre-decrement is similar but you end up with the value being one less than at the start.

However, when you post-increment, you still increase the value of the operand, but the result in the surrounding context is the value from before the increment! So:

```
short x = 4, y;

y = x++;
```

would produce `x` as 5 but `y` as 4. (And similarly for post-decrement.)

With that knowledge out of the way, let's look at how to code these for a `class` of our own. Let's keep working with the `Rational` `class` from before. As member functions, the prototypes of the two increment `operator`s — the decrements are left as an exercise — would look like this:

```
Rational & operator++();      // pre-increment
Rational operator++(int);     // post-increment
```

Wait. What's that unnamed `int` there? And why is one `return`ing a reference and the other not? One thing at a time. Hold your horses.

The unnamed `int` argument is to distinguish the overloads from one another. After all, the names of the two functions are the same and the `return` type is ignored during overload determination. So the only way to distinguish them from one another was to add an unused argument. Unused? Yep. We not only don't name it, we don't even use it! It is a placeholder only — no functionality.

As to the reference `return` versus not, the post-increment is sending back the original value of the operand and so that can't be the calling object as its value would have changed. But for the pre-increment, the value `return`ed is the new value and we can go ahead and return the calling object itself.

Since this object is guaranteed to still exist in the caller's context, we can send it back by reference. Also, this is how the built-in types behave and we want to be as much like them as possible.

What that reference means, of course, is that you are allowed to chain the pre-increment if so desired: `++(++x)` would have the same effect as `x += 2`.

So how would the definitions for these look? We typically do the bulk of the work in the pre-increment and then reuse that work in the post-increment's definition:

```cpp
Rational & Rational::operator++(void) // pre
{
    numer += denom;
    return *this;
}


Rational Rational::operator++(int)    // post
{
    Rational ret(*this);
    ++(*this);              // could also say operator++()
    return ret;
}
```

I didn't call the setter to check for cancellation because this is guaranteed to work based on the rules of basic arithmetic: we can't end up with a new cancellation due to this addition.

As noted in the comment, we could just say `operator++()` to call for pre-incrementation in the post-increment definition. I merely used the `this` pointer since I was using it already and because it was a little shorter to type.

What if you wanted to code these as non-members? Then the prototypes would look like so:

```cpp
Rational & operator++(Rational & r);      // pre-increment
Rational operator++(Rational & r, int);   // post-increment
```

Note that the unnamed `int` is still there, but moved to the $2^{nd}$ argument position.

And the definitions would change slightly as well:

```cpp
Rational & Rational::operator++(Rational & r) // pre
{
    r.set(r.get_numer() + r.get_denom(), r.get_denom());
    return r;
}


Rational Rational::operator++(Rational & r, int)    // post
{
    Rational ret(r);
    ++r;              // could also say operator++(r)
    return ret;
}
```

As an aside, the fact that pre-increment does the bulk of the work and post-increment reuses that work with one or two additional copy operations leads us to use pre-increment for most of our loop work. The only time most folks these days use post-increment is for special occasions or on built-in types where speed is always guaranteed by hardware. On `class` types — like `Rational` here or `iterators`, we always pre-increment instead.

## 11.6   A Case Study in Compatibility

To talk to the point of "thou should" make your operators compatible with built-in types whenever possible/necessary, I'll use the Rational class and operator+ in binary form. Seeing as the integers are a subset of the rationals in math, we should probably make them inter-compatible in the programming world, too.

We'll look at this in several passes to see several nuances of how these operations work and how constructors get involved as well. In all cases, the constructor and setter will call this function to make sure the Rational object is in a standard form:

```cpp
void Rational::normalize(void)
{
    // ensure we're in lowest terms
    long g = gcd(numer, denom);
    if ( g != 0L )
    {
        numer /= g;
        denom /= g;
    }

    if ( numer == 0L )                  // silly zero
    {
        denom = 1L;
    }
    else if ( denom < 0L )              // neg --> top
    {
        numer = -numer;
        denom = -denom;
    }
    else if ( denom == 0L )            // stupid zero!
    {
        denom = -1L;
    }
    return;
}
```

Here I've called for a greatest common divisor of the numerator and denominator. This can be your own gcd function or the one from the numeric standard library. Then I fix up some other issues that might make the object less presentable — like negative denominators.

(This technically has nothing to do with the compatibility issue or the operator overloading, but I will show calls to it so I wanted it here for completeness. I won't show the setter or getters, but they are pretty standard.)

### 11.6.1   First Pass

This first pass will involve a pretty plain constructor and 3 overloads of operator+. One will handle the adding of two Rational objects. Another will handle adding a Rational to a long. The third will handle adding a long to a Rational. Wait. Isn't that the same thing? Well, in mathematical terms, yes. But in the compiler's eyes, no. The compiler doesn't understand commutativity, you see. It's been taught that idea for the built-in types, but not for any types you create. So if a type is commutative, we have to explicitly show that in our code.

I'll make these member functions whenever possible. Here are the prototypes and the definition of

the constructor:

```
class Rational
{
    long numer, denom;
    void normalize(void);
public:
    Rational(long n = 0L, long d = 1L)
        : numer(n), denom(d)
    {
        normalize();
    }
    // getters and setter

    Rational operator+(const Rational & r) const;  // rat + rat
    Rational operator+(long n) const;               // rat + integer
};

Rational operator+(long n, const Rational & r);   // integer + rat
```

The only thing with the constructor is that it is kinda cool. It has, after all, three call signatures! It can be called with two, one, or even no arguments. Just keep this in mind as it comes into play in later passes...

So the definitions are pretty straight forward:

```
Rational Rational::operator+(const Rational & r) const
{
    return Rational(numer*r.denom + denom*r.numer, denom*r.denom);
}

Rational Rational::operator+(long n) const
{
    return Rational(numer + n*denom, denom);
}

Rational operator+(long n, const Rational & r)
{
    return r + n;
}
```

Notice that the first two use RVO to optimize away a local temporary and the third just rotates its arguments to call the earlier version. If we had `inline`d all of these, they would be uber efficient!

Also note that the middle one could be as coded or could invoke the constructor in either of these forms: `operator+(Rational{n})` or `*this+Rational{n}`. These would make a `Rational` anonymous object from the integer argument and that would force a call to the first overload using our calling object as its calling object.

Either way, we can now call on these operators like so:

```
Rational a, b, c;
long d;
```

```
c = a + b;
c = a + d;
c = d + a;
```

Since they `return` new objects arithmetically, we can even chain them together like so:

```
Rational a, b, c;
long d;

c = a + a + b + b;
```

Of course, if you are going to do a lot of adding the same `Rational` number to itself, you should probably invest in an `operator*` or two. *smile*

### 11.6.2  A Second Take

This second version of overloading `operator+` for intercompatibility between `Rational` and integers looks a lot like the previous one. In fact, it uses all the same code except for one thing: the middle overload for `Rational` plus integer is missing!

```
class Rational
{
    long numer, denom;
    void normalize(void);
public:
    Rational(long n = 0L, long d = 1L)
        : numer(n), denom(d)
    {
        normalize();
    }
    // getters and setter

    Rational operator+(const Rational & r) const;  // rat + rat
};

Rational operator+(long n, const Rational & r);   // integer + rat
```

And when we run the program having skipped the middle overload:

```
Rational a, b, c;
long d;

c = a + b;
c = a + d;
c = d + a;
c = a + a + b + b;
```

All works fine — no compile messages or anything! What gives? Did we not need that other overload after all? Well, it would seem we didn't here. So why is that? What is happening to make up for it?

Remembering what we said in that `operator+` about being able to create an anonymous `Rational` out of the integer parameter, we can start to see what happened here as well. The compiler likes the code we write to work out. And when it sees a type incompatibility, it looks really hard for a way around

that. This automatic conversion between types is called coercion. It uses roughly the same means as type casting would, but is done on our behalf by the compiler instead of explicitly by us.

One of the standard paths used in this coercion process is to call an acceptable constructor to make one object from another type of data. And that's exactly what happens here. Since our constructor can be called with a single integer argument, the compiler takes it upon itself to do so for us and — voila — a `Rational` is born to be used in a call to the first overload!

In other words, the code parses out as this:

```
c = a + d;
// c.operator=(a.operator+(Rational{d}));
```

The `operator=` used is the compiler-provided one, of course, since we didn't write our own this time.[7]

One might ask, can't we take out the non-member overload, too, and let the compiler coerce the left integer into a `Rational`? Unfortunately there is a rule in coercion that a calling object cannot be created. So for that left-hand value, we can't rely on the coercion of a new object. Or can we..?

### 11.6.3   Yet Another Version

This pass will make two furthers changes to the last one. I'm going to take out the last member `operator+` and alter the parameters for the non-member overload:

```
class Rational
{
    long numer, denom;
    void normalize(void);
public:
    Rational(long n = 0L, long d = 1L)
        : numer(n), denom(d)
    {
        normalize();
    }
    // getters and setter
};

Rational operator+(const Rational & r1 n, const Rational & r2);  // rat + rat
```

The rationale here is that, since a calling object can't be coerced into being but an argument can, we'll make both of the objects arguments and none as a calling object! It's brilliant!

Turns out that this works just fine and now we see things like this:

```
c = d + a;
// c.operator=(operator+(Rational{d}, a));
```

working just fine. All the earlier code works just fine, in fact!

Wow! Who'd have thought that with a little help from the compiler and its coercion via construction we could make our `class` interoperable with integers in one function?

---

[7]For more on writing an `operator=` of your own, see section 9.2.3.4 from the dynamic memory chapter (9).

## 11.6.4   What, Again?

But the above won't always work. There are classes where such a one-argument constructor would be undesirable or even harmful! What, for instance, would happen if the string class had a constructor that could take a single integer and turn it into some sort of horrible space-wasting run of null characters of length equal to the integer plus one?!  This could have happened if the standards committee had allowed the char to repeat on this constructor:

```
string(size_type count, char repeat_me);
```

to default to the null character.[8]

Now, I'm not saying that we shouldn't be able to construct Rational objects from integers — that would be silly!  But what if there were such a situation for some other class we were designing: the caller can construct an object from a single value when they really want to but it should never happen on accident.

The standards committee has us covered!  They have a keyword for that, in fact: explicit. If we place this mark before the head of a constructor, it tells the compiler that this constructor should never be used in a coercive way.  It can only be used when a programmer has explicitly asked for such a construction, you see.

This keyword only has useful meaning on a single-parameter constructor, but I have seen compilers accept it on any constructor. I'd skip that sort of thing, though, if I were you.

So what would this look like on the Rational class?  And what impact would it have on our operator+ experiment? Let's take a look:

```cpp
class Rational
{
    long numer, denom;
    void normalize(void);
public:
    explicit Rational(long n = 0L, long d = 1L)
        : numer(n), denom(d)
    {
        normalize();
    }
    // getters and setter

    Rational operator+(const Rational & r) const;  // rat + rat
    Rational operator+(long n) const;              // rat + integer
};

Rational operator+(long n, const Rational & r);   // integer + rat
```

Note that we are back up to three overloads of operator+ to make this all work.  This is because no Rational objects can just spring forth from integers mixed into the + operations anymore!

explicit is a powerful keyword so be careful how you use it!

---

[8]This constructor prototype is a simplification from a three-parameter constructor with lots of other things going on. You won't find it anywhere else.  But the third parameter actually does default, so it isn't needed for this discussion.

### 11.6.5  Summing Up

In summation,[9] we've seen that a normal, one-parameter constructor — or at least a constructor that can be called with only a single parameter — can be used along with as few as one `operator` overload to make your `class` intercompatible with a built-in type. But if your constructor needs to be `explicit`, you'll need up to two more `operator` functions to make that intercompatibility happen.

This count is per built-in type, btw. If making your own `class` to handle string-like operations, you'd want many of them intercompatible with both C-string and `char` if not also the real `string` `class`. That would need at least four extra or up to six extra overloads for each of those `operator`s!

## 11.7   += and Its Kind

Just as commutativity is only understood for the built-in types, short-hand `operator`s like += are understood only for the built-in types as well. So, just coding a + and an = won't automatically garner you the ability to += at will!

How do we code for these situations? Let's take a look at a mythical `Complex` `class` for an example. One way might be like this:

```cpp
Complex & Complex::operator=(const Complex & c)
{
    if ( this != &c )
    {
        real = c.real;
        imag = c.imag;
    }
    return *this;
}


Complex Complex::operator+(const Complex & c) const
{
    return Complex(real+c.real, imag+c.imag);
}


Complex & Complex::operator+=(const Complex & c)
{
    return operator=(operator+(c));
}
```

I went ahead and coded `operator=` even though no dynamic members were involved, just for completeness of the example code. The only thing mine does that the compiler-provided one wouldn't is to check the address of the calling object against that of the provided argument reference. The compiler's version would just blithely overwrite the member data without a care!

Anyway, so what's going on here is that both `operator=` and `operator+` are doing their own work and `operator+=` is relying on that work to be done well in its implementation. I also decided to avoid using `this` frivolously and just called the other `operator`s manually.

Some people don't like this version, however, because of the simple fact that there is a hidden temporary being created during the process. The result of `operator+`, after all, is an anonymous object and has no permanent residence. But we pass it to `operator=` all the same. It has to be held onto, therefore, for the length of time that `operator=` needs it until it can be thrown away at the end.

These people might take more kindly to this version, instead:

---

[9]Sorry, couldn't resist the pun...*chuckle*

```cpp
Complex & Complex::operator=(const Complex & c)
{
    if ( this != &c )
    {
        real = c.real;
        imag = c.imag;
    }
    return *this;
}

Complex & Complex::operator+=(const Complex & c)
{
    set_real(real + c.real);
    set_imag(imag + c.imag);
    return *this;
}

Complex Complex::operator+(const Complex & c) const
{
    Complex temp(*this);
    return temp += c;
}
```

Here we've switched the roles of operators + and += in terms of who relies on whom. This time operator+= is doing all the actual work and operator+ is reliant on the other.

Does this actually do better vis-á-vis temporaries? Yes and no. The temporary from before has been made more explicit here — no relation to the keyword — by making a temporary variable. But that just seems to me to hide the other temporary even better! The other one is still the result of operator+ itself. If used in other situations, it will still hold dearly to life instead of just being used and tossed aside as was intended. So there is still only one hidden temporary, but there is also an explicit temporary as well.

Our only saving grace might be if the compiler decides to use the facility of named-RVO. This optimization is much like RVO except here the compiler must realize that the local named variable is just being acted on and then used as the return value and thus can be conveniently created in the return area for efficiency. While RVO is a mandate by the C++ standard (C++17), named-RVO is still just a strong suggestion by the committee.

## 11.8   Subscript operator

There are still a few special operators to talk about, though. First up is operator[] — the subscript or indexing operator. This can be applied to an object to access its members. And those members don't have to be encapsulated arrays, vectors, or strings, either! They can just be normal fields/members of the class.

How? Well, the subscript parameter doesn't have to be a size_t, you see. You can designate that it be a char or string, if you like. Then the programmer using your class can request a member by name or initial. Let's examine this on our Rational class, for example:

```cpp
class Rational
{
public:
```

```
        long operator[](char member_init) const;
};
```

Here we've let the caller indicate they choice of member by a `char` parameter instead of a `size_t`. This makes more sense because designating 0 or 1 or whatever for numerator versus denominator is odd to say the least.

Also remember that this `operator` must be a member function by standard mandate!

The implementation could look like so:

```
long Rational::operator[](char member_init) const
{
    member_init = static_cast<char>(tolower(member_init));
    return member_init == 'n' ? numer
                              : member_init == 'd' ? denom
                                                   : -42;
}
```

I've lowercased the initial to make sure we don't worry about capitalization preferences of other programmers. Then I check if it is an `'n'` or a `'d'` — for numerator or denominator respectively, of course. The -42 is just an error flag in case they mistype and ask for member `'z'` or `'?'` or something other than `'n'` or `'d'`.

## 11.8.1   A Second Form

The really careful observer will notice that this `operator[]` is `const` and so won't change the calling object. Well, we also know that such a mark is considered when determining whether two member functions are overloaded or not, right? So, it is possible to overload `operator[]` a second time without the `const` mark.

This is often done to `return` members by reference for mutation. After all, the above form was kind of like an accessor (getter) an so it might be nice to have a mutator (setter) form as well.

This is done, for instance in the `string` and `vector` `class`es so that programmers can change the elements in those containers if they are not `const`ant containers.

Since this form will need to `return` a reference to whichever member, all those members need to be the exact same type. We'll also need some way to `return` a reference when they don't send a proper parameter. This bit is tricky, but we'll find a new tool invaluable here!

The basic idea is:

```
long & Rational::operator[](char member_init)
{
    member_init = static_cast<char>(tolower(member_init));
    return member_init == 'n' ? numer
                              : member_init == 'd' ? denom
                                                   : SOMETHING;
}
```

We just need to know what `SOMETHING` will be and how to code it.

Our first instinct is probably a local variable in the function.  But this won't work because that variable's memory will cease to exist when the function `return`s and we will no longer be able to refer to it!

The second typical thought is a member variable. This would work, but it would make there be an extra member for **every** object created of the `class` type. That seems excessive.

Okay, then a `static` member variable? This might work, but would still be `class`-wide and generally accessible unless we made it `private`. Hmm...what to do?

Well, it turns out that member variables and `const`ants aren't the only things that can be made `static`. We can also make functions' local variables `static`. This has the effect of making them stay around even between function calls. They are generally only accessible within the function.[10]  But we can `return` a reference to them since they do stick around between calls!

Let's see now:

```
long & Rational::operator[](char member_init)
{
    static long ERROR;
    member_init = static_cast<char>(tolower(member_init));
    return member_init == 'n' ? numer
                             : member_init == 'd' ? denom
                                                  : ERROR;
}
```

I could initialize it to, say, -42, but the user is unlikely to notice since they are using this for mutation purposes. And since a `static` local variable is only initialized once on the first function call, it wouldn't stick, either. So I'm just leaving it as garbage bits. It's the thought that counts, right? *grin*

Actually, I over-stated something just now. I said the caller was using this function for mutation. But that might not be the case. It turns out that the non-`const` version of a method is used for any non-`const` object calling the overloaded method name. It doesn't have to be the `const` overload for access, after all!

## 11.8.2   But Haven't We..?

Haven't we broken encapsulation now? Why yes, yes we have. Note that the programmer using the `Rational` `class` can not only put new values in their object with this non-`const` overload, but they are bypassing our old normalization code that made the rational be in lowest terms and look nice in a display. Se here, for instance:

```
Rational a{5,15};    // a contains 1/3

// but here...
a['n'] = 5;
a['d'] = 15;
// a now has 5/15 in it -- no cancellation!
```

Further, the programmer could put any old crazy thing in our members. As long as it looks like a `long` integer or some subset type, it'll go right in there:

```
a['n'] = '5';
a['d'] = (long)"15";
```

---

[10]This tool is often used to track how many times a function gets called for what is known as profiling. Profiling can help trace what parts of the program are using the most time and then you can focus optimization efforts on those codes instead of trying to optimize everything. This can save lots of time on a larger project.

Now the `Rational` object has 53 as its numerator and some memory address in `long` form as its denominator! *eek*

Is this a good idea? Of course not! Then why allow it? Well, if we did have a member `vector` or the like and we were passing through our `operator[]` to it, we could do so by reference unless we were also protecting the content of the container as well as the container itself. What? That is, thinking as if you were actually coding the `vector` `class` itself, you sometimes want to protect the overall container but don't care about the content inside so much. A `vector`, after all, doesn't care what is inside other than the type of it. What the `class` itself protects is the underlying dynamic array that holds the data — beginning and ending pointers, perhaps. The content themselves are irrelevant to the `vector`.

### 11.8.2.1  A Further Twist

Furthermore, there are also members known as `mutable`. These members are part of the `class` but don't need to be treated as `const`ant even when the rest of the data in the `class` is being so treated. What? Sorry, I'm just confusing you for fun now, aren't I? Well...

Let's take a new version of a `string` `class` that allows the programmer to choose whether comparisons are to be case sensitive or not at run time. This would be a great feature and we'd love to have it, right? But, in designing it, we think about `const`ant objects. Should they be locked in to being whatever default case was set up or should they be allowed to change their case sensitivity on the fly even while maintaining the `const`ancy of their content?

So, we have a `const`ant `string` with, say, `"Jason"` in it. And we are comparing that to another value, say, `"jason"`. If the original object were set in case sensitive mode, this comparison would fail to see their equality. The human using the program would become very irate and we'd lose their business! But if we allow the object to change its sensitivity without changing the contents, this becomes a breeze. Just change the sensitivity and compare away! The contents would not be affected by this shift, but the comparison result is all better now.

Another example is that of a `class` representing a set of values. When we go to print these values, we might want to use standard notation like curly braces around the elements and commas between them. But in another application, we might want to change that up to be spaces between and angles around the list. Hmm... These new `string` members we'll add to configure these things will possibly even change from place to place in the same application. Thus, we'll want them to have mutators and not just constructor parameters.

But should they be able to change when the object is `const`ant? Should the object's display be able to change when the set content is locked into certain values? I can't see a reason to do so, so we can set those members to be `mutable`.

This new keyword is merely placed before the member's type and let's the compiler know that — even in a `const`-marked function — this member can be changed. So we might have:

```cpp
class String
{
    mutable bool case_sensitive{false};
public:
    bool set_sensitivity(bool on_off) const;
};
```

Or:

```cpp
class Set
{
    mutable string before{"{ "}, after{" }"}, between{", "};
```

```
public:
    bool set_surround(const string & bef, const string & aft) const;
    bool set_between(const string & betw) const;
};
```

This tool is often used for any kind of maintenance data. Data that is about how we use the real data of the `class` but not that data itself.

Some of you are wondering why this is in the section on `operator[]`. Well, it was the most opportune place to put it. It does, after all, fall into the discussion of encapsulation and member protection here. Also, you could use a `const` mutator version of `operator[]` to allow these things to change with some sort of name/initial to do so. At least if they are all the same type like the `strings` for the set example.

### 11.8.2.2  What About any?

Some students say to me "I found this type on cppreference.com the other day called `any`. Couldn't we use that to allow for different types to be changed from the same `operator[]` with a reference `return`?" Wow! You do come up with stuff when you go a-searching!

`any` is a type found in the `any` library. It allows you to store literally any type of information in a single variable albeit still just one value at a time. To retrieve the information, you must use an `any_cast` which will either give you the value back or `throw` the exception `bad_any_cast`.

This cannot be used here because an `any` contains a **copy** of its initializer rather than a reference to it. If you make a change to the `any`, it won't affect the original. So you can't make changes to the original member variables — `mutable` or not — via the `return`ed any object.[11]

## 11.9  Typecast operators

All this talk about encapsulation violations with the reference `return`ing or non-`const` version of `operator[]` gets me thinking about another possible issue we've faced in C++ over the years. And it also happens to be an `operator` issue!

This time it is a whole classification of `operator`s known as the typecast `operator`s. These `operator`s have a distinct look and feel all their own and are used when you, for instance, do a `static_cast` of your `class` object into another data type.[12]

For instance, we could overload a typecast `operator` to approximate our `Rational` `class` objects as `double` values. This would look like this:

```
Rational::operator double (void) const
{
    return static_cast<double>(numer)/denom;
}
```

as a member function. As a non-member function, it would need a single `Rational` argument — preferably passed by `const&`.

Here we see that there is no `return` type listed as it is inferable from the name of the `operator` function. This, in turn, is the usual keyword `operator` followed by the desired type to turn the object into. The definition should be fairly self-explanatory by now.

The caller could use this function like so:

---

[11]Yes, the same issue applies to `variant` objects.

[12]This is as opposed to the other direction which would be a constructor call to a one-argument constructor.

```
Rational r;

cout << r << " is almost like " << static_cast<double>(r) << ".\n";
```

(I assume we've overloaded insertion (<<) for `Rational`s as above as well.)

#### 11.9.0.1  A Side Issue

There is a continuing debate about `operator˜` and a `Rational` `class`. Should it be reciprocal — flipping the numerator and denominator like it flips the bits in a built-in integer — or should it be approximate — like the tilde over an $\approx$ symbol?

I'll leave that for you and your colleagues to decide, but just wanted to point it out and this seemed like a decent time to do it. (It also speaks to the "make your 'clever' overloads … as clear and obvious as possible" rule. Your idea of clear and obvious might not be the same as that of other programmers on the team.)

### 11.9.1  So What Was That About Encapsulation?

Some might think this would be a good way to get out a C-string representation from a `string` object and wonder why we use the `c_str` method instead. Well, let's say `string` had this `operator`:

```
string::operator const char * (void) const
{
    return priv_char_star_member;
}
```

If this mythical method existed, a particularly lazy or vengeful programmer might do something like this:

```
strcpy((char*)static_cast<const char*>(str_var),
       "new information that won't fit");
```

Here they've gotten out a C-string version and cast that pointer to a writable variation with C-style casting! Truly evil! But further, they pass that writable version to `strcpy` as the destination and store there something that the current `string` object doesn't have room for. Diabolical!

What's to stop them from doing the same thing with `c_str`? Well, there are two provisions in the `c_str` description within the standard that rule this out. First, it says that the pointer will be invalidated (as with `iterators`) by passing a non-`const` version of the pointer to any standard library function. Second, it says that writing to the pointed to array is **undefined behavior**. This is a biggie and implies horrible things for the perpetrator.

## 11.10  Function Call operator

First of all, `operator()` is the function call `operator`. It is not the parentheses used to change order of operations or to group operations. It is not the parentheses used to prototype or define a function and delimit its arguments list. It is the parentheses used after a function name to pass actual arguments to the formal arguments and evaluate/run that function's code.

By overloading this `operator`, we make our `class` objects look and act like a function at times. After all, we'd place the `operator()` right against our object's name just like we would to call a function by name:

```
OOver obj;

obj( - - - )     // replace - - - with actual parameters
```

(I didn't put a semi-colon on that because I'm not limiting this function to a `void` `return`. We could further use its `return` value in a surrounding expression, after all.)

Since our objects now look and act like functions, many will call them function objects and our `class` a function object `class`.[13]

But you may ask, "Why would we do such a thing? Why make a `class` object look and act like a simple function?" Well, the answer is three-fold:[14]

- preserving state

- multiple access paths to state

- multiple concurrent sets of state

So what is 'state'? That's the set of variables that tell where a certain process was last at. If you got to a certain point in a process, you'll want to remember where you were so that when you come back after lunch or break or whatever, you can pick up where you left off. The note you leave for yourself — physical or mental — tells the state of the system in which the process works. Variables do the same thing in a computer program.

In the situations we are looking at here, a single function call represents but a step in the process — not the whole thing. The most common occurrence is that we are working on a list of information from either a file, a `vector`, or some other data source like a sensor hooked to the computer with a USB cord or Bluetooth. A call to the function will process a single item from this source of information and the state will keep track of how far that item got us in the overall process.

If we were adding or averaging the values, we would keep track of the total and count of values we'd calculated so far between calls, for instance. When the entire list has been seen and sent through the function, we'll look at the state to tell the results.

We'll treat the two approaches — functions and function objects — separately and see how each tackles these issues.

## 11.10.1   Function Objects

As a `class` object, we can solve these issues easily. Let's tackle each in turn. But first let's take a high-level view of our plan:

- Preserving State:

  We'll use member variables to hold onto state information. Since member variables of an object continue to exist between calls to any of its functions — even an `operator()` — they will be preserved between calls.

- Multiple Access Paths

  Since member variables are available not only to the `operator()` but to any of our member functions, any/all of them can use those state values for look-up or even changing them.

- Multiple Concurrent States

---

[13]Others will call them 'functors'. But it has been unclear to me whether they are saying the objects are the functors or the `class` is the functor. I'm not very fond of the word, so I really don't care to delve further.

[14]At least for now. When we get to the `template` chapter (14), we'll find out a whole other reason!

To have multiple concurrent sets of state, all we would have to do is to declare multiple objects of the same `class`. Each such object will have a separate and independent set of member variables to preserve a separate set of state within.

### 11.10.1.1   A Function Object class

Let's start with a function object `class` to keep track of the largest value seen in a list being processed. Remember that our `operator()` function will see only one value at a time from this list and some other part of the program will be calling on us to update the state with each new value brought into the program.

```cpp
class Largest
{
    long max_so_far, count;

public:
    Largest(void)
        : max_so_far(0), count(0)
    {
    }
    Largest(long max)
        : max_so_far(max), count(1)
    {
    }

    long operator() (long next)
    {
        count++;
        return max_so_far = count == 1 ? next
                                       : next > max_so_far ? next
                                                           : max_so_far;
    }
    long operator() (void) const
    {
        return max_so_far;
    }

    long reset(void)
    {
        long max = max_so_far;
        max_so_far = count = 0;
        return max;
    }
    long reset(long max)
    {
        count = 1;
        swap(max, max_so_far);
        return max;
    }

    long get_count(void) const
    {
        return count;
```

```
    }
};
```

Note that the member variables `max_so_far` and `count` record the current state of the maximum/largest finding process as each `next` value comes into the `operator()`. But don't be confused! The `class` has **two** `operator()` functions!

The second one with no arguments is for retrieving the most recently updated value of the `max_so_far` member variable. `count` is mostly for show and really just helps us at the first element of the list to make the right decision. After that it is not really useful/used. We relegate retrieving that value to a plain getter.

We also have two `reset` functions to parallel the two constructors. These take care of resetting to an empty or nothing-yet-processed list situation and resetting to a we've-seen-the-first-item-in-the-list situation, respectively. Both `return` the previous `max_so_far` to mimic the way the stream formatting functions would `return` the prior formatting settings before making the update to your requested new formatting setting. I always thought that was a cool technique and thought I'd use it here.

To determine a maximum for a set of values, let's say they are in a `vector`, we need a loop and a call to a function object of this `class`:

```
vector<long> data = { 4, 7, -2, 8, 2, 1, 9, 7, 4, -4 };

Largest max;

for ( auto d : data )
{
    max(d);
}

cout << "The largest item is " << max() << ".\n";
```

If we wanted to find another maximum for another `vector`, we don't even need a second object. As long as they aren't simultaneously coming in, we can just `reset` this object (`max`) and use it again:

```
vector<long> data2 = { 14, 17, -12, 18, 12, 11, 19, 17, 14, -14 };

max.reset();
for ( auto d : data2 )
{
    max(d);
}

cout << "The largest item is " << max() << ".\n";
```

### 11.10.2  Typical Usage

There are actually two typical usages of function objects when processing lists. One is as above as a countr or skimmer or just processor of the list's data. It skims off information about the data as it rolls by.

Another usage is as a producer, generator, or originator of list data. A producer takes data from a source and relays it one piece at a time to the rest of the program. This source could be a file, a `vector`, or some other source like an Internet connection to a server or user's browser somewhere.

As we've already explored the `Largest` countr, let's look at a typical producer. This one's source is a random sequence!

### 11.10.2.1  Producers

This is a producer function object `class`:

```cpp
class DieRoll // standard gamer dice information:  dDs+a
{
    const short dice, sides, adjustment;

public:
    // all error checking left to the reader!
    DieRoll(short s = 6)
        : dice(1), sides(s), adjustment(0)
    {
    }
    DieRoll(short d, short s)
        : dice(d), sides(s), adjustment(0)
    {
    }
    DieRoll(short d, short s, short a)
        : dice(d), sides(s), adjustment(a)
    {
    }
    // rolls the dice and returns the tops' pip total + adjustment
    long operator() (void) const
    {
        long total = adjustment;
        for (short d = 1; d <= dice; d++)
        {
            total += rand() % sides + 1;
        }
        return total;
    }
};
```

I've decided to make the members `const`ants to ensure that once a set of dice is created it won't get changed arbitrarily. (Wouldn't do to have that fancy magic axe suddenly doing less damage than it has all along, now would it?) Because of them being `const`, these members must be set in the member initialization list. Not everything can be done with the in-`class` initialization syntax! (Also remember that every object created can fill in these members with different values. The `const` just says that, once filled in via the constructor, the values can't change for the rest of the object's life.)

Although it might seem that the `const` on the `operator()` function itself relates to the above decision, it really is more a separate design decision to make it so that any object — `const` or not — can be used to roll the dice.

To produce a new value from the random sequence that is rolling a set of dice, just call `operator()` on the object:

```cpp
DieRoll standard, DnD{3,8,-2}

srand(static_cast<unsigned>(time(nullptr)));
```

```
cout << "Standard die roll from variable:  " << standard()
     << '\n';
cout << "3d8-2 roll:  " << DnD() << '\n';
cout << "Not-so-standard die roll from temporary object:  "
     << DieRoll{17}() << '\n';
```

Here I've set up a standard die roll of one 6-sided die by default and another object representing, perhaps, a battle axe wielded by someone who is a little too small to handle it. Then I set up the random number generator from `cstdlib` used in the `class` for simplicity.

Rolling these dice is just a matter of calling `operator`() as you can see. I even rolled a new die created on the spot which has 17 sides and no adjustment and is rolled alone.

### 11.10.2.2  An Example: Preserved State and Multiple Access Paths

If we combine these with the `Largest` object from before, we can find the largest value rolled over several experiments (as the statisticians would say):

```
// track 5 rolls of DnD-style dice for maximum value
for ( long count = 0; count != 5L; count++ )
{
    max(DnD());
}
cout << "Largest DnD-style die roll:  " << max() << '\n';

// now track 5 rolls of a standard 6-sider for maximum value
max.reset(standard());
for ( long count = 1; count != 5L; count++ )
{
    max(standard());
}
cout << "Largest standard die roll:  " << max.reset() << '\n';
```

Here the `max` object tracks the state between calls in the `for` loop and then reports with a different `operator`() call afterwards.

Then it is `reset` for another run with a different source/producer. Multiple access paths are also shown here with the multiple uses of the `reset` function and its overloads.

Further, we see preserved state in the producers themselves since the parameters for the production of new values is stored inside the objects and doesn't need to be passed again to the function each time a new value is desired.

### 11.10.2.3  An Example: Multiple Concurrent States

But what about multiple concurrent states? How can we show this facet?

Let's add two more `Largest` finding objects — one each for the two producer objects. I'll name them cleverly so we can easily keep track of which is for which.

```
Largest dnd, std;

// now track separate, concurrent maximums -- one on
// each sequence!  (five each)
```

```cpp
cout << "\nTracking two sequences concurrently:\n\n";
for ( long count = 0; count != 5L; count++ )
{
    dnd(DnD());
    std(standard());
}


cout << "Largest DnD-style die roll:  " << dnd() << '\n';
cout << "Largest standard die roll:  " << std() << '\n';
```

Once again we roll the various dice of the producers 5 more times and find the largest rolls during this time. But this time the producers are run side-by-side in lock-step as it were. Their results aren't separated in time but interleaved. We've tracked their sequences simultaneously instead of in tandem.

But this is hardly the way sources of information would arrive from, say, Internet connections or external data sensors. Those would be more sporadic and less regimented. How can we check for that? Well, I've now added a third producer called `flip` representing a coin rather than a die. It will tell us which source has come in this time over the course of the experiment:

```cpp
DieRoll flip{2};

// now we are tracking two sequences concurrently, but
// they are asynchronous in their arrival...  (50 total)
cout << "\nTracking concurrent, asynchronous sequences:\n\n";
dnd.reset();
std.reset();
for ( long count = 0; count != 50L; count++ )
{
    if ( flip() == 1 )
    {
        dnd(DnD());
    }
    else
    {
        std(standard());
    }
}

cout << "Largest of " << dnd.get_count()
    << " DnD-style die rolls:  " << dnd() << '\n';
cout << "Largest of " << std.get_count()
    << " standard die rolls:  " << std() << '\n';
```

As we can see, we have multiple access paths to preserved state in two sets of state simultaneously. A rousing success! (The two other sets of state show preservation but not really anything else at this point in the code.)

### 11.10.3 But Can't a Plain Function?

So now let's try to use a plain function and see how that goes, shall we?

Let's try to do the largest-finding scraper we'd done first with the function object approach. That'll be a minor challenge compared to a producer like dice rolling.

### 11.10.3.1   Round One

Here's a first attempt:

```
inline long largest(long next = 0)
{
    static long max_so_far = next;
    return max_so_far = next > max_so_far ? next : max_so_far;
}
```

Remember that a `static` local will only be initialized on the first call to that function, so that we will store `next` directly only that first time. Thereafter, the `?:` will naturally test things before updating.

An application could now call `largest()` to retrieve the largest value seen thus-far — assuming that 0 was less than the current largest state, of course. Or they could call `largest(new_value)` to test and set a new value into the largest state.

But there is currently no way to reset the largest state for a new sequence of values.

### 11.10.3.2   Allowing Reset

Since the state within this algorithm continues to grow bounded only by the data type itself, we need an altered approach. Let's try it this way:

```
inline long largest(long next = 0, bool reset = false)
{
    static long max_so_far = next;
    if ( reset )
    {
        max_so_far = next;
    }
    return max_so_far = next > max_so_far ? next : max_so_far;
}
```

Now they can `reset` by passing a second parameter of `true`.

But our guess of 0 for retrieval is not general enough, as was hinted at before. If they may have sent a sequence of negative values and then calling `largest()` would actually disrupt their accumulation of the largest value by arbitrarily setting it to 0. We've gotta try again…

### 11.10.3.3   Fixing the All Negative Case

Let's beef up that `bool` parameter again with an enumeration:[15]

```
enum LargestAction { Evaluate, Reset, Retrieve };

inline long largest(long next = 0,
                    LargestAction act = Evaluate)
{
    static long max_so_far = next;
    if ( act == Reset )
    {
        max_so_far = next;
    }
```

---

[15]For more on enumerations see section 11.11.2 below!

```
        else if ( act == Retrieve )
        {
            next = max_so_far;
        }
        return max_so_far = next > max_so_far ? next : max_so_far;
    }
```

*whew* Now they can call `largest(new_value, Reset)` to reset the accumulation sequence or `largest(???, Retrieve)` to merely look at the current state — the `???` indicating that the value passed is arbitrary and will be ignored. Calling `largest(new_value)` will as usual evaluate the `new_value` into the state — i.e. test and update if necessary.

But having to pass a first parameter for `Retrieve` seems clunky at the least. Not cool...*sigh*

### 11.10.3.4  Less Clunky But...

Let's give this another shot with a major overhaul. I didn't want to, but I've had to bring out the big guns:

```
static long max_so_far = 0;

long largest(long next)
{
    return max_so_far = next > max_so_far ? next : max_so_far;
}

long largest(void)
{
    return max_so_far;
}

long largest_reset(long new_value = 0)
{
    swap(new_value, max_so_far);
    return new_value;
}
```

With a global variable marked `static`, only the file in which this code resides can use it. Thus, this would go in an implementation file and the functions' prototypes would go into an interface file.[16] We can't even `inline` these functions since they need to access this global variable which has to be in a separate compilation unit to hide it from other programmers and their nefarious purposes!

This nearly covers all the features of the `Largest` function object `class` we originally defined. But there is still one hurdle to overcome, and it's a doozy!

### 11.10.3.5  Multiple, Simultaneous Sequences of Data?

We still suffer the problem of only being able to handle a single sequence's state at once. The function object `class` can be instantiated multiple times — concurrently — so that our program has multiple largest calculations happening simultaneously.

To do something like this with the function [set], we'd have to have much more book-keeping such as a `vector` of maximums. Further, we'd have to relate to the caller an index position to tell each of us

---

[16]Also note that I would have named the last function just `reset` if I'd been able to use a custom `namespace` here. But I thought I'd hold off on that since we won't talk about it until section 12.3.

which maximum was being looked at, reset, or adjusted on this call. This would not only be tricky for us, but also for the caller as they'd have to track the indices for each of their sequences.

This tracking could also be foisted off on the caller, of course. At first this option seems in poor taste, but it actually leads to cleaner design. They know the purpose of their concurrent streams and can, for instance, make an enumeration or even use a container with both string names and maximums in each position so that the maximums have a nicely associated name rather than have us assign them arbitrary numbers. It will make our caller's code bulkier, but more readable at the same time!

Either way, they'll have to set a maximum state into the function with `largest_reset`, update with any number of `largest(new_value)` calls available at this time, then retrieve and put into their container of maximums this state with `largest()`. And they'll have to do that for each maximum they want — that is for each sequence they need to track a maximum on — every time throughout the program!

### 11.10.4   In Summation

I think the winner is clear: the function object approach. It easily handles not only the state preservation issue for processing a list or sequence and allows for multiple access paths to that state, but it can be super easily made to handle multiple, simultaneous sequences in a single program without lots of bookkeeping mumbo jumbo.

And, not to foreshadow anything, but when we get to templates later (chapter 14), we'll start to see the true power of this technique!

## 11.11   operators for Types Other Than classes

There are three other ways to form new data types in C++ that can be used in operator overloading: structs, unions, and enumerations. It does not, sadly, work on a simple typedef or using alias.

### 11.11.1   structs and unions

It is possible to overload operators for these kinds of types. But I say so with a couple of warnings…

As mentioned in the previous volume, due to their by default public nature, structs are only typically used when we have complete control over the use of those public members. We used it there, for instance, to house elements for private vector members so that we could remove a parallel vector situation. We kept the struct usage internal to a class and never used it to interface with the other programmers in the program.

A union, if you've never heard of it, is like a struct where all the listed members overlap one another in a single memory space big enough for the largest member. So only one value is stored in the space at a time and we have to take care which one is valid for any particular instance of the union. This kind of thing is used sometimes in language translation.

union is even less used in modern designs than structs. Instead we focus on the use of the variant type introduced to the standard library in C++17. This is a type-safe way to store one of a set of possible types in a single object.

### 11.11.2   enumerations

It is also possible to overload operators for enumerations. Perhaps a slight discussion of this type creation mechanism is in order before we talk to overloads for it. I'll stick to basic ones here, but there are now other variations on this theme from recent standards releases.

### 11.11.2.1   Enumer-what?

An enumeration is a way to create a set of constants under a single type name. They are a subset of integers and we rarely care what their values are — although we can control them if we wish.

For instance, if we wanted to create a set of constants to represent the days of the week, we might do so like this:

```
enum WeekDays { THURSDAY, FRIDAY, SATURDAY, SUNDAY, MONDAY,
                TUESDAY, WEDNESDAY };
```

Here THURSDAY would take on the value 0 by default and each constant after that would increment by 1 so that WEDNESDAY would have the value 6.[17] All of these constants exist under the auspices of the type WeekDays and so could be stored in a variable of this type or passed to a function of this type and so on.

An advantage to this method of constant creation over just typing them up yourself is that they work nicely with switches. If you use a WeekDays-typed expression in the head of a switch, the compiler will check the cases and make sure that you didn't miss any of the possible values for this type.

If, on the other hand, you wanted sequential values starting somewhere else, like the months of the year, you could do something like this:

```
enum MonthNums { January = 1, February, March, April, May, June,
                 July, August, September, October, November,
                 December };
```

Here January is given the value of 1 explicitly and each constant thereafter is incremented by one more than the one before it such that December has the value of 12.

In an extreme case, you can give separate values to each of the constants like this:

```
enum MonthDays { Jan_days = 31, Feb_days = 28, Mar_days = 31,
                 Apr_days = 30, May_days = 31, Jun_days = 30,
                 Jul_days = 31, Aug_days = 31, Sep_days = 30,
                 Oct_days = 31, Nov_days = 30, Dec_days = 31 };
```

Don't worry, we'll take care of leap years in an if later. *smile*

You can also use specially valued enumerations for bit values. We'll see this in the appendix on bit manipulations (G).

### 11.11.2.2   Overloading operators for Them

Let's take as a quick example the case of simple traffic lights. These take on the possible colors green, yellow, and red. We could define this idea as an enumeration like so:

```
enum TrafficLight { Red, Yellow, Green };
```

Now that we have these, we might want to display them for the user. But if we just display Red on cout, for instance, we'll see 0 instead of the word Red or red. So we should overload an operator<< for this type like so:

---

[17]Why I set them up in this particular rotation is left as an exploration for the intrepid reader. I suggest you look into the Doomsday Algorithm of date repute.

```cpp
inline ostream & operator<<(ostream & ostr, const TrafficLight & t)
{
    const string trafficLightNames[3] = { "red", "yellow", "green" };
    return ostr << trafficLightNames[t];
}
```

Here we see an array of `string` objects paralleling the `enum`eration values and being subscripted by the `TrafficLight` argument itself. This automatic conversion to `size_t` is simple coercion and comes naturally to the compiler — no typecast `operator` required.

Another situation might arise as well. What if we are moving from one state of the traffic light to another and want to do this conveniently with ++? This would be troublesome if we let the compiler work it out as `Red` would be followed by `Yellow` instead of `Green`. Worse, `Green` would be followed by 3 instead of any true `TrafficLight` color!

Let's fix this with the following overloads:

```cpp
inline TrafficLight & operator++(TrafficLight & t)
{
    switch (t)
    {
        case Red:    t = Green;    break;
        case Yellow: t = Red;      break;
        case Green:  t = Yellow;   break;
    }
    return t;
}

inline TrafficLight operator++(TrafficLight & t, int)
{
    TrafficLight x = t;
    ++t;
    return x;
}
```

Quick quiz: which is pre and which is post? *smile*

A short driver to test this implementation might look like:

```cpp
TrafficLight t = Green;
cout << t << '\n';
cout << "vs.\n";
cout << static_cast<short>(t) << '\n';
cout << "\nAnd let's not forget that ++(++" << t << ") is "
     << ++(++t) << '\n';
```

## 11.12   Wrap Up

In this chapter we've learned about the idea of overloading `operator`s for a programmer-defined type. Such types include not only `class`es but `struct`ures, `union`s, and even `enum`erations.

We learned both general and special patterns for overloading various `operator`s and even what some new-to-us `operator`s do in C++.

Further, we've explored the fascinating topic of function objects. It might not seem as glorious now as it will in chapter 14 on templates, but we're getting there!

Don't forget to check out more on operators in the appendices G and I on bit manipulation and more advanced operators, respectively.

# Chapter 12

# Other Tools

This chapter deals with some tools that will help you with design, debugging, and finding elegant solutions to programming conundrums. Let's start with debugging, though, and work from there!

## 12.1 Assertions

Assertions come in two approaches from the two languages in our purview: C and C++. C used assertions at run-time to find problems as they were occurring. C++ decided to add the ability to use assertions to find problems before they occurred — during compile-time!

### 12.1.1 C: At Run-Time

Note that we already spoke in depth to the use of the `cassert` library's `assert` technique for debugging help at run-time in the previous volume. Please see there for more on this technique.

### 12.1.2 C++: At Compile-Time

C++11 added the compile-time assert capability called `static_assert`. This tool complements the C-style `assert` macro by allowing programmers to test things that are known at compile-time early in the system development rather than at run-time when it might fall on the user to see the problem.

It has two forms, either just `static_assert(bool_expr)` or `static_assert(bool_expr, "string")`. If using the string variant, the string becomes the error message when the assertion fails.

#### 12.1.2.1 Scope of a static_assert

Unlike the C-style `assert`, which goes inside functions, the C++ `static_assert` can go even at global scope, `namespace` scope, or even `class` scope, as well. This can be useful to test things like the number

of bits in a data type to make sure the plans you have for the system are acceptable or if they need to be changed:

```
static_assert(sizeof(int) == 4);
```

(Of course, if you want a 32-bit integer, why not just go with `int32_t`? This and other similar types can be found in the `cstdint` library header.)

