Full Public Release Version
FALL 2023

CS 246 COURSE NOTES

December 11, 2023

Amol Venkataraman
University of Waterloo
`avenkata@uwaterloo.ca` / `amolven16@gmail.com`
Discord: `amolven`

Notice to Reader

1. These notes have been created based on all the content covered in **CS 246** lectures in the **Fall 2023** term. These notes cover everything that was covered in lectures, as well as any more information that was deemed important.
2. For all the code examples provided, you can assume that all the required library imports and namespace setting have been done already unless mentioned otherwise. Any other code that can be assumed to have been run will also be mentioned before each example.
3. The content in these notes assumes prior knowledge in C and basic Data Structures and Algorithms, which was covered in CS 136.
4. Exam review resources are also available in this version of the notes, and start after the regular course notes end.
5.
6. If you discover any errors, please inform the author using the contact information provided on the first page.
7. These course notes contain **all** the content covered throughout the term.
8. This version of the notes contains **115** pages (including title and notice pages).

I hope you enjoyed these notes throughout the term, and I will you the best of luck on all your final exams! :)

You can access the latest version of these notes at: `https://app.amolven.me/static/Notes/CS_246_CN.pdf`.

Table of Contents

Introduction

In CS 136, the C99 standard was used. In this course, the C++20 standard is used.

C++ is 100% backwards-compatible with C, and every C program is also a valid C++ program. However, many aspects of C syntax is discouraged in C, since C++ has better functions, tools, and syntax, that faster and/or safer and/or more convenient.

Comparison between C and C++

Hello world in C99:

```
#include <stdio.h>

void main() {
    printf("Hello, World!\n");
}
```

Hello world in C++20

```
import <iostream>; // New C++20 syntax to simplify imports
using namespace std; // Adding this line means you don't have to prepend
commands with std::

int main() { // main function has to return int
    cout << "Hello, World!" << endl; // C++ syntax for printing
    return 0; // Default return value of 0
}
```

The `import` statement speeds