Or you can use it to test if a library has the proper version downloaded:

```
static_assert(LibNameSpace::Version >= 3);
```

This, of course, depends on the library supplying a nice `constexpr` value called `Version` in their `namespace`. For more on `namespace`s, see below.

#### 12.1.2.2    Beyond Standard Tests

The idea of type traits is pervasive in `template` metaprogramming, but we won't talk to that until much later (the end of chapter 14). But we can start small with the `numeric_limits` Boolean property `is_integer`. We can use this to test if a type that's been defined by a `template typename` or by `auto` is of integral type or not:

```
auto var{some_expr};
static_assert(numeric_limits<decltype(var)>::is_integer);
```

Here I've also used the new C++11 tool `decltype` that tells us what type a program entity or expression has without compiling it or executing it. This type is usable in declaring further variables, constants, argument, or `return` types.

## 12.2    exceptions

An `exception` is there to handle the exceptional case. That is, the case that should never happen but we coded to look out for it anyway. The `throw`ing of `exceptions` is more of an art form still than a science. There are lots of blogs and FAQs around the web that will tell you one thing or another. I'll try to distill the best of this advice here for you.

### 12.2.1    When to throw

You `throw` an `exception` when something bad happened that the local function can't really do much more with. This could be that you received an argument that can't be processed with (`invalid_argument`, `domain_error`, or even `out_of_range`). It could be that something happened when you were processing that can't be handled (`range_error` or `overflow_error`).

Whatever it is, you simply place the `exception class` you've selected in the `throw` statement inside your `if` that detected the problem. Make sure to give a descriptive string in the argument to the constructor:

```
if ( /* something went wrong */ )
{
    throw out_of_range{"function name -- details"};
}
```

And don't forget that there are consequences to `throw`ing an `exception` such as stack unraveling. . .

### 12.2.1.1 Stack Unraveling

When an `exception` is `throw`n, the function call stack will unravel to where an appropriate `catch` clause can be found. That is, function after function will be tossed off the stack and all their progress laid to waste until a function where a `try`/`catch` group with a `catch` that matches the `throw`n exception is found.

If this sounds expensive, it is. It is not something to be done lightly. Don't just `throw` an `exception` because you think it's cool to do so. Make sure you really can't handle the problem locally or in some less intrusive way first.

## 12.2.2 trying to catch

We've talked about this previously (in the earlier volume). But if you run code that might be `throw`ing an `exception` at you, it is important that you surround that code with a `try` block. This sets you up so you can actually `catch` the `throw`n exception if one comes.

The `catch` block comes after the `try` block and lists at least the type of the `exception` that you expected to be `throw`n at you. This should be listed in the documentation for the function. See, for instance, the listings for functions like string::at at a site like cppreference.com. It has a clearly marked section called "Exceptions" that tells which `exception` type is `throw`n and under what conditions it might occur.

### 12.2.2.1 catching by Name

If you want to `catch` the `exception` with a name, you can do so by either value or reference — even `const&`. The value is frowned upon as it will call the copy constructor and that might further `throw` if it is a memory situation that is the problem. But, once you give the caught `exception` a name, you can use the `what` method on it to recover that string that the `exception` was constructed with back at the `throw` site:

```
try
{
    // ... use string::at ...
}
catch (const out_of_range & ex)
{
    cout << "Function said:  " << ex.what() << '\n';
}
```

But that isn't much better than letting the program crash from not `catch`ing the `exception` in the first place. At least your program isn't dead. But the user is no more enlightened, so...

### 12.2.2.2 Multiple catch Blocks

If you have several possible `exceptions` that might be `throw`n in a `try` block, you can `catch` them one at a time with sequenced `catch` blocks all following the initial `try` block.

```
try
{
    // ... use string functions ...
}
catch (const length_error & ex)
{
    cout << "Function said:  " << ex.what() << '\n';
```

```
    }
    catch (const out_of_range & ex)
    {
        cout << "Function said:  " << ex.what() << '\n';
    }
```

There is even a catch-all syntax! If you put ... inside the `catch` head, it will `catch` any `exception` regardless of type. This is generally a bad idea, so avoid it as it could do you more harm than good!

It is also important to order these `catch` blocks correctly so that they don't keep another related `exception` that you've coded for from being caught. The reason is that they follow the rules of inheritance (talked about later in chapter 13).

#### 12.2.2.2.1   Inheritance and catch Blocks

Although we haven't talked to this specifically yet, inheritance is a way for us to tell the compiler that two `class`es are related to one another.[1] And then, if we `catch` the primary of those `class`es before we `catch` the secondary, the secondary `class` will never be caught. All instances of the secondary `exception class` will be caught by the primary `catch` block. Be wary of this!

#### 12.2.2.3   Re-throwing

If you don't want to completely handle an `exception`, but do want to add your two cents to the mix, you are more than welcome to `catch` the `exception` and then re-`throw` it. To do so, just `catch` like normal and then end your `catch` block with the single statement:

```
    throw;
```

This will re-`throw` the same exact `exception` you caught for someone upstream in the function call flow to `catch` again.

This can be useful if you need to clean up something that you'd been doing before things get too out of hand — like closing a file to free up its file handle to the OS or deleting some dynamic memory to avoid a memory leak. That sort of thing. This might, in fact, be the only place to use the ... `catch` syntax.

## 12.3   namespace Management

We've been using `namespace`s since the first day of C++ — literally! Remember these?

```
#include <iostream>

using namespace std;

int main()
{
    cout << "\n\t\tWelcome "
            "to C++!!!\n\n";
    return 0;
}
```

```
#include <iostream>

int main()
{
    std::cout << "\n\t\tWelcome "
                "to C++!!!\n\n";
    return 0;
}
```

We had to use the `std namespace` to get anything to print!

But what is a `namespace`? Could we use one for our own code somehow? When would that be a good idea?

---

[1]More on this in chapter 13.

## 12.3.1 What's in a namespace?

As we learned so long ago, a `namespace` collects together many identifiers that belong together so that they don't clash with other names than may look identical but refer to something totally different.

For instance, what if you were programming a game and there was a function to activate a player's power called `pow`. If the `cmath` library weren't surrounded in the `namespace std` we could have trouble!

What kind of names can go in a `namespace` you might ask? Any kind! `const`ants, functions, `class`es and other data types, even variables if you really needed to — but that's borderline tacky, so let's just not. Heck, you can even put other `namespace`s in there!

Let's explore...

### 12.3.1.1 std versus Global

First, let's talk about the two `namespace`s that are given to us: `std` for the standard libraries and the global `namespace` to hold `main` and all our regular work. The committee works hard to make sure there are no name conflicts within and between these spaces — at least when they come straight out of the box. Since everything we write goes into the global `namespace`, they can't guarantee everything.

But, if we cause a conflict, we can use scope resolution to get around it. That is, we can use `std::` on a library symbol to force the compiler to use that name if we make one that looks just like it for some odd reason:

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main(void)
{
    short pow = 2*2*2; // 2 to the power 3

    cout << "I got " << pow << " and the library gives "
        << std::pow(2.0, 3.0) << ".\n";

    return 0;
}
```

But this kind of conflict was caused because we let the `std namespace` out of the bag in the first place! It was that `using` directive's fault!

### 12.3.1.2 using Directives

Let's try to understand the `using` directive better — at least such as it pertains to `namespace`s. First, its purpose in the above form is to tell the compiler that we will be using lots of names from a particular `namespace` and so we don't want to have to scope them all. This brings all those names from the specified `namespace` out and mixes them with the global ones for faster and easier lookup.

If there were just one or two names from the other `namespace` we wanted to use, we could instead use this form of a `using` directive:

```cpp
#include <iostream>
#include <cmath>
```

```
using std::cout;

int main(void)
{
    short pow = 2*2*2; // 2 to the power 3

    cout << "I got " << pow << " and the library gives "
         << std::pow(2.0, 3.0) << ".\n";

    return 0;
}
```

This brings just the specified name into the global `namespace` mix instead of all the names from the other space. This can be really handy when only a few names need to be used instead of five or more.[2]

So far, though, I've shown the `using` directive placed at global scope — outside the `main` function or any function. We've also seen this directive placed inside a function when using `inline` functions in a library interface file. This doesn't mix the brought in names with the global `namespace` but rather with the local scope names.

In fact, you can use this directive in any scope you like except `class` scope — or the like.[3] It will only mix at the current scope level and then disappear from view when that scope ends.

## 12.3.2   Writing Your Own

Writing your own `namespace`s is pretty simple. You just have to remember the basic tenet from designing libraries: keep it all to a single theme! Don't cluster a lot of stuff into a single `namespace` just because it's convenient or all by the same coder or whatever.

To make a `namespace`, just place the name of the space after the keyword `namespace` and enclose everything to be inside it in a pair of curly braces:

```
namespace ThematicName
{
    // stuff to be inside namespace
}
```

Note that unlike a `class` definition, a `namespace` doesn't end in a semi-colon.[4]

Anything can be placed inside a `namespace`, btw:

- type definitions (including `class`es and `using` aliases)
- variable declarations/definitions
- `const`ant definitions
- function prototypes
- function definitions (`inline` or not!)

### 12.3.2.1   Starting and Stopping

A `namespace` doesn't have to be unique to a single file or place in a file, either. You can place some things inside a `namespace` at line 20 in a certain file and then close that off at line 30. Then, later on at

---

[2]Just a guideline, of course, but it sounds good, doesn't it? *smile*

[3]Not `struct` scope, either, for instance.

[4]This is, of course, because you can't declare a variable there optionally like you can when defining a type like a `class` or `struct`.

line 70 you can reopen the exact same `namespace` and add more items to it. This isn't necessarily the way we always do it, but see below for more on designing with `namespace`s.

#### 12.3.2.1.1  Designing with namespaces

Actually designing with `namespace`s can be a hurdle initially. What is typically done is to enclose the elements given in a library's header (`.h` or `.hpp`) in the `namespace` and then enclose any non-`inline` function definitions in the same `namespace` within the library's implementation (`.cpp`).

#### 12.3.2.2  Anonymity

Some `namespace`s are left without a name! This is called making an anonymous `namespace`. What could such a beast be used for?!

It makes the symbols global but only within the current file. This can be used instead of a `static` global declaration, for instance.

The main difference here is that only variables, `const`ants, and functions can be marked `static` whereas a `namespace` can also include type definitions.

#### 12.3.2.3  Nesting

When I said you can put anything in a `namespace`, I wasn't kidding! You can even nest one `namespace` inside another:

```cpp
namespace A
{
    namespace B
    {
        // stuff for nested namespace A::B
    }
}
```

If using C++17, this can be shortened to just:

```cpp
namespace A::B
{
    // stuff for nested namespace A::B
}
```

This can be done for organizational purposes like when a technical report feature is added before ratification. In many library implementations we had `std::tr1` well before technical report 1 was finalized and brought into the `std` `namespace` outright.

#### 12.3.2.3.1  inline

If you want a group of symbols to be collected together for organizational purposes but don't want to force anyone to have to place `using` declarations all about to use them, you can `inline` your `namespace`. This will mix the `namespace` content with that of the surrounding space for lookup purposes.

One nice place to use this is for library versioning. You can place the entire library in one `namespace` for general logistics and then place the current version of the library in an `inline`, nested `namespace` within that.

If using at least C++20, you can even do this more efficiently like so:

```
namespace A:: inline B
{
    // stuff for inline, nested namespace A::B (also just A)
}
```

### 12.3.2.4 Aliasing

If the name of a `namespace` gets too hairy to type — either by initial design or by nesting, you can use a `namespace` alias to give it a shorter name. The syntax is like a `using` alias but with the `namespace` keyword instead:

```
namespace new_name = original_namespace_name;
```

Here the `original_namespace_name` can include any scope qualifiers necessary to refer to a global or nested `namespace` as well.

## 12.4 string_views

We have two ways to represent strings of information in our programs now: the `string class` and the C-style string stored in a C-style array with a `'\0'` at the end. While the `string class` can take in a C-string, this copying is expensive in both time and space. So sometimes we might be wasting time writing overloads of a function to handle strings of information in two different forms. To simplify this and get rid of the excessive overloading, the C++ committee accepted the `string_views` proposal into C++17.

### 12.4.1 Compatibility

The `string_view class` provides a way to view a part of a string of information from afar, as it were. Often used on function parameters when a function needs to look at/examine but not change a portion of a string, it can represent a substring within either a `string class` object or within a C-string as necessary.

It does this by holding a pointer to the beginning of the view and a length/count of how many characters are in the view. This is sufficient and significantly cheaper to copy to a function than a whole `string class` object.

### 12.4.2 Utility

The `string_view class` provides much of the `const` utility of the `string class` such as iteration, element access, `length` determination, `substr` access, comparison, and `finding` functions. This means that a `string class` object passed to a `string_view` parameter won't suffer loss of functionality within the function just as if it were passed as `const&`.

In addition, when passing a C-string to a `string_view` parameter, the C-string data will suddenly gain these capabilities of the `string class`. While they are available in the form of `cstring` library functions, that requires the duplication of overloading. Having just the one function taking a `string_view` instead of one each for `string class` and C-string is far preferred.

### 12.4.3 In Action

Let's look at a quick example of a `string_view` processing function.

```cpp
inline void formatted_display(string_view s)
{
    for ( auto c : s )
    {
        cout << '\t' << c << '\n';
    }
    return;
}
```

And then this can be called from elsewhere in the program like so:

```cpp
formatted_display("apples");

string product{"Apple Macbook Pro"};
formatted_display(product.substr(6,7));
```

In each case the `string_view` parameter is initialized with an appropriate pointer and count for the initial string of characters. It's on par for the `string` but a step up for the C-string which could not normally use a range-based `for` loop!

### 12.4.4   And Also

And there are a couple of extra functions available in the `string_view` `class` as well: `remove_prefix` and `remove_suffix`. These advance the pointer or decrement the count respectively and so are quite efficient at their jobs. And they don't disturb the underlying data and so are still part of a constant view into the other string source.

## 12.5   Lambda Expressions

Lambda expressions are like tiny functions. We generally use them to pass as arguments to functions like `find_if` or `for_each` from the `algorithm` library. Such functions take a function and use it on each element in a container to either process those elements or identify properties of those elements. We'll talk about how to write such functions in chapter 14 on `template`s.

But, back to lambdas, let's look at various features of these tiny functions. And even how they can be as powerful as the function objects we discussed during `operator` overloading in chapter 11.

### 12.5.1   Basics

The basic syntax of a lambda is a pair of square brackets followed by a pair of parentheses where you list any parameters the lambda takes followed by a pair of curly braces where statements go to do the lambda's job. We'll start with the arguments and work our way around.

#### 12.5.1.1   Arguments and returns

The arguments to a lambda look like normal function arguments. They can be by value or by reference or even constant reference as necessary/desired. For instance, we could have the following situation:

```cpp
vector<string> words;
for_each(words.cbegin(), words.cend(),
         [](const string & w){ cout << w << '\n'; });
```

This would take each `string` from the `vector` and display it on a line by itself. The `string` comes into the lambda by `const&` to protect it from change and avoid copying as usual. Notice that `for_each` takes an `iterator` range instead of the `vector` straight up. This makes it more general and allows us to pass pointers as well as `iterators` thanks to the `template` mechanism.

Similarly, `find_if` can be used to identify elements of a container that meet certain criteria:

```cpp
vector<Date> birthdays;
vector<Date>::const_iterator match;
match = find_if(birthdays.cbegin(), birthdays.cend(),
                [](const Date & d){ return d.get_year() == 1970; });
if ( match != birthdays.cend() )
{
    cout << "Found birthday in same year as computer epoch!  " << *match
        << '\n';
}
```

Notice that the `return` type of the lambda is deduced automatically by the compiler from the `return` statement if any. If you want/need to, though, you can explicitly state it by using this syntax:

```cpp
vector<Date> birthdays;
vector<Date>::const_iterator match;
match = find_if(birthdays.cbegin(), birthdays.cend(),
                [](const Date & d)->bool{ return d.get_year() == 1970; });
if ( match != birthdays.cend() )
{
    cout << "Found birthday in same year as computer epoch!  " << *match
        << '\n';
}
```

This use of the arrow is odd and so is its placement, but you've gotta follow the rules, right?

### 12.5.1.2   Captures

What if, though, the year wasn't set by the application but entered by the user? How do we get our information into the lambda along with its parameter? We have to capture it! This is where the square brackets come in. Inside them you can list any local variables you'd like to use within the lambda comma separated and they will be available to the lambda throughout its lifetime. For instance, we could do this:

```cpp
vector<Date> birthdays;
vector<Date>::const_iterator match;
short target_year;                        // entered by user

match = find_if(birthdays.cbegin(), birthdays.cend(),
                [target_year](const Date & d)
                    { return d.get_year() == target_year; });
if ( match != birthdays.cend() )
{
    cout << "Found birthday in year " << target_year << ":  " << *match
        << '\n';
}
```

And we can even capture by reference to make changes to things:

```
vector<string> words;
vector<string>::size_type count{0};

for_each(words.cbegin(), words.cend(),
         [&count](const string & w){ cout << w << '\n'; ++count; });

cout << "Printed " << count << " words!\n";
```

Okay, that was a silly use — we could have asked `words` for its `size`. But it proves the concept, so...

You can also capture all local variables by value with `[=]` or all of them by reference with `[&]`. This isn't usually necessary and we prefer to list specific variables and types of capture.

If you'd like, you can call the captured variable by a different name within the lambda with a fairly simple syntax as well:

```
vector<string> words;
vector<string>::size_type word_count{0};

for_each(words.cbegin(), words.cend(),
         [&count = word_count](const string & w)
            { cout << w << '\n'; ++count; });

cout << "Printed " << word_count << " words!\n";
```

We can also capture by constant reference since C++14 with the syntax: `[&x=as_const(x)]`. Or with renaming: `[&y=as_const(x)]`.

### 12.5.1.3 Throw-Away versus Multi-Use

If you feel the need to use a lambda more than once, you can store it in a variable. The key to this working easiest is to use `auto` for the type:

```
auto predicate = [](const string & s){ return s.length() >= 5; };
```

Note the extra semi-colon after the lambda body to end the variable declaration and initialization. Now we can use predicate to pass to, say, `find_if` but not just for a single container, but over and over.

Is this better than writing an `inline` function? If you aren't capturing anything, it is probably a draw. But if you are capturing something, it is a definite improvement. More on this below...

### 12.5.1.4 auto Parameters

For some extra power, you can even mark the type of parameters as `auto` and the compiler will figure out what they are as the lambda is called each time. So, for instance, you could do the following:

```
double sum = 0;

auto totaler = [&sum](auto v){ sum += v; };

vector<double> numbs = { 4.2, 9.8, 12.4, 7.6, 18 };

for_each(numbs.cbegin(), numbs.cend(), totaler);
```

```
cout << "\nTotal is:   " << sum << ".\n";

cout << "\nResetting lambda total...\n\n";
sum = 0;

vector<long> numbs2 = { 12, 18, 42, 94, 62, 42 };

for_each(numbs2.cbegin(), numbs2.cend(), totaler);
cout << "\nTotal is:   " << sum << ".\n";
```

Here the `totaler` lambda is used first to add many `double` values and then to add some `long` values.

### 12.5.2   Mini Function Objects

The examples above with captures bring to mind the power of function objects to remember state and update it along the way. This is, indeed, true. So you might be able to get away with a lambda where a function object would have otherwise been the best deal. But note that the lambda mechanism cannot do multiple, concurrent states. Only the function object tool achieves this goal!

## 12.6   Wrap Up

In this chapter, we worked with `assertions` to debug program issues more quickly.  Then we switched gears and used `exceptions` to report errors at run-time effectively.

Next up was managing code with the idea of `namespace`s.  And finally we learned about the quick tool lambdas to make function and even function object-like entities for those many throw-away situations where crafting a nice function or function object `class` would be too time-consuming.

Hopefully you can use these tools in good stead in future projects.

# Part VI
# Polymorphism

# Chapter 13

# Run–Time Polymorphism

We've used polymorphism in the form of overloading and defaulting arguments for some time now. But run-time polymorphism is different in that it waits to take effect until the program is actually run by the user. The other two were handled by the compiler when the program was initially being put into binary form.

Run-time polymorphism requires the use of a tool called inheritance — so we'll explore the basics of this first. After that we'll look at run-time polymorphism itself. Then we'll wrap up the chapter with a look at more advanced features of inheritance.

## 13.1 Basics of Inheritance

Inheritance is a means to reuse code from old `class`es in new `class`es.

Well, that's a little shallow of a view, I suppose. But it is a useful view. And it is a place to start!

### 13.1.1 Concepts

When one `class` chooses to inherit from another for code reuse, it is known as the child `class` and the old one is the parent `class`. We use these biological terms because the idea comes from biological inheritance, of course. When you were born, after all, you were a bit your mom and a bit your dad. Now that you've grown, though, you've become more than that — and you've improved on the base you were given, of course. *smile*

Although programming inheritance does allow for multiple parent `class`es for a child `class`, we'll start with single parent examples. The child `class` chooses a `class` to be its parent because it it wishes to build upon or mould that `class`' behaviors and/or attributes. (Behaviors are methods (functions) and

attributes are data members (member variables), of course. More on the building and moulding later.) The parent doesn't even know there is a child `class`. But the child has seen its parent in action and knows that those are acts it can use as a basis for its own. (Okay, the designer of the child `class` saw the parent's code or usage and made the decision. Six of one, half-a-dozen of another. . . )

For instance, a color printer driver would probably start from a standard printer driver and then add the capabilities dealing with color. Most of the standard printer driver's behaviors of sending informa-tion down the wire and such don't need to be changed. Just enhanced with color information where appropriate/necessary.

### 13.1.1.1  Vocabulary

In terms of vocabulary, we will sometimes call the parent `class` the base, ancestor, or super `class`. Likewise we'll call the child `class` the derived, descendant, or sub `class`. The last of each list of names comes from set theory. The first of each trio is from the computer programming usage of an existing code base and deriving new code from it. The middle terms are just a formalizing of parent and child.

### 13.1.1.2  Reasons

As the child specializes the parent's provided code to its needs, it describes a finer subset of the objects its parent describes. Adding detail leads to a smaller set of possible objects, but this specificity was what the child was after. (We don't write color printer drivers for use on black-and-white laser printers, now do we?)

This follows the model of a typical classification system like from biology or geology or such. Think of the overall group animals — quite large but very vaguely described. At each level of the classification system — Animalia, Mammalia, ..., Canis, lupis — we add descriptive requirements that narrow the focus of each group. This makes each group smaller than the previous groups — a subset of it. But the number of descriptors has increased. So the more accurate/detailed your description, the fewer things in the resulting group.

### 13.1.1.3  Design Perspectives

From a design perspective, the main question facing us is composition or inheritance? That is, should you design as a `class` which contains its data or which derives from its data?

To help you decide, let's look at a classic example of the problem: point, circle, and cylinder.

Is a circle an object that has a point at its center or is it a point that has been extended out thru a radius?

Is a cylinder an object that has a pair of similar circles at each end or is it a circle that has been extended out thru a height?

This classic conundrum is one that has plagued programmers since the mid-late 60s. It isn't an easy argument and no clear winner has yet been found. The debate rages on.[1]

But how does it relate to composition and inheritance? Composition is a design by ownership: one `class` is composed of or owns an instance of another `class`. Inheritance is a design by specialization: one `class` is a more specific kind of thing than a previous `class`.

Therefore viewing a circle as an object that "has a" point at its center or a cylinder as an object that "has a" circle at each end is using a composition approach. Versus saying that a circle "is a" point extended thru a radius or a cylinder "is a" circle extended thru a height which is clearly inheritance.

In fact, software engineering uses these two catch phrases — "has a" and "is a" — to describe this predicament of composition vs. inheritance.

---

[1]Different implementations have been tried and we've found that each view works out better in different application circumstances.

## 13.1.2   Syntax

Theory is all well and good — and it's good for you to know — don't get me wrong! But I know you all want to get to the syntax: how do we do this in code? So here it is:

```cpp
class base
{
    double a;

public:
    base(double val = 0.0)
        : a{val}
    {
    }
    void base_only(ostream & os) const;
    void print(ostream & os) const;
};

class derived : public base
{
    long a;

public:
    derived(long val = 0L)
        : base{static_cast<double>(val)+1.2}, a{val}
    {
    }
    void print(ostream & os) const;
};
```

We start with a parent `class` named cleverly `base`. This has a member `a` of type `double`, a constructor, and two printing methods. Then we have the child `class` named `derived`. This `class` adds to the parent's goodness a new member variable named `a` of type `long`, its own constructor, and a printing method that looks suspiciously like one of its parent's methods.

Note the syntax with which the derived `class` makes its inheritance known to the compiler. We use a colon followed by the keyword `public` and the name of the parent `class`. This all follows the name of the child atop the `class` definition. There are several modes of inheritance, but we've chosen `public` here. This is not the default mode, but it is the most common. We'll see the other modes and their effects a little later (Section 13.3.3).

## 13.1.3   What Doesn't Come Along

Note also that, although we said in the introduction to this chapter that the child inherits everything from the parent, this is theoretically and is not exactly the case in C++. In C++, inheritance doesn't bring with it any constructor-like methods. This includes constructors and destructors. These have an intimate relationship with their `class` in assigning and cleaning up its members and won't go along for the ride.

Some people will tell you that the parent's `private` members are not inherited by the child `class`. This is not exactly true. The child `class` gets memory copies of data members from the parent that were `private`,[2] but cannot access any of the parent's `private` members directly. This makes accessors and mutators for parent members particularly important! However, since you cannot use the parent's `private` methods, you might as well not have inherited them, I suppose. *smile*

---

[2]That was all of them, right? You don't design with `public` data, do you? Good!

To make sure the parent's data members are properly initialized, we must call the parent's constructor or mutators to do so. If we take no action, the compiler will call the parent's default constructor on our behalf as a child object is constructed. But, if we want to construct our parent data members differently, we need to call a different parent constructor. It would be awkward and inefficient, after all, to let the compiler default construct the parent members and then turn around and mutate them in our constructor body!

To call a different parent constructor, then, we place the call in the member initialization list of the child constructor. An example of this is given above for `class derived` calling its parent's constructor with a non-default parameter.

## 13.1.4 What's It Look Like?

There are two ways to look at inheritance. One is via a memory diagram. This diagram shows what data members are inherited and where they reside in terms of which `class` brought them to the party. The other is an inheritance diagram which shows the organization of the `class`es involved from a base down to any derived `class`es.

### 13.1.4.1 Memory Diagram

The memory diagram to the side represents an object of the `class` type `derived` as coded above. It consists of two member rectangles and two containing boxes around them. The member rectangles are labeled with the name of the member and its type in parentheses. The containing boxes are simply labeled with the `class` type of that container in parentheses.

The `base` box is smaller as it contains just one member from that `class`. The box for `derived` is much larger as it contains not only its member variable rectangle but also the `base` box to represent its inherited memory.

The two containing boxes share an upper-left corner because they coincide in memory at the same address. This is, of course, because they are part of the same object. We'll see this more explicitly by running some of the upcoming code examples by printing the `this` pointer for our objects.
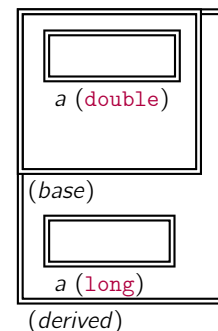


### 13.1.4.2 Inheritance Diagram

At right here you see another diagrammatic view of inheritance. This one is generically called an inheritance diagram or tree. So called because we modeled it after a family tree used to visualize human inheritance. The parent is placed at the top of the diagram — `base` here — and child `class`es are linked below this by lines sometimes called edges. These would spread out wider if there were more children immediately below the parent class. It would extend deeper if there were children of the top parent's children. We'll see examples of this shortly (section 13.1.5).
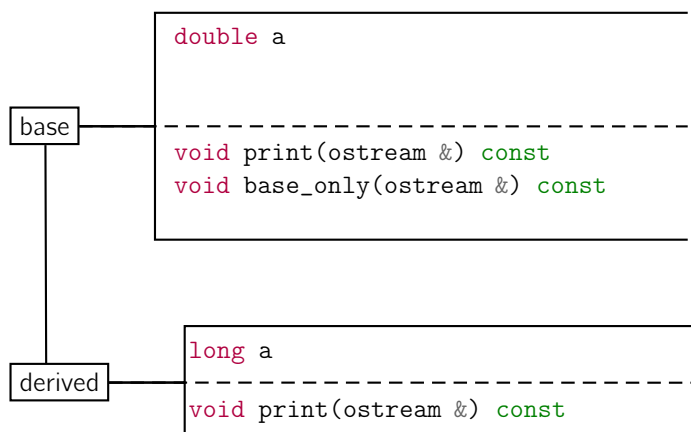


Optionally we mark off to the side the members of each class. Here we list the data members on top of a bracketed area and methods below a dotted line. This can get quite bulky so we don't always do it. But it is sometimes illustrative as it helps us visualize what members are inherited from above.

Some people like to put arrows on the edges connecting classes. I've not done that here because there is a bit of a tussle as to which end the arrowheads should be at. The most popular at this time is the UML-style diagrams which place the arrowheads at the parent showing that the child's methods and such are inherited from the parent. I'll leave it to you to decide which way the arrows should point in your diagrams.

### 13.1.5 Inheritance Hierarchies

Although the above is considered a minimal inheritance hierarchy, generally we talk about such a thing when there are 3 or more classes involved. For instance, we could give the class base above a second child like so:

```cpp
class derived_too : public base
{
    char a;

public:
    derived_too(char val = 'A')
        : base{static_cast<double>(val)-6.5}, a{val}
    {
    }
    void print(ostream & os) const;
};
```

In an inheritance diagram, this would look like so:

```
        base
       /    \
  derived   derived_too
```
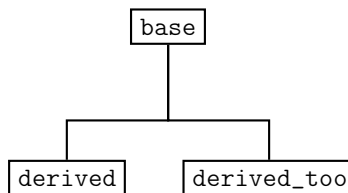
I've eschewed the sidebars on class members to keep it clean and readable here. Again, arrows are preferred, but I'm leaving the direction of them up to you. It could depend on the project leader for your team or on corporate mandates or who knows!

Another way to have a decent hierarchy would be to have the third class be a child of the first child like so:

```cpp
class derived_too : public derived
{
    char a;

public:
    derived_too(char val = 'A')
        : derived{static_cast<long>(val)+32}, a{val}
    {
    }
    void print(ostream & os) const;
};
```

In an inheritance diagram, this would look like so:

```
                    ┌──────┐
                    │ base │
                    └──────┘
                        │
                        │
                   ┌─────────┐
                   │ derived │
                   └─────────┘
                        │
                        │
               ┌─────────────┐
               │ derived_too │
               └─────────────┘
```
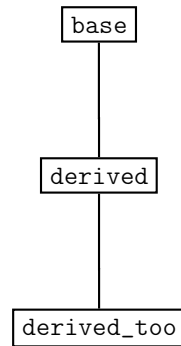
These two varieties can, of course, be combined to have almost any size and shape of hierarchy you can imagine. But, the larger the hierarchy, the harder it becomes to visualize for the human programmer. It becomes difficult to remember what capabilities you've received from parents and grandparents and so on, for instance.

It is also important to remember that a child chooses a parent in this system! This means the parents' don't know anything about their children or those `class`' members. And each child knows nothing about any siblings they might have, either!

## 13.1.6   Overriding and Hiding

When a child makes a method with the same name as a parent's method, there are two possibilities and two vocabulary terms we use: hiding and overriding.

When the signatures of the methods differ, we say that the child method hides the parent method. This makes it so that when a child tries to call the parent's version of the method, the compiler gives an error and says there isn't such a signature for that method name. Basically, from the compiler's perspective, it looks at the child class from the bottom of the hierarchy. With all functions having the same name aligned atop one another, the compiler can't see the parent's version through the child's.

We can get around this issue fairly easily, but the syntax is ugly. Let's say that our `derived` `class` from above had another `print` method with this signature: `void` `print(void) const`. This method would hide the parent's `print` method. We could still call the parent's method, however, but we have to add scope resolution to the method name in the call like so:

```
derived od;

od.base::print(cout);
```

Here we can call the `base` `print` method directly off a `derived` object. Without the scope resolution on the method name, the `derived` version of `print` would be called.

This brings to mind two other uses of things:

a) how does an overriding function use its ancestor function to help with the task if necessary?

b) there is another way to get through to a hidden function!

### 13.1.6.1   Implementing Overriding Functions

When you implement an overriding function that needs to call to its parent's version, you can't just call that function within your own. That would look like a call to yourself and this kind of unintentional recursion would be BAD! Instead, use the scope resolution notation to call your parent's method:

```
void derived::print(ostream & os) const
{
    os << "My long member is:  " << a << "\nAnd I inherited:\n";
    base::print(os);
    return;
}
```

You can't use the parent's a directly, either, — even with a scope resolution like `base::a` — because that variable is `private` to you as well!

### 13.1.6.2  Another Way to [Un]Hide

Another way for the `derived_too` `class` itself to unhide an ancestral method is to have a `using` declaration like so:

```
class derived : public base
{
    long a;

public:
    derived(long val = 0L)
        : base{static_cast<double>(val)+1.2}, a{val}
    {
    }
    void print(ostream & os) const;
    void print(void) const;
};

class derived_too : public derived
{
    char a;

public:
    derived_too(char val = 'A')
        : derived{static_cast<long>(val)+32}, a{val}
    {
    }
    void print(ostream & os) const;
    using derived::print;
};
```

This instructs the compiler that the parent's `print` method is to be visible to users of the current `class` as well. This makes it so that the caller doesn't have to do a scope resolution at all:

```
derived_too dto;

dto.print();
```

Awesome!

### 13.1.6.3  The Full Story on Overriding

What about the other `print` method that `derived` already has? Well, some would say that, since the signature is identical to that of the parent's method, the child method overrides the parent method. But

others — particularly those in C++ — still use the hides terminology.  I believe the reason for the schism is that most object-oriented programming languages are automatically polymorphic but C++ is not.  C++ allows you to choose polymorphism or plain inheritance.[3]  We'll see the effects of this in just a minute, in fact (section 13.1.7).  C++, you see, reserves the override terminology for polymorphic functions only.  But as the bulk of the OO community assumes polymorphic functions, they don't distinguish this possibility.

### 13.1.7  Weirdness

Going back to the earlier example:

```
derived od;

od.base::print(cout);
```

If you code up the functions for the above hierarchy, you can see the effects of the scope resolution on the `print` call on object `od` of `derived` type.  Let's say that the `base` method was coded as such:

```
void base::print(ostream & os) const
{
    os << "base type object has value " << a << ".\n";
    return;
}
```

Then, when we run the `do.base::print(cout)` call as above, this happens:

```
base type object has value 1.2.
```

So what happened to the rest of the data?  Where is the `long` member in this reporting?  Hold on!  One question at a time. . .

Normally, when `od` calls its own methods, its `this` pointer is a `derived*` typed pointer.  But when we did the scope operation on the `print` method to call specifically the `base` version, this turned `od`'s `this` into a `base*` version of itself.  The compiler had to do this because the `base` scoped function expects a calling object that is a `base`, after all.  How does this work?

Note how the `base*` pointer points to the address of the `derived` object in memory.  But it doesn't see the whole object!  It only sees the upper-left corner where the data inherited from `base` is located.  This restricted view allows any child `class` to be seen as a parent object.

The opposite is, however, not true and is disallowed by the compiler.  That is, you cannot look at a `base` object with a `derived*` pointer.  This would try to view more memory space than was actually allocated/initialized at that address.  This would be disastrous and so even the compiler says "No!" with a solid error.
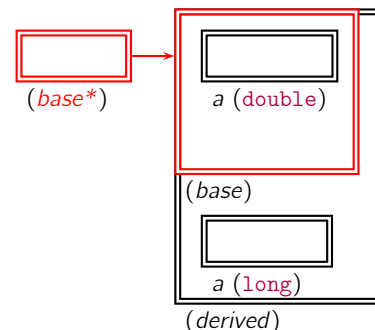


In fact, this issue won't just creep up when the compiler has to change the pointer type of `this` for a scope-resolved call.  It also happens at three other times.  See the below functions, for instance:

---

[3]The history behind this is that polymorphism used to be very computationally expensive.  Compiler writers have worked hard over the years, however, to streamline the process as much as possible.  Although still more expensive than a normal function call, a polymorphic function call is way faster than it used to be.

```
void app_print(const base & arg)
{
    arg.print(cout);
    return;
}

void app_print_two(base arg)
{
    arg.print(cout);
    return;
}

void app_print_three(const base * arg)
{
    arg->print(cout);
    return;
}
```

If called with the `od` `derived` object from before like so:

```
app_print(od);
app_print_two(od);
app_print_three(&od);
```

Then the `base` version of `print` will be called in all cases. This is because the arguments to all three functions believe themselves to be `base` objects or references/pointers to such. The only one that is correct is `app_print_two`, of course. It implicitly copies the `base` portion of the `od` object to a new `base` memory location. But with just inheritance, C++ can't tell. How can we fix this issue?!

## 13.2   Polymorphism

This section could equally well have been called **Overriding II — Overcoming the Weirdness**. But polymorphism is the more proper terminology. What is polymorphism? Well, as we learned in the first book in the series, polymorphism means literally "many forms". By this we mean that a function name can behave in different ways depending on the types/arguments involved with it. And we've seen examples of polymorphism in other aspects of C++ already: function and `operator` overloading, default arguments, and even basic `template`s from the earlier volume. But those are all handled at compile-time. This type of polymorphism that allows a function to have multiple forms based on an explicit inheritance relationship is also known as run-time polymorphism. This is because it doesn't take place during compilation like the others but while the program is being run by a user.

### 13.2.1   [Run-Time] Polymorphism in C++

We also mentioned above that many object-oriented languages are automatically polymorphic and C++ allows the programmer to choose such behavior. This is done by placing the keyword `virtual` on the prototype or `inline` definition of a function we would like to be polymorphic. For instance, let's say we had a `class` A that had two methods — f and g. Let's let f be a regular function and let's make g polymorphic:

```
class A
{
public:
```

```
        void f(void);
        virtual void g(void);
};
```

Nothing changes with respect to A `class` objects in the program at all. We call f and g normally and they act in no way differently here.

So what's the point? It's from the inheritance-based polymorphism. Let's take `class` B, for instance, that inherits from A:

```
class B : public A
{
public:
        void f(void);
        virtual void g(void);
};
```

Note that B also has f and g with the same call signatures as in the parent `class` (A). Now when we look at some calls to B's functions, we start to see the difference:

```
B b;
b.f();      // calls B::f
b.g();      // calls B::g
```

Oh, no, those are normally like that. I meant when we call B's functions with respect to a parent pointer or reference:

```
A & arb{b};
A * apb{&b};    // or we could dynamically allocate one with new B()

arb.f();       // calls A::f
apb->f();      // calls A::f
```

Wait, that still didn't do anything new! That's the same weirdness as before. Well, of course — that was the non-`virtual` function. Let's call g:

```
arb.g();       // calls B::g
apb->g();      // calls B::g
```

Now that's a horse of a different color! Suddenly, with that `virtual` keyword on it, calls to g remember whether the calling object — even when pointed or referred to — is of the parent or child type and calls the proper overridden method! This takes a lot of effort on the compiler's part, so it isn't necessarily something we do with all functions, but it is nice when we need it to work this way.

Again, this all works only via a parent pointer or reference and only when the keyword `virtual` is placed on the function.

### 13.2.1.1   What's in a Keyword

One other thing, once a function [signature] is made `virtual`, all children's overrides of this function are automatically `virtual`. The child `class` doesn't have to mark the function `virtual` again. The compiler (nor other programmers) won't be upset, however, if you continue to mark `virtual` further on in the hierarchy.

Also note that the `virtual` keyword doesn't have to appear on a non-`inline` definition of the function. The compiler remembers and hooks things up correctly anyway.

Another way to go is to use the keyword `override`. This is marked at the end of the function head instead of the beginning, however. This keyword also need not be repeated on non-`inline` definitions. (And it may be combined with the `virtual` mark if that rocks your boat!)

#### 13.2.1.2 The Destructor

And, BTW, of all the functions you could ever make `virtual`, the destructor is the most important one! If the destructor is not `virtual`, after all, the pointer we `delete` will just call the base `class` destructor which wouldn't clean up all the object's data were the pointer pointing to a descendant object rather than an actual base `class` object. With the destructor made `virtual`, however, the correct destructor is called and a lovely chain of destructor calls is automatically started which will follow our ancestry back up to the uber-parent. (This is akin to the chain of default constructor calls that would have happened had we not called appropriate constructors from our parentage in our member initialization lists for ourselves.)

What if you don't have any dynamic members? Who cares! Make an empty `virtual` destructor. If you care anything about your children or their children, you'll go to this minimal extra effort!

Okay, if you control the entire design of this hierarchy and you know that a certain `class` is terminal — at the bottom of the hierarchy — then you don't have to worry with it. But if you are just making a `class` in a library that someone using said library might derive from, please be considerate and make a `virtual` destructor.

#### 13.2.1.3 The VMT

The way this all works is a `virtual` method table (VMT). The basic idea is that there are entries in a 2D table where each row's first part indicates a type from the inheritance hierarchy and the last part indicates the address of a method in memory that is associated with that type. Such a table is only made with respect to `virtual` functions, of course, as it is a bit more expensive to have the program sift through this table for the right row to find the function address than to just fit the function address right into the code when it is called.

A full understanding of how this works is best left until you've taken assembly language and done a proper dispatch table.[4]

#### 13.2.1.4 A Full Example

On the companion website, you can find this example program which not only demonstrates the above points with runnable code, but further points out how the chain of constructors and destructors works as mentioned above. This is done by printing the `this` pointers of the objects in constructors and destructors. As noted in the comments, you can place this code side-by-side with your terminal and note which hexadecimal addresses correspond with which objects and trace what is happening there.

### 13.2.2 Polymorphic Dispatch

An interesting side-effect of `virtual` function processing is that a polymorphic function called from a non-polymorphic function still acts polymorphically! This is known as polymorphic dispatch for odd historical reasons.[5] I prepared an example, but it is a little long so I've placed it on the website here.

As you watch it run, note that the calls to `base_only` for the derived object `doo` are still polymorphic. There are other good notes in the comments as well. Be sure to read through those and maybe take some of the experimentation suggestions!

---

[4]Sometimes called a case table.

[5]I believe it goes back to something like the dispatcher for a taxi company or police department who does just the one job but sends out the proper person for the job and those drivers or officers are polymorphic as it were to adapt to the situation.

## 13.2.3 Abstract classes

When we inherit functions from a parent `class`, we effectively inherit a little language. It is a language that people who see a parent pointer or reference know will be spoken by the object at the end of that pointer. And if the functions are `virtual`, the objects will even be able to use the language appropriately to their current circumstances — rather than just mimicking what their parents always said.

Some `class`es even serve only a purpose of specifying the language all their children should speak. And if such a `class` has nothing specific to say about/in this language, it shouldn't introduce erroneous verbiage. By this I mean that we shouldn't put in a body to the function that does something incorrect or silly. Instead, it should make such `virtual` functions "pure". This way the language will be specified, but uncorrupted.

A pure `virtual` function is a `virtual` function with no definition. Not just a function prototyped to be defined later. Not just a function never called that we forgot to define. But a function which is planned to never be defined.

Why would you do this?! Isn't this insane? Well, not as insane as it sounds.

Take, for example the concept of a random distribution. This phrase brings to mind a set of values with corresponding probabilities. The primary functionality (spoken language) of such sets is generating (picking) a variate (value) from the set probabilistically. But, just knowing this doesn't give us a reasonable way to do this activity in a general way. We have to know more about the particular random distribution in order to generate a value from it. Is it uniform? Is it Poisson? Is it Weibull? More information will be necessary.

Similarly, all shapes have a concept of area: area under a curve, area of a 2D figure, surface area of a 3D figure, etc. But just knowing that something is a shape doesn't give us enough information to calculate its 'area'. I need to know what kind of shape it is in order to calculate the proper area.

A `class` having a pure `virtual` function, BTW, is known as an abstract `class`. A `class` which is abstract cannot be instantiated. That is, you cannot declare an object of an abstract `class` type. After all, if you did, what would the compiler do if you were to try to call that pure `virtual` function — the one without a definition! That would be all kinds of bad, right? So that isn't allowed — ever.

So an abstract `class` is just there to guarantee a language for all of its descendants as we mentioned before. How do you know they speak this language? Well, we can still point to or refer to them via the abstract `class` type. That is guaranteed by inheritance itself.

Oh, and any `class` that isn't abstract is sometimes also called concrete. If it is derived from an abstract `class`, it must have defined the pure `virtual` function in order to have become concrete.

I've prepared another long example. It is based on the random distribution idea above. As you look it over, you'll see that the `RandDist` `class` is abstract having the pure `virtual` function `generate`. In fact, that's almost all it has other than the `virtual` destructor — not even any member variables! That's because it doesn't know enough yet to need any data — it is just the concept of a random distribution. It just needs what it can describe to the compiler and that is that generating a random variate creates a `double` from nothing at all without harming the object doing the generating.

Note especially how the pure `virtual` function is designated: at the end of the function head an equal sign and a zero are appended. This `=0` syntax is the official C++ way to designate a function that will never be defined.

When you run it, you'll see that values are randomly generated in the proper way for the given [concrete] distribution type provided. Note also that the pointer used to `generate` variates doesn't have to be pointing to a dynamically allocated object. This is shown with the final two examples in main.

## 13.2.4   Polymorphic Containers

Let's build something from our last example that's a little bit more powerful. As you recall, we'd added the concept of an abstract base `class` to our repertoire allowing our entire hierarchy to speak a common language based on the [pure] `virtual` functions provided by the [abstract] base `class`.

But before, we just created and used individual objects of our children's types. Now let's go for something more ambitious.

We'll use code like this:

```cpp
const size_t MAX_DISTS = 10;
RandDist * dists[MAX_DISTS];
```

to create a polymorphic container of various `RandDist`-derived objects. This one is a C-style array for simplicity, but it could very well be an `array class` object or a `vector` as well.

This array can contain pointers to any object derived from `RandDist`. Its elements can't point to `RandDist` objects, of course, since that `class` is abstract. But any concrete descendants can be pointed to since the base type pointer can point to any derived type. This makes us able to 'contain' (indirectly) multiple [related] types of data!

How do we fill in this container? We would normally have the user choose which distribution they wanted to add from a menu, but let's keep it simple and just prove the concept. We'll place `new Uniform` objects in all even slots and `new DieRoll` objects in all odd slots like so:

```cpp
for ( size_t d = 0; d != MAX_DISTS; ++d )
{
    if ( d % 2 == 0 )
    {
        dists[d] = new(nothrow) Uniform;
    }
    else
    {
        dists[d] = new(nothrow) DieRoll;
    }
}
```

Then, when we go to display the results, we'll describe the variates `generated` with a simple `?:` operation:

```cpp
for ( size_t d = 0; d != MAX_DISTS; ++d )
{
    if ( dists[d] != nullptr )
    {
        make_em((d%2==0?"uniform":"dice"), dists[d]);
    }
}
```

(Recall the `make_em` function needs a description `string` to display above the `generated` variates.)

Why didn't I use a `?:` to fill the array in the first place? Well, that would have required some typecasting and we don't quite have the proper tool for that. We'll get to it in the next section, though, so stay tuned!

And let's not forget to clean up after ourselves:

```
for ( size_t d = 0; d != MAX_DISTS; ++d )
{
    done_with_it(dists[d]);
}
```

(Recall that the `done_with_it` function will `delete` each pointer and set it to `nullptr` for us as well.)

To save you having to code up these tweaks to the earlier program, I've posted it to the website here.

### 13.2.4.1  Enter dynamic_cast

The `dynamic_cast operator` will check if an actual object pointed to is of the type we are testing for. This looks something like so:

```
dynamic_cast<tested_for_type *>(pointer)
```

If the object pointed to is of the desired type, the address will be returned in the guise of a properly typed pointer. We can store this and use it to call functions more specific to the desired type.

In fact, 'is' here is quite general. To the compiler, a pointer is of the correct type if the true type is any descendant of the requested type. So when we store the new pointer, we can narrow our knowledge further and use a more limited language shared in a sub-tree of our hierarchy!

What if the actual pointer points to a different type? `nullptr` is returned instead of the original address.

How can we use this? Well, we can use it as a traditional typecast for dynamic objects like so:

```
for ( size_t d = 0; d != MAX_DISTS; ++d )
{
    dists[d] = d % 2 == 0
                ? dynamic_cast<RandDist*>(new(nothrow) Uniform)
                : dynamic_cast<RandDist*>(new(nothrow) DieRoll);
}
```

This greatly shortens the allocation of our objects from before. Also, technically only one of the pointers needs to be cast to `RandDist` here. Once that is done, the compiler realizes that the other half of the `?:` can be that type, too. But without the cast at all, the compiler is at a loss that the two seemingly unrelated types of `Uniform` and `DieRoll` can be made synonymous. I just put in both casts to make it look nicer.

But we don't have to keep it all regimented to even/odd positions anymore! With the testing facility of `dynamic_cast`, we can place the objects' pointers more randomly. Let's use a coin-flip to do it like so:

```
DieRoll flip(1,2);

// then inside the for loop:
    dists[d] = static_cast<short>(flip.generate()) == 1
                ? dynamic_cast<RandDist*>(new(nothrow) Uniform)
                : dynamic_cast<RandDist*>(new(nothrow) DieRoll);
```

Here we have to cast the `generate` result to `short` to account for the fact that all `generated` values are `double` but that type doesn't like to be exactly `==` anything. When the `flip` comes up a `1`, we'll store a `Uniform` pointer. And when it's a `2`, we'll store a `DieRoll` pointer.

But now, since we don't know where the various children objects are at in the array, we'll have to test for them with `dynamic_cast` when deciding the descriptions:

```cpp
for ( size_t d = 0; d != MAX_DISTS; ++d )
{
    if ( dists[d] != nullptr )
    {
        make_em((dynamic_cast<Uniform*>(dists[d]) != nullptr
                    ?"uniform":"dice"),
                dists[d]);
    }
}
```

Here we test for a `Uniform` object at the end of the `dists[d]` pointer and if it really was of that type, we describe with `"uniform"`. Otherwise we use `"dice"`.

This is still not the full power of `dynamic_cast` being used, but it is a good start. We'll see more on this in a couple of sections, too.

Oh, and here's the full example for you so you don't have to tediously make all those changes to the previous example.

### 13.2.4.2   But What About typeid?

The `typeid operator` can be used to access run-time type information.[6]  But this is available only if you *#include* the `typeinfo` library. This `operator` can identify any type in the entire program with a `type_info` structure object. In particular, it is typically used with its `==` `operator` to tell if the object at hand is of a particular type like so:

```cpp
if ( typeid(object) == typeid(double) )
{
    // object is really a double
}
```

This seems at first a little weird as wouldn't we already know something was a `double` or not? Yes, most of the time. But in a `template` (which you saw in the previous volume) or in a dynamically typed polymorphism situation (like we've been dealing with), it can make sense.

So, for instance, we could check if our `dists[d]` pointer from before actually pointed to a `Uniform` object with a bit of code like so:

```cpp
// ...in the for loop as before...
make_em((typeid(Uniform) == typeid(*dists[d])?"uniform":"dice"),
        dists[d]);
```

Some people are fond of using the `type_info`'s `name` method which returns a `string` representing the data type. But it is not just the name as you've typed it into your program. It has undergone a process called name mangling. This adds other information to the `string` that the compiler felt vital to know. For instance, on my compiler, the mangled `name` of the `DieRoll` `class` is `"7DieRoll"` and that of `Uniform` is `"7Uniform"`. But when I change the name of the `DieRoll` `class` to `Dice`, the `name` comes out as `"4Dice"`. It would seem my compiler wants to know how many characters are in the name of the `class` without having to count them..? Weird. . .

---

[6]The acronym RTTI stands for Run-Time Type Identification. . .

Anyway, here is an example on the website with the `typeid` implemented in the `for` loop instead of `dynamic_cast`. Notice that `dynamic_cast` is still used to initialize the polymorphic array elements in the first `for` loop. Be sure to play around with the currently commented out `.name()` variation to see what your compiler does with name mangling!

Although the `typeid` code is shorter than the `dynamic_cast` approach, it does require more heavy lifting behind the scenes. Because by having to *#include* the `typeinfo` library we've incurred a huge expense to gather and have available `type_info` data for all the data types used in your entire program — builtin types and all.

The `dynamic_cast` mechanism, on the other hand, is managed via the VMT mechanism that the compiler is already using to manage your polymorphic function calls. This is no more overhead and can lead to faster runs and smaller binary sizes. Also, as mentioned before, `dynamic_cast` has another property that `typeid` just can't replicate. More on that next...

### 13.2.4.3 When Not to be Polymorphic

But it occurs to us that the whole `dynamic_cast` versus `typeid` thing to determine the description could be avoided by just implementing a `string` member in the base `class`:

```cpp
class RandDist   // abstract [base] class
{
    const string desc;
public:
    virtual ~RandDist(void)
    {
    }

    RandDist(const string & d = "") : desc{d}
    {
    }

    const string describe(void) const
    {
        return desc;
    }

    virtual double generate(void) const = 0;   // pure virtual function
};
```

Now the programmer using any descendant of `RandDist` will be able to just ask it to `describe` itself. And it doesn't even need to be polymorphic! We just make sure the children pass their desired description `string` to their parent's constructor call like so:

```cpp
Uniform(double low = 0.0, double high = 1.0,
        const string & d = "uniform")
    : RandDist{d}, a{low}, b{high}
{
}
```

Here we've even let the programmer making the object name it differently if they so choose.

With this in place, our `make_em` becomes slightly different, of course:

```cpp
inline void make_em(const RandDist * p)
{
    cout << "Generating random " << p->describe() << " values:\n";
    for ( short i = 0; i != 9; ++i )
    {
        cout << p->generate() << ", ";
    }
    cout << p->generate() << '\n';
    return;
}
```

Now it doesn't need a `string` parameter and just calls the `describe` method to get the description for the values being generated. And, calling it is far simpler:

```cpp
make_em(dists[d]);
```

But why do we have all this polymorphism still hanging around? Well, we do still need the polymorphic behavior for `generate`, after all. What I'm saying with this change is that you need not rely on polymorphism for everything in your hierarchy. Sometimes simple solutions are best.

However, I also promised to finally show you that other use of `dynamic_cast`, so here that is. I added another function to the `DieRoll` class to retrieve the number of dice in the roll. Then I changed creating the child objects a little to change up how many dice were in each roll:

```cpp
DieRoll counts(1,10);  // from 1 to 10 dice in a roll

// ...down in the first for loop...
    dists[d] = static_cast<short>(flip.generate()) == 1
                ? dynamic_cast<RandDist*>(new(nothrow) Uniform(2.0,12.0))
                : dynamic_cast<RandDist*>(new(nothrow)
                                    DieRoll(static_cast<short>
                                        (counts.generate()),6));
```

Now the `DieRoll` children will all have different numbers of dice in them — anywhere from 1 to 10.

Finally, after the call to `make_em`, I placed this `if`:

```cpp
DieRoll * p = dynamic_cast<DieRoll*>(dists[d]);
if ( p != nullptr )
{
    count_sum += p->get_count();
    ++dice_count;
}
```

Here `count_sum` and `dice_count` are earlier initialized to zeros and are responsible for tallying the total dice in all rolls and the total number of `DieRoll` versus `Uniform` objects. Note that I can only call the `get_count` accessor via a `DieRoll` pointer. I could not have directly called `dists[d]->get_count()` because `dists[d]` is a `RandDist*` instead. The `dynamic_cast` not only vets the pointer as pointing to a particular descendant type, but also allows you to call type-specific member functions from that result.

I've placed the full example with a few extras and notes on the website for your enjoyment and further study.

## 13.3   Advanced Inheritance

We've talked at length about basic inheritance and how polymorphism helps to round out those sharp corners. But there are also more advanced topics in inheritance that might rear their ugly heads to your design process. In this section we'll look at those including inheriting from multiple parents, a new member access classification, non-`public` inheritance modes, and how `static class` members interact with inheritance.

### 13.3.1   Multiple Inheritance

Sometimes it becomes useful or some would even say necessary to inherit from multiple existing `class`es. For instance, a hovercraft might inherit from both land and air vehicles. Or a seaplane might inherit from both aircraft and boats.

Let's look at a basic example of this where a `derived` `class` inherits from two base `class`es at once:

```cpp
class base
{
    long a;
public:
    base(long A = 34L) : a{A}
    {
    }
    virtual ~base(void)
    {
    }
    void print(void) const
    {
        cout << "base:  " << a << '\n';
        return;
    }
};

class base_too
{
    long b;
public:
    base_too(long B = 43L) : b{B}
    {
    }
    virtual ~base_too(void)
    {
    }
    void print(void) const
    {
        cout << "base_too:  " << b << '\n';
        return;
    }
};

class derived : public base, public base_too
{
public:
```

```
        derived(long c) : base{c-12}, base_too{c+12}
        {
        }
        virtual ~derived(void)
        {
        }
};
```

Here we see that to inherit from multiple parent `class`es, we just comma-separate them after the inheritance colon. Also, we call both of their constructors in our member initialization list. Just watch that you call those constructors in the same order you inherited from those `class`es or you might suffer a warning about having to rearrange them!

#### 13.3.1.1   Overcoming Name Clashes

So, how can a `derived` `class` object call the two inherited `print` functions? They have the same signatures! It's as simple as using a scope-resolution override like we've done before:

```
derived d(6);
cout << "derived object contains:\n";
d.base::print();
cout << " & ";
d.base_too::print();
cout << '\n';
```

Here you'll get that the `derived` object contains a `base` value of `-6` and a `base_too` value of `18`.

#### 13.3.1.2   The Diamond Pattern

The only unfortunate thing about multiple inheritance is that sometimes you end up with a diamond pattern in your hierarchy. Often called the diamond of doom, it happens when two or more of a `class`' parents come from a common ancestor themselves. The simplest of these situations is modeled here:

```
class grandparent
{
    double g;
public:
    grandparent(double G = 12.43) : g{G}
    {
    }
    virtual ~grandparent(void)
    {
    }
    double get_g(void) const
    {
        return g;
    }
    void set_g(double G)
    {
        g = G;
        return;
    }
    void print(void) const
```

```cpp
    {
        cout << "grandparent:  " << g << '\n';
        return;
    }
};

class base : public grandparent
{
public:
    // reset unchanged
    void print(void) const
    {
        grandparent::print();
        cout << "base:  " << a << '\n';
        return;
    }
};

class base_too : public grandparent
{
public:
    // rest unchanged
    void print(void) const
    {
        grandparent::print();
        cout << "base_too:  " << b << '\n';
        return;
    }
};

// derived class unchanged
```

At first glance, it seems ridiculous and unnecessarily complex. But look, for instance at the input/output streams hierarchy for C++. It has a diamond pattern in it! Turns out that `istream` and `ostream` both derive from the common ancestor `ios`. Then, `iostream` derives from both `istream` and `ostream`! This can be seen in any number of diagrams on the web like this one from a stackOverflow discussion.

So what happens here? Let's examine:

```cpp
derived d(6);
cout << "derived object contains:\n";
d.base::print();
cout << " & ";
d.base_too::print();
cout << '\n';
d.base::set_g(d.base::get_g()-10);
cout << "derived object now contains:\n";
d.base::print();
cout << " & ";
d.base_too::print();
cout << '\n';
```

Running this fragment we see that the first display has both `base`'s `grandparent` value as `12.43` and `base_too`'s `grandparent` value as `12.43`. But in the second display, the `base grandparent` value has

changed to `2.43` while that from `base_too` is still `12.43`. This shows that, without some care, the two inheritance paths' common ancestor data can become out of sync.

We could, of course, handle this manually by always making the same change to both paths back-to-back. But this becomes tedious and is prone to error. A better way must be found!

To solve this problem, the C++ committee decided to reuse the `virtual` keyword in a unique way. And the solution comes in a unique place, as well, some feel. It is hardly a change at all for the `derived` `class` at the bottom of the diamond, but the middle-layer parents need to make a change to their inheritance modes from the common ancestor:

```cpp
class base : public virtual grandparent
{
    // rest unchanged
};
class base_too : virtual public grandparent
{
    // rest unchanged
};
```

Here we've added the `virtual` keyword to the inheritance mode `public`. Note that it can go on either side of the `public` and the compiler doesn't care. And this is all that is absolutely required to fix the problem. If we tweak this in our fragment of above, we'll find that both `base` and `base_too` have `grandparent` values of `2.43` now.

The compiler does this by forcing the two middle-layer parent's share their common ancestor when they are both inherited from at once. That's quite the effort for our benefit, so kudos to the compiler team!

But that relies on the shared `grandparent` portion being initialized twice: once by the `base` and then again by the `base_too` ctors. To alleviate this double effort and to clarify what the grandchild (`derived`) wants to have in its now solitary `grandparent` memory, we can also change the member initialization list of its constructor to call to the `grandparent` constructor. This is different than before because the compiler never let us call to a constructor two levels back before — only one level! But in the case of a `virtual`ized diamond pattern, the compiler allows it. It looks quite simply like this:

```cpp
derived(long c) : grandparent{c+43.2}, base{c-12}, base_too{c+12}
{
}
```

Note that the `grandparent` call must precede those of the parents or warnings may come forth!

You can find this full example on the website for further study, of course.[7]

### 13.3.1.3  How Not to Use Multiple Inheritance

Another way to avoid the diamond of doom is to just not use multiple inheritance at all. Instead, choose one parent and make yourself a data member of the other proposed parent's type. The problem with this solution is that you must take responsibility upon yourself to keep the common grandparent data synchronized. As mentioned above, this will be tedious and error-prone and is not the best choice.

---

[7]There is one change there: I `static_cast` the use of c to initialize the `grandparent` portion to `double` to avoid a warning from my compiler. Nothing significant, just didn't want you to be confused.

### 13.3.2   protected Membership

It turns out that a `class`' members don't have to be just `public` or `private`. There is a third access classification: `protected`. Members of a `class` whose access is marked `protected` are `private` from outside the inheritance hierarchy but `public` from inside the hierarchy. That is, they share some attributes with each of the other specifiers.

I wish I had a diagram to show how this worked, but the best one I've found is now lost in the web.[8] It had two neighboring houses with a single fence around the two yards. The houses themselves represented the `private` areas of two `class`es. The shared yard represented some `protected` members that the two `class`es had in common. And the fence kept the outside of the hierarchy from getting to the `protected` area. It was pretty slick. Hopefully my poor description can be enough for now. Maybe someday I'll learn to draw.

A stern warning: Do **NOT** make data members `protected`! Leave them `private`. You can have `protected` functions that provide easier or special access to your children, but not to those outside the family. But `protected` data members caused the Java designers to design a whole new GUI hierarchy because their first one could be corrupted and made ridiculous and/or ugly by misuse of `protected` data members.

### 13.3.3   Non-public Inheritance

Most inheritance is done in a `public` manner:

```
class Child : public Parent
```

But it is also possible to use other inheritance modes besides `public`. The two others are — perhaps not surprisingly — `private` and `protected`.

A `private` mode of inheritance makes all `public` or `protected` members of your parent `private` when viewed from your objects.

A `protected` mode of inheritance makes all `public` members of your parent `protected` when viewed from your objects.

In other words, the access to your parent's members is made more restrictive by the other two inheritance modes — when viewed from objects of your `class`.

Here is a table summarizing the changes:

| Inheritance Mode \ Parent Member Access | public | protected | private |
|---|---|---|---|
| public | public | protected | private* |
| protected | protected | protected | private* |
| private | private | private | private* |

Here the row labels represent the inheritance mode used and the column labels represent the access mode of parent members. The boxes inside the table, then, represent the access mode that appears to those outside the child `class` with respect to those members of the parent. The `private` access in the last column is starred because, of course, not even you can reach your parent's `private` members without using accessors or mutators — sort of uber-`private`.

This has limited applicability, of course. It is used fairly rarely. But we will see a good occasion to use it when discussing stacks and queues later on (chapter 18).

---

[8]I forgot to bookmark it or save the image and now I can't find it now matter what keywords I feed google!

## 13.3.4   static Members and Inheritance

One last nagging question involves `static` members of a `class` involved in inheritance. Clearly a child `class`' `static` members wouldn't affect the parent since the parent doesn't even know the child exists. But the other way around . . . how does the child `class` handle those members during inheritance?

One popular theory is that the child `class` will get copies of all the parent's `static` members and all child objects will share those copies separately from the parent `class`' copies. Others believe that all objects of both `class`es share one copy of the `static` members regardless of parent or child. Which is right?

We could try to use one of our conceptual models to ferret out the answer to this conundrum, but a fairly simple set of code should help us see what's going on as well. Just download those files and load them into your favorite environment to follow along below. Be sure to pay attention to the text comments and feel free to test it by changing the code comments in the two files of the `parent` library.

Let's look at the `parent.h` file first. The parent `class` creates three `static` members of increasing complexity to help with this example and to refresh our memories on handling `static` members of a `class`.[9] The `MAX` constant is initialized right away as it is simple enough to do so and constant. The other two are not so lucky.

The `grades` array is considered too complex to initialize within the context of the `class` itself. It must therefore be initialized outside. Similarly the `name` member variable cannot be initialized inside the `class`. This time, however, it is because it isn't constant — one-dimensional arrays are simple enough normally.[10]

Since both of these need to be initialized outside the `class` definition, you might try to do that right below in the `.h` file. But this leads to multiple definition errors from the compiler when things are being put back together after the separate compilation phase of things.

Thus, they must be placed in a separate implementation file of their own to keep them unique — not `#include`d multiple times. This is done in the `parent.cpp` file, of course.

The next file is `child.h` which simply defines a derived `class` with its own `static` constant member for good measure.

Both of these `class`es have `output` methods to display their components. The `A` parent `class` prints a random grade for the 'student' each time — just for fun.

The final file — `sm_inh_main.cpp` — calls on both of these libraries to test how the `static` members behave. First we make a parent object and test it. We `output` it, change its `static` member variable contents, and `output` it again. This seems to go off without a hitch. (Remember that the grade printed is random!)

Next we test the derived `class` and how it interacts with the parent's `static` member variable. (We don't test the member constants because we wouldn't be able to affect them anyway — even if they were copies in our local `class` space instead of being shared with the parent `class`.) If the child `class` has a separate copy of the `static` members, then the initial name should still be `"Bill"` as that member was originally initialized in the `parent.cpp` file. If not, it should be `"Alma"` as changed by the parent `class` object above. After changing the `name` itself, the child object re-`output`s itself to see if the change took.

Then, just to be extra certain we know what's going on, we `output` the parent object one more time to see if the child's change of `name` affected it as well.

As you can see when you run the program, the output might look something like this:

---

[9]Especially since we never talked about `static` arrays in a `class` after we learned about C-style arrays.

[10]In fact, the `name` member wouldn't even exist if we never initialized it! The compiler wouldn't bother making it real space if we didn't flesh it out more.

```
Parent object testing:
Bill D
Alma C

Please press [Enter] to continue...

Derived object testing:
Alma B
ME:  42
Jeff C
ME:  42

Please press [Enter] to continue...

How did changing the derived static member affect the parent?
Jeff A
```

You see that all changes were effective and so the parent and child classes both share a common memory space for their static members.

This could have also been ferreted out fairly easily with the set theory model of inheritance. As a subset of the original class' objects, the child class objects would clearly share the same static members as all their surrounding parent class objects.

## 13.4   Wrap Up

In this chapter we've learned about the basic concept and syntax of inheritance between classes. We've studied basic designs using inheritance and even making small hierarchies of related classes. But there was something off-putting when we looked at a derived object with a base pointer or reference and called methods the child had reworked — the parent's version was always called!

Thus we developed polymorphism to solve this issue. And with that power came further ideas like abstract classes and polymorphic containers to solve even more interesting problems than basic code reusability like inheritance solved.

Afterwards, we went back to inheritance and studied its deeper and more esoteric corners. Here we learned how to inherit from multiple parent classes at once and how to avoid/fix the diamond pattern that might creep up. We also learned the protected keyword and how it affected member access. And we reused this keyword and private to change up how members were inherited from a parent via alternative inheritance modes. Finally we looked at how a static member in a parent class is treated with respect to inheritance.

# Chapter 14

# Compile-Time Polymorphism

Compile-time polymorphism is achieved through the use of `template`s. So you should first review what we said of `template`s in the first volume. You could also review what we said of them later in the `vector` chapter (both this section and this one).

As you'll recall, `template`s can make quick work of implementing an algorithm in a generic way that applies to most any base type. Then it is all on the compiler to make the various overloads that lead to the polymorphic behavior of that algorithm within our program. That last bit is called instantiation of the `template` and creates binaries for each base type chosen.

Now that we've motivated and refreshed, let's deepen our appreciation of what this tool can do for us!

## 14.1  template Function Design

In learning to design well with `template`s, we must dig deeper into the requirements list hinted at in the first volume when we looked at the `swap` `template` there. In fact, let's start with the `swap` `template` and make an improvement.

That `template`, recall, was:

```
template < typename SwapT >
    inline
```

```
void swap( SwapT & a, SwapT & b )    // both arguments exactly same type
{
    SwapT c;                         // default constructor
    c = a;                           // \
    a = b;                           //  |-- assignment with self
    b = c;                           // /
    return;
}
```

It's requirements list is commented off to the side, even. And this `template` will work for many data types but not all. Let's take a fairly simple type like this one:

```
class Str_Pair
{
    string x, y;
public:
    Str_Pair(const string & a, const string & b) : x{a}, y{b}
    {
    }
};
```

This is a perfectly lovely `class` with a lot of potential. But it can't be `swap`ped with our `template`! Which of the requirements is missing? All `class`es get an `operator=` we've learned and that only needs changing if we dabble in dynamic memory which isn't happening here. When calling `swap`, both arguments would be of the same type, so that's not the problem. It must be the constructor thing. Why can't this `class` default construct?

Remember the detailed rules about constructors: the copy constructor is always supplied and the default constructor is supplied unless you write one of your own. We've broken the "unless" clause here! We supplied our own constructor to divvy the arguments into the members properly. Now we can't default construct. And what sense would that make for this `class`, anyway, right?

Can we fix this issue? Sure we can! Looking back to the first volume's own example `swap` overloads, we see that they weren't written in quite this way, either. They were all copy constructing their temporary helper variable! Let's do that:

```
template < typename SwapT >
    inline
void swap( SwapT & a, SwapT & b )    // both arguments exactly same type
{
    SwapT c{a};                      // copy constructor
    a = b;                           // \___ assignment with self
    b = c;                           // /
    return;
}
```

Now we require the `template` type to be copy constructible and that will either always be supplied by the compiler or it will be supplied by the programmer with dynamic members in their `class`! In fact, we now see that calling `swap` with two `Str_Pair` objects works just fine!

## 14.1.1   A Common Language

Like [run-time] polymorphism, `template`s can basically enforce a common language amongst the different types which can be used to instantiate a `template`. Polymorphism specifies the language all of a `class`'

descendants will share via `virtual` functions. A `template` specifies a language via its `template` type requirements and then data types attempting to instantiate are merely tested to see that they indeed know the proper language before they are allowed a binary form. (The `template` mechanism is often referred to as compile-time polymorphism. That's why some people don't just call polymorphism that but rather run-time polymorphism.)

So the requirements list is the common language that all participants in the `template` must speak. We can use this much more general approach[1] to allow a piece of code to work not just with types that are related via inheritance, but which all provide the same functionality. Like, if all the types had a `print` member that took an `ostream&` and didn't change their calling object, it wouldn't matter if they were part of the same hierarchy. We could provide something like this for them with ease:

```cpp
template < typename PrintT >
ostream & operator << ( ostream & os, const PrintT & p )
{
    p.print(os);                          // print const member function
                                          // taking ostream (&)

    return os;
}
```

Now all of them can be inserted into a standard stream like `cout`! No need to retrofit all of them individually if our design were consistent in the first place. *smile*

## 14.1.2   Containers of a Feather

Let's talk about a simple linear search algorithm and how we can make it work with all kinds of containers. We've got the original design built with C-style arrays and it looks like this:

```cpp
template <typename ElemT>
size_t linsearch(const ElemT arr[], size_t len, const ElemT & val)
```

This version of the linear search `template` function takes a built-in type array of elements and the element to find. The base type of the array must, of course, match that of the element.

But, not all containers of elements are built-in type arrays. `vectors` and `strings` are `class` types which overload `operator[]` to behave like built-in type arrays.

To alleviate this problem, we can try to add a new `typename` slot to our `template`:

```cpp
template <typename ElemT, typename ContT>
size_t linsearch(const ContT & cont, size_t len, const ElemT & val)
```

Yes, it is possible to have multiple `typename`s on a single `template`. I just opened up whole new worlds for you, didn't I? *smile*

This now works with `class`-type containers and yet doesn't break our ability to work with built-in type arrays. "But wait," you say, "you cannot refer to an array!" Well, no, but I can refer to a pointer to the beginning of the array. And all arrays degrade to a pointer when passed to a function, so... *grin*

But those worried about the compatibility of `string::size_type` and `size_t` (or its compatibility with various `vector size_type`s) will want more. They'll want a third `template typename` to represent the position/size type for the container:

---

[1]So general, in fact, that we call using `template`s generic programming!

```
template <typename ElemT, typename ContT, typename PosT>
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val)
{
    PosT p = PosT{};
    while ( p < len &&              // could use !=, but < is safer
            cont[p] != val )
    {
        ++p;
    }
    return p;
}
```

That should fix their little red wagon!

### 14.1.2.1 The Empty Curly Syntax

But what's the `PosT{}`? Oh, that default constructs an anonymous `PosT` object with which we then copy construct p. Why not just `PosT p;`? Because built-in types don't default construct unless special circumstances — like us calling their default constructor explicitly — occur, remember?

Can we get rid of it? Sure:

```
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val,
               const PosT & beg)
{
    PosT p = beg;
```

That gets rid of it, but now they have to specify where to begin searching. If they really wanted to start at the beginning by default, we'd have to supply a defaulted argument like this:

```
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val,
               const PosT & beg = PosT{})
```

Now it's back! \*shrug\* Guess we couldn't get rid of it after all. . . \*grin\*

### 14.1.2.2 And the const& on Arguments?

But why have I put all these `const&` on the arguments? Well, since they are going to be filled in by the compiler at the callers request, we don't know how simple or complex of a type they may be. They may be extremely costly to copy. Therefore, I avoid the copy construction by passing them by `const`ant reference. (`const` because I don't need to modify any of these arguments, of course.)

### 14.1.2.3 Whither Requirements

Before we move on, what exactly is required by the types in this `template`? Its requirements list looks like this:

- `ContT`:
  An `operator[]` taking `PosT` and returning T. T is an auto-type or automatically deduced `typename`. It has requirements, too:

- `T`:
  `operator!=` taking an `ElemT`-compatible value or `const&`. I say compatible because both of these means of passing arguments can accept anything an `ElemT` can convert to via typecast `operator`s or constructors. This `operator!=` must also return something `bool`-like — for use in the `&&`. I

say `bool`-like for similar reasons — anything that be converted to a `bool` can be returned here. That's almost anything but `void`, of course.

- `ElemT`:
  No requirements. . . weird. . .  (Well, unless `T`'s `operator!=` accepts it by value: then it'll need a copy constructor.)

- `PosT`:
  Needs a default constructor, a copy constructor, an `operator<` returning something `bool`-like (for use in the `&&`), and an `operator++` in prefix form.

Although this seems at first a rather long requirements list, this is actually rather shallow as requirements list go. Even with its automatically deduced `typename` and all the hand-wavy `bool`-like business. It could get much worse! What if `T` doesn't actually have the `operator!=` but rather `ElemT` overloaded it as a non-member? That should work, too. What if either `operator!=` or `operator<` actually returns a new deduced type `U` that then overloads `operator&&` to combine itself with the other `operator`'s result?! The compiler can handle all of this — and more! We may not even be able to conceive of the oddness that can result from this simple linear search construct. Luckily the compiler is infinitely patient and tedious if not clever.

#### 14.1.2.4   Another Container Style

Another thing we can do is to allow for pointer/`iterator` -styled access to a container for searches. This requires a separate `template`, of course, with quite different requirements:

```cpp
template <typename ElemT, typename PosT>
PosT linsearch(const PosT & beg, const PosT & end_before, const ElemT & val)
{
    PosT p = beg;
    while ( p != end_before && *p != val )
    {
        ++p;
    }
    return p;
}
```

Now this overload's typename requirements list is:

- `PosT`:
  A copy constructor, an `operator!=` returning something `bool`-like for use in the `&&`, an `operator*` — unary — returning `T`, and an `operator++` in prefix form.

- `T`:
  `operator!=` taking `ElemT` — by value or `const&` — and returning something `bool`-like for use in the `&&`.

- `ElemT`:
  Nothing required.

The `PosT` should be instantiable on either pointers or `iterator`s. Pretty nice, eh?

#### 14.1.2.5   Overloading templates

These two `template`s can overload one another because of SFINAE — "substitution failure is not an error". Basically, the compiler tries both `template`s one at a time and the first to work is the victor. If neither work, then we have a mismatch to a call that reports as ambiguity.

## 14.2 Functions as Arguments

But there is one more thing we can do to help this `template` function out. Right now it is dependent on `operator!=` for locating the object within the collection. If the caller wants to search on more complicated criteria than simple [in]equality or doesn't have an `operator!=` available, they might want to pass a function returning a `bool` to tell us what to consider [un]equal.

Wait! Did you just say, "pass a function?!" Like, pass one function as an argument to another function?! What are you, nuts? That can't be possible! Au contraire, mon frère! It is. . .

```cpp
template <typename ElemT, typename ContT, typename CompF, typename PosT>
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val,
               const CompF & equal)
{
    PosT pos = PosT{};
    while ( pos < len &&
            ! equal(cont[pos], val)    // cont[pos] != val
          )
    {
        ++pos;
    }
    return pos;
}
```

Now there are four `typename`s — each with its own requirements list:

- `ContT`:
  Needs an `operator[]` taking `PosT` and returning `T`.

- `CompF`:
  Needs an `operator()` taking `T` and `ElemT` (both by value or `const&`) and returning something `bool`-compatible (i.e. non-`void`).

- `ElemT`:
  No requirements. . . still weird. . .

- `PosT`:
  Needs a default constructor, a copy constructor, an `operator<`, and an `operator++` in prefix form.

So what does `CompF` look like before it gets here? It could be a regular function. The caller would just use its name in the call without applying parentheses to call it there:

```cpp
inline bool same(double a, double b)
{
    return abs(a-b) <= 1e-6;
}

void else_where(void)
{
    const size_t MAX{50};
    double arr[MAX];
    size_t used_arr{0}, found_at;
    // fill in arr...

    found_at = linsearch(arr, used_arr, 12.5, same);
}
```

But as long as it meets the specifications above of taking a T and an ElemT — or types convertible from these — and returns a bool-compatible value, we'll be happy to use it in our linear search template function!

## 14.2.1 Function Objects

Thus it could be a function object which would, of course, be just a variable and not need parentheses. (It could even be a lambda as we talked about in section 12.5!)

So letting a template parameter be another function is amazing power. We can put off some of the decisions in our algorithm design until later and just take a function to handle that when it gets finalized.

For instance, take this function template:

```cpp
template <typename ContT, typename PosT, typename FuncT>
void foreach(ContT & cont, const PosT & len, FuncT call_me)
{
    for ( PosT c = 0; c != len; ++c )
    {
        call_me(cont[c]);
    }
    return;
}
```

Its purpose is to walk through a container and apply some function to each element. This can be used to work with the values individually or to even modify them since we took the container by pure reference instead of our usual const&.

As an example, we could use this to display a list of double values in a special format like so:

```cpp
void print_nice(double x)
{
    ios_base::fmtflags rememberF = cout.setf(ios_base::right|
                                             ios_base::showpoint|
                                             ios_base::fixed);
    streamsize rememberP = cout.precision(3);
    cout << setw(7) << x;
    cout.setf(rememberF);
    cout.precision(rememberP);
    return;
}

void else_where(void)
{
    double data[MAX_D] = { 3.1897, 0.0004, 2.9991 };
    size_t num_data{3};

    foreach(data, num_data, print_nice);
}
```

For more on the use of single | to set up multiple format flags simultaneously, see chapter G.

Here the result would look like this:

```
3.190  0.000  2.999
```

Nice and neat! We could also use it to total the elements in a container like so:[2]

```cpp
class Total
{
    double s;
public:
    Total(double x = 0.0) : s{x} {}
    double operator()(double x) { return s += x; }
    double operator()(void) const { return s; }
    double reset(double x = 0.0) { swap(x,s); return x; }
};

void else_where(void)
{
    long a[MAX_A] = { 10, 14, 6, 2, 8 };
    size_t count{5};
    Total sum;

    foreach(a, count, sum);
    cout << "Total is " << sum() << ".\n";
}
```

And running this we see:

```
Total is 0.
```

Hey! What happened? Well, the `Total` object `sum` was passed to the `foreach` function by value instead of reference. This caused the total to update locally inside the function and then be thrown away as it returned! To fix this, we just need to adjust the `foreach` `call_me` parameter to be by reference:

```cpp
template <typename ContT, typename PosT, typename FuncT>
void foreach(ContT & cont, const PosT & len, FuncT & call_me)
```

This time around we'll see the result as:

```
Total is 40.
```

There is a possible problem now, though, with our earlier `print_nice` example. Some older compilers won't like passing a plain function to a reference parameter. And sometimes your company might be stuck on an older compiler due to pricing issues. To fix this is a bit of a kludge. We would have to take the `call_me` parameter by `const&` and then make the `Total` `class`' internal `s` member `mutable`.[3] Then mark the `operator()` that updates `s` as `const` and you've got it all working again!

But using `mutable` to mark core data is not a good design. It's just not the done thing! Best thing to do: negotiate a compiler update with IT. *smile*

## 14.3   Requirements Filling

So, just out of curiosity, what do the requirements look like as they are filled in? Well, it's more than curiosity, it can also come in handy when reading error messages from compilers having trouble fulfilling requirements from a call. So let's explore that.

---

[2]Please don't use the sideways style for normal code development!!! This is just to make the code fit better in the flow of the text. Without it, the sample was way too long!

[3]For more on this keyword, see section 11.8.2.1 from earlier.

We saw in our first volume that the swap `template` was filled in during instantiation with types that matched the requirements. So if I called swap with two `short` integers, I'd get swap`<short>` as a binary version of the function.

But how does this extend to multiple `typename template`s like we've been doing? Just like you might expect, the instantiation just has multiple actual types filled in and comma-separated. So if we look at a call like this:

```
template <typename ElemT, typename ContT, typename PosT>
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val)
{
    // as before...just reminding us of the head
}

void else_where(void)
{
    const size_t MAX{50};
    short arr[MAX];
    size_t used_arr{0}, found_at;
    // fill in arr...update used_arr

    found_at = linsearch(arr, used_arr, 12);
}
```

We'll find that the instantiation is linsearch`<short,short*,size_t>`. Notice that they follow the order specified in the `template` head and not their use in the function head.

But when it comes to a call with a function parameter, it can be more tricky. For instance, what is the type of a regular function? They have them, but we've never looked at them. I suppose it's about time. First, a word of warning: a function resolves to a pointer type. After all, a function's code must reside somewhere in memory. And so their data types are pointer types. But they look really weird at first glance and will require some explanation. But I think we should just throw you into the deep end and then teach you to swim if that doesn't suffice. *smile*

Here are a few functions we know and love and their respective data types:

| Function Prototype | Function Pointer Type |
|---|---|
| char * strcpy(char * dest, const char * src); | char * (*)(char *,const char *) |
| int toupper(int ch); | int (*)(int) |
| time_t time(time_t * result_now); | time_t (*)(time_t *) |
| void srand(unsigned int seed); | void (*)(unsigned int) |
| int rand(void); | int (*)(void) |

Here we see that the function name is basically replaced with a parenthesized pointer. Once you get used to it, it isn't so bad.

So, then, what would be the instantiation for this call?

```
inline bool same(double a, double b)
{
    return abs(a-b) <= 1e-6;
}

void else_where(void)
{
    const size_t MAX{50};
```

```
    double arr[MAX];
    size_t used_arr{0}, found_at;
    // fill in arr...and used_arr

    found_at = linsearch(arr, used_arr, 12.5, same);
}
```

That call would be instantiated as `linsearch<double,double*,bool(*)(double,double),size_t>`.

To contrast, the call here:

```
class Total
{
    double s;
public:
    Total(double x = 0.0) : s{x} {}
    double operator()(double x) { return s += x; }
    double operator()(void) const { return s; }
    double reset(double x = 0.0) { swap(x,s); return x; }
};

void else_where(void)
{
    long a[MAX_A] = { 10, 14, 6, 2, 8 };
    size_t count{5};
    Total sum;

    foreach(a, count, sum);
    cout << "Total is " << sum() << ".\n";
}
```

would instantiate as `foreach<long*,size_t,Total>`.

And what about the pointer/iterator version of `linsearch`? How does that one work? Let's check it out:

```
    vector<string> vs = { "apple", "neato", "stuff", "laughing", "dude",
                          "sweet" };
    linsearch(vs.begin(), vs.end(), "stuff")
```

The instantiation of this can come out as either `linsearch<char[5],vector<string>::iterator>` or sometimes as `linsearch<string,vector<string>::iterator>` depending on the compiler's current whim. *smile*

But then we try it with pointers and see:

```
    char s[10] = { 'a', '9', '`', '\'', '\n', '-', 'q', '*',
                   '8', 'I' };

    linsearch(s, s+10, 'q')
```

This gives us a failure to compile! Clearly `s` is an array and should degrade to a pointer for the function call, but at least on my two compilers, it checks into the `template` as `char[10]` instead. There

is a conflict, therefore, between this parameter and the s+10 which is definitely a `char*`. To fix this, we can either do explicit instantiation:

```
linsearch<char,char*>(s, s+10, 'q')
```

Or we can force `s` to look more like a `char*`:

```
linsearch(&(s[0]), s+10, 'q')
```

Both of these work and get the right result.

### 14.3.1 Full Disclosure

The C++ committee did update the function pointer syntax in C++11 to make it somewhat less daunting. For instance, now we could do the data type of `rand` as simply `int(void)` without having to deal with the parenthesized star syntax at all.

But note that if you want to store such a function address, you need to add the star back on:

```
using RandType = int(void);

RandType * ptr_to_rand = &rand;
```

Or we could just do this:

```
using RandType = int(void);

RandType * ptr_to_rand = rand;
```

And the compiler will implicitly convert the `rand` function to a pointer without need of the address operation.[4]

## 14.4 Another Approach

Above we let the caller use our `linsearch` `template` that used `!=` directly for most types and just call the overload with a function argument when they needed to compare things like `double` that don't like `!=`. This approach is valid, but there is another.

The second approach would use a `template`d helper function and only provide the equal-testing function parameter version of `linsearch`. Such a helper could look like this:

```
template <typename CompT>
inline bool Equal(const CompT & a, const CompT & b)
{
    return a == b;
}
```

But this would just give warnings for `double` and the like or crash if the type had no `==` operation. So what do we do with those types? We recall from the `vector` revisit of `template`s in the first volume about specializing a `template` and make such for our Equal `template`:

---

[4]This is really a messy area and need not be mastered unless you really need to store a function for a while. Just let the `template` mechanism figure it out and be happy.

```cpp
template <>
inline bool Equal(const double & a, const double & b)
{
    return abs(a-b) <= 1e-6;
}
```

Note: if your compiler complains that `double` is converting to `int`, make sure you `#include` cmath and think about adding `std::` on the call to `abs`.

But how do we use these? I tried:

```cpp
char s[10] = { 'a', '9', '`', '\'', '\n', '-', 'q', '*',
               '8', 'I' };

linsearch(s, 10, 'q', Equal)
```

And it crashed at compile time! It said, basically, that it couldn't use `Equal<unknown type>` in the call. That's because we didn't tell it what type to use in the `Equal` `template`! We've got to explicitly instantiate it like so:

```cpp
linsearch(s, 10, 'q', Equal<char>)
```

And, to use this for `double` and the like, we do the same:

```cpp
vector<double> d = { 4, 4.1, 4.199999, 4.2, 4.3, 4.4 };

linsearch(d, d.size(), 4.2, Equal<double>)
```

### 14.4.1 But That's Too Accurate

Those who tried the above code already have noticed that the last search finds the 4.2 at position 3 and not the 4.199999 at position 2. This might seem odd at first, but at $10^{-6}$, there aren't enough 9s to round us up. We'd need to cut the precision of the match to make it round. Looking over our available tool-set, we find that this function solution is inadequate to the job and so a function object approach might be the better tack.

Here is such a function object `class`:

```cpp
class EqualP  // P for predicate...
{
    double epsilon;
public:
    EqualP(void) : epsilon{1e-6}
    {
    }
    EqualP(double e) : EqualP{}
    {
        set_epsilon(e);
    }
    double get_epsilon(void) const
    {
        return epsilon;
```

```
        }
        bool set_epsilon(double e)
        {
                epsilon = e;   // should we check for epsilon of 0.0?  hmm...
                return true;
        }
        bool operator()(double a, double b) const
        {
                return abs(a - b) <= epsilon;
        }
};
```

Now we can see its effectiveness in calls like:

```
linsearch(d, d.size(), 4.2, EqualP{})
linsearch(d, d.size(), 4.2, EqualP{1e-10})
linsearch(d, d.size(), 4.2, EqualP{1e-2})
```

The first two still find the value at position 3 but the last one is so slack that it rounds up and find the value at position 2.

## 14.4.2   But That's Tedious

If, again, this `linsearch` is now our only `linsearch`, then we could also change the head of the `template` to by default use `Equal` when a comparison function wasn't specified:

```
template <typename ElemT, typename ContT, typename PosT, typename CompF>
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val,
               const CompF & equal = Equal<ElemT>)
```

Note that we've still used explicit instantiation, we just don't know yet to what type! But there is one thing missing. We have to tell it that the `CompF` can default as well:

```
template <typename ElemT, typename ContT, typename PosT,
          typename CompF = bool(const ElemT &,const ElemT &)>
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val,
               const CompF & equal = Equal<ElemT>)
```

There, now that will compile and run. But the syntax of the `CompF` default is a bit much. We can make it a bit better with the `decltype` tool. Basically you give `decltype` an expression and it tells you the type you would use to declare that result. It looks much cleaner, as you can see:

```
template <typename ElemT, typename ContT, typename PosT,
          typename CompF = decltype(Equal<ElemT>)>
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val,
               const CompF & equal = Equal<ElemT>)
```

To prove it all works, here is a short code sample. It even works with the specialization!

### 14.4.3  A Variation

This talk of defaulting `template typename`s has me thinking that being `Equal` doesn't mean you have the same data type all the time. For instance, `5 == 5L` is `true` even though the left value is `int` and the right value is `long`. We can make a simple tweak to our `template` to make this possible:

```
template <typename LeftT, typename RightT = LeftT>
inline bool Equal(const LeftT & a, const RightT & b)
{
    return a == b;
}
```

Now `Equal` can be called with two different types if need be:

```
if ( Equal(5,5L) )
{
    cout << "int and long are equal!\n";
}
```

## 14.5  Failure versus Success

So we've seen many successful `template` instantiations so far. And we saw one failure with the pointer/`iterator` version of `linsearch`. But what actually happens when an instantiation fails? The messages change from one compiler to another, of course, but hopefully they are readable and you can interpret them to head toward a solution.

What's more concerning is what happens when an instantiation should have failed but didn't? What do I mean? Let's look at this `linsearch` call:

```
string::size_type um_len(const string & s, const string & t)
{
    return static_cast<string::size_type>(llabs(s.length() - t.length()));
}

void else_where(void)
{
    vector<string> vs = { "apple", "neato", "stuff", "laughing", "dude",
                          "sweet" };

    linsearch(vs, vs.size(), "stuff", um_len)
}
```

This function is clearly not comparing anything and doesn't even return a `bool`. But this call will instantiate and returns 4 as the answer. Position 4 is clearly not `"stuff"` but `"laughing"`. So what's the deal?

We'll have to analyze this a little more deeply. When `linsearch` reaches a position in the container, it passes that value and the searched for value to the function argument in that order. So we are generally looking at a call to `um_len("apple","stuff")` and so on — advancing to the next slot to change `"apple"` to the next and next and so forth.

Let's look at this first call. First, the lengths of each `string` are subtracted and the absolute value of this difference is taken. For `"apple"` and `"stuff"`, the difference is 0 which is its own absolute value. This is then cast to a `string::size_type` for the `return`.

When that call gets back to `linsearch`, this result is used in a `!` operation. `!` expects a `bool`, of course, and receives a `string::size_type` — an `unsigned` integer type! Instead of being ruffled, it just coerces the value into `bool` form. The 0 it received has no 1 bits and so it turns that into a `false`. Then, applying the `!` to that, it gets `true`, which pushes the loop around one more time.

This process continues with `"neato"` and `"stuff"` — both of which are the same `length` as the `"stuff"` to be found and so they just push the loop around again. Finally, `"laughing"` is reached which is longer than `"stuff"` by 3 and so we get a value in the `!` that has 1 bits.[5] This turns into a `true` which the `!` negates to `false` and this stops the `&&` which stops the loop!

Thus, we discover that the use of `um_len` in the `linsearch` function is not only a possibility, but it finds the first entry not of the same `length` as the searched for value in a container full of `strings`.

Can we protect from this nonsense? Not easily. This might — *MIGHT* — be possible with the new C++ `concepts` mechanism in C++23. But I haven't got a compiler to test it with. For now, we have to take it in stride as an accident waiting to happen or the most clever usage pattern ever invented!

## 14.6 Making a Whole class a template

`template`'ing a whole `class` comes in two basic variants: all `class` functions are `inline` and some `class` functions must be non-`inline`.

### 14.6.1 All inline Functions

If a `class`' functions are all `inline`, making a `template` out of it is relatively easy. Note the following function object `class`, for instance:

```cpp
template <typename SumT>
class Total_c
{
    SumT sum;
public:
    Total_c(const SumT & s = SumT{}) : sum{s}
    {
    }

    SumT operator() (void) const
    {
        return sum;
    }

    template <typename ItemT>
    SumT operator() (const ItemT & i)
    {
        return sum += i;
    }

    SumT reset(SumT s = SumT{})  // not const& because we swap into it!
    {
        swap(s,sum);
        return s;
    }
};
```

---

[5]3 in binary is `00000011` for an 8-bit space.

We just add the `template` head on top and use the new `typename` throughout where it might need to be. But one thing has changed for this type — its name. We don't see it in this all-`inline` definition, but when we instantiate a function object from it, we see:

```
Total_c<double> d_tot;
Total_c<string> s_tot;
```

Here we note that the `typename` must be filled in for instantiation to take place. The compiler has no context to deduce the type from otherwise. Even if we used an initializer on the construction like so:

```
Total_c d_tot{0.0};
Total_c s_tot{string{}};
```

The compiler will fail to deduce the `typename` for the `template` unless we are using at least C++17. Note also that the initializer for the `s_tot` object must be a `string class` object and not a literal C-string!

Examining the `template class` in more detail shows us that we've nested one `template` inside another here. The `operator()` that takes an argument takes one of a different type than the SumT for the `class` itself. The reason being that you can add say, `short` values to a `long` total or `char` or C-string values to a `string` total. The individual values don't have to be the same as the result type. This nesting of `template`s is also known as the `template` method pattern and is pretty useful in many places.

### 14.6.2   Some non-inline Functions

Not having an example handy with large enough functions to keep them not `inline`, we'll just assume that we want a couple of the above functions to not be `inline`. Let's start with the constructor. It raises two issues. One with the name of the `class`' scope and contrasting this with the name of special functions like constructors and destructors. The scope turns out to be the type of the `class` and the special functions' name is just that of the `class` itself.

We've always just assumed these were synonymous, but we were mislead by the simplicity of the non-`template class`es we were working with. The scope being actually the type of the `class` means it needs the instantiation name used instead of just the `class` name. Since we won't know a specific instantiation until the programmer using the `template` fills one in, when we write the code we use the `typename` from the `template` head like so:

```
template <typename SumT>
Total_c<SumT>::Total_c(const SumT & s) // default removed since not inline
    : sum{s}
{
}
```

Note, also, that the `template` head has to reappear on top of all non-`inline` function definitions.

Another key example to look at is our nested `template` function for adding a new value to the `sum`:

```
template <typename SumT>
template <typename ItemT>
SumT Total_c<SumT>::operator() (const ItemT & i)
{
    return sum += i;
}
```

Here we see that the two `template` heads are stacked or at least space separated. The scope/type of the `Total_c` `class` hasn't changed, though, with the new head added — it is still just `Total_c<SumT>`. The second `template` head applies only to this function and so it is the function's type that has changed to include that `typename`. Luckily that won't affect us since it is deducible from the argument.

Keep the type name rule in mind, though, when non-`inline` function takes an argument of the `class`' type or returns a `class` type value/reference as well. For instance, if we had an `operator=` for the `Total_c` `class` for some reason, it would look something like this:

```
template <typename SumT>
Total_c<SumT> & Total_c<SumT>::operator=(const SumT & s)
{
    sum = s;
    return *this;
}
```

This is about as bad as this gets, so...

### 14.6.3   Separate Compilation

In the case where all the `class`' function are `inline`, we have no problems putting it into a library. Just put the whole `class` definition in the interface/header file and away you go.

But when the `class` has non-`inline` methods, we have an issue. Tradition holds that such methods need to be defined in a separate source file (the implementation file) to maintain encapsulation and to keep our implementation secrets safe from the competition. If we were to eschew these concerns, we could define the non-`inline` functions outside the `class` definition — below it, in fact — within the header file and we'd be done.

But we don't want to eschew tradition so easily! We want it to separate! So, we went to a lot of research as a community and found that we couldn't get the compiler to recognize separated code as `template` code and still be able to instantiate the `template` later. Thus we did what any self-respecting programmer would do: we tricked the compiler.

We invented a new file extension/type and a new kind of inclusion guard. Unfortunately these aren't standard throughout the community, but they are fairly easy to spot and understand once you know what to look for.

Let's start by talking about the new extension. This needs to indicate that it contains `template` function definitions that is to be `#include`d into another file and so many go with extensions like `.inc`, `.defn`, or `.templ`. We avoid extensions like `.tmp` or `.temp` because those are considered temporary files on many systems and might be deleted by file cleaners!

We take all the non-`inline` method definitions and place them alone into the `.defn` file. They can `#include` any other libraries they need, but they cannot employ a `using` directive as they will end up `#include`d themselves!

For the above `class`, this might look somewhat like this:

```
total.defn File

template <typename SumT>
Total_c<SumT> & Total_c<SumT>::operator=(const Total_c<SumT> & s)
{
    sum = s;
    return *this;
}
```

```
template <typename SumT>
template <typename ItemT>
SumT Total_c<SumT>::operator() (const ItemT & i)
{
    return sum += i;
}
```

Once we have picked an extension, we need a new inclusion guard symbol. This one is to tell the compiler that it cannot separate `template`s into multiple files.[6] This can be wordy or terse, as long as it is readable. I often go with something like `TEMPL_CANT_SEP` — possibly with full words on the first and last.

Then we put the following inclusion-guarded code into the bottom of the interface file:

From total.hpp File

```
#ifdef TEMPL_CANT_SEP
    #include "total.defn"
#endif
```

Now to compile the whole application, we have two possible approaches. The most portable is to simply *#define* the include guard symbol before each *#include* of the library containing the `template class`' definition. This would look something like so:

From Driver C++ File

```
#define TEMPL_CANT_SEP
#include "total.hpp"
```

Or, instead of *#define*'ing this all over the place, we can define it once for the whole compile process. The most common way to do that is with the `-D` command-line parameter which most compilers seem to support. This command-line parameter to the compiler will define a symbol that follows it — typically with no spaces separating it — in all separately compiled source files this build. So it would look like `-DTEMPL_CANT_SEP` on your command-line. If compiling in a GUI IDE, however, finding where to put such a global define or command-line parameter is difficult sometimes. Hopefully your instructor will be able to help with this.

Note that a separate `.cpp` file is not required for this library as it would end up empty anyway.

Since this process is a bit complex, I've placed a complete example on the website.

### 14.6.4   Making friends

If you feel the need for your `class` to make `friend`s for efficiency at the cost of data security, you can still do that with a `templated class`. The process is a little tedious to implement but guaranteed to work in any compiler as it has been part of the standard since C++98. There are three steps:

  i) declare the `templated class`

 ii) declare the function/`class` that is to be the `templated class`' friend

iii) declare the `friend`ship inside the `templated class`' definition as normal *except* that you must place empty angle brackets — `<>` — between the `friend`'s name and its argument list

Let's look at another `class` that's been turned into a `template`: a `Set class`. This `class` has also overloaded `operator<<` for printing to an `ostream`-derived output stream. Let's make this a `friend` as

---

[6]This seems at first absurd, but the compiler doesn't even realize this is a goal, so. . .

an example. The steps look something like this in the interface file:

```
From set.hpp File

template <typename ElemType>
class Set;

template <typename ElemType>
std::ostream & operator<<(std::ostream & out, const Set<ElemType> & set);

template <typename ElemType>
class Set
{
    friend std::ostream & operator<< <>(std::ostream & out, const Set & set);
};
```

Note that the `friend`ship declaration doesn't have the function's `template` head on it since it had the same one as the `class` it has been made `friend`s with. Since the function takes a `class` argument, it would pretty much have to have the `class`' `template` `typename`(s) in tact and so be itself a `template`.[7]

When you go to define the `friend` function, it looks pretty normal again:

```
From set.defn File

template <typename ElemType>
std::ostream & operator<<(std::ostream & out, const Set<ElemType> & set)
{
    size_t i;
    out << " { ";
    for ( i = 0; i < set.cur_elem; ++i )
    {
        out << set.the_set[i] << ", ";
        if ( (i+1) % 7 == 0 )
        {
            out << "\n   ";
        }
    }
    out << '}';
    if ( i % 7 == 0 )
    {
        out << '\n';
    }
    return out;
}
```

Note that the `Set` argument has the full type name here as it did during declaration of this function. It did not have the full type name during the declaration of `friend`ship...

### 14.6.4.1  An Alternative Approach

In newer compilers, there is a shorter syntax that allows you to combine the not only the declaration of the `friend` function with the declaration of `friend`ship, but even define the `friend` function right there `inline`, too! This would look like so:

---

[7]The only question remaining is if it has to be exactly the same `template` head or if it can add other `typename`s for its own use... Sounds like a great place for a little test app to prove a concept!

---

**From set.hpp File**

```cpp
template <typename ElemType>
class Set
{
    friend std::ostream & operator<<(std::ostream & out, const Set & set)
    {
        size_t i;
        out << " { ";
        for ( i = 0; i < set.cur_elem; ++i )
        {
            out << set.the_set[i] << ", ";
            if ( (i+1) % 7 == 0 )
            {
                out << "\n    ";
            }
        }
        out << '}';
        if ( i % 7 == 0 )
        {
            out << '\n';
        }
        return out;
    }
};
```

---

Here is an example program for you to check out since it got a little long to publish here.

## 14.7   Overloading vs. Specializing

When using a library with `template`'ing going on and you aren't getting the results you expect, how can you best resolve it? There are a couple of approaches, let's explore which is best. I've worked up two sample programs on the website for you to download and play with. They can be found here and here.

In the first one we see a `template` for finding the `minimum` value of two arguments followed by a similar function. The similar function might look at first like it is overloading the `template` as we did above. But it doesn't really. Remember that the plain function here is just called `min` whereas the `template` is really called `min<Data>` — very different names. But we see this kind of thing a lot and sometimes just call it overloading because we are human and we are a little sloppy. Just be aware that you might need to defend yourself to others on such grounds.

So the main for both programs is the same and looks like this:

---

```cpp
int main(void)
{
    char e[10] = "Jason";
    char f[20] = "jason";

    cout << "   minimum (plain):  " << ::min(e,f) << endl;
    cout << "minimum (explicit):  " << ::min<const char *>(e,f) << endl;
    cout << "minimum (typecast):  " << ::min(static_cast<const char *>(e),
                                             static_cast<const char *>(f))
         << endl;
    return 0;
```

---

```
    }
```

Here we see three calls to a `min` function of some sort. One is 'plain' where we give just the argument names and let the compiler figure everything out. The second is an explicit instantiation of the `template` version. And the third uses typecasting to tell the compiler what we think the arguments should look like for the function.

On this first program that utilizes 'overloading', we see the following results:

```
    minimum (plain):  I'm just for const char *!!!
Jason
minimum (explicit):  I'm generic!!!
jason
minimum (typecast):  I'm just for const char *!!!
Jason
```

Here the right answer (the capitalized value) is given by both the plain call and the typecasted call. The `template` version failed because it just compared the addresses of the variables instead of their contents.[8]

In the second code, we see two `minimum` functions and again one is a `template`. The other this time is a `template` specialization for `const char *`. We run these again through the `main` above and now get this result:

```
    minimum (plain):  I'm generic!!!
jason
minimum (explicit):  I'm just for const char *!!!
Jason
minimum (typecast):  I'm just for const char *!!!
Jason
```

Again, the one that called the `template` gets the wrong answer, but this time it is for different calls! This time the plain call does the `template` whereas last time it was only the explicit instantiation that did it! In both programs, though, the typecasting worked beautifully to get us to the function that would work.

Lesson learned: use typecasting to help guide the compiler to the right function for the job.

## 14.8 Non-Type template Parameters

The things that the compiler deduces to match a `template` don't have to be just types. There are also non-type `template` parameters. These do have to be 1D discrete types like for a `switch` head, but they can still be useful all the same — just like a `switch`!

In the provided example, you can see two functions to `swap` the contents of two C-style strings. The non-`template` uses an arbitrarily large constant to size the temporary helper C-string. This is, of course, a poor design as it takes up way too much space most of the time and wouldn't work on some cases, still. We also fear that the calls to `strcpy` will overrun at some point and that's never good!

This version is at first commented out as it provides a more exact match to the compiler's taste than does the `template` version despite the `template` being listed first. If you uncomment it and run the program, you'll see that when it prints the sizes of the parameters with the `sizeof operator`, you

---

[8]Also, because of this, your mileage may vary on this result. The order in which variables are stored in the function activation record is compiler and OS dependent and so can change from one situation to another. Be careful the conclusions you draw and test with the capital on each variable before assuming it is working right.

get something smaller than the arrays were declared to be. This isn't because the contents is shorter as it doesn't vary from value to value. It is actually the size of a pointer on your system. Remember that arrays degrade to pointers when passed to functions.

Commenting that function back out, we recompile to use the `template` with the non-type parameter. This looks like so:

```
template <size_t N>
void swap(char (&a)[N], char (&b)[N])
{
    cerr << "my non-type template swap...'" << a << "' (" << sizeof(a)
        << ") & '" << b << "' (" << sizeof(b) << ")\n";
    char c[N];
    strcpy(c, a);
    strcpy(a, b);
    strcpy(b, c);
    return;
}
```

Here a `size_t` is deduced and used in the `template` instantiation. In this case, the deduced value is the declared size of the C-string parameter. The compiler can do this because the array was declared in the same scope as the call — main. And we also took special care with the formal parameters themselves and made them refer to the original arrays. This reference mark had to be parenthesized because it would otherwise have been assumed part of the array's base type and arrays of references are illegal.

Once deduced — and note we are requiring that both parameters have the exact same size! — we can use this constant known at compile time to size the temporary helper array inside the `swap` function itself! Now it won't waste memory or fall short. It will always be the exact right size! And we can use `strcpy` without fear of overrun, too!

The careful observer will note that just putting the references on the array parameters was enough for the compiler to remember the sizes of the declared arrays. But to do so we must fill in the [] on those parameters with some literal or constant number — making `sizeof` redundant. Having the `template` deduce the array size makes the function work with a variety of arrays again.

## 14.9 Metaprogramming Basics

`template` metaprogramming is the use of `template`s to run code at compile time. Think about it. That's just insane, right?!

But it's true. When deducing `template` information — especially non-type `template` information — we can make use of it to write bits of code that will run during compilation. I've got two examples of this that seem tractable at this level. For future exploration, I'd recommend something like the boost libraries site. They have tons of examples and do amazing work that sometimes finds its way into the standard libraries!

### 14.9.1 Improving the swap Function

Let's start where we left off above. There was a test in the main for the `swap` example that couldn't run because it tried to `swap` two C-style strings that were of different physical lengths. Let's make that sort of thing work!

We'd need two non-type `template` parameters to deduce the separate sizes of the C-string arrays coming in like so:

```
template <size_t N, size_t M>
void swap(char (&a)[N], char (&b)[M])
```

And then we'll need to decide which of these is larger to size the temporary array with. This can be done with a simple ?: test, of course, like so:

```
char c[N > M ? N : M];
```

Since both N and M are compile-time constants, either is able to be used to size the temporary array.

This kind of decision being run by the compiler at compile-time is just amazing and allows us to do all kinds of coding we couldn't do before. And it isn't limited to just this. We can encapsulate this decision into a package for easy reuse — even at compile-time:

```
template <size_t A, size_t B>
struct Max_of
{
    enum { value = (A > B ? A : B) };
};
```

Here we've coded up a templated struct with a single enumerated constant called value. This constant was initialized with our ?: decision from before. Now we can use this in the declaration of c from before like so:

```
char c[Max_of<N,M>::value];
```

Since both N and M are compile-time constants, when they are used by Max_of, they can initialize the constant value member. Then this compile-time known constant is used to declare the size of the temporary array. It's like magic!

Anyway, here's the full code to check on your own if you like.

#### 14.9.1.1   Another Approach

Some folks wonder if another approach isn't better. Some folks would want to just 'overload' a normal swap template with a char * version that uses strlen and dynamic allocation to do the temporary array sizing like so:

```
void swap(char * a, char * b)
{
    size_t a_len{strlen(a)}, b_len{strlen(b)};
    char *c{new(nothrow) char[a_len > b_len ? a_len : b_len]};
    // swap with strcpy and strncpy as above
    delete [] c;
    return;
}
```

This approach is poor for two main reasons: a) strlen is slow and b) allocation and deallocation take time away from our user. Just say no.

### 14.9.2   Improving Random long Generation

Another classic way to use template metaprogramming is specialization on the values a deduced type can have. This could be something simple like bool or something more interesting like an enumeration.

We'll take it easy and use `bool`. Since `bool` has only two possible values, we'll only need the main `template` and one specialization.

What will we do with it? Let's endeavor to make random `long` values more appropriately spread for our system. You see, most systems today use 32-bit `rand` values but `long` might be 64-bit. If we try to use the built-in `rand` to make those kinds of values, it won't be able to spread out far enough to generate all the desired data.

We'll start with the code to call our `template` so the `template`'s code itself will make more sense. A basic `rand_range` overload would be written like so:

```
inline long rand_range(long min, long max)
{
    return rand_range_helper< (RAND_MAX >= LONG_MAX) >(min, max);
}
```

Here I'm testing the `RAND_MAX` constant from the `cstdlib` directory against the maximum value of `long` as defined in `climits`. (I could have used `limits`' `numeric_limits` facility, but I chose to go this way since I had to use the old C constant for the random maximum anyway.)

Now we call our `template` — `rand_range_helper` explicitly instantiated with the value of this comparison which will be either `true` or `false`. We just need to put the right code into the main `template` and the specialization to match the situation we find ourselves in.

In the `true` case, the random number system has enough range to handle the possible requested bounds. But in the `false` situation, we'll have to do something special to make it work. I'm going to make the main `template` the `true` version like so:

```
template <bool>
inline long rand_range_helper(long min, long max)
{
cerr<<"true version called\n";
    return rand() % (max - min + 1) + min;
}
```

Here we just generate a random value as we usually would when things are going right. But in the `false` specialization we'll do something new:

```
template <>
inline long rand_range_helper<false>(long min, long max)
{
cerr<<"false version called\n";
    return ((static_cast<long>(rand()))<<32|rand()) % (max - min + 1) + min;
}
```

Notice that to make this a specialization, we must list the special value in the angle brackets after the function name. There is no other context for the compiler to deduce it from.

Here we use bit manipulations as from chapter G to splice together two randomly generated values. (There is a bit of an assumption that the random values are exactly half as wide as the `long` values on the system. This isn't without merit, but might bite you in the butt on some platforms. Be careful!)

If we download this sample program and compile/run it, we should see proper values generated. If you see `true version called` on the screen, then you've got a large enough random number generator for your `long`s already. If you see `false version called` many times, you have had to use the bit-manipulation method to meet the spread. Either way it should work. But in the case of the `true` version

being called, you won't see any values outside the `RAND_MAX` boundary whereas for the `false` version you should.

## 14.10 static template Members

As we learned in section 13.3.4, `static` members of a `class` can sometimes behave oddly under new tools like inheritance or `template`s. So let's explore how they work under `template`s with this simple test `class`:

```cpp
template <typename DataT>
class StaticMemberTest
{
public:
    static double stuff;
    static DataT data;

    void print(ostream & out = cout) const
    {
        out << "I have '" << data << "' & '" << stuff << "'.\n";
        return;
    }
};

template <typename DataT>
double StaticMemberTest<DataT>::stuff = .42;

template <>
short StaticMemberTest<short>::data = 4;

template <>
string StaticMemberTest<string>::data = "alpha";
```

As we can see, we've got two `static` data members and we've initialized them below the `class`. The `double` member is initialized once and the `template`d member is specialized for both `short` and `string`. Now we just need a `main` to test it all with:

```cpp
int main(void)
{
    StaticMemberTest<short> a;
    StaticMemberTest<string> s;

    cout << string(35,'*') << "SHORT" << string(35,'*') << '\n';
    cout << "a:  ";
    a.print();
    StaticMemberTest<short>::data = 42;
    cout << "a:  ";
    a.print();

    cout << string(35,'*') << "STRING" << string(35,'*') << '\n';
    cout << "s:  ";
    s.print();
    StaticMemberTest<string>::data = "betazoid";
```

```
    cout << "s:  ";
    s.print();

    cout << string(35,'*') << "STUFF" << string(35,'*') << '\n';
    StaticMemberTest<string>::stuff = 4.2;
    cout << "a:  ";
    a.print();
    cout << "s:  ";
    s.print();

    return 0;
}
```

And running the test program, we see that all of the instantiations' `double` members were initialized to `0.42` and that their `DataT` members were properly initialized and changed. But in the final phase, we see that changing the seemingly shared `double` member via the `string` instantiation, only its member changed — the one for the `short` instantiation remained `0.42` as before.

Thus we find that each separate instantiation of the `class` will have its own memory location for each `static` member — whether that member is of known type or is `template`d itself.

## 14.11   templates and Inheritance

How can `template`s be mixed with inheritance? Just about any way imaginable! Well, at least as many as I could come up with. Let's start with a basic `class`:

```
class B1
{
};
```

From here we can derive a `template class` like so:

```
template <typename T>
class D1 : public B1      // template inherits from non-template
{
};
```

(I've placed members inside the `class` to test instantiation, but it doesn't affect the results — it works either way. So don't sweat it. But if you want, you can tweak a full example by just adding a main that declares objects of the various types to these `class` definitions and adding members to them.)

So you can get a `template` inheriting from a non-`template`. We can also inherit a non-`template` from a `template`:

```
class D2 : public D1<long>     // non-template inherits from template
{
};
```

Here D2 is a non-`template` because we've explicitly instantiated our ancestor `template` to D1<long> making it no longer a `template`.

And we can inherit a `template` from a `template` like so:

```cpp
template <typename T>
class D3 : public D1<T>    // template inherits from template -- alike
{
};
```

Here the D3 `class` is of the same kind of `template` as the parent `class` — both T as specified by the programmer using the `template`. But the child `template` need not be the same instantiation type as the parent as we see here:

```cpp
template <typename U>
class D4 : public D1<double>  // template inherits from template -- differ
{
};
```

Here D4 has a `template` type but the parent `template` has been instantiated to D1<double> so that it doesn't share the same type necessarily.

I've also tested this with a main declaring variables of all these types, but that didn't seem to prove anything beyond just making sure the above compile. Still, that's a lot of variability in inheritance patterns as you mix in `template`s in different ways!

## 14.12   Wrap Up

Compile-time polymorphism — or `template`s in C++ — is an amazing tool that leads to all sorts of powerful techniques. We learned things from basic function design and using the common language model to making a `template` out of an entire `class` their `friend`ships. And we talked about separate compilation of `template`s. We focused some on both overloading and specializing techniques and calls and looked at `template` parameters that weren't even data types at all! Then we dabbled in meta-programming techniques and reviewed how the `template` mechanism interacted with prior things like `static` members and inheritance.

# Part VII

# Data Structures

# Chapter 15

# Algorithm Analysis

Over the years, we've developed many algorithms to solve various problems. In fact, we've developed many algorithms at times to solve the same problem. This might have happened via independent development or in competition with one another. However it happened, we need a way to compare two algorithms that accomplish the same task to see which is right for us to use in our current application. In steps algorithm analysis.

## 15.1   Just the Basics

There are two main types of analysis: time analysis and space analysis. Time analysis looks at an algorithm with respect to how long it takes them to run on a problem of a certain size. Space analysis looks at an algorithm with respect to how much memory it takes while running on a problem of a certain size — beyond the actual problem data, of course. Space analysis isn't as hard so we'll leave that as an exercise and focus on time analysis.

These comparisons are troublesome to perform on hardware since machines run at different speeds and programmers code with more/less efficiency than one another. Not to mention compilers optimize better/worse than one another. Oh, and there may be multiple processes/users taking up CPU time and throwing the timings off. And... Well, let's just say it would be easier if we could analyze algorithms 'offline' — mathematically rather than on a particular coded version. Besides, if we analyze them mathematically, it gives us a single analysis to cover any implementation language chosen.

To do a time analysis we count the number of critical operations performed on a problem of a certain 'size'.[1]  Exactly what is a critical operation is given to you at this stage and will become increasingly intuitive as your experience grows. This counting gives us a function of problem size that determines how long — in terms of critical operations — a particular algorithm will take to solve the problem.

---

[1]Size is quoted here to indicate that this isn't always easy to determine. We'll stick with the simpler cases in this text, however.

## 15.2   Best, Worst, Average

Further, when looking at these critical operations, we are interested in the algorithm's behavior in three different cases: best case, worst case, and average case behaviors. In the best case behavior, everything in the input will be perfect to give the most efficient run of the algorithm. In the worst case scenario, exactly the opposite is true — we'll look at the least efficient run of the algorithm for the inputs given. And then we consider what will happen in the rest of the input situations — what will happen on average.

To do these analyses, we can break down the different possible inputs and assign them each a probability of occurrence. Then we multiply the critical count for each input by its probability and add them up:

$$\sum_{\text{all inputs } i} p_i \cdot c_i$$

This will give us the proper weighted critical counting function for each of the three cases. What, then, are the probabilities? Well, for best case analysis, we set the probability of the best possible input to 1 (aka 100% likely) and all other inputs' probabilities to 0 and the sum reflects just the count for the most efficient input. Similarly for a worst case analysis.

For an average case analysis, we typically assume all inputs are equally likely to occur and then we can factor out the $p_i$ multiplicand to outside the summation and just add the critical counts and multiply by the common probability once.

BTW, average and worst are the most sought after measurements. Best case situations are deemed rare and inconsequential. Average case happens a lot and worst case can affect your operations deeply and so those are the most impactful to our decisions.

### 15.2.1   A Quick Example

Let's take a quick look at linear search, for instance. Typical code for this was found in our discussions from chapter 14:

```cpp
template <typename ElemT, typename ContT, typename PosT>
PosT linsearch(const ContT & cont, const PosT & len, const ElemT & val)
{
    PosT p = PosT{};
    while ( p < len && cont[p] != val )
    {
        ++p;
    }
    return p;
}
```

But for analysis purposes, we typically look at a more generic view called pseudocode like this:

```
LinearSearch(container, length, value)
{
    p = 1
    while ( p <= length AND container[p] != value )
    {
        p = p + 1
    }
    if ( container[p] != value )
```

```
    {
        p = 0
    }
    return p
}
```

Very like the C++ code, but more general and easily implemented in any high-level language. Only the `<=`, subscript, and `!=` notations from C++ remain and are often substituted with math notations instead. I've not done that here for simplicity of typography.

We take as our critical operation here the comparison of the `value` to the `container` element. The integer assignments, updates, and comparisons are going to be done at CPU speeds and only comparing the target data might be more expensive and thus affect our timing much more.

A quick thought process will let us know that the best case for this algorithm is when we find the data in the first position and are immediately done. Thus the count for this situation is 2 comparisons and that is our result: $T(length) = 2$ in the best case. But this is a little bulky for our math kin so we usually term the `length` parameter as $n$ for the math: $T(n) = 2$. Thus, for the best case, we achieve a constant amount of work no matter the number of data in the `container`.

Similarly, we look at the worst case possibility and find that the most comparisons of target data is done in the case where we either find the target `value` in the last position or can't find it in the `container` at all. Both of these lead to a count of $n + 1$ comparisons. Setting the probabilities of each to 0.5 (so that they add to 1 as probabilities must), we find that the worst case is: $T(n) = n + 1$. So in the worst case, we take a linear amount of work and it will totally depend on the number of data in the `container`.

For the average case, we make each outcome equally likely. But what are the possible outcomes? They are the positions from 0 to $n$ where the last position indicates a not found condition. These take from 1 to $n + 1$ comparisons with the $n + 1$ comparisons being repeated twice. Doing some math, we see that there are $n + 1$ outcomes and so, if each is equally likely, their probabilities are all $\dfrac{1}{n + 1}$. So our summation looks like this:

$$\sum_{i=0}^{n-1} \frac{1}{n + 1} \cdot (i + 1) + \frac{1}{n + 1} \cdot (n + 1)$$

You'll learn lots about how to tackle this kind of thing in discrete mathematics when you get there. Suffice it to say that this boils down to: $T(n) = \dfrac{n}{2} + 1$. So it takes just over half the list on average to find the data we are looking for. This is still linear, but less so than the worst case due to the slope being shallower.

## 15.3   Sequence versus Nesting

An easy way to estimate the critical count function of an algorithm is to look over its loops. Count out how many times each loop operates. If loops are sequenced — one right after another, add their counts. If loops are nested, multiply their counts. (This is due to the fact that the inner loop will complete all of its iterations for every one of the outer loop's iterations.) For instance, the following algorithm:

```
for (i = 1; i <= n; ++i)
{
    for (j = 1; j <= n; ++j)
    {
        // do something interesting
```

```
      }
   }
   for (i = 1; i <= n; ++i)
   {
       // do something else interesting
   }
```

Would give us $n^2 + n$ as the estimated critical operations count.

When the interesting bit in the center of the loop(s) changes the loop condition variable(s), things get a little tricky. We'll talk more about that kind of thing in discrete math.

## 15.4   Oh, Omega, Theta

So now that we have a critical operations counting function, how do we compare it to that of another algorithm? This is done with some mathematical techniques to categorize the algorithms with respect to several standard reference functions. Once we know to which reference function each algorithm compares, we will then know how the two algorithms compare with one another.

The categorization works by placing our algorithm functions into one of three sets: $O$, $\Omega$, or $\Theta$ also known as Big-Oh, Big-Omega, and Big-Theta, respectively. Let's look at each in turn.

If a function falls into the set Big-Oh with respect to a reference function, then we know that our function's output grows at worst as rapidly as the reference function. That is, it will grow that rapidly or less rapidly as the size of the problem input grows larger.

If a function falls into the set Big-Omega with respect to a reference function, on the other hand, then we know that our function's output grows at least as rapidly as the reference function. That is, it will grow that rapidly or even more rapidly as the size of the problem input grows larger.

When a function falls into the set Big-Theta with respect to a reference function, it specifies that the two functions grow at approximately the same rate.

These three sets split the first quadrant of the x-y plane into three areas. We focus on the first quadrant only, of course, because neither time nor problem size can be negative. Here is a diagram of the split:
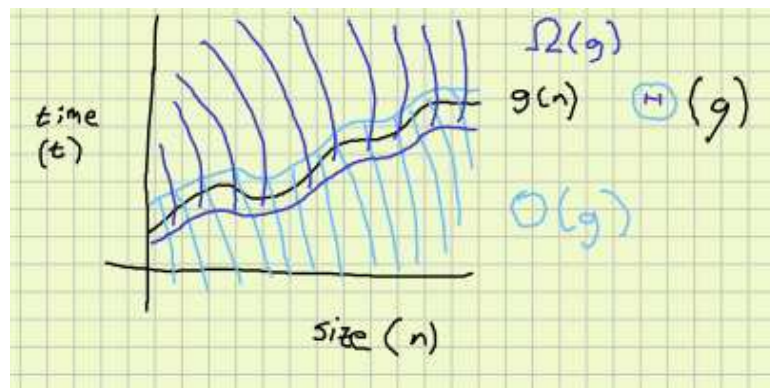


All of these can be off by a constant multiplier and it is considered fine. Also, we generally don't see 'lower-order' terms as significant in the process. Both of these things come about due to limits from calculus so your mileage may vary. What we mean is that $3n^2 + 4n + 8$ will still be Big-Theta of $n^2$. In more appropriate notation: $3n^2 + 4n + 8 \in \Theta(n^2)$.[2]

Big-Theta is clearly ideal, but it can be tricky to arrive at such a classification for certain algorithms. Many research hours have been spent making this easier for larger and larger classes of functions, but it doesn't always apply to your algorithm. When you are in such a situation, you will find that a Big-Oh classification will be much easier.

---

[2]This will be explained in lots more detail in a discrete math course/book. We're not too interested in the details at the level of this text.

In fact, this was so for so long, that many of our analyses are in terms of Big-Oh instead of Big-Theta. Many [older] programmers will be more interested in the Big-Oh just because that is what they are used to seeing available. So if you know something is Big-Theta from your current studies, don't fret that your colleagues keep saying Big-Oh. Just nod and smile and know you are more correct. *smile*[3]

## 15.5   Standard Reference Functions

The most commonly used reference function are in this table:

| Function | Notes |
|---:|---|
| $n!$ | factorial |
| $c^n$ | exponential |
| $n^c$ | polynomial |
| $n\log(n)$ | $n$ times logarithm of $n$ |
| $n$ | linear |
| $\log(n)$ | logarithmic |
| $c$ | constant — often listed as just 1 |

Three things about the table:

- The table is organized in terms of decreasing time complexity. That is, the top function grows fastest as input size gets larger and the bottom function grows slowest. Each function is in Big-Oh of all of the ones above it, in fact. (As well as being in its own Big-Oh, of course.)

- Why is $n$ separated from the other polynomials? There is that slightly worse than $n$ and better than $n^2$ function of $n\log(n)$ in there. This is often used and cannot be ignored! (Similarly, the constant function is separated from the polynomials as well because $\log(n)$ comes between it and $n$.)

- The $c$ represents any constant. For the last category, it can be any known value no matter how large. As a constant function, it will still not grow at all as the size of the input grows — its slope is zero, after all.

The line between exponential and polynomial draws an arbitrary line that delineates the reasonable-speed algorithms — below — and the unacceptable-speed algorithms — above. In fact, this line is often drawn amidst the polynomials between $n^3$ and $n^4$. Some would draw it even lower! But it can depend heavily on your hardware and the expected size of problems you are going to encounter.

### 15.5.1  A Look Back at Linear Search

So where does linear search fall in all of this. Well, from the above, we see that linear search's average time analysis of $T(n) = \dfrac{n}{2} + 1$ falls in $\Theta(n)$ as does its worst time analysis of $T(n) = n + 1$. At least its best time analysis is $\Theta(1)$.

## 15.6   Validating an Analysis

Once you have an analysis — and let's face it, it isn't very accurate with our estimation techniques — you'll probably want to verify it. Well, we can take a tip from Calculus (limits and convergence) to do it with implementation timing data.

The idea is to time your algorithm implementation[4] at several different sizes of input. (Remember to take a median of times at each input size instead of just a single reading!) Then divide each by the

---

[3]BTW, if a Big-Theta eludes you, you can also search for a Big-Omega proof to match your Big-Oh reference function and having "sandwiched" the two, you'll have proven your Big-Theta. It is, after all, the intersection of Big-Oh and Big-Omega.

[4]For more on timing program events, see appendix F.

analysis function you came up with in your Big-Oh or Big-Theta work evaluated at that input size.

This quotient can do one of three things:

- If, as you do this for the different input sizes, the results get larger and larger (diverge), then you have underestimated your analysis — the algorithm actually is growing more quickly than you thought.

- If, on the other hand, the results get smaller and smaller and run toward zero (converge to zero), then you have overestimated your analysis — the algorithm actually is growing less quickly than you thought.

- Finally, the results might get closer and closer to a particular non-zero value (converge). This denotes that you have a good estimate in your analysis!

Good luck!

## 15.7   Wrap Up

In this chapter we have discussed algorithm analysis basics. We've covered a wide range of terminology related to the task and basic estimation and validation techniques. We looked at classifying our analysis function into categories based on standard reference functions. And we tried it all out on linear search — a classic and yet simple algorithm.

Hopefully it gave you a healthy respect for the purpose of analyzing algorithms and how tricky it can be. But you will learn more in future studies, never fear!

# Chapter 16

# Recursion

Recursion is when a function calls itself — either directly or indirectly! We learned this briefly in the first volume, but only to say don't do it because it was too tricky. Now we'll learn how to do it properly.

Why is it so difficult/tricky? Well, think about when do you stop calling yourself? Ah. . . This is especially true for indirect recursion which often happens accidentally when two functions call one another in a cyclic fashion.

Further, recursion directly correlates to two mathematical techniques: proof by induction and recurrence relations.[1] As in those systems, you will need to identify three things:

- a base case (also called a point of reference, starting/stopping position, etc.)

- a reduction step (reducing the size of the original problem before passing it along to yourself)

- a recursive call (actually calling yourself to finish the job)

Finally, there are four rules of programming with recursion:

- Know when to stop calling yourself! This relates directly to your base case.

- Make the problem smaller before calling yourself. This relates directly to your reduction step.

- Call yourself and trust that the call to yourself worked. This is the hardest step because that trust is hard to come by.

- Don't waste previous calculations!!! (If at all possible.)

## 16.1 Design

This leads us to the question of how to design such an algorithm? It can be viewed as the following generic steps:

---

[1] In fact, proof by induction is useful to prove a recursive algorithm correct and recurrence relations are used to do algorithmic analysis of recursive algorithms!

```
given an initial problem of size n
unless n is minimal,
   break this into a number of     \
       smaller problems of the      |--- reduction step
       same nature and then        /
       apply this approach to      \__ recursive
       each                        /    call
   put the smaller case answers    \
       back together to answer our |__ build
       size n problem -- include   |   phase
       any size n specific info    /
else
   just use a canned response for  \__ base
       simple case(s)              /   case(s)
```

The three important things are listed there in the side labels along with a fourth. This fourth is not a classically listed part but is nevertheless a crucial step in the process — putting the recursive call's answer and the current step's information together to form the current step's answer. Without it all else would be for naught!

Let's look at a few examples:

### 16.1.1   Factorial

The classic example to start with is calculating a factorial. It looks something like this:

```cpp
long factorial(short n)
{
    return n < 0 ? -1          // error catch...
         : n <= 1 ? 1          // base case -- enhanced!
         : n*                  // build phase
             factorial(        // recursive call
                     n-1       // reduction step
                   );
}
```

I've spaced it out to label all the parts because they are rather tightly woven together in this simple algorithm. The first branch is actually an addition to the general plan — catching issues caused by using a `signed` type for the argument. Factorials are only defined for non-negative inputs, after all. But the magnitude of the produced factorials grows so quickly that a 32-bit `long` can only hold up to 12! correctly. So I just went lazily with `short` for the input type.[2]

The next branch is the base case but I enhanced it since the first two factorial values are actually identical. So both 0! and 1! are 1 and why make a separate step for this.

The final branch is the rest of the algorithm. It reduces the size of the problem by subtracting 1 from our input value, passes that off to our recursive call, and then takes that answer and multiplies it by our original input to get our answer.

---

[2]A 64-bit `long long` could have handled up to 20! but that isn't much gain and would have left the same basic type choice for the argument, so...

## 16.2  Visualization

To visualize recursion in action, many people swear by many methods. But the most common seem to be index cards and memory diagrams. I've tried to combine those below in a side-by-side fashion with the current index card on the left and the memory diagram of the function call stack on the right. Let's start with one of the most mundane cases: `factorial(1)`.

Calling `factorial(1)`

```
long factorial( 1 )
{
    return 1 < 0 ? -1
         : 1 <= 1 ? 1
         : 1 * factorial( 1-1 );
}
```

```
+=========+
| n = 1   | factorial(1): 1
+=========+
```

You can see that on the left the code has been filled in with 1 everywhere the parameter `n` would have been. (These ones were colored red to differentiate them from the regular code which also had ones.) We can then trace which branch actually executes and get the return value for the function listed on the right side (alongside the function's memory area on the call stack).

But there wasn't any recursion happening with this call, so let's explore a deeper case: `factorial(3)`.

Calling `factorial(3)`

```
long factorial( 3 )
{
    return 3 < 0 ? -1
         : 3 <= 1 ? 1
         : 3 * factorial( 3-1 );
}
```

```
+=========+
| n = 3   | factorial(3): 3*____
+=========+
```

It at first seems nothing has changed, but note that as we evaluate the branches, we end up in the recursive branch hanging on or waiting for the result of the recursive call. This leads us to call for `factorial(2)`. And now things look more like this:

Calling `factorial(2)`

```
long factorial( 2 )
{
    return 2 < 0 ? -1
         : 2 <= 1 ? 1
         : 2 * factorial( 2-1 );
}
```

```
+=========+
| n = 2   | factorial(2): 2*____
+=========+
| n = 3   | factorial(3): 3*____
+=========+
```

Now there are two calls on the stack and both have reached their third branches and are waiting on a recursive evaluation! Let's keep going to the next call — this one to `factorial(1)`. Although it seems like we've been here before, this time we are at a call stack depth of three:

Calling `factorial(1)`

```
long factorial( 1 )
{
    return 1 < 0 ? -1
         : 1 <= 1 ? 1
         : 1 * factorial( 1-1 );
}
```

```
+=========+
| n = 1   | factorial(1): 1
+=========+
| n = 2   | factorial(2): 2*____
+=========+
| n = 3   | factorial(3): 3*____
+=========+
```

Finally, we reach the base case branch and find an answer.  We then return the 1 to the previous function and this happens:

Calling `factorial(2)`

```
long factorial( 2 )
{
    return 2 < 0 ? -1
         : 2 <= 1 ? 1
         : 2 * factorial( 2-1 );
}
```

```
+=========+
| n = 2   | factorial(2): 2*__1_=2
+=========+
| n = 3   | factorial(3): 3*____
+=========+
```

Here we receive the 1, multiply it by our waiting 2, and get the return value to be 2 overall.  This leads us back to the earlier function call state but with an answer:

Calling `factorial(3)`

```
long factorial( 3 )
{
    return 3 < 0 ? -1
         : 3 <= 1 ? 1
         : 3 * factorial( 3-1 );
}
```

```
+=========+
| n = 3   | factorial(3): 3*__2_=6
+=========+
```

Now we multiply our waiting 3 by the returned 2 to get our return value of 6.  This goes back to the caller and we are done.

Using index cards, you would also get a feel for the function call stack because you could stack the recursive call cards on top of the previous cards as you went.  If you do one deep enough, like `factorial(6)` or so, you'd really get the feel and vision of the function call stack getting deeper and deeper as you went.

## 16.3   Other Examples

There are many other examples we could do of recursive functions.  And we will see more in upcoming chapters. But for now let's just look at a few more that add something interesting to the picture.

### 16.3.1   A Mystery Guest

This first function is a bit of a mystery for you. (No reading ahead to find out who done it! *smile*)

Here's the function:

```cpp
size_t mystery(const char * s)
{
    return s == nullptr || *s == '\0' ? 0
                                      : 1 + mystery(s + 1);
}
```

What do you think it does? Take a few minutes to puzzle with this for real. Don't just read on. It's an important skill to be able to reason out what a piece of poorly documented code does at times.

So, once you've got your idea, let's talk it through. We've clearly got a C-string pointer coming in. We know this due to the `const char*` argument type and the null character (`'\0'`) test. This value for the first `char` in the C-string — an empty C-string — leads to a 0 result as a `size_t`. And so does a pointer value of `nullptr` — a C-string that isn't just empty but nowhere! Then, for the recursive case, we see that the `char` that we just saw that wasn't the `'\0'` is counted in as 1 more than the recursive call.

But that recursive call is weird, right? We pass `s+1` instead of a subtraction! Thinking about this for a second, though, we realize that adding to a pointer will move it toward the end of the array it points into. So this is advancing the pointer through the C-string toward the eventual null character and that base case!

So, putting it all together, we see that an empty (or non-existent) C-string results in 0 and any non-null `char` adds 1 to a running total. Sounds like a `strlen` replacement if there ever were one!

What's interesting about this example? Well, two things. Firstly, it has an additive reduction step instead of being subtractive or divisive. That's pretty unusual. Secondly, it gives us a `strlen` replacement that respects the presence of `nullptr`s in our world. Turns out that the standard library `strlen` will crash if given a `nullptr` instead of just reporting 0 like it probably should!

### 16.3.2  Binary Search

Another classic start is binary search. I thought we'd explore it as a pointer-based version:

```cpp
const size_t max_size_t = numeric_limits<size_t>::max();

template <typename DataT, typename PtrT>
size_t binsearch(const PtrT * beg,    // inclusive
                 const PtrT * end,    // exclusive
                 const DataT & target)
{
    size_t half_way = (end - beg) / 2,
           found;
    const PtrT * mid = beg + half_way;
    return end - beg <= 0 ? max_size_t      // max_size_t when not found!
         : *mid == target ? mid - beg
                          : *mid < target ? (found = binsearch(mid+1, end, target),
                                            found == max_size_t ? found
                                                                : half_way + 1 + found)
                                          : binsearch(beg, mid, target);
}
```

Sorry to shrink the font on you, but the nesting was getting hairy. *smile*

So what's going on here? Why the deep nesting, anyway? It looks like the `target` not being found and being before the middle are pretty straightforward. But when the `target` is after the middle, we have an issue.

Returning a `size_t` offset is hard this time because on the recursive call, the `beg` parameter would have been mapped from the prior call's `mid+1` pointer. Thus, if we just returned the recursive call result directly, it would be offset by over half the container's length! So, we have to adjust it by adding `half_way+1` to it. But we can't just add this because the `target` might not have been found at all! If we added `half_way+1` to `max_size_t`, we'd end up returning just `half_way` due to integer wrap-around! Thus the hideous nesting. . .

So what have we learned from this example? One thing is that sometimes you can have more than one recursive branch. Like here we have before middle and after middle calls. And another is that you sometimes don't have additional work to do with the recursive result. Like here we just return the recursive result when the `target` is potentially before the middle somewhere.

## 16.4   Single versus Multiple

But that last example also brings to mind that sometimes we do have the need to perform multiple recursive calls to get our current answer put together. In binary search, only one of the two calls would execute. But here I'm talking about a situation where we need two or more calls at the same time. A classic example is the Fibonacci numbers sequence. It is defined mathematically as:[3]

$$
\begin{aligned}
fib(0) = fib(1) &= 1 \\
fib(n) &= fib(n-1) + fib(n-2) \quad, \quad n > 1
\end{aligned}
$$

As you can see, each number after the second requires two prior numbers to be added together. A visualization might help. Here is a visual depiction for $fib(5)$:
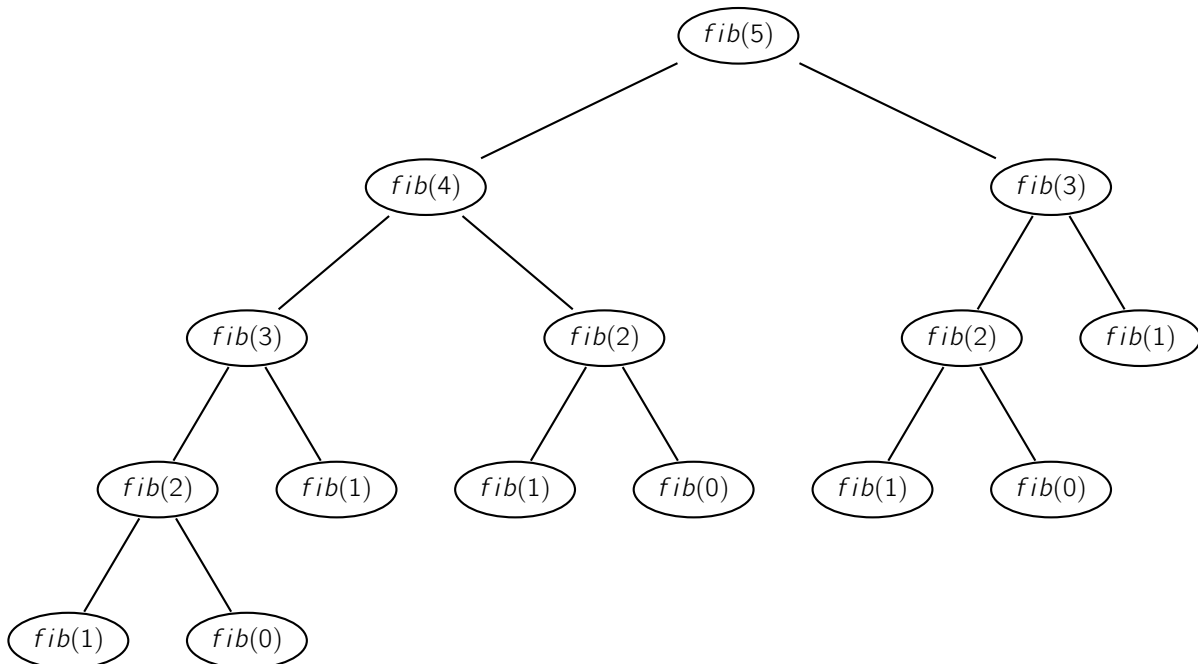


Depending on the context of the problem, this repeated recursion can be quite expensive in terms of recalculation. This goes against our fourth directive from before: don't waste previous calculations!!!

---

[3]Some would say $fib(0) = 0$ instead. This leads to the same sequence but with a 0 in front. *shrug*

## 16.5   Costs

The above discussions lead us to the idea that recursion is sometimes rather costly. So why do we do it? Is it needed? Is it useful?

Turns out recursion is exactly as powerful of a solutions tool as our old friend repetition — aka looping. So we'd need tools to turn recursive definitions and algorithms into looping solutions to remove it. Sometimes this is simple and sometimes not.

In the meantime, there are other tools to remove the redundant calculations involved in some recursions. So we'll explore those, too.

But first we'll look at the last big cost in recursion: the limits of the function call stack.

### 16.5.1   Stack Overflow

As I said, recursion is limited by the size of the function call stack, of course. And there is a limit on this size in all compiled systems and many interpreted systems as well.

To show this, I've created a broken recursive function which has no base case to stop it. It will therefore overflow its function call stack after so many calls. To track this, I've installed a `static` `size_t` variable to count how many times this function has been called. To help cut down the number of `calls` it takes to experience a stack overflow, there is an `array` declared on each invocation of the function. The size of this `array` can be changed easily between runs to make there be 2, 200, or even 20000 `double`s as you desire. You'll find that the larger the `array`, the less calls it takes to eat up the function call stack.

```cpp
void broken_recursion(void)
{
    static size_t calls = 0;
    ++calls;
    double array[200] = { 0.0 };
    cerr << "I've been called " << calls << " times.\n";
    broken_recursion();
    return;
}

int main(void)
{
    broken_recursion();
    return 0;
}
```

When compiled on my system, I get a warning about `array` being unused. That's by design, so I'm ignoring that for now. When I then run it on my system with an `array` size of 20'000, I get 52 runs before the crash (which is called a Segmentation fault on my Linux box). At a size of 200, I get 5185 runs before the crash. So it is approximately proportional. Also note that your mileage may vary as your macOS or ChromeOS or Windows may have a different maximum stack size. The point is that all systems have a maximum and so this is a perpetual issue with recursion and large problem sizes.

We'll look at tackling this issue in chapter 18 when we talk about the stack data structure.

### 16.5.2   Tail Recursion

Tail recursion is an optimization performed by many compilers. When they see a recursive function coded in just the right way, they can automatically transform it into a looping form. The basic idea is that the

last statement in the function needs to contain the recursive call. Our `factorial` function from before would be a prime example:

```
long factorial(short n)
{
    return n <= 1 ? 1          // base case -- enhanced!
           : n*                // build phase
               factorial(      // recursive call
                       n-1     // reduction step
                   );
}
```

Since the recursive call is in the last statement, many compilers will rewrite this automatically as something akin to:

```
long factorial(short n)
{
    long f = 1;
    while ( n > 1 )
    {
        f *= n;
        --n;
    }
    return f;
}
```

We won't study here how this transformation is accomplished, but we've now learned to benefit from it.

#### 16.5.2.1   Full Disclosure

In the interest of completeness, there is a fancier form of tail recursion that many people espouse on the web. It involves transforming the function to have a helper argument which often means having a helper function for the recursion itself like so:

```
long fac_helper(short n, long ans)
{
    if ( n <= 1 )
    {
        return ans;
    }
    return fac_helper(n-1, ans*n);
}

inline long factorial(short n)
{
    return fac_helper( n, 1L );
}
```

Another approach in C++ would be to supply the `ans` parameter with a default argument. Then the second function wouldn't be needed.

This slightly fancier version of tail recursion is another learning curve up, though, so I wouldn't expect you to master it just yet. *smile*

## 16.5.3  Memoization

Although it looks like we just misspelled memorization, the title of this section is a real word. It historically comes from a process involving interoffice memos (or memorandums). The use of these memos to remember that something had already been done and therefore didn't have to be done again — rather we just looked at the answer from the memo that recorded it.

So we memoize results in programming by remembering all of the previous calculations so that they need not be re-computed! In C++ this is easily accomplished with a `static` local `vector` or the like:

```cpp
long factorial_memoized(short n)
{
    static vector<long> fact = { 1L, 1L };
    return n < 0 ? -1
        : n < fact.size() ? fact[n]
        : *(fact.insert(fact.end(),
                        n * factorial_memoized( n - 1 )));
}
```

See how we initialize the `vector` with our base cases and then in the second branch (the first still being error checking) we look to see if the answer should be inside the `vector` already. If it is, we just use it right away. If it isn't, we proceed to the third branch and calculate and `insert` it.

The `insert` is worth extra care in reading, though. It puts the new value in front of the `end` position of the `vector` — thereby extending the `vector` by one. Then, `insert` returns the position just added by `iterator` which we dereference with the ∗ operation and return.

Another, but slightly trickier way, would be to use `push_back` followed by subscript:

```cpp
    : (fact.push_back(n * factorial_memoized( n - 1 )),fact[n]);
```

Note, though, the terrible use of the comma `operator` here to separate the `push_back` from the subscript in the last branch. This is terrible code, of course, and really hard to read! We could make it work like so:

```cpp
long factorial_memoized(short n)
{
    static vector<long> fact = { 1L, 1L };
    if ( n >= fact.size() )
    {
        fact.push_back(n * factorial_memoized( n - 1 ));
    }
    return n < 0 ? -1                      // same old error catch
        : n < fact.size() ? fact[n]   // should now always fire
        : -1;                          // just in case the push_back failed
}
```

But this is a little clunkier and doesn't use the tools we've learned to their utmost.

At any rate, this memoization causes any second call for a value to be more-or-less instantaneous and that makes a huge difference! This is especially true for multiple recursions like the Fibonacci numbers above.

## 16.6   Wrap Up

In this chapter, we learned about the idea of recursion and its components. We learned to design recursive functions from scratch. And we saw recursion in action!

But, we also saw that recursion doesn't come without costs. Luckily for it, there are many places in upcoming data structure processing where recursion is just so elegant and hard to avoid, we use it regularly. So don't fret! Your new-found knowledge will not be wasted!

# Chapter 17

# Linked Lists

Why study a new data storage technique? Aren't arrays working fine for us? Especially the dynamically growing ones stored inside `vectors` and `strings`? Yes and no. There are things arrays do well and things that can be improved — sometimes drastically.

By holding each piece of data separate and linking them together with some sort of link like maybe a pointer, linked lists aim to provide for improved memory management patterns as well as speedier insertion and removal compared to a traditional array. However, they have slowed access patterns due to having to follow pointers from element to element.

But there are other attributes that cause us to think critically about our decision to use a linked list versus an array in a particular application.

Below is a chart that denotes these phenomenon:

| Action | Array | Linked List |
|---|---|---|
| insert (at known location) | $O(n)$ | $O(1)$ |
| remove (at known location) | $O(n)$ | $O(1)$ |

(Continued)

| Action | Array | Linked List |
|---|---|---|
| access element | $O(1)$ | $O(n)$ |
| excess memory | 0-100% | $100 \cdot \frac{\text{sizeof}(\text{void*})}{\text{sizeof}(Data)}\%$ |
| holes in memory | multiples of one another | all same size |
| 'talks' with OS | few (amortizes) | many (all nodes separate) |

Above the double line are the standard things folks talk about with regard to linked lists as compared to arrays. But those below the double line are less well disseminated.

### 17.0.1 Data Structures Algorithms

Clearly the three vital activities of insertion of new data, removal of old data, and locating data are those above the double line. And as clearly, linked lists win on the first two of these and lag behind on the third. So, just from the standpoint of "what am I doing with the data?" in the application, we have some hint at what we should be choosing.

If we are doing a lot of insertion and/or removal of data, but less search, a linked list is great! But if our goal is to load all the data once and then search it lots and lots, a linked list might not be so ideal.

### 17.0.2 Memory Issues

In terms of memory usage patterns, the linked list wins by leaving behind all the same sized holes. This will aid in later dynamic lookups for new places to store data. The array, on the other hand, grows in multiples — for amortized growth — and leaves behind holes only useful if shrinking later as well.

But from the point of view of how often do these allocation requests come up, the array seems to win the day. It amortizes the need for allocation down to a constant feel whereas the linked list is clearly linear as every individual piece of data will require a separate allocation.

Finally, in terms of memory waste, we have a cautionary tale. The array is potentially wasting a lot of space depending on how much of the allocated space is utilized to store true data. In the linked list we only allocate space for true data except for some pointers we use to link them together. As long as these pointers are not ridiculously larger than the data themselves, we are wasting relatively little memory, therefore. (Note to self: don't make linked lists of simple `char` data! After all, `char` is only 1 byte in size and pointers are at least 4. By that token, a list of `short` is not that good of an idea, either.)

There are two different ways to put this together, of course. We'll look first at dynamic management of a linked list.

## 17.1 Dynamic

In a dynamic linked list, we'll be using pointers to link together the data. We call each memory block a node and the two basic fields are the data field and the next pointer that tells us the location of the next piece of data. When we reach the last node, a `nullptr` is used to signal the situation that there is no next piece of data. Such a design leads naturally to `class`es as below.

## 17.1.1 In Code

There are actually two approaches to designing a linked list in `class` form. The classic approach uses two `class`es and another uses just a single `class`. We'll look at the classic approach now and the other after some experience has brought us to a revelation.

### 17.1.1.1 The Classic Approach

The classic approach uses separate `class`es for each of the node and the whole list. The node is quite simple with just two members: the data and the next pointer. The list also holds just two members: a counter for how much data we've got in the list — totally optional, but useful and easy to maintain — and a pointer to the first item in the list — the so-called head of the list:

```cpp
template <typename DataT>
class LinkedListNode
{
    DataT data;
    LinkedListNode * next;
public:
    // need ctors
    // ...but do NOT do dtor (unless defaulted/private/broken
    //                        do NOT delete next or it will
    //                        take down the whole chain following
    //                        this node with it!)
    // likewise, the copy ctor and op= should default because
    // we aren't managing our own pointer here --- the next
    // class will do that for us!

    LinkedeListNode(const LinkedListNode & lln) = default;
    LinkedListNode & operator=(const LinkedListNode & lln) = default;
    ~LinkedeListNode(void) = default;

    // accessors, mutators
};

template <typename DataT>
class LinkedList
{
    size_t count;
    LinkedListNode<DataT> * head;
public:
    // need ctors, dtor, op=
    // insert, locate, remove
    // accessor for count
};
```

The only real improvement to be made to this design is to nest the node `class` inside the `private` area of the list `class` so that its details are hidden and the application programmer knows that they don't need to know how the nodes are stored — just that the list holds their data. This is a minor design change, though, so it is left as an exercise. *smile*

## 17.1.2 In Memory

The diagram to the right shows a linked list of nodes without the initial `head` pointer. Just imagine a `class` object off to the left whose `head` member — with another `count` member alongside — points to the first node in this list.
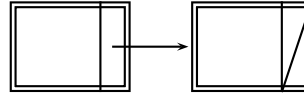
But in that diagram, you see nodes with relatively large `data` paired with a `next` pointer. And the first of these has its `next` pointer actually pointing at the second node in the list. The second node's `next` pointer is, though, `nullptr` to show that there is no following item in the list.

### 17.1.3  Typical Actions

The typical actions mentioned above in the intro were inserting new data, removing old data, and searching for data within the storage structure. Let's examine each in turn.

#### 17.1.3.1  Insertion

Insertion into a linked list at a known insertion point is actually quite simple. For instance, let's say we had the list at right containing 15 in one node and 67 in the next. And let's say we wanted to insert a 42 between the 15 and the 67. Well, we'd just need to take a few simple steps:
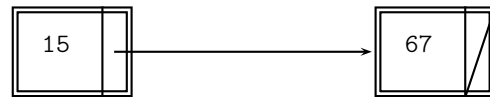
1 Create a new node for the 42.
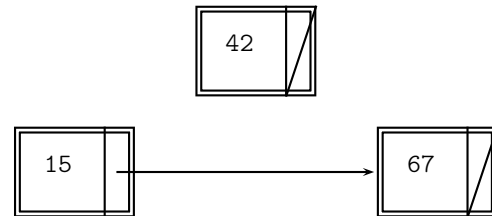
2 Set the 42 node's `next` pointer to point to the 67 node (that is, initialize it to be the 15 node's `next` value).
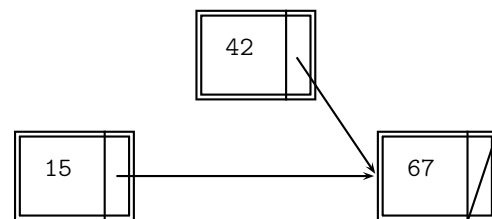
3 Update the 15 node's `next` pointer to point to the 42 node.
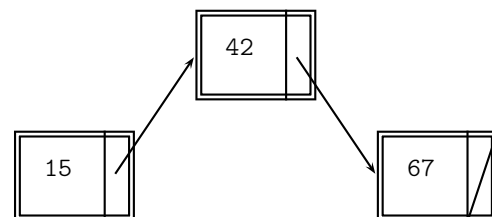
Compare to insertion into an array where all following elements have to be shifted over to make room for the new data and you see the improvement!

This can be coded pretty simply if we have a good constructor for the node `class` that takes the data and the next pointer (probably defaulted to `nullptr`). It can look like so:

```
LinkedListNode<short>* prev_node;  // set up to point to the node with 15
LinkedListNode<short>* new_ptr = new LinkedListNode<short>(42, prev_node->next);
prev_node->next = new_ptr;
```

Of course, `short` is a bit small compared to a pointer on most systems as well. Again, we should really only consider linked lists for larger data items like `strings` and other `class` objects.

### 17.1.3.2 Removal

The process of removing a piece of data is almost as simple. Once you find the data and have a pointer to the node that precedes it, you just follow the next few steps. Let's say you had the situation from above with 15, 42, and 67 in the list currently and wanted to remove the 42 this time.

1 Set a separate pointer to point to the node to be removed.
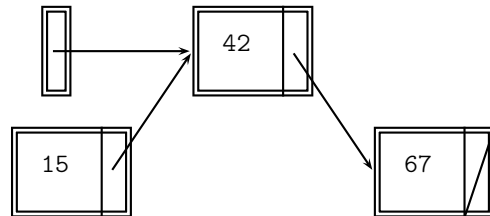
2 Update the prior node's `next` pointer to point to the node being removed's `next` pointer value.
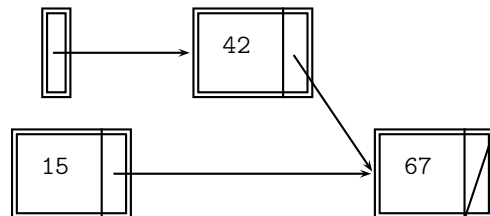
3 `delete` the node to be removed via the separate pointer.
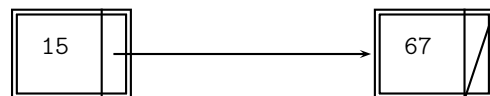
Again, compare this to the process for removing data from an array and it is amazingly simple and efficient! Recall that in an array, we'd have to scoot all remaining elements down to overwrite the removed data (and one another).

The code is equally simple:

```
LinkedListNode<short>* prev_node;  // set up to point to the node with 15
LinkedListNode<short>* rem_ptr;    // set up to point to the node with 42
prev_node->next = rem_ptr->next;
```

```
    delete rem_ptr;                         // safe because the dtor of the node
                                            // class is defaulted!
```

### 17.1.3.3  Searching

The question then becomes, how do you get to the node with 42 or to the position you want to add the 42 in the first place? Let's tackle these one at a time.

For removal, we need to find the node with the value to remove and the node before that in the list. This requires two pointers and we just need to advance them along until we reach the proper positions. Of course, the pointer for the target node should be initialized to the `head` of the list so we are looking at nodes actually in the list. But the pointer to the previous node should be initialized to `nullptr` and should then update as the target pointer does but a step behind:

```
    LinkedListNode<short>* prev_node = nullptr;
    LinkedListNode<short>* rem_ptr = head;        // done in the scope of LinkedList

    while ( rem_ptr != nullptr &&         // this would mean we'd looked past the end
            rem_ptr->data != target )     // for above target == 42
    {
        prev_node = rem_ptr;              // remember from whence we came
        rem_ptr = rem_ptr->next;          // advance to next position in list
    }
```

Just be careful if the `target` value isn't found in the list — don't try to remove it!

Also notice how we advance to each `next` position with the assignment of the removal pointer to its `next` pointer. This is the mainstay of linked list processing!

Not too hard, but definitely slow. Of course, if you didn't know the location of the value in an array, it would take at least a binary search to find it. So we aren't too far behind. *smile*

But what if the 42 wasn't in the list yet and we wanted to put it in order. We'd need a loop something like this one:

```
    LinkedListNode<short>* prev_node = nullptr;
    LinkedListNode<short>* after_node = head;    // done in the scope of LinkedList

    while ( after_node != nullptr &&     // this would mean we'd looked past the end
            after_node->data <= target )  // still less than
    {
        prev_node = after_node;           // remember from whence we came
        after_node = after_node->next;    // advance to next position in list
    }
```

Here, again, caution should be taken to make sure the insertion will happen inside the list or at its `head`. If we are inserting in front of the `head`, we need to update the `head`, too!

Another thing we can do that is like searching is just visiting every node in the list and doing something — maybe we print the `data`. That is even simpler as we don't need to track the previous node's location:

```
    LinkedListNode<short>* curr_node = head;
    while ( curr_node != nullptr )
    {
```

```
        cout << curr_node->data << ' ';
        curr_node = curr_node->next;
    }
```

Again, a simple `next` pointer update to advance in the list from node to node until the `nullptr` is reached.

## 17.2 Recursion and Linked Lists

In staring at the linked list diagrams from earlier (section 17.1.2), we notice that each node of the linked list has a `next` pointer that points, effectively, at a slightly shorter linked list of nodes — a sublist of sorts. This lends itself naturally to the thought of recursively processing the linked list. We can even come up with elegant solutions to some problems like the printing problem in the last section.

### 17.2.1 Printing Forward

To print a linked list from `head` to `nullptr`, we can utilize a recursive routine like this:

```
template <typename DataT>
void print_in_order(LinkedListNode<DataT> * head)
{
    if ( head != nullptr )
    {
        cout << head->data << ' ';
        print_in_order(head->next);
    }
    return;
}
```

It's a nice `private` method for the node `class`, in fact. It would probably have a `public` interface method associated that had no `head` argument.

### 17.2.2 Printing Backward

And what if we wanted the content backwards instead of forwards? Maybe they've asked to see our sorted list sorted the other way round? We wouldn't re-sort it, of course: just walk it backwards! This can be done effectively by just reversing the order of the recursive call and the `cout`:

```
template <typename DataT>
void print_in_reverse(LinkedListNode<DataT> * head)
{
    if ( head != nullptr )
    {
        print_in_reverse(head->next);
        cout << head->data << ' ';
    }
    return;
}
```

Now it walks to the end of the list before it starts printing and prints on each return from a recursive call — placing it back at a previous node.

### 17.2.3   But A Long List...

Yes, if a list is really long, we'll likely run out of [function call] stack space and overflow. So many want to see how to print the list in reverse without recursion. Turns out we need a little help with this. Help from another data structure known as — of all things — a stack! We'll come back to this issue in that chapter (18).

### 17.2.4   A Different Approach to Design

The recursive processing above leads to a natural second approach to linked list design. In this approach, we utilize the sublist idea from above. To emphasize sublist relationship, we merge the two `class`es of the above design into a single one. This loses the `head` pointer and the `count` of held elements. The `next` acts as the `head` of every sublist. And storing the `count` in each node would be highly wasteful and terribly inefficient to keep maintained.

But it begs the question of who holds the true `head` of the list?! Turns out it doesn't matter. Anyone holding a node holds the `head` of a linked list. If you want the uber-`head`, talk to the `main` function! *smile*

#### 17.2.4.1   A Clear Design Winner?

On one hand, the classical approach has two `class`es to maintain — even if one is nested inside the other versus the recursive approach which has only the one `class` to work with. But recursive thinking is a bit of a pain for many programmers and so that approach comes with some mental overhead as well.

Also, the classical approach has a clear delineation of duties as to which `class` is responsible for what. The recursive approach makes this less clear-cut and, again, can cause some programmers mental anguish.

Overall, I'd say there isn't a clear winner between these two approaches to linked list `class` design. But keep both in mind just in case they come in handy someday!

## 17.3   Static

The static in the title of this section is not referring to `static class` members or `static` local variables or any of that. It is simply referring to a non-dynamic situation. That is, we are going to make a linked list without pointers. We'll store it in a fixed-sized array for good measure. That way this approach will allow us some of the improvements of the earlier linked list but in non-dynamic form for when there just isn't enough memory to do dynamic allocation.[1]

How can we get by putting our data in a fixed-sized array and still get the benefits of linking? Won't we just have to shift everything around like normal? Not like this! We'll keep some sort of link attached to each piece of data in the nodes. And each array element will be a whole node.[2]

### 17.3.1   Fixing the Pointers

So how do we replace the pointers if our data are stored in an array? Well, with `size_t` position markers, of course! So each piece of data is paired with a `size_t` which tells what position holds the next piece of data. In fact, we call this member of the node `class` next as well!

And what of the `nullptr` ending? That is taken care of with a special flag value that can't be a position in the array: the maximum `size_t` value. Since `size_t` is `unsigned`, this can be gotten in one of two ways. Old-style programmers would cast a `-1` to `size_t` knowing it will wrap around to the

---

[1]This will often happen on embedded systems but rarely in a more mainstream platform.
[2]In ancient designs, the links would be in an array parallel to the data. But we've long since given up such clunky designs and use `class`es as often as possible. For more, see that section back in the first volume of this series.

largest value. But modern designers will use the `numeric_limits` `template` — yep, that's a `template`, too! — to get the `max` value.

Let's say we had the same data as in our earlier linked list: 15 and 67. It would look like this:

| 15 | 1 |
|---|---|
| 67 | -1 |
| 0 | -1 |
| 0 | -1 |
| 0 | -1 |

And the `head` member of the list would be 0, of course. (Note that I'm being lazy and coding the end-of-list marker as −1 even though it is old-fashioned. This is because the actual value is unknown to us until it is compiled so I'm not sure if it is 32-bit or 64-bit on any given system.)

### 17.3.2 Inserting New Values

So how would we get the 42 in there as we did before? Well, once we know it should go between 15 and 67, we simply add it to the `count` position of the array and fix up the `next` links. We don't have to insert it between the two current pieces of data because they have links to tell you what order things are really in. So we simply overwrite the `0,-1` in position 2 of the array with `42,1` and update 15's `next` link to be 2. (Then increment `count`, of course.)

| 15 | 2 |
|---|---|
| 67 | -1 |
| 42 | 1 |
| 0 | -1 |
| 0 | -1 |

Now the list goes from 0 to 2 to 1 and then stops.

As always, watch for having to update `head` when the new data should go before our current `head` (15 here)!

#### 17.3.2.1 An Aside

BTW, this technique can also be used to organize large data with sorting that doesn't move the actual data around. We simply sort the links into the right order and leave the data items where they fell originally. Note how the data in our array are not in order, but the linked list of them actually is.

### 17.3.3 Removing Old Values

To remove an old value — like that 42 there, we just take the `next` links and fix them and then decrement the `count`. So, once we know that 42 came from 15, we change 15's `next` to be that of 42 and then set 42's `next` to −1 for safety. Decrementing the `count` completes the process:

| 15 | 1 |
|---|---|
| 67 | -1 |
| 42 | -1 |
| 0 | -1 |
| 0 | -1 |

We don't actually have to remove the 42 unless we want. (Just like `delete`'ing that node earlier didn't change the data in that memory location from the heap. . . )

### 17.3.4 Free versus Taken Spots

What if things were more extreme than this? What if the list looked like so before 42 was removed:

| 15 | 2 |
|----|----|
| 67 | -1 |
| 42 | 1 |
| 3 | 4 |
| 9 | 0 |

Now, when we remove the 42, it looks like so:

| 15 | 1 |
|----|----|
| 67 | -1 |
| 42 | -1 |
| 3 | 4 |
| 9 | 0 |

Further, `count` is 4 so we'll be putting the next insertion into slot 4 in the array. That would overwrite the 9 from the second list position! Not cool!

We need a way to track what positions are actually not in use instead of just placing the new data in the same position as the value of `count`. Thus is born the free list. It is a second linked list interwoven amongst the slots with the data linked list. It has its own head but we call it `free` or `free_head` instead to distinguish it from the actual data `head`. In this scenario, we'll know where the new piece of data can be laid down with ease.

### 17.3.4.1 Revisiting Insertion

After removing 42 above, we'd set the `free` head from `-1` as it was when the list was just full to 2 where the 42 had resided. Then, when we go to insert the new data wherever it may land in the data list, it will be put into `array[free]` and `free` will be updated to whatever that slot's `next` used to be. That is, we use the head of the free list to tell us where to put any new piece of data and just move it along. Thus, when we create the empty list to start the above process, it would look like so:

| 0 | 1 |
|---|----|
| 0 | 2 |
| 0 | 3 |
| 0 | 4 |
| 0 | -1 |

And `free` would be 0 and `head` would be `-1`. As 15 arrives, we put it in position 0 since that position is free and change `free` to be that slot's `next` which was 1. Also we change the `next` there to what `head` was since this is an insertion at the start of the data list. Then we update `head` to position 0, as well. Finally, we update `count`, as well:

| 15 | -1 |
|----|----|
| 0 | 2 |
| 0 | 3 |
| 0 | 4 |
| 0 | -1 |

As we follow along, we end up back here:

| 15 | 1 |
|----|----|
| 67 | -1 |
| 42 | -1 |
| 3 | 4 |
| 9 | 0 |

But now the `free` list starts at 2 and we know to put the next piece of data that arrives there. Let's say it was -2. It would look like this after insertion:

| | |
|---|---|
| 15 | 1 |
| 67 | -1 |
| -2 | 3 |
| 3 | 4 |
| 9 | 0 |

And `free` would be −1! (And `head` would be 2, but we already covered that idea.)

### 17.3.4.2 Revisiting Removal

Well, how did 42 get removed in the first place? It would have looked like so beforehand:

| | |
|---|---|
| 15 | 2 |
| 67 | -1 |
| 42 | 1 |
| 3 | 4 |
| 9 | 0 |

And we found that 15's `next` links to the 42, so we change its `next` to be 1 like 42's is. And then we change 42's `next` to what `free` currently is since it's going in front of the `free` list now. Finally, change `free` to be 2 where 42 is located. (Yes, yes, and decrement `count`.) This leaves us with this array:

| | |
|---|---|
| 15 | 1 |
| 67 | -1 |
| 42 | -1 |
| 3 | 4 |
| 9 | 0 |

And `head` is still 3 and `free` is now 2.

## 17.4 Other Linking Styles

There are other linking styles than what we've seen so far. What we've seen is typically called single linking or a singly linked list. Two other styles are prominent as well: double linking and circular linking. Let's discuss each in turn.

### 17.4.1 Double Linking

As you can see at right, doubly linked lists have two pointers per node instead of one. The second pointer is called the previous or `prev` pointer as it points to the previous node in the list.
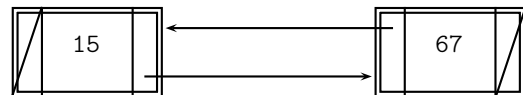


The list itself often contains a `tail` pointer in addition to the `head` pointer, as well. This pointer tells the location of the node with a `nullptr` next pointer. (Whereas the `head` tells the location of the node with a `nullptr` prev pointer.) In the diagram above, the node with 15 is the `head` and that with 67 is the `tail`.

In code, this looks something like so:

```cpp
template <typename DataT>
class DoublyLinkedListNode
{
    DataT data;
    DoublyLinkedListNode * next, * prev;
public:
```

```
    // need ctors
    // ...but do NOT do dtor (unless defaulted/private/broken
    //                        do NOT delete next or it will
    //                        take down the whole chain following
    //                        this node with it!)
    // likewise, the copy ctor and op= should default because
    // we aren't managing our own pointer here --- the next
    // class will do that for us!

    DoublyLinkedeListNode(const DoublyLinkedListNode & lln) = default;
    DoublyLinkedListNode & operator=(const DoublyLinkedListNode & lln) = default;
    ~DoublyLinkedeListNode(void) = default;

    // accessors, mutators
};


template <typename DataT>
class DoublyLinkedList
{
    size_t count;
    DoublyLinkedListNode<DataT> * head, * tail;
public:
    // need ctors, dtor, op=
    // insert, locate, remove
    // accessor for count
};
```

The insertion and removal routines are similarly efficient, but their constant number of operations is higher because of the need to connect twice as many pointers.

Searching, on the other hand, is twice as easy as we no longer need to track the previous node during the search — it is given as a member of the node now!

So why the extra pointers? What good can it do? One thing is management of the list is simpler except the insert and remove routines are slightly more complex.

Another thing is that a list with any kind of locality information — such as a list sorted low to high — can start the search at the end of the list closest to the target data instead of always from the `head`. This can improve practical performance quite a bit!

### 17.4.2 Circular Linking

The earlier lists we've seen were all also known as linearly linked lists. They go from the `head` to the `nullptr` or `tail` as the case may be. The list at right, however, is circularly linked. Its final node points back to the `head` node in a never-ending cycle.
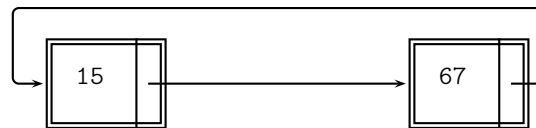


For this reason, the `head` is often replaced with a current or `curr` pointer. This keeps track of the place the list was last processing. To determine if an algorithm has processed all the nodes, you merely check that the `next` pointer isn't the `curr`. If it is, then you've gone all the way around.

This style of list linking is good for situations that don't need a special place to start and where all the information is just as important as one another. It can be hard to do with a singly linked list, but is pretty easy with a doubly linked list. (It's the insert in front of `curr` that makes the single linked version difficult. Re-linking the end to the new `curr` is tedious.)

### 17.4.3 Two Dimensional?!

If you cross two linked lists at a single node — one 'left-right' and the other 'top-down', then you have what is sometimes called a mesh. This mesh is interesting because it can easily be sparse. That is, you can skip nodes in the grid that is formed if they are unimportant or default to certain values like 0s. This can be used to good stead in applications where there are many such data to cut down on the memory a 2D array would have taken up storing all those unnecessary data!

## 17.5 Wrap Up

To sum up, linked lists aim to improve on the insertion and removal speeds of arrays at the cost of some trouble traveling from item to item. The single `next` pointer can be augmented with another to form a more flexible search and improve the management of searching/processing the list. If the application calls for it, you can even make such a list circular in nature!

# Chapter 18

# Stacks & Queues

Not all data structures are general storage techniques like the linked list or array. Stacks and queues have very restricted access patterns that make them suitable for special situations in algorithm design, for instance. Let's examine each and their potential applications in turn.

## 18.1   Stacks

The idea of a stack comes from the dispenser for plates at a cafeteria. This mechanism is a set of springs holding a flat metal piece on which the plates are placed. As more plates are added, their weight stretches the springs and the earlier plates are pushed down into a metal cylinder or 'well'. Then, as patrons take plates off the top of this stack of plates, the springs pull back up and raise the plates toward the top of the well.

The bleak reality of it, however, is that the raising of the plates is not smooth like it would be in a physics experiment or calculation. Springs in the real world wear and distend. They gain a shuttering or jarring effect. Thus, the plates don't rise smoothly to the top but in jumps. So you might take a plate or even two before the spring raises the plates at all, in fact. Then, suddenly, the next plate causes the stack to pop up toward the top of the well.

### 18.1.1   Basic Operations

This imagery caused us to name the basic operations on a stack data structure push and pop. We push new data onto the stack for storage and pop data off the stack when we need to process it. Both of these operations happen at the same place in the storage — the designated top of the stack. We often give looking at the top element without removing it as an extra operation, in fact.

Other operations that are sometimes given in a stack data structure are empty and maybe full, size/count, and clear/erase. The last would just remove all elements from the stack making it empty.

Note that none of these operations will allow the programmer using the stack to delve below the top element! There is no access or removal other than at the top. This mimics the workings of a stack of

books, which I think most of us know will topple over into a right mess if you try to pull out a book from the middle or bottom!

## 18.1.2  Visualization

In discussions of stacks, we often diagram them like so:

```
|___|   |___|
| a |   | b |
|---|   |---|
        | a |
        |---|
```
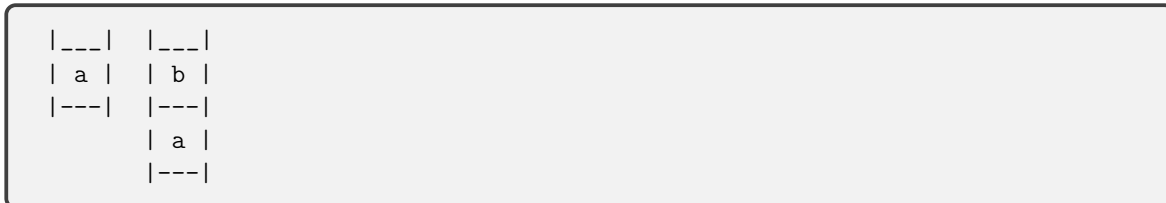
Here we see a stack holding a value a before and then the stack again after a value b has been pushed on as well. The diagram resembles a vertical array with an open cell at the top.

## 18.1.3  Implementation Details

It turns out that it is quite simple to implement a stack data structure in either an array or linked form. We'll explore these one at a time.

### 18.1.3.1  Static

The diagram above leads one to easily view a stack as an array-based structure. We have some flexibility as to implementing the `push` operation. We can either initialize `top` to 0 and code `push` as "store and then increment `top`" or initialize `top` to -1 and implement `push` as "increment `top` and then store".

Once this is decided, we'll know how to code `pop` as it undoes the `push` operation. We'll also know how to code the operation that peeks at the `top` element — either at `top` for the latter method or at `top-1` for the former. Further, we will know how to implement `empty` and `full`.[1] `empty` would be either `top==-1` for the latter or `top==0` for the former. And `full` would be `top==size` or `top+1==size` respectively.

Maybe a little table would help:

| Operation \ Initialization | top=0 | top=-1 |
|---|---|---|
| push | store then increment | increment then store |
| pop | decrement then copy | copy then decrement |
| get_top | top-1 | top |
| empty | top==0 | top==-1 |
| full | top==size | top+1==size |

Either way we initialize, we end up with the `top` at the end of the array and there is instantaneous access to this location so all operations are $O(1)$.

### 18.1.3.2  Dynamic

If we want similar performance from our linked implementation, we'll want to reconsider the location of the `top` element — at least for a singly-linked list. If we move the `top` to the `head` of the list, it will always have instantaneous access as well. And inserting before the `head` is $O(1)$ as well and so is removal of the `head`. `empty` becomes a `nullptr` check. And `full` always returns `false`.

All in all, this implementation is even easier than the array because there's no confusion as to the initialization of `top`.

---

[1]Although `full` will be always `false` if our array can grow dynamically.

But remember that your choice may be made by the memory system available to you at run-time in your application and deployment platform.

### 18.1.4  Applications

There are many applications of the stack data structure.  Here are a few basic ones:

- the function call stack follows the rules of the stack data structure

- used with a loop to emulate recursion without the fear of call stack overflow (your data stack can be on the heap and should grow pretty near infinitely)

- palindrome checking can be done with a stack: `push` half the string onto the stack and compare as you `pop` those elements off but continue forward through the string

- parenthesis — or other bounding symbol — matching: `push` open symbols on the stack and `pop` when a matching close symbol is seen; if the stack ends up `empty`, then the symbols were balanced

- evaluation of mathematical expressions can be effected with one or two stacks (either just operand stack or operand and operator stacks depending on the expression entry method); we'll see more on this near the end of chapter 19

#### 18.1.4.1  Emulating Recursion

The use of stacks to emulate recursion bears some examination as it is an oft-used technique.  Let's look at a simple case to start: printing a linked list backwards non-recursively.  This is basically:

```
Stack<LinkedListNode<blah>*> aux;

t = head;
while ( t != nullptr )
{
    aux.push(t);
    t = t->next;
}

while ( ! aux.empty() )
{
    aux.pop(t);
    cout << t->data << ' ';
}
```

Here we've created an auxiliary stack data structure cleverly named `aux` to store all the node pointers. These reside from `top` to bottom of the stack in reverse order since the last one `pushed` was the `tail` of the linked list and the first one `pushed` was the `head` of the list.

Then we start a loop to `pop` each pointer off and print its `data`.  This continues until the stack becomes `empty`. Again, the pointers in the stack were in reverse order from how they are actually linked in the list. So this prints the elements from the nodes in reverse.

### 18.1.5  Nomenclature

Because of their general processing characteristics — as discussed in the previous section, stacks are often called FILO or LIFO structures (first-in-last-out or last-in-first-out).  These names are often used in place of 'stack' when discussing designs for software and algorithms. You'll want to recognize them if not use them yourself.

## 18.2   Queues

The queue data structure is named for the British name for lining up at a checkout or event. Instead of a line and lining up, they say you are in queue and queuing up.[2]

The basic idea, of course, is that you enter the queue from the end or rear and exit the queue at the front. There are no backsies or cutsies in the data structure, of course.

### 18.2.1   Properties and Operations

The basic data structure needs two core operations much like the stack did: putting data in and taking data out. These are called enque and deque respectively.[3]

In addition to these two basic operations, we also typically provide the usual gamut of empty, full, count, and maybe a peek at next operation.

When depicting a queue, it might look something like this:

```
-------------------
 | a | b | c | d |
-------------------
```

Here the front or head of the queue is on the left and the rear or tail is on the right. This may, of course, change with cultural norms around the world — not sure on that. If we added an e to the queue, for instance, it would then look like so:

```
-----------------------
 | a | b | c | d | e |
-----------------------
```
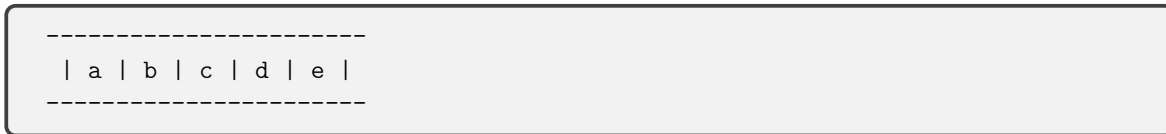
And if we remove a for processing, the queue would look like so:

```
-------------------
 | b | c | d | e |
-------------------
```

### 18.2.2   Implementation Details

As before, we can implement a queue in either an array or in a dynamic linking fashion with pointers. We'll explore each in turn.

#### 18.2.2.1   Static

Using an array — dynamic or not — is not so straight forward as it might at first seem. We'll focus on a fixed-size array because dynamic growth is just overhead here and the problems we will face are not aided by it.

##### 18.2.2.1.1   A First Attempt

Our first attempt will be the naive approach. Here we would just try like we did with the stack to use the array normally with the `head` and `tail` starting at 0 and the `tail` moving forward during each `enque`.

---

[2]There is a mildly interesting history here involving heraldry and tails as the term migrated from Latin to French and finally landed in English.
[3]Some modern library implementations like that in C++ have changed this to `push` and `pop`. I'm going to stick with the traditional names here, though.

But we see from a quick case study that this is not going to work in the long run. Look at this diagram, for instance:

```
head              tail
   0   1   2   3   4
   ------------------
   | a | b | c | d |
   ------------------
```

Here the queue is `full` and can't take any more `enque`s.[4] If we then start to process the data, `head` will advance forward as well and come to this situation:

```
                  head
                  tail
   0   1   2   3   4
   ------------------
   | - | - | - | - |
   ------------------
```

Although the data are still physically there, I've dashed them through to remind us that they are no longer there logically — they've been `deque`d and processed. But now the queue is `empty` again yet we have no where to `enque` new data coming in!

How can we fix this problem? We must be able to implement a queue with an array!


### 18.2.2.1.2  Modulo to the Rescue

It turns out that modulo will aid us greatly here. We need to take advantage of all that space in front of the `head` that we lost while `deque`'ing. We could, of course, do an `if` and subtract the array size each time we overshot, but that would get troublesome. Instead we use the power of modulo arithmetic to wrap the positions of `head` and `tail` as needed around from the size of the array to 0. (Remember that our array is of a fixed size. Growing it dynamically would just waste more space.)

Our structure might now look like the diagram at right. This is our concept of the `full` position where `head` is the first bit of data and `tail` is just past the last. The reason `tail` is past the last is the same as for the stack — we store and then increment when we are initializing the `head` to 0.
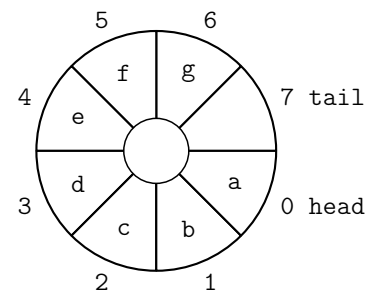
Why, then, is this `full`? Isn't there another slot to use? Sure, but think of what would happen if we used that. We'd store and then increment `tail`. In this modulo variation, all increments are done under a modulo by the size of the array (8 here). So `(7+1)%8==0` and we would be at the same position as `head`. What happens when `tail==head`? That's right: the queue is empty! This would mean we were full of data but had none at the same time. Not cool. Thus, our concept of `full` needs to change to `(tail+1)%size==head`.


### 18.2.2.1.3  Operations Summary

Here is a table summarizing the operations. I hope you find it helpful.

---

[4]Note how `tail` was pushed off the edge of the data with an increment after store. This is the same as in the stack when initializing to 0.

| Initialization <br> Operation | tail=head=0 |
|---|---|
| enque | store then increment tail |
| deque | copy then increment head |
| peek_front | head |
| empty | tail==head |
| full | (tail+1)%size==head |

And remember that those incrementations are to be done modulo the size of the array!

#### 18.2.2.2 Dynamic

Dynamic implementation is actually quite easy. We just utilize a doubly-linked list pattern so that tail operations are faster and we're there! The enque is simply inserting at the tail. deque is removing from the head. We'd do a nullptr check for empty. And full would always return false. Just remember to keep a size_t in the class that increments on enque and decrements on deque for use in the count operation.

### 18.2.3 Applications

Queues have many applications in the real world. Let's just talk to a few here:

- print jobs line up as they arrive to be processed

- email routing would most likely be performed in a first-come-first-served basis: a queue

- processes/threads might queue up to await being run through the CPU; this would depend on the OS, of course

- the keyboard buffer is a circular queue (15 characters by default, but upgraded by most modern OSes)

### 18.2.4 Nomenclature

Because of their general processing characteristics, queues are often called a FIFO or LILO structure (first-in-first-out or last-in-last-out). As with stacks, these designations are often used in discussion to mix it up a bit and should be remembered and recognized alongside the official name 'queue'.

## 18.3 Inheritance from List

We talked about using a linked list pattern for implementing both stacks and queues earlier. To make it an even simpler job, you can not only use composition to create the stack or queue with a linked list member but even use either private or protected inheritance from a linked list to make your stack or queue class.

What this would do would be to give you all the linked list goodness within your class but hidden from those using you in their program. The linked list functions would be your protected or private ones now! And your public functions could easily call the right ones from the linked list to do the job without any fear that a programmer using your class could mess up the rules and get to the middle of your data!

## 18.4 Wrap Up

In this chapter we've learned about two data structures that adapt or specialize the list idea from the previous chapter (17) to new purposes. The stack data structure models a stack of information where only the top is accessible at any given time. And the queue models a line of patrons waiting to see

a movie or buy their groceries such that new data comes in to the end of the line and older data are processed from the front of the line.  These two specializations are quite useful in real-world situations.

# Chapter 19

# Trees

We are familiar with trees in computer science from our experiences with inheritance (back in chapter 13). They are typically drawn with a root at the top and leaves dangling off the bottom of the diagram. It's just that with the data structure, we are dealing with data in each branching junction instead of a `class`.

## 19.1  General Properties

To the right is a small tree to start our discussion. The value a is at the top or root of the tree. It has a left branch to the value e and a right branch to the value c. Each of these values is also known as a child of a and a is likewise known as their parent. Just like with inheritance, you see, we use both tree and genealogical terminology just haphazardly. We also intersperse computer terminology such as that the circles with the data in them — where the branches meet — are called nodes.
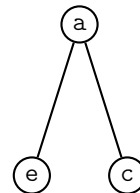
We like to talk about many physical properties of the tree from time to time, so we'd best take care to define them here at the outset. Let's first define the height of a tree. The tree above is of height 2 because there are 2 levels of data in it. If the tree stopped at just the value a and it had no children, it would be of height 1. But the level need not be full to count in the height. If either e or c were not there but the other were, the tree would still be of height 2.

Similarly, each node in a tree has a property called depth. The root is considered to be at depth 0. It's children are each at depth 1. Etc. We also use this numbering system to number the levels, typically. So all depth $d$ nodes are on level $d$.

A tree also has a branching factor[1] that tells how many children each node is allowed to have. The tree above was at least branching factor 2 since a had two children. If it were actually 2, then we would call this a binary tree. Other arities exist such as ternary or branching factor 3, quadrary or branching

---

[1]Sometimes this is called the arity of the nodes just like we spoke of arity in `operator`s earlier — chapter 11.

factor 4, etc. But past ternary, most people just use the branching factor, a hyphen, and the suffix "ary" such as 4-ary for a tree that can have as many as four children per node.

As it turns out, a $b$-ary tree can have $b^d$ nodes on level $d$. If we sum this up for all possible $d$ values like so:

$$\sum_{d=0}^{h-1} b^d$$

We can find out how many possible nodes there can be in a tree of height $h$. If this tree is binary, we can calculate the sum easily and get $2^h - 1$ nodes possible in a binary tree of height $h$. Similarly, in a tree with $n$ nodes, the minimal height of a binary tree is:

$$\lfloor \log_2(n) \rfloor + 1$$

A tree where every node with children has all possible children is called full. A tree that is full to its penultimate level and — at least in computer science — has all children on the last level filled in from the left side toward the right without skipping a spot is called complete.

We are also sometimes concerned with how well balanced a tree is. This measure requires us to look at each node in turn and check the heights of the 'trees' rooted at its children. If these subtrees have heights within 1 of each other, then their parent node is considered balanced. If they differ by 2 or more, then it is unbalanced. (A missing subtree — one where the child doesn't exist — counts as having height 0.)

The idea that any node in a tree can be seen as the root of a subtree itself leads one to believe that recursion will be used to process trees somewhat heavily. This is, indeed, true!

Interestingly, there is an algorithm to transform any arity of tree into a binary tree without loss of relative connections between nodes. This is a topic, however, for a subsequent full course in data structures. But because of this algorithm, we'll mostly be studying binary trees.

## 19.2 Implementation Details

As before, we can implement trees as either dynamically-linked structures or in array form. The dynamically-linked structure will be similar to a doubly-linked list in that it will have two pointers for the children, but they will be pointing down and out instead of off to the sides. So the tree is really a two dimensional linked structure instead of a linear one. This will become evident as we explore larger trees than just a height of two like the one before.

### 19.2.1 Dynamic

Below is a sample of code to implement a binary tree in C++. I've gone the classic route with separate node and tree `class`es. (Another approach would be a recursive design with just a tree `class`.) An option here would be to make the node `class` a `private` nested `class` inside the tree itself. This would hide implementation details and make your design more robust to changes down the line.

In memory, this could look something like the figure to the right of the code below. Here we see a memory version of the tree from the properties section above. There would, of course, be another pointer from the `Tree` `class` itself pointing to the `a` node as the `root` of the tree.

```
template <typename DataT>
class TreeNode
{
    DataT data;
    TreeNode * left, * right;
public:
    // need some ctors... but we do NOT need
    // the big three:  NO dtor, copy ctor, or
    // op= -- have those default:
    TreeNode(const TreeNode & tn) = default;
    TreeNode & operator=(const TreeNode & tn)
        = default;
    void ~TreeNode(void) = default;
    // as with a list node, the tree itself
    // will handle our dynamic [de]allocation
    // accessors, mutators
};

template <typename DataT>
class Tree
{
    TreeNode<DataT> * root;
public:
    // need ctors, dtor, op=
    // insert, remove, locate, ...
};
```
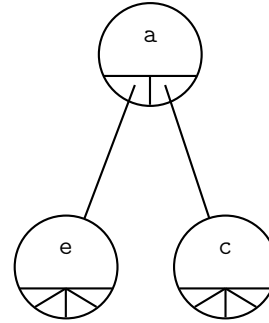
### 19.2.2  Static

To store a tree in an array, we simply map the nodes to positions within the array like so:

- the `root` goes into position 0

- for each node — including the `root` — place that node's `left` child in position $2*p+1$ where the node is at position p; place the node's `right` child into position $2*p+2$

- repeat previous step until all nodes are placed

This would map the above tree to an array like so:

```
------------------
| a | e | c | - |
------------------
  0   1   2   3
```

The dash in position 3 represents that no data is there yet and the extra dashes on the border at the right end represent that the array may grow in future to hold more data.

There is one more useful formula as well: the parent of the node at position c is located at position $(c-1)/2$. Note that this is integer division and will truncate odd values!

All of the above works best if the tree is complete so that there are no gaps in the array where we, for instance, skip position 3 here and put data in position 4:

```
-------------------------
| a | e | c | - | q | - |
-------------------------
  0   1   2   3   4   5
```

It can be done, but it is just icky! With `char` it isn't too bad as we can store some unused character like `'\0'` in the skipped spot, but other data types don't have such a 'null' value to use, typically.[2]

## 19.3 Traversals

Much of what we do with trees involves the process of traversing — or visiting all nodes within — the tree to perform some action. Even the above steps to place a tree into an array can be seen as a breadth-first traversal application.

There are essentially four classic traversal algorithms: pre-order, in-order, post-order (aka depth-first), and breadth-first. Most use recursion, but not all!

Further, the 'order' traversals all share a common trait, too. They always visit the current node's children from left-to-right. The reason they are different is the order in which they process the node itself. When the parent/current node is processed before the children, it is a pre-order traversal. When the parent is processed after the children, it is a post-order traversal. And when the parent is processed in between the children, it is an in-order traversal.

As we explore each of the four in turn, we'll practice on this tree:
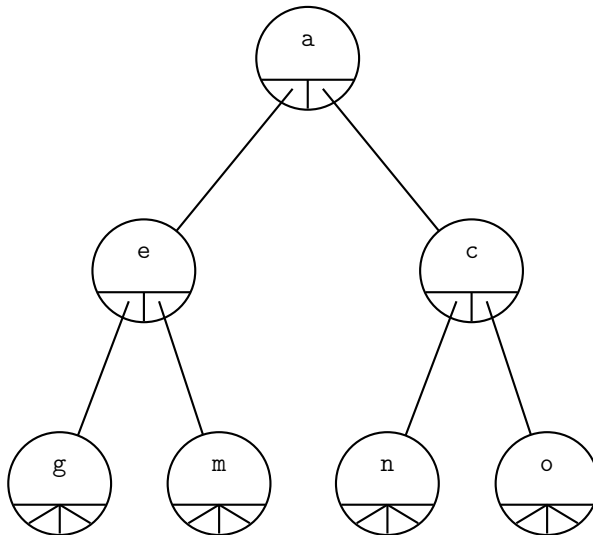


### 19.3.1 Pre-Order Traversal

Pre-order traversal is easily defined recursively like so:

```cpp
template <typename DataT>
void print_pre_order(TreeNode<DataT> * root)
{
    if ( root != nullptr )
    {
        root->Data().print();
        print_pre_order(root->Left());
```

---

[2]Well, floating-point types have `NaN`, but integers are out of it as is `bool`.

```
            print_pre_order(root->Right());
        }
        return;
}
```

This version always prints the data in the node using the `Data` accessor and a `print` method. Note that the incoming `root`'s data is processed first and then we move to the left and right children in that order. It is tempting to check the left and right pointers for a `nullptr` value before recursing, but that is over-optimizing. The base-case check will catch that situation during the recursive call — let it!

Printing the above tree with this function would result in the following result on screen:

```
a e g m c n o
```

If we wanted to make the processing more general, we could pass a processing function to the `template` as well:

```
template <typename DataT, typename ProcF>
void pre_order(TreeNode<DataT> * root, ProcF do_this)
{
    if ( root != nullptr )
    {
        do_this(root->Data());
        pre_order(root->Left(), do_this);
        pre_order(root->Right(), do_this);
    }
    return;
}
```

Remember, don't overthink the recursion too much. Just trust that it will do the job correctly. If we have a valid node, we process its data first and then recursively process the data of each child from left-to-right. That's the very definition of a pre-order traversal. We need do no more!

### 19.3.2   In-Order Traversal

To perform an in-order traversal of a tree, we do something like this:

```
template <typename DataT, typename ProcF>
void in_order(TreeNode<DataT> * root, ProcF do_this)
{
    if ( root != nullptr )
    {
        in_order(root->Left(), do_this);
        do_this(root->Data());
        in_order(root->Right(), do_this);
    }
    return;
}
```

Again, the left-to-right nature of the children processing is the same. Only the order in which the `root`'s data is processed has shifted.

If we passed a proper printing function to our `do_this` parameter, we'd see this on screen:

```
g e m a n c o
```

### 19.3.3 Post-Order Traversal

For post-order traversal, just shift the processing of the `root`'s data again:

```cpp
template <typename DataT, typename ProcF>
void post_order(TreeNode<DataT> * root, ProcF do_this)
{
    if ( root != nullptr )
    {
        post_order(root->Left(), do_this);
        post_order(root->Right(), do_this);
        do_this(root->Data());
    }
    return;
}
```

Now the data for the `root` node is being processed after both children have been handled.

This traversal is also known as depth-first traversal because it goes as deeply into the tree as possible before returning to the `root` for processing.

If we passed a proper printing function to our `do_this` parameter, we'd see this on screen:

```
g m e n o c a
```

### 19.3.4 Breadth-First Traversal

This traversal is different from the rest. It walks from node-to-node across each level from top-to-bottom in the tree. To do so is not recursive at all, you may realize, as we aren't processing a subtree but a level of siblings and cousins! To do this, we need the help of a queue data structure like so:

```cpp
template <typename DataT, typename ProcF>
void breadth_order(TreeNode<DataT> * root, ProcF do_this)
{
    if ( root != nullptr )
    {
        Queue<TreeNode<DataT>*> nodes;
        nodes.enque(root);
        while ( ! nodes.empty() )
        {
            root = nodes.deque();
            if ( root->Left() != nullptr )
            {
                nodes.enque(root->Left());
            }
            if ( root->Right() != nullptr )
            {
                nodes.enque(root->Right());
            }
            do_this(root->Data());
```

```
            }
        }
        return;
    }
```

Here we take the `root` provided and place it in the queue. Then we check that the `nodes` queue still has something in it and proceed to `deque` it. We examine the left child and right child in turn to see if they exist. If they do, we `enque` them one after the other. Then we process the `root` we just took from the queue.

As we loop back to the top, we check that there are more nodes in the queue. If so, we continue the process. If we were to have a `do_this` function for printing, this procedure applied to the test tree would look like this:

```
a e c g m n o
```

## 19.4   Derivative Data Structures

Although we haven't talked yet about adding things to the tree, we generally do that in special ways for special trees. These derivative data structures borrow from the tree basis and add what is called an invariant. The invariant is a property that must hold `true` at all nodes in the tree.

We'll explore three data structures derived from the tree below: binary search trees, heaps, and expression trees.

### 19.4.1   Binary Search Trees

The idea behind the binary search tree is to physically embody the binary search algorithm. It can search its contained data in logarithmic time when things are at their best. At worst, it will degrade to linear performance. We'll discuss how to avoid this below.

#### 19.4.1.1   Invariant

The invariant that helps the BST achieve its goal of embodying the binary search algorithm is that the data in the left child must be less than the parent's data and the data in the right child must be greater than the parent's data. This simple mantra keeps the tree organized such that we can quickly find any item in the tree or realize it isn't there.

Here is an example BST for us to explore:
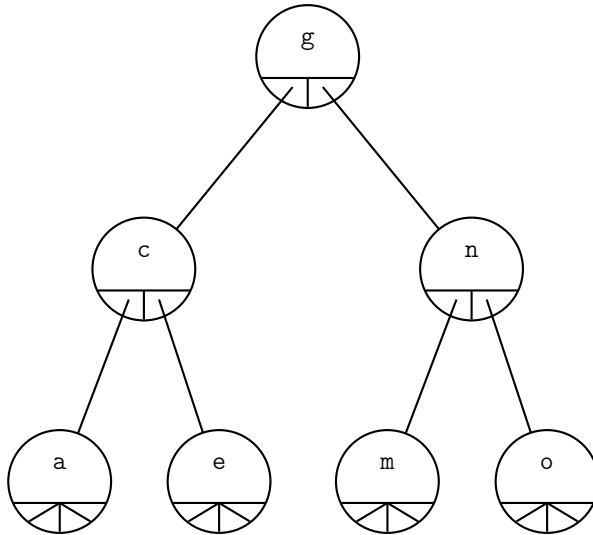
Note that c is less than g and n is greater than g. This pattern continues to the c subtree and the n one as well. In the cases of subtrees like a, we say that the invariant holds vacuously.[3] That is, there are no children, so the data there might as well match the invariant. Or, put a slightly different way, since there are no children, we can say anything we want about them.

### 19.4.1.2　Searching for Data

Let's test out the binary-search-ness of this structure and its invariant! If we start at the root and look for e, we first note that this value is less than g and so we move to the left branch. There we see that e is greater than c and so we move to the right branch. And there we find that e is found and we are done!

At each comparison, we eliminated half the remaining tree by moving either left or right. This is just like the binary search algorithm would do in a sorted array!

Even better, if we can't find something, we will know where to link the new node!

### 19.4.1.3　Inserting Data

Let's say we wanted to insert i into the above tree. Then we would start by looking for it in the tree. If it isn't already there, we'll add it. We'll discuss duplicate data later.

i is greater than g so we move right. i is less than n so we move left again. Finally, i is less than m so we move left. Here, though, we find a `nullptr`. So, we create a new node for i and link it to m as its left child. In code, this can look pretty straightforward:

```cpp
template <typename DataT>
void insert(TreeNode<DataT> * & root, const DataT & value)
{
    if ( root == nullptr )
    {
        root = new TreeNode<DataT>(value);
    }
    else if ( value < root->Data() )
    {
        insert(root->Left(), value);
    }
    else
```

---

[3]Vacuously here because there are no children — the set of children is empty or a vacuum.

```
    {
        insert(root->Right(), value);
    }
    return;
}
```

For this to work, the `Left` and `Right` accessors must return their pointers by reference so that we can modify them when we hit `nullptr` at the bottom of the tree. Note we are taking the `root` by reference.

Further, this function would be a `private` or `protected` member of the `Tree` `class`. There would be another overload of `insert` that took just a `value` to place in the tree in the `public` area of the `class`. It would call this one with the actual `root` of the tree.

#### 19.4.1.3.1 An Aside

The only potential problem here is that some compilers would complain at our use of `root` as the function parameter name claiming it shadowed the member variable. If this happens, just change one or the other. Changing the parameter will affect less of the `class`, of course, but either can work. Some typical changes would be to add `m_` to the member variable or to add `p_` to the parameter.

Another possibility is making the parameter name capitalized. This route isn't advised as it is prone to error! After all, we could easily change some of the `root` occurrences and miss one or two. Then we'd be altering/following the overall `root` instead of the `Root` parameter we'd intended to alter or follow!

#### 19.4.1.4 Removing Data

To remove data, you might think to modify the `insert` routine like so:

```
template <typename DataT>
void remove(TreeNode<DataT> * & root, const DataT & value)
{
    if ( root != nullptr )
    {
        if ( value < root->Data() )
        {
            remove(root->Left(), value);
        }
        else if ( value > root->Data() )
        {
            remove(root->Right(), value);
        }
        else
        {
            delete root;
            root = nullptr;
        }
    }
    return;
}
```

But this assumes that the value to be removed is a leaf of the tree![4] This assumption is poor, to say the least.

---

[4]A leaf is a terminal child with no further children. This is as opposed to an internal node that has more children below it.

Let's say we were removing the `c` from our earlier tree, for instance. Just `delete`'ing it would leave both `a` and `e` as orphans — a memory leak! We've gotta check in that `else` whether there are nodes below this one or not!

And what do we do when there are? We've gotta bring up one of the descendants to replace us that will preserve the invariant across the entire subtree. A daunting task at first glance. But it turns out to not be that bad. We just need to go either left and then as far right as possible or vice versa. For instance, `c` could be replaced by either `a` or `e` and still leave a correct subtree.

But let's test it further up. Let's remove `g` from the uber-root position. Here we can't just go one place left or right. `c` would certainly be less than everything to the right of `g`, but it wouldn't be greater than everything in the left tree: `e` is over there, after all! That's why we must go left once and then as far to the right as possible. This would take us to the right-most element in the left subtree which will be the greatest thing less than the node to be removed.

Similarly, if we prefer, we could go to the right one and as far left as possible from there. This would result in the least thing greater than the node being removed.

Which you choose is dependent on which child exists. For instance, if we were to remove `m` right now, you couldn't do the right and left as far as possible maneuver because `g` doesn't have a right child at the moment!

So, to fix our above red-framed code, we'd do something like this:

```cpp
template <typename DataT>
TreeNode<DataT> * & greatest_left(TreeNode<DataT> * root)
{
    TreeNode<Data> * & t = root->Left();
    while ( t != nullptr && t->Right() != nullptr )
    {
        t = t->Right();
    }
    return t;
}


template <typename DataT>
TreeNode<DataT> * & least_right(TreeNode<DataT> * root)
{
    TreeNode<Data> * & t = root->Right();
    while ( t != nullptr && t->Left() != nullptr )
    {
        t = t->Left();
    }
    return t;
}


template <typename DataT>
void remove(TreeNode<DataT> * & root, const DataT & value)
{
    if ( root != nullptr )                      // removal value not in tree
    {
        if ( value < root->Data() )             // move left -- it's less
        {
            remove(root->Left(), value);
        }
```

```cpp
        else if ( value > root->Data() )        // move right -- it's greater
        {
            remove(root->Right(), value);
        }
        else                                     // we found it!
        {
            TreeNode<DataT> * & left = greatest_left(root);
            TreeNode<DataT> * & right = least_right(root);
            if ( left != nullptr )
            {
                root->Data() = left->Data();   // move data up from descendant
                remove(left);                   // remove descendant node
            }
            else if ( right != nullptr )
            {
                root->Data() = right->Data();  // move data up from descendant
                remove(right);                  // remove descendant node
            }
            else
            {
                delete root;                    // remove us -- we're a leaf
                root = nullptr;                 // clean up pointer
            }
        }
    }
    return;
}
```

Of course now we've called a helper routine that removes a node regardless of content. This will be simpler but similar — it just won't check the data value. It will only move the proper descendant up and remove that moved node from the tree. In fact, it does just what the above `else` branch does! Let's move that out. . .

```cpp
template <typename DataT>
TreeNode<DataT> * & greatest_left(TreeNode<DataT> * root)
{
    TreeNode<Data> * & t = root->Left();
    while ( t != nullptr && t->Right() != nullptr )
    {
        t = t->Right();
    }
    return t;
}


template <typename DataT>
TreeNode<DataT> * & least_right(TreeNode<DataT> * root)
{
    TreeNode<Data> * & t = root->Right();
    while ( t != nullptr && t->Left() != nullptr )
    {
        t = t->Left();
    }
```

```cpp
        return t;
}


template <typename DataT>
void remove(TreeNode<DataT> * & root)
{
    if ( root != nullptr )                    // not necessary if we're careful
    {
        TreeNode<DataT> * & left = greatest_left(root);
        TreeNode<DataT> * & right = least_right(root);
        if ( left != nullptr )
        {
            root->Data() = left->Data();  // move data up from descendant
            remove(left);                 // remove descendant node
        }
        else if ( right != nullptr )
        {
            root->Data() = right->Data();  // move data up from descendant
            remove(right);                 // remove descendant node
        }
        else
        {
            delete root;                  // remove us -- we're a leaf
            root = nullptr;               // clean up pointer
        }
    }
    return;
}


template <typename DataT>
void remove(TreeNode<DataT> * & root, const DataT & value)
{
    if ( root != nullptr )                       // removal value may be in tree
    {
        if ( value < root->Data() )            // move left -- it's less
        {
            remove(root->Left(), value);
        }
        else if ( value > root->Data() )       // move right -- it's greater
        {
            remove(root->Right(), value);
        }
        else                                   // we found it!
        {
            remove(root);
        }
    }
    return;
}
```

Yea! No longer red-framed! Remember, these routines are all `private` or `protected`. Further, the one for a `value` would be called from a `public` routine that just took a `value` to be removed from its caller.

### 19.4.1.5   Achieving Balance

The performance of the binary search tree relies on the balance of the tree. If it strays from minimal height — a logarithm based on the number of nodes in the tree, if you recall — then it will veer toward linear performance on searching. (And since insertion and removal also depend on searching, they'll get slower, too.)

Every time we `insert` or `remove` data we alter the height of a subtree. Sometimes this is fine like the addition of `i` or the removal of `c` earlier. These didn't change the balance of the tree because they were just ±1 changes from already perfect balance.

But if we further add `j` to the tree as at right, we'd see things go off by 2:

Now the subtree at `m` has heights 2 and 0 for its subtrees and is imbalanced! We can also start to see things leaning right a little and heading toward a linked list effectively on that side. Definitely bad things ahead!
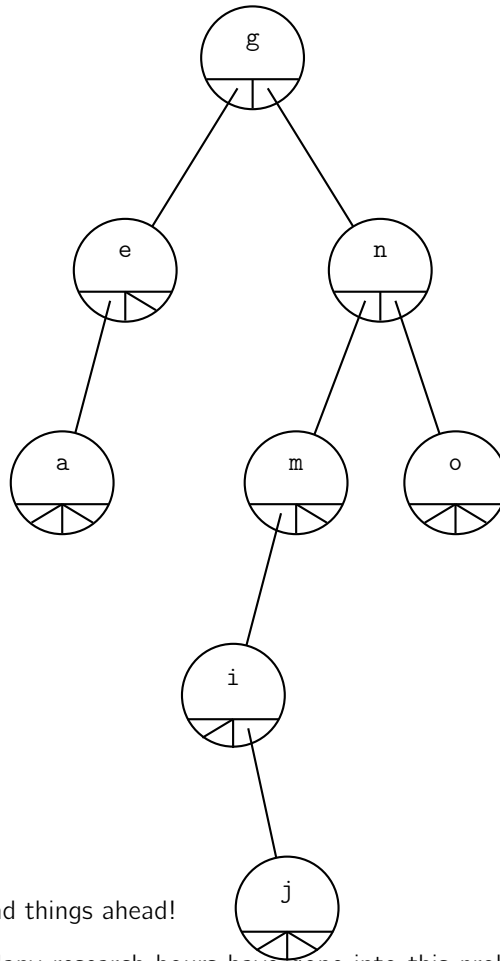
How can we stop this madness?! Many research hours have gone into this problem and there are many techniques to solve it. This area is known as self-balancing BSTs. Some notable algorithms in this field are AVL trees — named after the creators — which uses rotations to restore balance to a skewing tree and red-black trees which uses a node coloring scheme to rebalance. We'll study these further in a later course.

### 19.4.1.6   Handling Duplicate Data

The above all assumes unique data items. What if there are duplicate keys?[5] If the keys are all that there is to the data, just add a counter to the nodes to count how many copies of that key are in the tree.

If the key is one piece of data out of many that describe the data objects we store — like an id number in a student record, then make a nice linked list as an extra item in the nodes and prepend the new data object to the list as it arrives at the node. (Or you could append it if using a doubly-linked list...)

This linked list technique will require an adjustment to the search result of `true` or `false` as it is now. You'll have to search the linked list based on further information in the objects to see if you can find a more exact match. After all, they all share the same key we've been using to build the BST in the fist place. Maybe search on first name now since the id numbers are all equal somehow.[6]

To speed the search amongst duplicate data objects, you could make that linked structure a BST instead of a simple linked list. That would speed up searching on the secondary key value. (And if that key has clashes, make those nodes have BSTs on a tertiary key. Etc.)

---

[5]A key is simply a value on which you are searching.

[6]No self-respecting school would let there be duplicate id numbers, but similar things are bound to come up in real-world code, so be prepared!

#### 19.4.1.7 Sorting

A side benefit of the BST's method of storage is that a simple in-order walk through the tree will hit the data in sorted order! So if we happen to need to see the data in order, having placed it all in this tree isn't a detriment. (Some would have thought it so since we don't have a nice sorted array anymore, but that turns out to not be a problem!)

### 19.4.2 Heaps

The purpose of the heap data structure is to make locating the extrema of the data as easy as possible. It makes sure that the root of the tree is always the smallest thing in the whole tree. Well, at least for a min-heap or smallest heap. There is a second variation called a max-heap or largest heap which makes sure of the exact opposite.

#### 19.4.2.1 Invariant

The node invariant for a heap is that the node's data is smaller (or larger) than that of both of its children. Making sure this invariant doesn't change for any node in the tree makes sure the absolute smallest (or largest) thing in the tree is at the root. Also note that there is nothing implied about the relative order of the children of the node. They could be left greater than right or vice versa! This is very different from the BST above.

Here is a heap for us to work on in the upcoming sections. Notice how the invariant is held at each node but the left-right order of the children is not consistent.
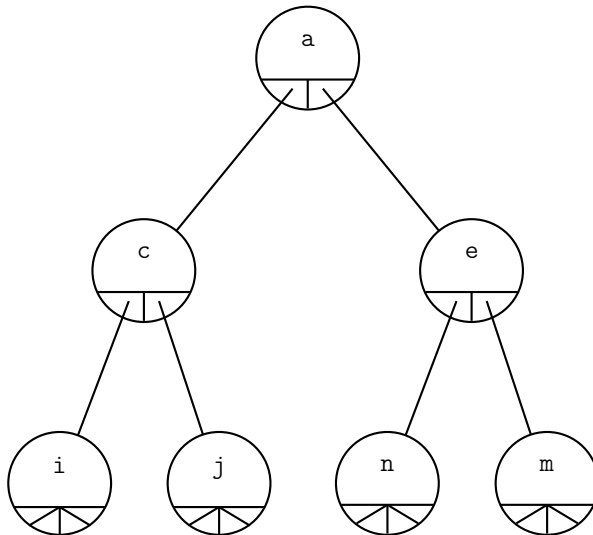


As the treatment of heaps is somewhat different in dynamically-linked form and array form, we'll need to visualize this in both ways. Here's the array version for your convenience:

```
--------------------------------
| a | c | e | i | j | n | m |
--------------------------------
  0   1   2   3   4   5   6
```
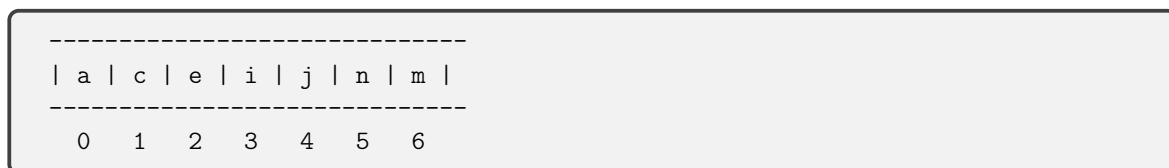
Note also that we will not talk about searching a heap as the only thing of interest in the whole tree is the root element.

#### 19.4.2.2 Inserting Data

Inserting a new piece of data is differently done on dynamically-linked and array-based heaps. Let's discuss each in turn.

### 19.4.2.2.1 Dynamic

The insertion point in a dynamically-linked heap needs to be the right-most position on the lowest level. This will keep the tree complete and therefore as short as possible. This position is found based on always going to the left subtree if both subtrees are either empty or the same in height and right otherwise. This relies on the height of a subtree being updated on the return trip up from the recursively determined insertion point. This is always one more than the maximum height of the children.

Don't forget to drop the new data into the right node on the way down!

```cpp
template <typename DataT>
inline size_t height(const TreeNode<DataT> * node)
{
    return node == nullptr ? 0 : node->Height();     // Height or 0
}


template <typename DataT>
void insert(TreeNode<DataT> * & root, const DataT & new_item)
{
    if ( root == nullptr )                           // found bottom
    {
        root = new TreeNode<DataT>{new_item};        // put it in
    }
    else
    {
        if ( new_item < root->Data() )               // should data go here?
        {
            swap(new_item, root->Data());            // replace and take
        }
        if ( root->Left() == nullptr )               // no left child
        {
            insert(root->Left(), new_item);          // insert left
        }
        else if ( root->Right() == nullptr )         // no right child
        {
            insert(root->Right(), new_item);         // insert right
        }
        else if ( root->Left()->Height() ==          // children are same
                    root->Right()->Height() )        // height
        {
            insert(root->Left(), new_item);          // insert left
        }
        else                                         // right must be shorter
        {
            insert(root->Right(), new_item);         // insert right
        }
        // update height after insertion
        root->Height() = max(height(root->Left()),   // taller child's
                            height(root->Right()))    // height
                        + 1;                          // plus one
    }
    return;
}
```

Again, these routines would be `private` or `protected` ones in the `Heap` `class` itself that was called

from a `public` method that just took the `new_item` to be inserted.

Some people call the above a bubble-down operation.

### 19.4.2.2.2  Static

Placing a new item in a heap that is in an array is done from the bottom up. We know exactly where the lower-right-most item is because it is at the end of the data in the array — which we've been tracking so as not to go off the end of the array or to know when to grow it if it is dynamic. Then, once that is done, we bubble the new data upward to its final resting position by swapping it with its parent as needed to maintain the invariant.

Let's add g to the heap by way of example. The data is added as the right-most node on the lowest level like so:

```
---------------------------------
| a | c | e | i | j | n | m | g |
---------------------------------
  0   1   2   3   4   5   6   7
```

Now we percolate it upward by checking its parent and sibling to see who is smaller. It has no sibling as it is a left child of i, so we just check it against i to find it is smaller. Then the two values are swapped and we follow this same procedure up to the parent node:

```
---------------------------------
| a | c | e | g | j | n | m | i |
---------------------------------
  0   1   2   3   4   5   6   7
```

This time we find that the parent — c — is already the smallest of the three and so we stop.

Since putting the new data into the right-most position on the lowest level keeps the tree complete, this keeps the height logarithmic in nature and the trek is at most a logarithm in length.

In code this process looks something like this (using a `vector` for convenience):

```cpp
template <typename DataT>
void insert(vector<DataT> & heap, const DataT & new_item)
{
    heap.push_back(new_item);
    auto child_pos{heap.size()-1},          // new item position
         parent{(child_pos-1)/2},           // its parent
         right{2*parent+2},                 // right child of parent
         left{right-1};                     // left child of parent
    child_pos = left;
    if ( right < heap.size() &&             // right exists and
         heap[left] > heap[right] )         // right is smaller
    {
        child_pos = right;                  // pick smaller child
    }

    while ( child_pos < heap.size() &&      // child exists and
            heap[parent] > heap[child_pos] ) // child is smaller
    {
        swap(heap[parent], heap[child_pos]);
```

```cpp
        child_pos = parent;                         // move up tree
        parent = (child_pos-1)/2;                   // get new parent
        child_pos = 2*parent+1;                     // get left child
        if ( child_pos+1 < heap.size() &&           // sibling exists and
             heap[child_pos] > heap[child_pos+1] )  // sibling is smaller
        {
            ++child_pos;
        }
    }
    return;
}
```

Some people call the above a bubble-up operation.

Because it walks up just a single path to the root of the tree and we've kept the tree complete, this is a logarithmic process.

### 19.4.2.3  Removing Data

Removal is always from the `root`. But once there is no value there, we must bring up values from below that preserve the invariant. This is going to take a different approach for dynamically-linked heaps and heaps stored in an array. We'll take each in turn...

### 19.4.2.3.1  Dynamic

To remove the `root` of a heap stored dynamically is a two-step process. First we need to find the right-most node on the lowest level and move its data to the `root` node. Then we bubble-down that new `root` value to its proper resting place. Let's look at the second step first, though, as it is the easiest.

```cpp
template <typename DataT>
DataT remove_min(TreeNode<DataT> * & root)
{
    DataT min;
    if ( root != nullptr )                          // somewhere to remove from
    {
        min = root->Data();                         // remember minimum value
        TreeNode<DataT> * bottom                    // remove right-most node
                {remove_lower_right(root)};         // from lowest level
        root->Data() = bottom->Data();              // copy data from there
        delete bottom;                              // clean up node memory
        TreeNode<DataT> * child{root->Left()};      // find smallest child
        if ( child != nullptr &&                    // we have left
             root->Right() != nullptr &&                // and right
             child->Data() > root->Right()->Data() )  // and right is less
        {
            child = root->Right();                      // move to right
        }
        while ( child != nullptr &&                 // we have a child
                root->Data() > child->Data() )      // and it is less than us
        {
            swap(root->Data(), child->Data());      // swap data
            root = child;                           // move down
            child = root->Left();                   // find smallest child
```

```
            if ( child != nullptr &&                // as above...
                 root->Right() != nullptr &&
                 child->Data() > root->Right()->Data() )
            {
                child = root->Right();
            }
        }
    }

    return min;                                    // return min (or default)
}
```

This routine would be `private` or `protected` in the `Heap` `class` as usual. It would be called by a `public` version that takes no argument and just returns our result passing us the uber-root.

But what of that `remove_lower_right` function? Let's tackle it next. It can be a slight modification of our `insert` routine from earlier which also sought out the lower-right-most node as an insertion point. But here we actually seek the node there instead of the next position after that. Thus the modification.

```
template <typename DataT>
TreeNode<DataT> * remove_lower_right(TreeNode<DataT> * & root)
{
    TreeNode<DataT> * ret_val;
    if ( root == nullptr )                          // nowhere to remove
    {
        ret_val = nullptr;                          // return nothing
    }
    else                                            // something is here
    {
        if ( root->Left() == nullptr &&             // both left and right
             root->Right() == nullptr )             // are empty
        {
            ret_val = root;                         // this is our target
            root = nullptr;                         // remove from tree
        }
        else if ( root->Right() == nullptr )        // only right is empty
        {
            ret_val = remove_lower_right(root->Left());  // look left
        }
        else if ( root->Left()->Height() ==         // both here and
                  root->Right()->Height() )         // of equal heights
        {
            ret_val = remove_lower_right(root->Right()); // look right
        }
        else                                        // left is taller
        {
            ret_val = remove_lower_right(root->Left());  // look left
        }
        // update height after removal
        root->Height() = max(height(root->Left()),  // taller child's
                             height(root->Right()))  // height
                         + 1;                        // plus one
    }
    return ret_val;
}
```

```
}
```

Note how this one updates the height of the subtrees after the removal on the way back up the recursive descent. Again, this should be a `private` or `protected` member of the `Heap` class.

Overall, we see that both steps are logarithmic as they walk down our complete tree even though one uses recursion and the other uses a loop. So the overall process is logarithmic as well.

### 19.4.2.3.2 Static

Here we'll follow the same algorithm as above, but it will be much easier to find the right-most node from the lowest level because of our calculable addresses.

```cpp
template <typename DataT>
DataT remove_min(vector<DataT> & heap)
{
    DataT result;
    if ( ! heap.empty() )
    {
        auto child_pos{heap.size()-1};              // get lower-right node
        result = heap[0];                           // get minimum
        heap[0] = heap[child_pos];                  // move lower-right to top
        heap.resize(heap.size()-1);                 // shrink away last position
        auto parent{child_pos};                     // a parent position
        parent = 0;                                 // reset to root
        child_pos = 1;                              // reset to root's left child
        if ( child_pos+1 < heap.size() &&           // right exists and
              heap[child_pos] > heap[child_pos+1] ) // right is smaller
        {
            ++child_pos;                            // pick smaller child
        }

        while ( child_pos < heap.size() &&      // child exists and
                heap[parent] > heap[child_pos] )    // child is smaller
        {
            swap(heap[parent], heap[child_pos]);
            parent = child_pos;                     // move down tree
            child_pos = 2*parent+1;                 // get left child
            if ( child_pos+1 < heap.size() &&           // sibling exists and
                  heap[child_pos] > heap[child_pos+1] ) // sibling is smaller
            {
                ++child_pos;                        // pick smaller child
            }
        }
    }
    return result;
}
```

There was one cheap trick there, I'll admit. I used `auto` to declare the `parent` position with initialization from `child_pos` and then reset it immediately to 0. This avoided the need to use a `size_type` which would have required a `typename` helper. Again, a little sneaky, but in some circumstances worth it.

#### 19.4.2.4   Building a Heap

Building a heap from raw data is done by calling the `insert` routine as the data arrives. This process is pretty straight forward and leads to a time analysis of $n \log(n)$. In addition, it doesn't need to change for either dynamically-linked heaps or for array-based heaps. Or does it..?

#### 19.4.2.4.1   Static Can be Faster

The above is considered a top-down build because we insert from the `root` position and things bubble down after that. But it turns out that putting the data in the array first and then rearranging them into a heap structure can be much faster — linear, in fact!

The idea is to start at the parent position of the last element and make sure that little collection of nodes is a heap — follows the heap invariant. Then proceed to the left along that level to do the same to all the other little subtrees there. As we finish that penultimate level, we move up to the level above on the right and continue the process. Even though this might make us reprocess the earlier heaps we built from the lower level, it ends up being linear overall instead of $n \log(n)$. Lucky for us, the proof of this result is beyond the scope of this volume. *smile*

What does this look like in code? Pretty straightforward, actually. Finding all those parent positions is just subtraction. A breadth-first search in an array is just a walk along the array, after all!

```cpp
template <typename DataT>
void add_root(vector<DataT> & heap,
              typename vector<DataT>::size_type new_root)
{
    auto active{new_root};                        // make active parent
    auto child{2*active+1};                       // get left child
    if ( child < heap.size() )                    // if child exists
    {
        if ( child+1 < heap.size() &&             // if sibling exists
            heap[child] > heap[child+1] )         // and is smaller
        {
            ++child;                              // pick smaller sibling
        }

        while ( child < heap.size() &&            // child exists and
                heap[active] > heap[child] )      // child is smaller
        {
            swap(heap[active], heap[child]);      // bubble down
            active = child;                       // move down heap
            child = 2*active+1;                   // get left child
            if ( child+1 < heap.size() &&         // sibling exists and
                heap[child] > heap[child+1] )     // is smaller
            {
                ++child;                          // pick smaller sibling
            }
        }
    }
    return;
}


template <typename DataT>
void build_heap(vector<DataT> & heap)
{
```

```
    for ( auto current{(heap.size()-1-1)/2}; // start at parent of last element
          current+1 > 0; --current )          // moving down to first element
    {
        add_root(heap, current);              // add new rooted subtree in heap
    }
}
```

Just one thing here merits a bit of note. Remember that all `size_types` are `unsigned` and so when we subtract 1 from 0 we don't go negative but around to the top and a very large positive value. Thus the test on the `for` loop not being simply `current>0` but, rather, `current+1>0`. This makes sure we didn't drop down below 0 to the very large positive value that is 1 less than 0. (Because adding 1 to it will take it back to 0, and not be greater than 0.)

The helper function `add_root` here looks awfully like our removal routine, though. That's because they both bubble-down a value. We could even use this new routine as a part of `remove_min` like so:

```
template <typename DataT>
DataT remove_min(vector<DataT> & heap)
{
    DataT result;
    if ( ! heap.empty() )
    {
        auto child_pos{heap.size()-1};        // get lower-right node
        result = heap[0];                     // get minimum
        heap[0] = heap[child_pos];            // move lower-right to top
        heap.resize(heap.size()-1);           // shrink away last position
        add_root(heap, 0);                    // bubble new root down
    }
    return result;
}
```

Nice!

### 19.4.2.5 Sorting

The heap data structure leads naturally to a sorting algorithm as well. It is called, unsurprisingly, heapsort. The basic idea is that you have the smallest element in the topmost position, so put that where you need it to be in the resulting sorted list — front or rear depending on your application needs — and then get the next and the next and so on... This process is overall $n \log(n)$ since the `remove_min` process is logarithmic and there are $n$ data to place.

The process destroys the heap we built, but gives us a container of data that is in sorted order, so it's probably worth it.

### 19.4.2.5.1 Dynamic

The above pretty much says it all for dynamically-linked heaps, but they are a bit different than the array-based ones as you'll see. Just to be clear, a min-heap will give you low-to-high order in your result and a max-heap will give you high-to-low order in the result. At least when using a dynamically-linked heap.

### 19.4.2.5.2 Static

Three things are noteworthy about doing the sorting process in an array-based heap. First, we typically leave the sorted result in the same array as the data was originally found. This is easy enough to do if

we just swap the `root` and that lower-right-most node's data when doing the `remove_min` operation and don't truly `resize` the container but just don't count those positions any more. This can be done with an extra parameter to `add_root` to denote the `size` of the container instead of relying on a method of the container. We'd typically do this in an array, anyway. But our examples above were all `vector` for convenience, so...

Second, I wanted to point out that using a min-heap will actually get you a high-to-low order result instead of vice versa. So you need to watch out since that isn't quite as sensible to most as with the dynamically-linked heap. (Same goes for max-heaps and making the result a low-to-high order.)

Third, heapsort on an array-based heap can be seen as selection sort — pick the next item to put in place and move it there. But heapsort is faster because it has a supporting data structure underneath instead of a linear search for the next smallest/largest like selection sort. That linear build of a heap on the data really pays off in $n \log(n)$ performance rather than quadratic!

### 19.4.3  Expression Trees

Trees can also be used in a really nice way to represent an arithmetic (or even algebraic with some other help) expression in your program. With this tree representation, you can display it back out, evaluate it, manipulate it for optimization — per a compiler, etc. Yes, this is the kind of thing compilers do for programs as well: representing at least the mathematical expressions internally as trees but possibly the whole program.

How is this done? The means of crafting a tree from a standard input is a bit advanced for this volume, but we still want to explore some of the uses of such trees, so let's proceed!

#### 19.4.3.1  Representation

We can represent a mathematical expression like $4 + 3 * 5$ in a tree like so:
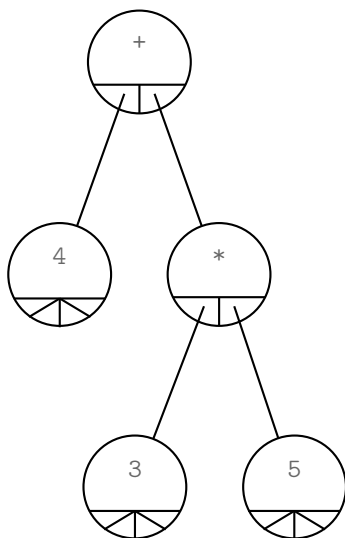


Note how the $*$ comes below the $+$. This is how precedence is represented in the tree — higher precedence operators come lower in the tree then lower precedence ones. If we have parentheses to change the order, it looks like this tree for $(4 + 3) * 5$:
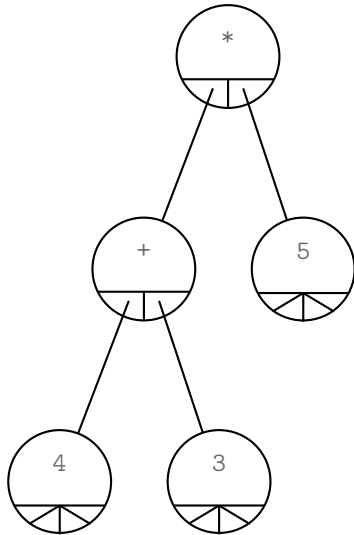
Note that the parentheses themselves don't appear in the tree. Their presence in the original expression is evidenced by the fact that + is now lower than ∗.

### 19.4.3.2 Traversal Results

There are three standard order traversals for trees and they all have useful results when applied to an expression tree. Let's tackle in-order first.

#### 19.4.3.2.1 Infix Expressions

An in-order traversal of an expression tree will result in an infix notation expression. This is the notation you are used to since kindergarten. *smile* With such a walk, the original expression is retrieved — mostly. Looking at this on the first tree above, we get the original expression of $4 + 3 * 5$ back out exactly.

But when we try it on the second tree, we get $4 + 3 * 5$ just like the other one. What gives?

Remember, the tree encodes parentheses as precedence levels within the tree and our standard in-order walk doesn't take that into account. So, for this kind of tree, we need some extra code to add an open parenthesis on the way down an edge when the precedence of the child is normally lower than that of the parent. And when unrecursing from that scenario, print a close parenthesis, too. This will give us the desired result.
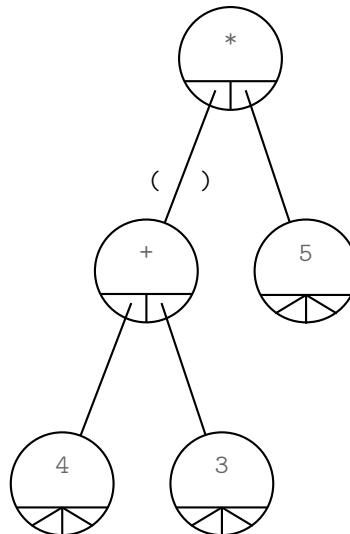


#### 19.4.3.2.2 Prefix Expressions

Another kind of walk is the pre-order walk. Printing in this approach leads to an expression representation known as prefix notation. Here the operators are before what they operate on instead of in between like with the infix notation. The magic of prefix notation is that there are never parentheses required to change precedence. The order of the operators and operands says it all! Why didn't we use this instead all our lives? Why this sick obsession with parentheses? I can't tell you that. I just don't know!

But, let's look at a couple of these just to satisfy our curiosity. The first expression above that had no parentheses in infix form looks like this in prefix: $+$ 4 $*$ 3 5. But the expression with parentheses looks like this in prefix form: $*$ $+$ 4 3 5.

As you can see, there are no needed parentheses. The mere arrangement of the operators and operands makes things clear. In the first expression, for instance, the $+$ has to wait for a second operand until the $*$ is done. And in the second, the $*$ has to wait for the $+$ to be done before it has even a first operand.

### 19.4.3.2.3  Postfix Expressions

But prefix notation isn't the only magical notation requiring no parentheses. There is also postfix notation which results from a post-order traversal of the expression tree. The no-parentheses tree looks like 4 3 5 $*$ $+$ in postfix form and the parentheses tree looks like 4 3 $+$ 5 $*$. Here we hold onto operands until an operator comes along and then we use the top with the operator.

### 19.4.3.2.4  Nomenclature

Although poor from a modern cultural standpoint, if you are looking for further information on these techniques, it would be useful to know that prefix notation is also known as Polish notation and that postfix notation is also known as reverse Polish notation (RPN). They should have the name of the actual Polish mathematician that worked them up — Jan Lukasiewicz, but we just abbreviate with the name of his nationality.[7]

### 19.4.3.3  Evaluation

We can also use the tree form or its traversal results to evaluate an expression.

### 19.4.3.3.1  In-Tree

If we walk through a tree in post-order form, we can evaluate the original expression as we go. A number evaluates to itself and an operator works on the two children that just got evaluated. When you finish the root of the tree, you have the result for the whole expression!

Walking through the parentheses-less tree above, for instance, we get 4 first and hold onto it. Then we get 3 and 5 and hold them, too. When we finally see $*$, we take the two children and multiply them to get 15. Now we unrecurse to the $+$ where we retrieve the children who were 4 and 15 and evaluate to 19. This is made easy by the recursive nature and having a helper variable in the nodes that holds the node's evaluated value.

A similar walk through the parentheses-containing tree gives 35. Give it a try!

### 19.4.3.3.2  After Traversal

You can also do evaluations with either a prefix or postfix version of the tree. The prefix version takes two stacks: one for operands and one for operators. It also has some slightly complex rules for when to operate on operand stack elements. So we'll leave that to future study.

In a postfix version of an expression tree, you can make due with just a single stack for operands. As soon as an operator is seen, you work on the top two entries in the stack and push the answer back on the stack. At the end of the expression, you should have one entry on the stack and that's your answer!

Walking through the parentheses-less tree above we got, for instance, 4 3 5 $*$ $+$. Evaluating this with a helper operand stack, we see the steps:

---

[7]It turns out that mathematicians do this all the time. Another famous example is the so-called Chinese Remainder Theorem which should really be Sun Tzu's Remainder Theorem! You'll study this in discrete mathematics, almost certainly.

```
                |---|
                | 5 | \
         |---|     |---|   |- * \_    |---|
         | 3 |     | 3 | /        \_ | 15| \
   |---|     |---|     |---|               |---|   |- + \_    |---|
   | 4 |     | 4 |     | 4 |               | 4 | /        \_ | 19|
   +---+     +---+     +---+               +---+               +---+
```
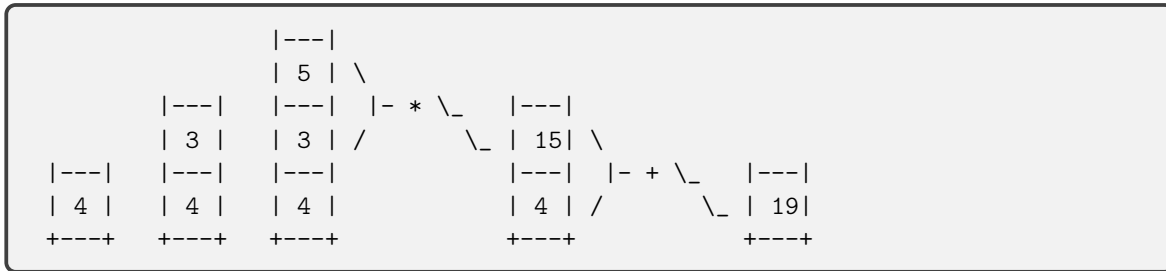
Evaluation of the parentheses-containing tree is left to the reader. (Hey, you've gotta practice these skills sometime! *smile*)

## 19.5   Wrap Up

This chapter has been a whirl-wind tour of tree data structures. We started with the general idea of a tree to store data and various mathematical properties such trees have. Then we moved on to looking at basic operations on general trees including traversals of various sorts.

Then we started looking at particular data structures based on the tree concept. Our first was binary search trees or BSTs. These let us search a set of data in logarithmic time if the tree stays balanced. But standard operations like inserting and removing could easily skew the tree. That'll lead us in future to code self-balancing trees like AVL trees. BSTs also give an automatic sort technique as well.

Next up we explored heaps and how they could be used to find the smallest or largest piece of data in a collection instantaneously. This led us to a very efficient sorting technique.

Finally, we explored some about expression trees which are key to compiled language performance. We looked at properties of these trees and how traversals led to different forms of mathematical notation. We even learned how to evaluate an expression tree in two different ways.

All-in-all, a glorious romp if not a complete one. That's right, there are more tree-based data structures out there to explore!

# Appendices

# Appendix A

# Setup

This appendix will help you set up your local environment for compiling C++ programs — an integrated development environment (IDE) — and a connection to a Unix server if your school provides one.

Rather than show a specific setup for each IDE for each platform, however, I'm going to advocate the blanket use of VS Code on all platforms. This will make things easier if you do decide or need to move back and forth between multiple platforms in your programming. It doesn't have its own compiler, though, so we'll have to install one of those alongside it on each platform.

The instructions below are broken down into sections based on your desired operating system. Feel free to scroll down or simply click here to jump straight to your OS: Windows, macOS, Linux, and ChromeOS.

## A.1    Windows

You've chosen your OS wisely. This is one of the most popular environments around. It is also very fun to work in as it has the most games available for it! (Wait, isn't that counter-productive?)

As to job potential, there are probably more Windows shops out there than anything else. So don't worry! Your time learning Windows skills will certainly be rewarded.

Now let's get you set up for programming!

### A.1.1    IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with Microsoft-provided tools, but we'll take a different route to give you a portable experience in case you have to work on other platforms in the future.

### A.1.1.1   The MinGW Diversion

We'll start by installing the MinGW compiler utilities via a suite called MSYS2. Scroll down to 'Instal-lation' and download the proffered `.exe` file. Don't worry with the signature check unless you are into that kind of thing. Run the installer, hit 'Next' twice to accept the defaults, and finally 'Install'. Finally click 'Finish' to exit the installer and run `MSYS2`.

Once in, you'll receive a little window with a couple of lines that say your computer's name and your user name and ends in a cute little dollar sign. This is a command window typical on a Unix machine — but you're getting one on Windows — congratulations! Here you can enter any basic Unix commands. We'll run the package manager: `pacman` — cute, eh?

To do so, type this:

```
pacman -Syu
```

at the dollar-prompt and hit Enter — note the capital 'S'! This will start a few parallel checks and download them after you hit Enter to accept the default 'Y' response.[1] If you want to more easily complete this process, you can pin `MSYS2` to the taskbar before hitting Enter a second time.

`MSYS2` will close after that is finished, but we need to do more in that dollar-prompt! Run `MSYS2` again from either the taskbar or your Start menu. Run the above command again to get other necessary components.

This time the `MSYS2` terminal doesn't close so you can go on to the next step. Now type:

```
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
```

(Sadly, even this simple command won't copy-paste well from the PDF. Be extra careful typing it!) When you hit Enter, many packages will be listed. You don't technically need them all, but I'd just accept the bunch for ease of use.

Now we need those tools in our Windows `PATH` — where Windows looks for executable commands by default. This can be done by running Control Panel, type 'edit path' in the 'Search' bar, and click the first resulting suggestion. Here we can double-click the 'Path' variable in the top window. A new window appears with all your current `PATH` entries listed out. Click 'New' to the right and enter:

```
C:\msys64\mingw64\bin
```

and hit Enter twice. (Note there are no spaces in there!)

Now go to the Start menu and run the command prompt (`Cmd`). At its terse greater-than-prompt, type:

```
g++ --version
```

to verify you have all you need for this book. (Don't worry, it actually depended on quite a few of the other installed items. It wasn't just that one we needed.) If it says command not found, we've got issues. Please talk to your instructor right away.

### A.1.1.2   VS Code Time!

Now go to the VS Code site and find the 'Download' button. (For me it was on the upper-right side of the screen just to the right of the Search field.) From the choices, choose the left-most one under the big Windows logo. This will get you the installer. It also takes your browser to a 'Getting Started' page

---

[1]Sometimes typing Y or y at this prompt will fail. It is not repeatable and they can't track it down.

just for Windows. We'll just use our own steps below, though. No need to read that now — or maybe ever. (But it can't hurt to bookmark it for later. . . )

But, before you go below, double-click the installer in its download location — or however you like to run a newly downloaded installer.

Click 'I agree...' and then 'Next' three times. I turned on the 'Create a desktop icon' box before my next 'Next' — up to you. Now click 'Install' and finally 'Finish'. This will run it and you can now proceed to section A.5. But don't forget to come back sometime and do the Unix software setup if you are using that at your school!

## A.1.2    Unix Server Connection

If you are working in a Unix environment at your school, then I've got the instructions for you here to make that happen. We'll need two tools: one for entering commands like compiling and execution of your programs and one for file transfer to either back up your programs from the Unix server or upload your local VS Code creations to the Unix server.

### A.1.2.1    Command Processing

We'll start here by downloading a nice terminal. Windows does include the app we need, but it isn't very pretty and Windows users like pretty, right? So we'll download a nice one. It's called `Putty`. You can get it from the author's site. Amongst other links and information, you'll see the link next to 'Download' that says 'Stable'. Click that. While there is a lot to the Putty suite of programs, we only need one file. Scroll down to the 'Alternative binary files' section and click the first 'putty.exe' link. This is the 64-bit version for a typical Intel machine. If you've got something different, you'll be used to picking the right thing or you can ask your instructor for help.

Once that's downloaded, move it from where it went (your `Downloads`, perhaps?) to the `Desktop` or somewhere easily accessible. You can also make it part of your taskbar by right-clicking its icon once you run it and selecting Pin to taskbar . Go ahead and run it now and we'll configure it a little.

There really isn't much to configure, but we'll save a session to make it easy for later. `Putty` actually starts in the 'Session' tab, so that's perfect. Now fill in the 'Host Name' bar with the name of your school's Unix server. This should be three words separated by dots and the last one is probably `edu`. The 'Port' field should already say 22 which is exactly what modern Unix servers want. This is for an SSH connection — the Secure SHell protocol. This makes all information sent across the connection encrypted for safety. (You'll also notice that SSH is selected in the radio buttons below the 'Host Name' and 'Port' bars.

Now type a name for your school in the 'Saved Sessions' line. Something you can remember what this is for even after you've collected several dozen Unix machine names. *grin* *chuckle* Just kidding. But make it a good name anyway. Once done, click the 'Save' button at the right side.

Right now you can connect to your Unix server with either an Enter or clicking 'Open' or by double-clicking the saved session name in the box below 'Saved Sessions'. Next time you use `Putty`, you'll probably just double-click to connect.

One other bit of configuration bears noting: the scrollback buffer. This is relatively short at first — just a couple of thousand lines. And some programs will give **LOTS** of errors and warnings. So we want to probably increase this to 10,000 or even 100,000. This setting is located in the 'Window' tab. To get there, look to the left and see the tree of tabs available. Click 'Window' and just below the middle of that tab is the 'Lines of scrollback' input. Just change it to some reasonably large value. No millions or anything! To save this change we have to return to the 'Session' tab, make sure your saved session is selected still, and click the 'Save' button. If your saved session wasn't still selected, you will probably have to 'Load' it, change the scrollback setting again, and return here to 'Save' that change.

Now, to test this out, you'll need to know your Unix server's fingerprints.[2] Your instructor or IT folk should have given them to you for reference. If you don't check them here you may subject yourself to a man-in-the-middle attack![3] This fingerprint check is only done for the first connection to the new server. Once you answer 'yes', they are saved with the session and always checked automatically from here in.

When prompted, enter your username for the school's Unix server at the 'login at' prompt and then your password at the next prompt. You hit Enter after each one — not Tab as some interfaces expect. Note that your password won't show up as you type for extra security. You'll have to be a really good typist and trust that you've done it right. *smile*

Once in, you'll receive a little window with a line that says your computer's name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine. Here you can enter any of the commands your instructor/IT staff gave you to edit, compile, or run programs on the Unix machine.

When you are done, type `exit` and hit Enter to log out of the connection.

### A.1.2.2   File Transfer

But someday you'll want to download those files you edited on the remote server or upload those you've developed in your VS Code environment locally. To do this you'll need a good file transfer program/app. The best one I've found for Windows is WinSCP. The SCP part is short for Secure Copy Protocol. It is also the typical command used in a Unix environment for secure copying of information — albeit in all lower case.

Once at the site, you can click the 'Download Now' button — it's big and green about the center of your window, perhaps. The information on the next page tells us lots of crazy details but note that it integrates well with `Putty`! That's nice. Again, hit the big green 'Download WinSCP' button. This time the download actually starts. Save it in a known place (your `Downloads`, perhaps?). Now go there and run it.

Whenever I have to install it afresh, I just click 'Accept' and then 'Next' twice and finally 'Install'. After a few files are installed, it notices you have `Putty` configured already and offers to import those sessions — say 'Yes' and then 'Okay'. When the final dialog comes up to 'Launch' and/or 'Open' the website, you can uncheck the website open button. I've got your back!

Clicking 'Finish' will run `WinSCP` and it opens to the connection window. There you'll see your `Putty` session and a 'New Site' icon. Double-click your session and you'll be prompted for your 'Username' and then for your 'Password'. If there's a window in between, your Unix administrators have set up a special message for those who connect to the machine. Just click the 'Never show this banner again' box and then click 'Continue'. At this point, you'll have two panels. The left one shows files from your own machine and the right one shows files from your Unix account.

The left one defaults to your `My documents` folder. This can be changed by double-clicking on the `..` entry at the top of the file list. This entry stands for the folder above the currently displayed one. If, on the other hand, you want to go into some other folder that is already shown, double-click that one, by all means. The same goes for the right panel. Once you've navigated one side or the other to the files you want to transfer, just click, Shift-click, or Ctrl-click the files to select them. Grab the selected files and drag them to the other panel and let go. Make sure you aren't hovering over a folder when you let go! That will transfer the files to that folder instead of the displayed folder.

One quick but important fact before we close down `WinSCP`: don't try to double-click a PDF to view it from inside `WinSCP`. It will try to view it as a plain text file which really is not the case! Instead either find it in the file manager or right-click and choose Open to launch it into your favorite PDF viewer.

To end the session, just click the corner X button. But this first time it'll bring up a dialog. Here we'll check 'Never ask this again' in the lower-left corner and click 'Yes'.

---

[2]Of course computers have fingerprints! People are always touching them! *chuckle*
[3]I actually like the moniker "monkey-in-the-middle attack", but it just isn't as popular for some reason.

Don't worry about finding `WinSCP` again, it saved an icon on your `Desktop` for your convenience. *smile*

## A.2   macOS

You've chosen your OS wisely. This is one of the most stable and secure environments around. It is easy to work with and has a sturdy Unix undercarriage.

Now let's get you set up for programming!

### A.2.1   IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with Apple-provided tools, but we'll take it a step further to give you a portable experience in case you have to work on other platforms in the future.

#### A.2.1.1   The Xcode Diversion

First install Xcode from the App Store. I know, I know: I said we would be doing VS Code. But remember, it doesn't have a compiler of its own. And do you know how you get a compiler on a Mac? That's right: you install Xcode. It's annoying, but it works.

The icon for Xcode is an A like the App Store's but in blueprint form — like we're building it, get it — with a hammer on top. In my search it was the second hit right after a story item called "What the Heck is Xcode?". Click the 'Get' button and wait. And wait. And wait. It has been known to take hours. I hope you got yourself some coffee and went potty before you clicked. Well, I suppose you have time to do it now. Go on, I'll wait for it for you.

There, now just drag that to the `Applications` folder for safe keeping. Now open a Terminal. This app, if you've never used it, is located in the `Utilities` folder inside the `Applications` folder. Once opened, you'll receive a little window with a line that says your computer's name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine.[4]

Now type the following at this dollar-prompt:

```
xcode-select --install
```

This should start installing the command-line tools for Xcode. These are what we need to run underneath VS Code. But don't worry, they don't take nearly as long as Xcode itself did.

If you had already installed Xcode and done something from the command-line, this step would report an error asking you to run a Software Update. In that case, you'd have to log in — with your Apple user name and password — at Apple's developer site. This would give you access to download the latest set of command-line tools. Click 'Account' in the upper-right-ish of the screen. Log in. Verify yourself with two-factor authentication. And then you can get the latest command-line tools. If you are told there are no operating systems to download even though you clearly just clicked the 'Download Tools' button, just click 'More' in the upper-right. This should bring you to Xcode and such. Scroll down a bit until you hit the "Command Line Tools for Xcode xx.x". The version number should agree with what you just got from the app store, if not, scroll until you find one that does. All older versions are available here for download.

The command-line tools are much smaller than the full Xcode, as I said. Just double-click the `.dmg` file[5] that results. This will do a verify and then open up. Double-click the `.pkg` file inside to install the command-line tools themselves. You'll probably have to give permission with your Mac password at some point.

---

[4]You did know your Mac was Unix under the hood, didn't you? *smile*
[5]I believe that stands for Disk iMaGe. It's like a little downloadable USB drive.

### A.2.1.2  VS Code Time!

Now go to the VS Code site and find the 'Download' button. (For me it was on the upper-right side of the screen just to the right of the Search field.) From the choices, choose the right-most one under the big Apple logo. This will get you the Universal binary that should run on Intel or M1 Macs just fine. It also takes your browser to a 'Getting Started' page just for Macs. We'll just use our own steps below, though. No need to read it now — or maybe ever. (But it can't hurt to bookmark it for later...)

But, before you go below, unzip the file you downloaded by double-clicking it. Drag the resulting app (with a blue ribbon on its icon) to your application tray for easy access. Now you can run it and proceed to section A.5. But don't forget to come back here to do the Unix software setup if your school uses such!

## A.2.2  Unix Server Connection

If you are working in a Unix environment at your school, then you are in luck by being on a Mac! The tools you need to connect to your school's Unix server are built in for command processing and you only have one download to make if you want to transfer files from your Mac to the server and/or back.

### A.2.2.1  Command Processing

We begin by opening a Terminal. This app, if you've never used it, is located in the `Utilities` folder inside the `Applications` folder. Once opened, you'll receive a little window with a line that says your computer's name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine.[6]

At this cute little dollar-prompt you can type commands to the computer like the `ssh` command. This is short for Secure SHell. It will give you a nearly identical dollar-prompt except it'll be one connected securely — encrypted — to a remote Unix machine. You just need the machine/host name for the school's Unix server and its fingerprints for verification.[7]

To connect to your school's Unix server, just type this at the command prompt:

```
ssh youraccount@hostname
```

Of course, you must replace `youraccount` and `hostname` with the name of your account on your school's server and that server's host/machine name respectively. When the connection has been established for the first time, `ssh` will prompt you with a set of fingerprints. Verify these against those given to you by your school's IT site or your instructor. If they don't match, you may be subject to a man-in-the-middle attack![8]

When prompted, enter your password for the school's Unix server. Note that your password won't show up as you type for extra security. You'll have to be a really good typist and trust that you've done it right. *smile*

Now you can type any commands your teacher told you about to edit, compile, or run files on the Unix system. If you need more help, I recommend finding a good Unix tutorial you like. I'm partial to text references, but lots of younger folk enjoy a good YouTube video. *shrug* To each their own...

To disconnect from the remote machine, just type:

```
exit
```

at the dollar-prompt. This will place you back in your own `Terminal` on your own Mac. Before you quit the `Terminal` app, though, you might want to pin it to your app tray if you'll be using it a lot.

---

[6]Yes, I said all this before, but some people don't read both parts!

[7]Of course computers have fingerprints! People are always touching them! *chuckle*

[8]I actually like the moniker "monkey-in-the-middle attack", but it just isn't as popular for some reason.

You can pin a currently running app by right-clicking its icon in the tray, selecting $\boxed{\text{Options}}$ and finally $\boxed{\text{Keep in Dock}}$. You can then drag it to your happy location amongst the other apps, if you want.

Once you're done, you can just $\boxed{\text{command}}$+$\boxed{\text{Q}}$ as usual to exit the app.

### A.2.2.2   File Transfer

But someday you'll want to download those files you edited on the remote server or upload those you've developed in your VS Code environment locally. To do this you'll need a good file transfer program/app. The easiest one I've found for my Mac is FileZilla. When you go there, you'll want the Client — not the Server. But don't grab the big green button for macOS right away! Instead, click the little text at the bottom that reads "Show additional download options". You want to do this because the green button gives you 'value-added' software add-ons. Once on the next page, you can click the first link for the `.tar.bz2` file. This is a compressed format similar to `.zip` which you may be more familiar with.

Once this downloads, double-click it to uncompress it. Then drag the resulting app to your app tray or `Applications` folder as you see fit. Now run the app and let's get configuring!

First let's declutter the interface a bit. Go to $\boxed{\text{View}}$ and click $\boxed{\text{Local directory tree}}$ to undo the checkmark by it. Then go back into $\boxed{\text{View}}$ — see the checkmark is gone? — and click $\boxed{\text{Remote directory tree}}$. Two panels just disappeared from the main window and that's less to look at. Depending on how much it annoys you, you can also go back into the $\boxed{\text{View}}$ menu and click to uncheck the $\boxed{\text{Transfer queue}}$ panel.

On the left side now is your local directory. I believe your home directory is the default, but it's been a long time since I was able to do a fresh install, so it may be your `Documents` folder now. You can see little folder icons next to any subfolders/subdirectories and little pieces of paper next to other files. At the top of the subfolders list is one called `..` — yes, just two dots. This represents the 'parent' directory of the current one — i.e. the one above this on the way to the `Macintosh HD` and then the machine itself! Sorry, got a little carried away. But, for instance, if you were in your `Documents` directory, you'd double-click `..` to go to your home directory.

Time to connect to the school's Unix server! Above the directory/file panel a little ways is the connection bar. This has places for the Host (aka host name or machine name), your username on the school's Unix server, your password there, and the port to connect with. You should have been given the first three by your teacher or IT personnel. The port will be 22 for an `ssh`-style connection. When used for file transfer like this one will be, we usually call it an `sftp` connection — Secure File Transfer Protocol.

Once you've filled in all the spots, click the 'Quickconnect' button. This should start a scroll of information between the connection bar and the file panel. You'll be prompted for two things: whether to save your password — I wouldn't, but it's your computer/choice — and to check the fingerprints of the remote machine. As with the `ssh` in the `Terminal` above, you should have been given fingerprints by your instructor or IT staff. Make sure it is right to avoid that monkey in the middle!

Once connected, you'll see your remote account's files in the right panel next to your local ones. Now you can navigate each side by double-clicking folders (including `..` if needed) to find your program source codes. Once you find them, just drag and drop them to the other side. If you let go over a folder, you'll place the files inside it instead of just where the other panel was at. I say 'files' plural because you can use $\boxed{\text{shift}}$-click or $\boxed{\text{command}}$-click to grab multiple files at once. `FileZilla` may ask you if it is okay to copy the files, if so, just check the "Don't ask again" box and say 'Okay'.[9]

When you are done, just $\boxed{\text{command}}$+$\boxed{\text{Q}}$ and next time you open up `FileZilla` it will remember where you've been. Click the drop-down arrow next to 'Quickconnect' to find your history. Just choose the one that starts with `sftp://` and has your user name and the school's host name in it and `FileZilla` will connect anew!

---

[9]Aren't those things annoying?!

# A.3  Linux

You've chosen your OS wisely. This is one of the most stable and secure environments around. It is also very educational for Computer Science students and fun to work in!

As to job potential, there are more Linux shops out there than people realize. So don't worry! You time learning Linux skills will almost certainly be rewarded.

Now let's get you set up for programming!

## A.3.1  IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with free Linux tools, but we'll take it a step further to give you a portable experience in case you have to work on other platforms in the future.

### A.3.1.1  The g++ Diversion

First we must install the actual Linux compiler for C++: g++.[10] We can do this from the command-line or from a graphical interface. The instructions for this vary from distribution to distribution, but I'll show you how it is done on my Ubuntu box and you can easily look up how to install packages on your own distro.

The graphical package installer (manager) on Ubuntu is Synaptic. (I had to look this up as I had taken it out of my app tray/launcher long ago.) Its icon is a box/package with a big green arrow pointing down in the corner. Click this to get started. You may have to enter your password to prove you are worthy to install things. If it doesn't give you this option, you might have to revert to the command-line. Once you get started, put 'g++' in the search and select the top hit. You do not need the `multilib` version at this juncture.

And on the command-line (see A.3.2.1 below for how to open a command prompt), we simply run these two commands one after the other:

```
sudo apt-get update
sudo apt-get install g++
```

and hit Enter when it asks to install other packages as well.

Once this is done, you'll have g++ installed and can make sure by running:

```
g++ --version
```

in the terminal. It should show 9 or higher on a typical system these days. 10 is a little better, but if it didn't come with your default package management system, you don't wanna take on installing it by hand!

Now it is time to put an IDE in front of this puppy!

### A.3.1.2  VS Code Time!

To get VS Code, you must visit the VS Code site. It isn't handled by the system package manager. (Although it will update automatically after it is installed!)

Once there, find the 'Download' button in the upper-right side of the screen just to the right of the Search field. From the choices, choose the middle one under the big Linux Tux logo (the penguin!). You'll actually have to choose if you are on a Debian-based system or a Red Hat style system. If you

---

[10]Well, there is also `clang++`, but g++ is the typical recommendation unless you need something particular.

don't know, you'll have to wing it and come back to get the other one should you fail this time around. It also takes your browser to a 'Getting Started' page just for Linux. We'll just use our own steps below, though. No need to read it now — or maybe ever. (But it can't hurt to bookmark it for later. . . )

Find where you downloaded it and either double-click it from your file manager or run this from an Ubuntu command-prompt:

```
sudo dpkg --install ...
```

Fill in the name of the downloaded file for the . . . here. To find the installed application can be daunting. On my Ubuntu box, for instance, when I hit the `command`/`Windows` key on my keyboard, a menu of apps comes up and I can search. Typing in 'code' — the executable name for VS Code — gives just the one hit. You can also type `code` into the terminal if you've just installed from there.

Now go below to the VS Code configuration section (A.5) for how to set this environment up. But don't forget to come back here and set up the Unix server connection software if your school uses such!

### A.3.2   Unix Server Connection

If you are working in a Unix environment at your school, then you are in luck by being on Linux! The tools you need to connect to your school's Unix server are built in for command processing and you only have one download to make if you want to transfer files from your Linux box to the server and/or back.

#### A.3.2.1   Command Processing

Now open a terminal. This app, like you've never used it, is located in various forms on various distributions. For instance, when I hit the `command`/`Windows` key on my keyboard, a menu of apps comes up and I can search. Typing in 'terminal' brings up some eight choices — and I don't think that is all of the ones that are on my system! Once you pick one and open it, you'll receive a little window with a line that says your computer's name and your user name and ends in a cute little dollar sign. This is a command window on a typical Unix machine.

At this cute little dollar-prompt you can type commands to the computer like the `ssh` command. This is short for Secure SHell. It will give you a nearly identical dollar-prompt except it'll be one connected securely — encrypted — to a remote Unix machine. You just need the machine/host name for the school's Unix server and its fingerprints for verification.[11]

To connect to your school's Unix server, just type this at the command prompt:

```
ssh youraccount@hostname
```

Of course, you must replace `youraccount` and `hostname` with the name of your account on your school's server and that server's host/machine name respectively. When the connection has been established for the first time, `ssh` will prompt you with a set of fingerprints. Verify these against those given to you by your school's IT site or your instructor. If they don't match, you may be subject to a man-in-the-middle attack![12]

When prompted, enter your password for the school's Unix server. Note that your password won't show up as you type for extra security. You'll have to be a really good typist and trust that you've done it right. *smile*

Now you can type any commands your teacher told you about to edit, compile, or run files on the Unix system. If you need more help, I recommend finding a good Unix tutorial you like. I'm partial to text references, but lots of younger folk enjoy a good YouTube video. *shrug* To each their own. . .

---

[11]Of course computers have fingerprints! People are always touching them! *chuckle*
[12]I actually like the moniker "monkey-in-the-middle attack", but it just isn't as popular for some reason.

To disconnect from the remote machine, just type:

```
exit
```

at the dollar-prompt. This will place you back in your own terminal on your own Linux box. Before you quit the terminal app, though, you might want to pin it to your app tray if you'll be using it a lot. You can pin a currently running app by right-clicking its icon in the tray, selecting Lock to Launcher . You can then drag it to your happy location amongst the other apps, if you want.

Once you're done, you can just 'exit' as you did above but this time to exit the app.

### A.3.2.2   File Transfer

But someday you'll want to download those files you edited on the remote server or upload those you've developed in your VS Code environment locally. To do this you'll need a good file transfer program/app. The easiest one I've found for my Linux machine is FileZilla. But don't go there except for help/reference. Use your package manager to download/install the app. If you like the graphical manager, just search for 'filezilla' and click 'Install'. If you like the command-line installer, the package name is typically 'filezilla' and it will install at least one other package for a helper library as well. On my Ubuntu box I ran:

```
sudo apt-get install filezilla
```

and then hit Enter to accept the extra package as well.

Next run the app from the command-line or the menu or a search. That all depends on how you like to do things and how you have your system configured. I ran mine from the command-line:

```
filezilla
```

Once it is running, you can right-click it in the tray and choose Lock to Launcher to keep it handy for later up/downloads.

Now let's get configuring!

First let's declutter the interface a bit. Go to View and click Local directory tree to undo the checkmark by it. Then go back into View — see the checkmark is gone? — and click Remote directory tree . Two panels just disappeared from the main window and that's less to look at. Depending on how much it annoys you, you can also go back into the View menu and click to uncheck the Transfer queue panel.

On the left side now is your local directory. I believe your home directory is the default, but it's been a long time since I was able to do a fresh install, so it may be your `Documents` folder now. You can see little folder icons next to any subfolders/subdirectories and little pieces of paper next to other files. At the top of the subfolders list is one called `. .` — yes, just two dots. This represents the 'parent' directory of the current one — i.e. the one above this on the way to the root directory (/)! Sorry, got a little carried away. But, for instance, if you were in your `Documents` directory, you'd double-click `. .` to go to your home directory.

Time to connect to the school's Unix server!  Above the directory/file panel a little ways is the connection bar. This has places for the Host (aka host name or machine name), your username on the school's Unix server, your password there, and the port to connect with. You should have been given the first three by your teacher or IT personnel. The port will be 22 for an `ssh`-style connection. When used for file transfer like this one will be, we usually call it an `sftp` connection — Secure File Transfer Protocol.

Once you've filled in all the spots, click the 'Quickconnect' button.  This should start a scroll of information between the connection bar and the file panel. You'll be prompted for two things: whether to save your password — I wouldn't, but it's your computer/choice — and to check the fingerprints of

the remote machine. As with the `ssh` in the `Terminal` above, you should have been given fingerprints by your instructor or IT staff. Make sure it is right to avoid that monkey in the middle!

Once connected, you'll see your remote account's files in the right panel next to your local ones. Now you can navigate each side by double-clicking folders (including .. if needed) to find your program source codes. Once you find them, just drag and drop them to the other side. If you let go over a folder, you'll place the files inside it instead of just where the other panel was at. I say 'files' plural because you can use Shift-click or Control-click to grab multiple files at once. `FileZilla` may ask you if it is okay to copy the files, if so, just check the "Don't ask again" box and say 'Okay'.[13]

When you are done, just Control + Q and next time you open up `FileZilla` it will remember where you've been. Click the drop-down arrow next to 'Quickconnect' to find your history. Just choose the one that starts with `sftp://` and has your user name and the school's host name in it and `FileZilla` will connect anew!

# A.4    ChromeOS

Was this a choice or was it forced on you by circumstances? Yeah, I thought so... Well, let's get you set up for programming!

## A.4.1    IDE

Most people, when working on coding, want a nice integrated development environment where they can do all their tasks at once. This can be done with basic Linux tools — even on ChromeOS, but we'll take a different route to give you a portable experience in case you get to work on other platforms in the future.

### A.4.1.1    The Linux Diversion

First we'll install Linux on your ChromeOS box. This won't work on terribly old Chromebooks, but not much will, am I right?

Get to your Settings page and look for Linux. It might say "Beta" on it, that's fine. Mine was hidden in Advanced setup under Developers! Click to turn this on and follow the prompts to set it up. The only thing I changed was my name. But when it asks for space, make sure you give it as much as you can spare. I upped mine because I had very little free and it almost ran out during the VS Code configuration! 4.1Gb seems to have satisfied it for now...

When that finishes, you'll be placed into a terminal window. This is a terse prompt that allows you to type commands to your system without clicking yourself to death. It defaults to your chosen name followed by an at sign and then 'penguin'. Then it probably has a colon and a tilde followed by a dollar sign. This is the classic dollar-prompt we talk about in Chapter 2 so much. Now you have one on your Chromebook!

#### A.4.1.1.1    A Compiler

The next step is to install the compiler utilities from the command-line. Type the following two commands one after the other:

```
sudo apt-get update
sudo apt-get install g++
```

The second command will stop with a `Yn` prompt. Just hit Enter/return to accept the `yes` default.

A while later, it will return you to the dollar-prompt. Type this command to check the installation:

---

[13]Aren't those things annoying?!

```
g++ --version
```

If it says command not found, we've got issues. Please talk to your instructor right away.

### A.4.1.2 VS Code Time!

I hope you didn't close the terminal window yet! We need one last thing from there to install VS Code. We need to find out what kind of machine your Chromebook is. Some run on Intel or AMD chips and some run on ARM chips. You have to decide which you have before downloading the right one from the VS Code website. To determine your machine type, just run this command at the dollar-prompt:

```
dpkg --print-architecture
```

Be careful when reading it. 'amd64' looks an awful lot like 'arm64'. But it's an important distinction!

Now go to the VS Code site and find the 'Download' button. (For me it was on the upper-right side of the screen just to the right of the Search field.) From the choices, choose the middle one under the big penguin logo (that's Tux, the Linux mascot). Make sure to select the one from the `.deb` row that matches your architecture. If you were 'arm64', choose 'ARM 64', of course. If you were anything else, choose '64 bit'. This will get you the installer. It also takes your browser to a 'Getting Started' page just for Linux. We'll just use our own steps below, though. No need to read it now — or maybe ever. (But it can't hurt to bookmark it for later. . . )

But, before you go below, double-click the installer in its download location. The system offers to install it with Linux, select Install and then OK.

To run VS Code, go to the dollar-prompt once more and type this command:

```
code
```

You can now proceed to section A.5. But don't forget to come back here to finish setting up software to connect to a Unix server if your school uses such!

## A.4.2 Unix Server Connection

If you are working in a Unix environment at your school, then I've got the instructions for you here to make that happen. We'll need two tools: one for entering commands like compiling and execution of your programs and one for file transfer to either back up your programs from the Unix server or upload your local VS Code creations to the Unix server.

### A.4.2.1 Command Processing

We'll start here by making your terminal talk to the school's Unix server for processing commands. This is how one typically compiles and executes programs in a Unix environment — from the terminal. If you've closed the terminal, re-open it by swiping up on the task bar at the bottom of your screen and selecting first 'Linux Apps' and then Terminal from there. Its icon looks like a greater-than followed by an underscore for whatever reason.

At the dollar-prompt in your terminal, you can type commands to the computer like the `ssh` command. This is short for Secure SHell. It will give you a nearly identical dollar-prompt except it'll be one connected securely — encrypted — to a remote Unix machine. You just need the machine/host name for the school's Unix server and its fingerprints for verification.[14]

To connect to your school's Unix server, just type this at the command prompt:

---

[14]Of course computers have fingerprints! People are always touching them! *chuckle*

```
ssh youraccount@hostname
```

Of course, you must replace `youraccount` and `hostname` with the name of your account on your school's server and that server's host/machine name respectively. When the connection has been established for the first time, `ssh` will prompt you with a set of fingerprints. Verify these against those given to you by your school's IT site or your instructor. If they don't match, you may be subject to a man-in-the-middle attack![15]

When prompted, enter your password for the school's Unix server. Note that your password won't show up as you type for extra security. You'll have to be a really good typist and trust that you've done it right. *smile*

Now you can type any commands your teacher told you about to edit, compile, or run files on the Unix system. If you need more help, I recommend finding a good Unix tutorial you like. I'm partial to text references, but lots of younger folk enjoy a good YouTube video. *shrug* To each their own...

To disconnect from the remote machine, just type:

```
exit
```

at the dollar-prompt. This will place you back in your own terminal on your own Chromebook. Before you quit the terminal app, though, you might want to pin it to your app tray if you'll be using it a lot. You can pin a currently running app by right-clicking its icon in the tray, selecting Pin . You can then drag it to your happy location amongst the other apps, if you want.

Once you're done, you can just 'exit' as you did above but this time to exit the terminal app itself.

### A.4.2.2   File Transfer

But someday you'll want to download those files you edited on the remote server or upload those you've developed in your VS Code environment locally. To do this you'll need a good file transfer program/app. The easiest one I've found for my Linux machine is FileZilla. But don't go there except for help/reference. Use your package manager to download/install the app. If you like the graphical manager, just search for 'filezilla' and click 'Install'. If you like the command-line installer, the package name is typically 'filezilla' and it will install at least one other package for a helper library as well. On my Ubuntu box I ran:

```
sudo apt-get install filezilla
```

and then hit Enter at the `Yn` prompt to accept the extra packages as well.

Next run the app from the command-line or the menu or a search. That all depends on how you like to do things and how you have your system configured. I ran mine from the command-line:

```
filezilla
```

Once it is running, you can right-click it in the tray and choose Pin to keep it handy for later up/downloads.

Now let's get configuring!

First let's declutter the interface a bit. Go to View and click Local directory tree to undo the checkmark by it. Then go back into View — see the checkmark is gone? — and click Remote directory tree . Two panels just disappeared from the main window and that's less to look at. Depending on how much it annoys you, you can also go back into the View menu and click to uncheck the Transfer queue panel.

---

[15]I actually like the moniker "monkey-in-the-middle attack", but it just isn't as popular for some reason.

On the left side now is your local directory. Your home directory is the default. You can see little folder icons next to any subfolders/subdirectories and little pieces of paper next to other files. At the top of the subfolders list is one called . . — yes, just two dots. This represents the 'parent' directory of the current one — i.e. the one above this on the way to the root directory (/)! Sorry, got a little carried away.

Time to connect to the school's Unix server! Above the directory/file panel a little ways is the connection bar. This has places for the Host (aka host name or machine name), your username on the school's Unix server, your password there, and the port to connect with. You should have been given the first three by your teacher or IT personnel. The port will be 22 for an `ssh`-style connection. When used for file transfer like this one will be, we usually call it an `sftp` connection — Secure File Transfer Protocol.

Once you've filled in all the spots, click the 'Quickconnect' button. This should start a scroll of information between the connection bar and the file panel. You'll be prompted for two things: whether to save your password — I wouldn't, but it's your computer/choice — and to check the fingerprints of the remote machine. As with the `ssh` in the `Terminal` above, you should have been given fingerprints by your instructor or IT staff. Make sure it is right to avoid that monkey in the middle!

Once connected, you'll see your remote account's files in the right panel next to your local ones. Now you can navigate each side by double-clicking folders (including . . if needed) to find your program source codes. Once you find them, just drag and drop them to the other side. If you let go over a folder, you'll place the files inside it instead of just where the other panel was at. I say 'files' plural because you can use Shift -click or Control -click to grab multiple files at once. `FileZilla` may ask you if it is okay to copy the files, if so, just check the "Don't ask again" box and say 'Okay'.[16]

When you are done, just Control + Q and next time you open up `FileZilla` it will remember where you've been. Click the drop-down arrow next to 'Quickconnect' to find your history. Just choose the one that starts with `sftp://` and has your user name and the school's host name in it and `FileZilla` will connect anew!
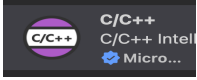
## A.5   VS Code Setup

Now that you have VS Code installed on your platform, you'll want to configure it for best practices. Follow these steps and all should be well:

1) Got to either the Code or File menu — macOS and Windows/Linux/ChromeOS, respectively — and choose Preferences ⟩ Settings under that. Now use the 'Search settings' box to find the following bold items and change them as directed:

     i) **Editor:Detect Indentation** should be unchecked. Having this on can really mess up the next two settings.

     ii) **Editor:Tab Size** should be set to something between 3 and 5 inclusive. The typical 8 is way too deep when we get to nested control structures later on.

     iii) **Editor:Insert Spaces** should be checked. Tab characters save a little disk space, but can really mess up displayed code in other environments.

     iv) **Files:Eol** should be set to \n. This is a platform neutral setting and won't mess up things when moving from a Windows to a Linux venue, for instance.

     v) **Files:Insert Final Newline** should be checked. This is a recent push in the industry for a text file standard. Also makes some code displays look nicer.

     vi) **Files:Encoding** should be UTF-8. This is an industry standard for text files now.

---

[16]Aren't those things annoying?!

vii) **Files:Auto Save** should be 'afterDelay'. This is my personal taste, but you definitely don't want it 'Off' which was the default when I installed last! (If you set it to 'afterDelay', you can also set the delay itself in the next item down. It defaults to every second which is really fast. (Note that the delay is set in milliseconds — not seconds.)

viii) **Editor:Rulers** will ask you to 'Edit in settings.json'. Just click that and inside the square brackets (`[]`), type 75 or 80 or whatever your teacher suggests. (I have my students set it to 75 for our local system. 80 is fairly common as well.) This phantom vertical line will tell you visually when you should be wrapping lines of code that get too long for good code display. Save the file but don't close out of it yet, we'll come back to it shortly!

2) Open the Extensions sidebar (the icon is three boxes together and one floating to the side). Here search for C++ in the 'Search Extensions in Marketplace' bar. One is just 'C/C++' — not 'Extensions', not 'Themes', and not 'C++ Intellisense'. Mine looks like the figure at right. Click the 'Install' button. You may have to restart VS Code afterwards.

3) Back over to the `settings.json` file, place a comma after the close square bracket on `"editor.rulers"` if one is not present. Now add this line on the line after that:

```
"C_Cpp.default.cppStandard": "c++17"
```

If any more lines come after it except the close curly brace, place a comma after the `"c++17"` above. (You can use `"c++20"` if you have a compliant compiler for this. Check your documentation or the compiler's website.)

Now save the `settings.json` — but don't close it — we're still not done there!

4) Now open Extensions, click the Search bar, and type rewrap. There are a few, but you want the one with the backwards S and a vertical bar to the side of it. Install that. You can read more about it in its description later, but for now just go back to the Settings tab and search for rewrap there, too. Check the **ReWrap:Auto Wrap: Enabled** setting so that it actually becomes Enabled. This is very handy for long comments so you won't have to wrap them manually at the ruler we set up before.

Note: you may have to restart VS Code here as well.

5) Now click the stacked sheets of paper icon in the upper-left corner of the window to get to the file explorer. You can then close the Extensions and Settings tabs, too — but leave the `settings.json` file open! Open a new folder by choosing the 'Open Folder' button and choose/create the place you want to write a program.[17] You will have to accede to trust the authors in this folder. You are safe here. Go ahead and trust yourself. *grin* Seriously, make sure you check the 'Trust the authors of all files in the parent folder...' box, too.

Now right-click (two-finger click on a trackpad) the 'settings.json' in the title bar — the one with the X next to it that would close the file — just don't close it! This brings up a menu of choices of what to do, of course. We want to do the first of the 'Reveal' commands. This will open a file browser window for your OS where the `settings.json` file is located. You should see it and several folders there. (The second one just makes it appear in the File Explorer within VS Code.)

Next click the single sheet with a + on it near the top of VS Code's File Explorer window. It will appear only when your cursor is in the blank area beneath your folder's name — the one you created to place your program above. This wants a name for this new file. Call it `tasks.json` — case is important here! Once you hit Enter/return on the file name, you have a nice blank file to work with. Now we need to put this in it:

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
```

---

[17]I'd recommend making a first folder for your course or your exploration of this book and then subfolders within this for each of your projects/programs.

```json
        // for the documentation about the tasks.json format
        "version": "2.0.0",
        "tasks": [
            {
                "type": "shell",
                "label": "clang++ build active file",
                "command": "clang++",
                "args": [
                    "-std=c++17",
                    "-g",
                    "-stdlib=libc++",
                    "-Wall",
                    "-Wextra",
                    "-Wfloat-equal",
                    "-Winline",
                    "-Wunreachable-code",
                    "-Wredundant-decls",
                    "-Wconversion",
                    "-Wwrite-strings",
                    "-Wcast-qual",
                    "-Woverloaded-virtual",
                    "-Weffc++",
                    "-fno-gnu-keywords",
                    "-pedantic",
                    "-Wparentheses",
                    "-Wshadow",
                    "-Wold-style-cast",
                    {
                        "value": "*.cpp",
                        "quoting": "escape"
                    },
                    "-o",
                    {
                        "value": "${fileBasenameNoExtension}.out",
                        "quoting": "escape"
                    }
                ],
                "options": {
                    "cwd": "${fileDirname}"
                },
                "problemMatcher": [
                    "$gcc"
                ],
                "group": {
                    "kind": "build",
                    "isDefault": true
                }
            }
        ]
    }
```

Sadly, your mileage may vary for pasting from a PDF. I've had mixed results myself. I'll keep this and the below files on the website for you to copy/paste from or download as you see fit.[18]
Now, if you've set up for Windows, ChromeOS, or Linux, change where it says `clang++` to `g++` (there are two places — lines 8 & 9). If you've set up for Windows, change the `.out` extension on line 36 to `.exe` instead. (And with a compliant compiler, you can change out the 17 on line 11

---

[18]When clicking to the website link from a web browser, I recommend a right-click to open it in a new tab/window so you don't lose your place in the PDF.

with 20 as before.)

Last edit: if you are on Windows, ChromeOS, or Linux, delete line 13 for the `--stdlib` setting. This is a `clang++` only thing and that is the command-line compiler on Mac. The rest of the settings can stay for either compiler. Now make sure you've Saved the file.

Then we want to 'Reveal' this file in the browser. Right-click (two-finger click on a trackpad) the name of the file at the top with the X next to it and choose the first 'Reveal' option again. Now you have two browser windows open — one with the `settings.json` and one with the new `tasks.json`. Go to the `tasks.json` folder and drag it to the `settings.json` folder. This should move the file. If not, you can feel free to delete the one from your just made folder, we won't need it anymore. You can also close the `tasks.json` file in VS Code as it isn't where we put it from there anymore.

6) Next we need to edit the `settings.json` file from before. Go to the last setting — it should be the `rewrap.autoWrap.enabled` we just did — and add a comma after it. Now paste this from either here or the launch file from the website:

```
"launch": {
    "version": "0.2.0",
    "configurations": [
        {
            "name": "clang++ - Build and debug active file",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}/${fileBasenameNoExtension}.out",
            "args": [],
            "stopAtEntry": true,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": true,
            "MIMode": "lldb",
            "preLaunchTask": "clang++ build active file"
        }
    ]
}
```

Again, change `clang++` if necessary on lines 5 (name) and 15 (preLaunchTask) and change the `.out` on line 8 (program) if needed as well. And for Windows, ChromeOS, and Linux also change the `lldb` on the MIMode line (14) to `gdb`.

Save the `settings.json` file and we're done configuring!

7) Finally, we're ready to test this thing! Click the stacked sheets icon to the far left of the window to get back to the File Explorer sidebar. Now move the cursor to the blank area below your folder name again and click the sheet of paper with a + on it at the top of the sidebar. Call this new file `welcome.cpp`. Paste this code in there:

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "\n\t\tWelcome to C++!!!\n\n";

    return 0;
```

```
    }
```

(If having trouble copy/pasting from the PDF, this file is available on the website as well.) Save and choose `Terminal` ⟩ `Run Build Task...`. A little window appears below your editor with a bunch of text. If you look carefully, you'll recognize the bits and pieces from the `tasks.json` file we made. If all goes well, you'll see this at the bottom:

```
  Terminal will be reused by tasks, press any key to close it.
```

Click in that window and hit `Enter`/`return` to close it. Now choose `Terminal` ⟩ `New Terminal`. This gives you a command prompt in which you can ... wait for it... run commands. We'll run our program. In macOS, ChromeOS, or Linux type `./welcome.out` or for Windows type `./welcome` to run the program. You should see this:

```
                    Welcome to C++!!!
```

If anything has gone wrong, please let your instructor know ASAP so they can help you fix the issue. If they need help, they can email me. *smile*

8) One final note about programs. Make sure each program you tackle is in a separate folder/directory from one another or the above techniques won't work.

## A.5.1 Normal Workflow

### A.5.1.1 Terminal Management

Once you've opened a new terminal window, it should come back each time you reload VS Code. Next time you compile, the compile terminal will overlay it. Just clicking it and hitting Enter as before will close the compile terminal and put you back to your command terminal. There's no need to repeat the new terminal window part each time.

### A.5.1.2 File Management

The above setup is geared toward working on a program whose files are all in a folder/directory together. This is helpful once you reach that part of chapter 4 with separate compilation and is not a bad idea otherwise.

Don't expect to place all your C++ programs in the same folder together and all will work well. It would be disorganized and won't work with the above tasks setup or debugging setup.

#### A.5.1.2.1 Opening Files

Speaking of having multiple files open at once, there are actually two types of open files. If you single-click to open a file, it will be open to view. But if you then single-click to open another file, it will replace the first file! This can get annoying when trying to open multiple files at the same time. To get around it, make sure you double-click to open a file and this will leave it open even if you don't edit in it before opening a second file.

## A.5.2 Recommended Extensions

The above are almost all mandatory setups except perhaps the ReWrap extension. But the following are all completely optional. They will, however, make certain things easier to do within the VS Code environment and so are strongly encouraged.

- If your teacher gives PDF handouts or makes you prepare a PDF to hand in to them, the extension "PDF Preview" is very handy as it let's you view PDF files from within VS Code.

- If you are prone to spelling errors, then the "Code Spell Checker" extension is your friend. It will check not only your plain text files, literal strings, and comments, but also your variable, constant, and function names. The extension knows many common abbreviations for things so camel-case with short-hand can work without complaints!

- To make the difference checking suggested in Chapter 4 easier, you can install the "Partial Diff" extension.
  To use this extension, you select a section of code, right-click, and select "Select for Text Compare" from around the middle of the popup menu. Then select the second section of code you want to the first to, right-click, and select the "Compare Text with Previous Selection" option — just below the previous option.
  To configure the colors, you need to edit the `settings.json` file. Remember, to open "Settings" use Ctrl + , or command + . Then search for 'rulers' and click the "Edit in settings.json" link. Now, go to the bottom and add a comma after the last thing and then this:

```
"workbench.colorCustomizations": {
    "diffEditor.removedTextBackground": "#FF000055",
    "diffEditor.insertedTextBackground": "#ffff0055"
}
```

  A color box will appear in front of the two color settings. You can then click them to get a color chooser dialog. Note the 55 after the normal color codes is called an alpha channel and can be tweaked as well. I found that tip on StackOverflow.

- If you want to take your comments to the next level, I strongly recommend the "Doxygen Documentation Generator" extension. This let's you do fancy comments ala JavaDoc or JSDoc. Making the hyperlinked form will require the doxygen application, but this should be easy enough to get for most platforms. For more, see the application's homepage.

- If you want help making a nice inclusion guard for your library interface files, try out the "C/C++ Include Guard" extension. This extension needs a little configuration to match the styles discussed in these volumes.
  Open the Settings with Control/Command and Comma and click the Extensions arrow to open it and then click to the C/C++ Inclusion Guard section. Here scroll down to "Comment Style" and change it to "None". Then continue down to "Macro Type" and change it to "Filename". Now uncheck the "Remove Extension" box just below. And finally add a "Suffix" of "`_INC`".

- For smooth operations between your VS Code installation and a remote system via the SSH protocol, I recommend the "SSH FS" extension. With good bit of configuration it can make editing remote files in place a breeze.

## A.6   Wrap Up

That concludes our setup instructions. If you have any trouble, contact your instructor immediately for help! It is very important that you have a working environment sooner than later in a course like this. Without practice, you don't really learn anything in the long run, after all.

# Appendix B

# Debugging Tips

We've talked some about debugging programs before:

- Printing values with `cerr` at crucial junctures.

- Using an `ignore` with `cin` to pause the program during a [nearly] infinite loop.

- Using `assert` to test function parameters.

- Using `cerr` to determine which of many function calls caused an `assert` to fail.

But there are software-based debugging systems other than `cerr` and `assert`. Two such systems are `gdb` and `lldb`. These debuggers work to trouble-shoot your program in either a `gcc`-based or `clang`-based build. If you've been working based on the instructions from above for VS Code setup, you should be able to debug your program within that environment with the provided setup. (Windows and Linux were set up to use g++ from the `gcc` compiler suite and macOS was set up to use `clang`++ from the `clang` compiler suite.)

The steps to debugging programs with a debugger are to set watches and breakpoints and to either step into or over lines of code. The exact way to do these things varies from one debugging system to another, but we'll walk you through debugging a program in VS Code as set up above here.

## B.1   Starting a Debugging Session

We already set up a debugging session above. We just need to start it going. To do so, choose the `Run` ⟩ `Start Debugging`.

When I did this the first time, I had to give special permission on my macOS box to use the `lldb` and for my program to access part of my files for whatever reason. I said yes to both and thereafter I was only asked about the file permissions on each rerun. Kind of annoying, but I'm not sure how to turn that on permanently.

## B.2   Setting Watches and Breakpoints

Breakpoints are easiest to set. What do they do? Well, they stop your program from running. Sadly, this would be applying the brakes, but we have instead a breakpoint. *shrug* What're ya' gonna do?

This is done by hovering the mouse to the left of the line number in the VS Code editor window. A red dot appears and you just click on it to set that line as a breakpoint. Then, when the program is run in debugging mode within VS Code — not separately in the terminal, it will stop there and let you see the values of variables at that point in the execution. This can be terribly helpful in finding problems during a troublesome project. The variables and their values are located to the left of the editor window where the list of files once was.[1]

There you see variables, watches, the call stack, and breakpoints. The variables panel changes as you move from function to function to reflect the currently visible local variables. The watch panel shows variables or function arguments you'd really like to keep a special eye on. More on that in a bit. The call stack panel shows what functions have been called to this point in the program. You'll note that they are stacked up from oldest on bottom to newest — currently running — on top. This is just as described back in chapter **??**. Finally, the breakpoints panel is just a textual list of the red dots you've made in your code.

Note: to look inside a `string`, `vector`, or `array` `class` variable, you'll have to open the little greater-than arrow after its variables entry during the run.

Unless, you are interested in a specific part of said container. In that case you can set a watch on an expression to reach that part. Just type out something like $v[2]$ to see the $3^{rd}$ slot in a `vector` named v, select it, and right-click to "Add watch" on it. Or, while the program is running in debug mode, click the plus sign in the watch panel to add a watch expression.

The expression can look into a container with the [] operator or look at a `bool` expression to see if it is `true` or `false` at the moment. It can involve any kind of operations, basically, on any currently known — in scope — variables, constants, or values. This is what makes the watch panel more than the variables panel. Sure, you can watch a variable, but you can also watch any kind of expression involving a variable, too.

## B.3   Stepping Through Code

Making the code run during a debugging session can be a bit tricky at first. There are four methods of movement listed in the debug control bar atop the editor window. They are the four icons in light blue. The green circular arrow restarts the program from scratch and the red square stops it right where it is at.

The first icon that looks a little like a play button runs the program until the next breakpoint or the end of the program. The second one with the arrow hopping over the dot will execute the current statement and stop at the next one. This operation is called "step over" in debugging terms. If the statement calls one or more functions, step over will execute them all and then land on the next statement.

The third blue icon, however, is known as "step into". This one looks like an arrow pointing into a dot. This means execute the current statement, but if it contains function calls, dive into them to see how they are working inside as well. On some systems you have to be very careful with step into because it might step into the standard library codes as well as your own. That doesn't seem to be the case on my macOS install of VS Code on top of `clang++` and `lldb`.

But with my Linux install on top of `g++` and `gdb`, it tries to step into the standard code and gives an error at the end about not being able to open a file. In fact, once I tried that to see what it would do, the only time I don't get that error at the end of the program run is when I hit the play button to end things. If I try to step out — even over! — it give the "can't open file" error.

The last button is "step out". I haven't gotten this to work, but it is supposed to run until the current function `return`s.

---

[1]Don't worry, you can get back to the file browser by clicking the top icon (two overlapping sheets of paper).

## B.4   Wrap Up

I hope this short introduction to advanced debugging using an automated tool has helped whet your appetite for this topic.

# Appendix C

# Essential Unix Knowledge

As implied in the setup appendix above (appendix A), a lot of times introductory programming courses use a command-line interface like the Unix (or Linux) terminal prompt. This appendix attempts to give you the basic knowledge necessary to navigate this environment with relative comfort. We won't talk to compiling commands, as that would require knowledge of your local setup and what compiler(s) are installed. But we'll talk naming, navigation, and essential commands that will get you around the terminal in reasonable style.

## C.1 Paths and Filenames

Unlike in a graphical interface (or GUI), the command-line interface deals a lot with path and file names. In a GUI, after all, you just see a dialog box and click on the right one. At best you type most of the filename in an input line when naming the file. But in the terminal we type these things out a lot so it is good to be comfortable with the basics.

First, terminal folk call folders directories. This is an older name but more traditional. And knowing this will make some of the mnemonic names more clear later.

Second, separators that go between folder or directory names and the eventual filename are different in a Unix environment than in Windows. The Unix community uses the same separators as in a web address: /. This is the opposite slash as Windows uses — when you even get to see them.[1]

Third, a path relative to the current folder can be entered starting with the subfolder name. But an

---

[1]But modern Windows is pretty accepting and will transform a path with the Unix/Web separators to its own form on the fly.

absolute path needs a leading / on it to indicate it is with respect to the entire drive.[2]

Fourth, case is SENSITIVE in the Unix terminal. Upper and lower case are considered different and must be entered exactly. So be careful what you name your files if you don't want to have to shift your life away!

Finally, spaces and quotes are allowed in path and file names, but are a little tricky to work with. Let's talk about that specially.

### C.1.1  Spaces and Quotes

To use spaces or quotes in a file or folder/directory name, you have to take one of two tactics. The first is to escape the space or quote. No, you don't have to outrun it in a death race. You just place the standard escape character in front of it. We'll learn about the escape character and how it is used in C++ programming in chapter **??** — section **??**. But for now know that it is used at the command-line prompt as well.[3]

What is the escape character and how do we use it? It is a backslash: \. And we place it just before the space or quote we want to make special. You see, normally the space is used by the command prompt to separate 'words' in the input command, so treating them as part of a filename is relatively special. We'll see how quotes are used by the command prompt in a second.

So if you had a file named 'my file' and you wanted to use a Unix command on it, you could type that name as:

```
my\ file
```

Or, if you prefer, you could surround it in quotes which the command prompt uses to group together space-containing phrases for use as single things in a command:

```
'my file'
```

Double quotes would work just as well, of course. We often just use singles because then we don't have to shift. Seems silly, but when you spend all day typing, you find shortcuts.

And if you need a quote in a filename or path component, you can escape that as well:

```
Jason\'s\ file
```

You cannot, however, just surround a quote in a name with more quotes! They can't even just be escaped! You have to take it to the shift-level. Then you can do a switcheroo:

```
"Jason's file"
```

That's the gist for now, but see below for more on special treatment of spaces and quotes!

### C.1.2  Special Folders

There are two special folders/directories that actually exist within every folder on the drive. One represents the parent folder of the current one. It is called .. — just the two dots. We'll use this later to navigate around the system a bit.

---

[2]Actually, it is the file system to which many drives could be attached. But more on that in an actual course on operating systems.

[3]Unix was made by programmers for programmers to use and you'll find lots of the same conventions used at its prompt as are used in common programming languages. In fact, the teams that developed Unix and the programming language C overlapped quite a bit.

The second is called . — just a single dot. This represents the current folder/directory. It is used when you want to make sure a file is used relative to the current folder and not with respect to somewhere else on the drive.

### C.1.2.1   More on Relative Paths

We spoke earlier about absolute and relative paths. But now we can use the special directories . and .. to make other relative path references like:

```
'./file in this folder'
```

Or perhaps:

```
'../folder beside this one/file over there'
```

These are both relative paths — one relative to the current directory and the other relative to its parent folder.

The parent path can also be used multiple times to dig your way back up a deeply nested folder structure. Like:

```
'../../../folder three up from here/file over there'
```

There is no need, however, to repeatedly use . in a relative path.

## C.1.3   Wildcards

Sometimes we want to group many similarly named files together. We can't just shift-click or Windows-click or Command-click them to group them together like we would in a dialog box, though. We have to group them by the commonality of their names. For instance, if you had a set of files all ending in the extension cpp, you could specify them all at once with:

```
*.cpp
```

This use of the asterisk indicates to the command prompt that you intend to use all the files whose names end in a period and then the letters cpp. The dot isn't even explicitly required and is often left off by speedy programmers:

```
*cpp
```

This wildcard character as we call it can replace any sequence of characters — even spaces or quotes — in a file name or path component. You can, for instance, refer to all subfolders in the current directory whose names start with lib by the wildcard pattern:

```
lib*/
```

The ending slash tells the command prompt that you are referring to folder names instead of file names.

There is also the wildcard question mark. This wildcard represents any single character. So if you had five numbered files starting with the word data and ending in a dat extension, you could group them all into a single command with:

```
data?.dat
```

This would match all of the numbered files `data1.dat`, `data2.dat`, etc.

# C.2    Basic Navigation

So what do you do with all these paths and file names? You can do lots of commands with them at the command prompt! Let's look at those dealing with basic exploration, organization, and navigation. Throughout these examples we'll use the sample folder structure at right.  Here we have the ~ directory where you initially log in and below it are two more folders.  One is for `code` and the other is for `data`.  Each has a couple of subfolders below it as well.  The `code` ones are organized for the parts of this book and the `data` are just broken down by `old` and `new` data.  (The style of the diagram was taken from an actual command-line tool called `tree` that makes what we call directory or folder trees.)

```
~
|-- code
|    |-- Aggregation
|    |-- Appendix
|    `-- Flow Control
`-- data
     |-- new
     `-- old
```

## C.2.1    Listing Folder Contents

Let's start with `ls` whih is a command that lists the contents of the specified folder or the current folder if nothing is specified.  Like most Unix commands, this is a terse mnemonic for its purpose.  We don't really like to type a lot — as you may be gathering.

So, we could do:

```
ls
```

To list the contents of the current directory and we'd see:

```
code        data
```

Or we could do:

```
ls code
```

to list the contents of the specified folder (`code`).  Here we'd see:

```
Aggregation     Appendix      'Flow Control'
```

We can even list just a particular file like this:

```
ls file\ name
```

This only lists the file's name, though:

```
'file name'
```

and we already knew that. To see more information about the file, we need to request a longer listing format. This is done with the command-line flag `-l` like so:

```
ls -l file\ name
```

Now we see not only the name but also the permissions, the date of creation, the file size in bytes, and much more!

```
-rwxr--r-- 1 user  user 1317 May 26  2022 'file name'
```

Two other flags we often use with `ls` are `-a` to list all files including those normally hidden and `-h` to list those file sizes in human-readable form with k, m, g, etc. prefixes. If you want more than a single one of these at a time, you can merge them without the dashes like so:

```
ls -lah
```

The order of the flags is irrelevant. Just that they are present after a dash makes them work.[4]

```
drwxr-xr-x 1 user  user 4.0k Jul 28 15:31 .
drwxr-xr-x 1 user  user 4.0k Jul 28 15:06 ..
-rwxr--r-- 1 user  user 1.1k May 26  2022 'file name'
```

I've cut us off after the first file to just show the relevant parts.

## C.2.2   Tab Completion

Since we get tired of escaping and quoting all of our spaced names, though, we also invented a tool to help. It is called Tab completion and is pretty straightforward. You type part of a filename and then hit the Tab key. The command prompt then tries to find a file with that part of the name at the start and completes it for you as far as it isn't in conflict with another name.

So let's say you had those five data files from before. You could type just d and hit Tab. The system would then complete `data` for you and beep or at least stop. This indicates that there is confusion at that point and more information is needed. If you hit Tab a second time immediately, the system will display all the files it is considering here so you can see the conflict. Like so:

```
$ ls -l data
 data1.dat   data2.dat   data3.dat   data4.dat   data5.dat
```

Here I've added a $ to indicate the command prompt. Many will end with this character while others will use % instead.

Once the list is shown you can type one or more characters to clear up the confusion and hit Tab again. So if you typed 3 and hit Tab, the system would complete `data3.dat` for you.

When the system completes to the end of a filename, it puts a space after the name. When it completes to the end of a folder name, it puts a / afterwards.

## C.2.3   Making Folders

But we probably don't want all our files in a single folder. I suppose it is a style of organization, but it isn't much of one — no offense. To make new folders, we have to remember that they are typically called directories in Unix and so the command is `mkdir` to make a directory:

```
mkdir new\ folder\ name
```

---

[4]If you want to name a file with a dash at the front, you'd have to use an escape or quoting to refer to it at the command prompt due to this convention of prefacing flags with a dash.

Then you can arrange you files to be in different folders based on their purpose. This can be done in a file transfer tool (which could also make folders, probably) like the WinSCP or FileZilla espoused above in the setup appendix for various operating systems. Or you could do it at the command line for convenience.

To make the directory structure in our sample diagram above, we would do the following:

```
$ mkdir code
$ mkdir code/Flow\ Control
$ mkdir code/Aggregation
$ mkdir code/Appendix
$ mkdir data
$ mkdir data/old
$ mkdir data/new
```

Note that the commands just return a new prompt with no message as to success or failure or what was done. Unix, again, is a very terse and programmer-oriented land.

## C.2.4  Handling Files

There are two commands to handle files. There is `cp` to copy files leaving one copy where it started and making a new copy in a new location:

```
cp 'current file' 'new location/'
```

And then there is `mv` to move files removing the original and placing it in the desired new location:

```
mv 'current file' 'new location/'
```

As an added bonus, you can use `mv` to rename files that you've decided need a new name as well. Consider it moving the file to a new name:

```
mv 'current name' 'new name'
```

I'll teach you to remove files, but be forewarned: it is typically permanent as there isn't normally a trash can to get things back out of on Unix. The command is `rm` — mnemonic for remove — and you just give it a filename like so:

```
rm 'file name'
```

But remember that there is **NO** undelete! The file is simply removed and that's that!

## C.2.5  Changing Folders

Finally, how to you place your terminal into a new folder so that you can use the files you placed there without excessive path typing? Again, recalling the directory thing, we have `cd` to change directories:

```
cd new\ folder
```

Placing a trailing / is optional. You can also use `..` as the destination to move back to the parent folder, of course.

As a special usage, you can also move all the way back to your original login directory by just typing `cd` without a target folder name:

```
cd
```

You can even travel multiple layers at a time like so:

```
cd data/new
```

And Tab completion can complete folder names as well, so feel free to do a `cd` to `code/Fl` and hit Tab to finish that out for you. *smile* (As an added bonus, when Tab completing from a `cd` command, only folder names are expanded!)

## C.3 Common Commands

There are many common commands in Unix that will help you do certain tasks or utilize files in certain ways as well. Anything from getting free of an infinite loop (see chapter **??**) to displaying file contents on the terminal screen and beyond.

### C.3.1 Stopping a Runaway Program

If you find yourself with a program run amok, you can stop it by pressing the control key and simultaneously the C key. This closes the errant program rather than copying a selection. Weird, right? (Likewise, Control + V won't paste in a terminal.)

### C.3.2 Getting Help

There is a manual in most all Unix environments available from the command prompt. The command to access it is `man` and you just give it a command name and it shows you the manual page one screen at a time right there in the terminal. For instance, you can read much more about `ls` by doing:

```
man ls
```

To get to the next screen, just hit the spacebar. To quit once you've seen enough, hit q.

If you are unsure of a command, you can also use the flag `-k` to find a command that talks about a topic. So if you want to see all the commands related to doing listings, you could do:

```
man -k list
```

Among them you'll find `ls` with a `(1)` after it. This 1 designates the manual section the command is in. If that command is unique, then no section is needed. But if you ever man something that has more details in a later section, the first found section is always displayed! To get to the later section entry, just put the section number before the command name:

```
man 1 ls
```

### C.3.3 Displaying Files

If you have a plain text file that you want to display on the terminal screen, you can us the `cat` command:

```
cat 'plain text file'
```

This command name might not seem mnemonic at first, but it kinda is. The creators were considering that displaying the file was like attaching its contents to the end of the screen. So they thought of it as

concatenating the contents to the screen. And since that's too big for a command, they shortened it to `cat`.[5]

This is not good to do with a binary file like a compiled program! It can really mess up your terminal to the point you'll have to open a new one.

### C.3.4  Paging Long Files

If the file is really long, it will, of course, scroll your terminal up pretty far and you'll have to use the mouse or trackpad to scroll up with it. If you'd like to see it just a screen at a time like the `man` command did, you can use the `less` command:

```
less 'long text file'
```

Again, use spacebar to see another screenful and q to quit before or at the end.

What is this name mnemonic of? Well, the original command was `more` for 'one more screen'. But someone came along with a competing command they thought more beneficial and called it `less` because, as the adage goes, 'less is more'.[6]

### C.3.5  Converting Line Endings

Many times you will have created a file on Windows and are trying to transfer it to Unix for use. This works much of the time if you've set up VS Code or some other environment as recommended in the setup (section A.5) to use Unix line endings. But if you didn't, you can still work with these files by running them through the terminal command `dos2unix`. This command is named from the old Microsoft OS from before Windows: MS-DOS.[7]

```
dos2unix 'Windows file to convert'
```

This command will convert the line endings from the Windows 2-byte default to the Unix 1-byte variant. This will help them display on screen correctly with `cat` and also help them be processed by your program as we do in chapter **??**.

### C.3.6  Formatting Files

Sometimes we've formatted a file so that it won't look nice in certain environments. While the terminal typically wraps longer file lines during a `cat` from one screen line to the next, it often looks odd. And if we are trying to preserve it (see transcript recording below), can even be chopped off when lines are too long. That's why I espoused using a line guide around 75-80 in the VS Code configuration section (A.5).

But if you ended up with a file with longer lines and want it to look good on the terminal screen, you can use the Unix command `fmt` instead of `cat`. It will wrap lines to a given line length automatically and at word boundaries so it looks nice. A blank line in the text file is treated as a paragraph boundary so that you can have more than a single block of text in the file.

This command can be entered as:

```
fmt file\ to\ format
```

for a default 75-length line or you can specify a line length with a flag like so:

---

[5]Perhaps one of them had a pet they loved? *shrug*
[6]I don't make this stuff up — I swear!
[7]DOS stands for Disk Operating System, if that kind of trivia interests you. *smile*

```
fmt -50 file\ to\ format
```

Here we've specified to wrap lines to about 50 characters long.

If you have control of your Unix machine, you might want to upgrade your toolset to use the `par` command instead. This formats paragraphs really nicely and even does full justification instead of left justification like `fmt` always does. It is a little different in how it works, though:

```
par -j1q0 <'file to format'
```

The `-j1q0` sets you up for full justification if you like that sort of thing. Leave it off it not. If you want a width different than 72, use the `-w99` flag — just fill in the 99 with your desired line width.

The `<` there is called a redirection indicator and takes contents from a file and redirects it into the given command as if you had retyped all that at a prompt for the program. We could have alternatively used a command-line pipe like so:

```
cat 'file to format' |par -w50 -j1q0
```

The single vertical bar here takes the output from the first command and sends it to the second command as if you had retyped it at that command's prompt.

### C.3.6.1 Caveat/Warning

Keep in mind that there is no easy automation for formatting code. There are specialized utilities, but they don't come standard on any platform. Neither `fmt` nor `par` can handle formatting code files!

## C.4 Programmer Tools

Some commands are especially useful to programmers like recording a terminal transcript or searching for content in a set of files.

### C.4.1 Recording a Transcript

Sometimes a programmer is experiencing something that is hard to describe. If they want another programmer's input on the situation and can't just call them over to look, they might send a screenshot. But sometimes it involves lots of interaction or even dynamic interaction such that a simple screenshot won't do. In these situations a transcript is called for!

The command `script` will record a transcript of the terminal session in a file so that you can later send that file to someone else for review. The file by default is called `typescript`, but this can be changed at the command line. For the default name, just run:

```
script
```

To record to a file other than `typescript`, simply give that filename on the command-line:

```
script 'transcript name'
```

The nice thing about the default name is that you don't have to think about a name and it just overwrites the next time you run `script`. The bad thing about the default name is that it is overwritten the next time you run `script`. If you are preserving a transcript to send to another programmer or the like, having it automatically overwritten might be a bad thing. Worse, you might have two `script` commands running at the same time.

This can happen if while you are recording you realize something else needs to happen and you go do that in another window and when you come back to the terminal you think, "Well, I better start this recording now." It happens a LOT. I see it all the time. And when two `script` commands are actively writing to the same transcript file, it is a mess!

If the session needs to be more dynamic, you can record timing information with the `-T` flag. This should go before the filename if that version is used. Then, when the person on the other end gets your transcript, they can use `scriptreplay` to replay exactly what happened on your terminal in theirs.

BTW, the transcript file is a bit text and a bit binary so it won't look nice with just `cat` or even `fmt`. You'll have to do a little more work to prepare it if you want to print it or PDF it. It was designed to work well with `scriptreplay`, so...

## C.4.2 Searching Files for Text

After you've programmed for a while, you'll find that you have many codes you want to reuse in new programs. Chapter **??** on writing functions has lots of tools for easier code reuse, but you still might find yourself with a function name and no idea which file it was in. `grep` to the rescue!

This command will allow you to search through files for a word or phrase with ease. Just do:

```
grep word *cpp *h
```

This will search through all of your C++ files in the current folder for any occurrence of the given `word`. Or, if it is a phrase, you can put that in quotes like so:

```
grep 'short or long phrase' *cpp *h
```

And that text will be found instead. The matching line will be printed with some highlighting on what was searched for. If searching more than a single file like we did here, the line will be prefaced by a filename and a colon. If you'd like more context, you can use the flags `-C9` for common context or `-A9` and `-B9` for after and before context. Just change the `9` to your number of lines. So to see 3 lines of context before and 2 after, you could do:

```
grep -B3 -A2 'short or long phrase' *cpp *h
```

To make it a little easier to find in the editor, though, you can also have `grep` print line numbers with the `-n` flag.

Or maybe you don't want so much information — just the names of the matching files so you can then load them in the editor and look them over there. This can be achieved with the `-l` flag (a lowercase L).

But if this weren't enough, we can also use `grep` to search for fancy patterns! After all, who can remember the exact name of a function written weeks ago — we might just have an idea of how we named it or some inkling of a phrase in its comments.

To handle these situations, we'll need the regular expressions tool. In fact, this is where `grep` gets its name: global regular expression print.

### C.4.2.1 Searching Files for Patterns

Whereas wildcards allowed you to specify groups of files all at once, a regular expression — regex for short — can allow you to match many texts all at once. For instance, you could match any of `hello`, `hallow`, or `hollow` with the regex `h.llo.*` in your search.

### C.4.2.1.1  Regular Expression Basics

In regex patterns, the . can match any single character much like a ? in path or filename wildcards.[8] And the ∗ modifier says to match 0 or more of the preceding pattern. So .∗ says 0 or more of any character and plays the role of just ∗ in a wildcard match.

But that's just the tip of the iceberg! There are lots more things to match with. You can use groups of characters at a particular position, for instance. So if the above matching `hollow` was a problem, you could exclude it by using `h[ae]llo.∗` as your regex. Now only an `a` or an `e` will match between the `h` and the `llo` parts.

The order of the characters in the square brackets is irrelevant — only a single one of them will match at a time unless you put ∗ after the close bracket. You can also use ranges of characters in the square brackets like `[0-9a-fA-F]` to match any hexadecimal digit.[9] Note that the match is case sensitive and we had to list lower and upper case A through F to match either.

If you need to have a group include the dash character, you can place it first in the brackets. So this `[-.?]` will match any of dash, period, or question mark. This is also a way to match the dot without it meaning 'any single character'. Another is to use the escape slash. So `Hello\.` will match `Hello` followed by a period. (Again, case sensitive!)

But we've barely scratched the surface! There are ways to represent repetitions of 1 or more, up to *m*, *n* or more, exactly *n*, or even *n* to *m* of the previous pattern. There are ways to group parts of the pattern and repeat them. There are ways to match multiple possible patterns at a time. There are ways to match across line boundaries as well. And so much more!

There are any number of fine tutorials on regex pattern crafting online. Some are video and some are text-based. Use whatever works best for you to learn more on this powerful and fascinating topic!

### C.4.2.1.2  regex Everywhere!

Regular expressions are found all over the place, in fact. `grep` uses them to search files from the command line. `less` can use them to search the file being viewed — use `/` to start a search and `n` to repeat the search. And even VS Code can use them to search an editor file — see that .∗ off to the side of the `Control` + `F` / `command` + `F` dialog?

But do be careful — especially when searching for a tutorial — to make sure you are using the right flavor of regex. There are several — almost as many as there are programming languages! `grep` can use any of 3 flavors, in fact. And VS Code uses one for the editor search and another for the search in multiple files dialog. The most popular flavors are Javascript/ECMAscript and Perl/PCRE. But there are others and more coming everyday!

## C.5  Wrap Up

In this appendix we've taken a brief overview of the Unix (or Linux) terminal interface. We've discussed issues as basic as forming file and path names and as complex as programmer focused tools. In between we looked at some file handling and other common commands. I hope you put your new-found knowledge to good use soon!

---

[8]Technically the . cannot match an end-of-line character — the one stored for `Enter` or `return` and represented by `'\n'` in code.

[9]Hexadecimal is base 16. Since we ran out of digits at 9, we add the letters A through F to represent the next 6 'digits' in this base.

# Appendix D

# Input and Numeric Formats

Input is a very important aspect of any program so that we can gather the values the user actually wants us to work with this time around. As we implied in the program design section (**??**), we are basically making a general word problem solver: give us a particular word problem and we make a generalized solver for it. So having the numbers and such that the user needs this time is very important.

In this appendix we'll explore more about the input process and numbers in particular.

## D.1 The Keyboard Buffer

First off, input doesn't just magically appear in our variables. It starts out as a sequence of keystrokes — characters — sitting in a buffer. What's a buffer? Well, it's a place where things sit waiting to be processed. Sadly, this usage of the word matches no other definition of the word, so I have no parallels for you to draw from.

But envision it as a sequence of characters waiting to be worked on like so:

| User Types | Buffer Contents |
|---|---|
| Hello | 'H', 'e', 'l', 'l', 'o', '\n' |
| 4213 | '4', '2', '1', '3', '\n' |

Note that every buffer ends in a newline keystroke. This is because cin's extraction operator doesn't start working until the user hits ⌷Enter⌷/⌷return⌷ when they are done typing.

The input of char data is pretty straight forward. >> just puts the next non-whitespace keystroke in the char variable.

But, as you can see, even numbers are broken down by keystroke before being turned into actual data inside a variable where they can be added, etc. So the next question, then, is how does cin do that kind of thing?

## D.2 Basic Input of Numbers

Let's take the second example from above and walk through how >> does that translation into an integer for the program to work with. It begins by setting an accumulator to 0.[1] This will be where we add up

---

[1]Accumulator here is fancy talk for holding onto a sum or total.

the number that the user intended step-by-step.

Next we look at each digit's `char` in turn and both translate it to a number and put it in the proper place in the accumulator. The translation process could be done with simple subtraction, it turns out. But it is a little weird to subtract `char`. So we'll use the helper from section **??** to make our subtraction between ASCII values instead.[2]

Let's call the next digit's `char` from the buffer `next` and we'll call its translated value `digit`. Then what `>>` does is essentially:

```
digit = static_cast<short>(next) - static_cast<short>('0');
```

This tells us how far from `'0'` the `next` digit is. If it were a `'0'` itself, then it would be 0 away, for instance. If it were the leading digit from above, it would be 4 away. And so on. . .

Next we need to put this in the proper place in the `accumulator`. This part is a little tricky sounding at first, but once you see it move a couple of digits along, it comes together.

We start by shifting (multiplying) the accumulator by 10:

```
accumulator = accumulator * 10;
```

Again, this will make more sense after you've done a couple of digits, but for now, we have the initial 0 value times 10 is still 0. Then we add the `digit` translated above:

```
accumulator = accumulator + digit;
```

For the example from the table above, this gives us 4.

Next we move to the next position in the buffer and repeat.[3] Thus we'll translate `'2'` into 2, multiply the `accumulator` by 10 getting 40, and add the 2 to get 42.

Although not complete, things are already starting to take shape. The leading `'4'` from the user's number is now a step higher in the digit places. It started in the ones place and has now moved to the tens place thanks to the multiplication by 10 of the `accumulator`. As this process continues, it will continue to shift over until it falls eventually into its final place of thousands.

The process from the beginning, then, is:

| Next `char` | Translated | `accumulator` **Shifted** | Accumulation |
|:---:|:---:|:---:|:---:|
| `'4'` | 4 | 0 | 4 |
| `'2'` | 2 | 40 | 42 |
| `'1'` | 1 | 420 | 421 |
| `'3'` | 3 | 4210 | 4213 |

Neat! Well, I think so, anyway. . .

## D.3   Numeric Formats

So how flexible is this system? That is, what can a number typed by the user consist of? Well, integers can start with a leading plus or minus sign or not. They can then be any sequence of digits. But when it comes to placing it into memory, the `accumulator` has to fit into the data type's slot. That is, it has to be between the minimum and maximum bounds for the variable's type. These are shown in a table in section **??** for the integer types.

---

[2]For more on ASCII and the traits of it that make this process work, see appendix E.
[3]We keep going until we reach the first thing that isn't a numeric digit `char`.

We often represent such specifications in a format known as a regex or regular expression. The regex for the above integer specification is just `[+-]?[0-9]+`. The square brackets here denote a set of values any one of which is allowed. The dash in the second brackets means a range of values. The question mark means the previous set is optional. And the plus means the second set has to have at least one representative but might have more. It's all quite complicated and suitable for a later course to review in more detail. But for now, I just wanted to show you the regex for the floating-point numbers for comparison:

```
[+-]?([0-9]+(\.[0-9]*)?|[0-9]*(\.[0-9]+))([eE][+-]?[0-9]+)?
```

Wow! That's a mouthful! We see a couple of more notations as well here. Basically, the parentheses are used for grouping as usual. The slash as on a `char` escape says the next bit is different than normal. Our problem is that in a regex a period or dot normally matches any single character and we want it to represent itself here. So we escape the typical meaning to make it be itself. Then the asterisks or stars say to match zero or more of the previous item as opposed to one or more like the plus said earlier. Finally, the vertical bar says to match either the left thing or the right thing.

Let's break that down:

```
[+-]?           # optional sign (like on integer)
(               # group for number itself
    [0-9]+      # either 1 or more digits
    (\.[0-9]*)? #       possibly followed by a . and
                #                0 or more digits
 |              #   OR
    [0-9]*      #       0 or more digits
    (\.[0-9]+)  #       followed by a . and
                #                1 or more digits
)               # one of these has to be there!
(               # group for scientific notation
    [eE]        # either an e or an E
    [+-]?       # optional sign (as int and above)
    [0-9]+      # 1 or more digits
)?              # sci-not is optional
```

I've added comments off to the side after pound signs as is conventional for these things.

Note that we are including the possibility for scientific notation here and that the E normally required to be capital can be lowercase as well. Also note that the number can start with just a dot followed by decimal places and need not have a leading `0` on it. Similary it can end in a dot with no trailing decimal places.

Again, the final accumulated value is subject to fitting into the desired data type. Please see section **??** for more on floating-point data type limits.

Interestingly, these formats are the same as used for the compiler when looking at literals in source code!

## D.4  Wrap Up

This was, sadly, a mere brief delving into this topic of the input buffer and particularly how numbers are translated from a sequence of `char`acters to actual numbers we can work with in arithmetic ways. You'll learn more in later courses, though, so don't fret!

# Appendix E

# Character Encoding

How are letters and symbols stored in the computer? Are numbers always numbers? In this appendix we'll answer such questions!

The answer to the first question is fairly complex so we'll break that down below. The answer to the second is simply no. Numbers start their lives — whether at the user's keyboard or in our own source code — as sequences of individual `char` keystrokes. They are then converted as per the process gone over in appendix D.3.

So that brings us to the question of how are the individual digits stored in the computer? That falls right in line with the first question!

There are actually many answers and which is right depends on your system and sometimes your needs on that system. There are three main players in the field of mapping human letters, digits, and symbols to binary memory values: EBCDIC, ASCII, and Unicode. While others exist, they are less prominent and you may never encounter them.

## E.1   ASCII

ASCII is the American Standard Code for Information Interchange. It is essentially a chart by which the letters, symbols, and digits we hold dear are mapped to numeric values for computer memory storage. When a keystroke is hit, it maps to one of these values via the chart. When such a memory value is displayed, it is mapped via the chart to a display form.

As you can see via the link above, ASCII comprises the English alphabet in both upper and lower case forms, the 10 Arabic digits for forming numbers, and various punctuation marks and other symbols we find useful on a daily basis. It even has slots for spacing characters like the Spacebar, Enter, and Tab keys. Essentially, it has the keyboard values and a few others.

But there are also a few items below the Spacebar called control characters. These are/were used for controling various things between parts of the computer. Some were primarily used for modem communications and may still be used in some communication applications today. Others were intended to control aspects of printing on paper. Since we do little of either today, these codes are little used anymore. But they are still there because ...well, why remove them?

309

## E.2  EBCDIC

EBCDIC is the Extended Binary Coded Decimal Interchange Code.[1] It is another chart that maps human symbols to binary memory values. It is primarily used on older mainframe computers whereas ASCII and Unicode are mostly used on personal computers like a Windows™ box, a Mac™, or a Chromebook™.

It's arrangement was quite different and can give some difficulty in common operations like detecting what group a character belongs to or changing the case (upper versus lower) of a letter. More on that in a bit (section E.4). This is because of the gaps (see the grayed out spaces in the chart above) between parts of the alphabet for instance.[2]

## E.3  Unicode

The Unicode Standard (a set of translation tables that aim to be a universal collection of all human symbols) is for more advanced programs that need heavy internationalization. It is represented in C++ by the `wchar_t` data type and the associated wide strings (`wstring` for the `class` type). A full discussion is beyond the scope of this text, but please see online for more detailed readings on the matter.

The main thing to know about it is that its first — bottom-most — table is essentially the ASCII table although it is called the Latin-1 or Basic Latin subset.

## E.4  Contiguous Runs

The main nice thing about both the ASCII and Unicode systems is the contiguous runs for things like the alphabet and the digits. This makes detecting that we are alphabetic or even which case or a digit easier and even makes it easier to convert upper and lower case back and forth when necessary. The fact that the digits are contiguous in both of these systems and in EBCDIC makes converting from characters to numbers easier as well.

## E.5  A Stern Warning

Don't try to remember the numeric versions of the symbols, letters, etc — that's for uber-geeks. Normal programmers — GOOD programmers — use single-quoted literals in their source code to refer to particular character values they need. So use `'a'` instead of 97, for instance. One reason is that the numeric valules change from system to system — ASCII to EBCDIC, notably. If you worry at all about portability of your code — and you should! — then don't code with numeric values for characters. Always use single-quoted literals instead.

## E.6  Wrap Up

Hopefully this exploration of different ways to encode human symbols in the computer has proven interesting. Feel free to explore more on your own!

---

[1]Slightly redundant there, eh?

[2]While the gaps contain valid codes, they don't align with the ASCII table and that was the purpose of the chart I linked above. If you read elsewhere on the page it explains the other bits of the code in more detail.

# Appendix F

# Timing Program Events

The idea of this appendix is to talk about automating the task of finding out how long certain parts of the program are taking to execute. There are different methods available depending on how far you've come in your studies. Let's start basic with the `ctime` function: `time(nullptr)`.

## F.1   Using time(nullptr)

Since `time(nullptr)` returns the number of seconds since a fixed point in the past — the computer epoch of January 1, 1970, we can use two readings from it to tell how far apart they are and therefore how long the task between the two readings took!

Um, what? Just call the function twice — once before and once after the code is run. Then subtract to find the number of seconds elapsed. Oh...why didn't you say that in the first place?

```cpp
time_t start, end;

start = time(nullptr);

// do code to be timed

end = time(nullptr);

cout << "That took " << end-start << " seconds.\n";
```

But, since most code events are blindingly fast, we might not have even a single second for some of them. We'll need to do one of two things with this idea: time lots of repetitions of the event and take an average or find out how many times the event can run in a single second and take the reciprocal.

### F.1.1   Method One

The first way is to average many repetitions of the code event we want to time:

```cpp
time_t start, end;

start = time(nullptr);

for (long i = 0; i != LOTS; i++)
{
    // do code to be timed
}

end = time(nullptr);

cout << "That took " << (end-start)/static_cast<double>(LOTS)
     << " seconds.\n";
```

You just have to decide how much LOTS should be. You might even make this a program input and adjust it on different runs to see if you get better results.

### F.1.1.1  Dealing with Data Re-Organization

One thing to watch out for, of course, is making the timed event in the loop consistent. For example, if you were trying to time a sorting algorithm with a short-circuit condition (like bubble sort), timing this sort directly 10000 times would really only time the whole algorithm once and the single short-circuit loop 9999 times. A single loop through a vector isn't very slow — nearly instantaneous for most vectors. So your timing is likely to still be 0.

To adjust for this circumstance, you can do one of two things: replace the original vector contents between sorts or randomly shuffle the data between sorts. Although it may not seem like it, either of these approaches will give you comparable timings. The problem is, this copying or shuffling must be done inside the timing loop and so your timing will reflect not only the sorting done, but the vector re-organization as well. To compensate for this, simply time the re-organization separately and subtract this from your 'sort' timing:

```cpp
time_t start, end;
double reorg_time;

start = time(nullptr);
for (long i = 0; i != LOTS; i++)
{
    // re-organize vector
}
end = time(nullptr);
reorg_time = (end-start)/static_cast<double>(LOTS);

start = time(nullptr);
for (long i = 0; i != LOTS; i++)
{
    // re-organize vector
    // sort vector
}
end = time(nullptr);

cout << "That took " << (end-start)/static_cast<double>(LOTS) - reorg_time
     << " seconds.\n";
```

Even if you are comparison-timing many sorts, the re-organization timing only needs to be done once.

You can also improve your timing by making the data larger — not the actual values, but the number of data elements. If the thing you are timing is `vector` processing, for instance, you can increase the number of elements in the `vector` that are being processed.

## F.1.2   Method Two

This method of timing program execution also uses the `time(nullptr)` function from the `ctime` library, but it takes a different approach.  In the previous method, we ran our code many, many times and calculated how many seconds this took — on average.  Another way to look at this idea is that we aren't sure how many times it will take running our code to make a second, so we've instead just run our code long enough to make many seconds and divided out how many times we ran to get an average:

$$\frac{seconds}{runs} = \text{sec/run}$$

However, since we expect our code to run multiple times each second, we could simply run it enough times so that the current second changes.  Then, by counting how many runs that took (i.e. how many runs we did in a second), we can calculate:

$$\frac{1}{\frac{runs}{second}} = \text{sec/run}$$

That's just what we wanted and it only took us a second instead of the several/many seconds the previous method took!

```cpp
double reorg_time;
unsigned long count;
time_t start;

count = 0;
start = time(nullptr);
do
{
    // re-organize/setup
    count++;
} while (time(nullptr) == start);
reorg_time = 1/static_cast<double>(count);

count = 0;
start = time(nullptr);
do
{
    // re-organize/setup
    // do code to be timed
    count++;
} while (time(nullptr) == start);

cout << "That took " << 1/static_cast<double>count - reorg_time
    << " seconds.\n";
```

### F.1.2.1   Adjusting for System Differences

Well, sort-of...  In fact, because of all the things computers are always doing, we'll not likely get an accurate reading all the time. To fix this, we should take several such readings and then find the median number of times the code ran in a second.[1] Even so, this method can still be done in a consistent number of seconds (say 9-10) instead of the more arbitrary time that the previous timing method took.

We should still take account of re-organization/setup time for this method, but we can use this method to time both the setup and actual code running. *smile*

So, what might this look like? Something like this, perhaps:

```cpp
vector<unsigned long> counts;
double reorg_time;
unsigned long count;
time_t start;

for (short rep = 0; rep != 9; rep++)
{
    count = 0;
    start = time(nullptr);
    do
    {
        // re-organize/setup
        count++;
    } while (time(nullptr) == start);
    counts.push_back(count);
}
// sort counts vector
reorg_time = 1/static_cast<double>(counts[4]);

counts.clear();
for (short rep = 0; rep != 9; rep++)
{
    count = 0;
    start = time(nullptr);
    do
    {
        // re-organize/setup
        // do code to be timed
        count++;
    } while (time(nullptr) == start);
    counts.push_back(count);
}
// sort counts vector
cout << "That took " << 1/static_cast<double>(counts[4]) - reorg_time
     << " seconds.\n";
```

And as your knowledge grows, you'll likely find more and more ways to time events on different systems.

---

[1]See the elsewhere for information about taking a median of a set of data.

## F.2  Using chrono

The `chrono` library and `namespace` come with C++ and offer timings down to the nanosecond depending on hardware support. This process is a little more syntax heavy than that for using the `time(nullptr)` function, but well worth it! The basic code can be:

```cpp
auto start{ chrono::steady_clock::now() };
// do code to be timed
auto end{ chrono::steady_clock::now() };

chrono::duration<double> elapsed_seconds = end - start;
cout << "That took " << elapsed_seconds.count() << " seconds.\n";
```

But to get more precision out of it than decimal seconds, just use a more precise clock and change up the duration with a cast:

```cpp
double nano, milli;

auto start{ chrono::high_resolution_clock::now() };
// do code to be timed
auto end{ chrono::high_resolution_clock::now() };

milli = chrono::duration_cast<chrono::milliseconds>(end - start).count();
cout << "That took " << milli << " ms.\n";

nano = chrono::duration_cast<chrono::nanoseconds>(end - start).count();
cout << "That took " << nano << " ns.\n";
```

Of course, you don't need any of the `chrono::` if you have a `using` directive for it.

### F.2.1  What About Those Other Issues?

The `chrono` timer is often so accurate that you don't need multiple loops around your code to get an accurate time for it. That doesn't mean that the median method wouldn't help with system congestion issues, but you generally don't need it.

## F.3  Wrap Up

No matter what way you choose to time a program event — some bit of code, always remember to have fun!

# Appendix G

# Bit Manipulation

Coming Soon!

## G.1   Basics

Coming Soon!

### G.1.1   What's a Bit?

Coming Soon!

### G.1.2   What's a Flag?

Coming Soon!

### G.1.3   How's It Come Together?

Coming Soon!

## G.2   Types of Manipulations

Coming Soon!

### G.2.1   Setting a Flag

Coming Soon!

317

## G.2.2   Testing a Flag

Coming Soon!

## G.2.3   Unsetting a Flag

Coming Soon!

## G.2.4   Toggling a Flag

Coming Soon!

## G.2.5   Masking for Groups of Bits

Coming Soon!

# G.3   Benefits and Examples

Coming Soon!

# G.4   Other Features

Coming Soon!

## G.4.1   Multiplying and Dividing

Coming Soon!

## G.4.2   Swapping

Coming Soon!

# G.5   Helpers

Coming Soon!

## G.5.1   enumerations

Coming Soon!

### G.5.1.1   More Details

Coming Soon!

#### G.5.1.1.1   The Underlying Type

Coming Soon!

#### G.5.1.1.2   enumeration classes

Coming Soon!

### G.5.2    structures

Coming Soon!

## G.6    Wrap Up

Coming Soon!

# Appendix H

# Files Extras

## H.1   Formatting

Coming Soon!

## H.2   Tieing Streams Together

Coming Soon!

## H.3   Index 'Files'

Coming Soon!

## H.4   filesystem Library

Coming Soon!

## H.5   Binary Files

Coming Soon!

## H.6   Wrap Up

Coming Soon!

# Appendix I

# Advanced Memory Management

Coming Soon!

## I.1   Memory Management

Coming Soon!

### I.1.1   Allocation and Deallocation

Coming Soon!

### I.1.2   Dereferencing

Coming Soon!

## I.2   Smart Pointers

Coming Soon!

### I.2.1   Making Our Own

Coming Soon!

### I.2.2   From the Standard Libraries

Coming Soon!

## I.3   Move Semantics

Coming Soon!

## I.4 Wrap Up

Coming Soon!

# About the Author

Jason James graduated with his BS in Computer Science (minor in Applied Mathematics) and his MS in Computer Science (Theory emphasis) both from the then University of Missouri at Rolla (UMR) now MST (Missouri University of Science and Technology). While working on his PhD (Artificial Intelligence emphasis; ABD), Jason taught introductory programming to engineering students using both FORTRAN and C++. He also taught Freshman and Sophomore topics in Computer Information Systems at an adjunct campus of Columbia College during this time.

When he moved to Chicagoland and the wonderful world of William Rainey Harper College, Jason focused on teaching Freshman and Sophomore Computer Science courses and the occasional mathematics course — especially Discrete Math. During his twenty years at Harper, Jason has also taken over as chair of the Computer Science department; served on committees for Curriculum, Academic Standards, and Testing and Placement; co-mentored the Robotics/Engineers club; and started a Computer Science club last Fall (hopefully converting to an ACM Student Chapter this Spring), but teaching remains his focus and heart.

Jason maintains membership in the international Computer Science group: the ACM (Association of Computing Machinery). He also keeps up his membership in the IEEE (Institute of Electrical and Electronics Engineers, Inc.). In both he is especially interested in their education-focused sub-groups. Jason also tries to attend conferences of the CCSC (Consortium for Computing Sciences in Colleges) whenever he can get away.

When not playing Dungeon & Dragons with his wife and friends, he and his wife take care of their two beautiful sons and two ornery cats. In his *spare* time, Jason enjoys developing formulae for counting the results of dice rolls — with an eye toward a 'nice' standard deviation formula; using the LaTeX type-setting system to prepare lecture supplements and exams; creating lecture supplements and assignments online; and the development of a calculator language & its interpreter which are being applied to classroom management software (such as a gradebook and an exam analyzer). He's also recently taken to writing — converting those lecture supplements into an OER textbook for [at least] his students at Harper College.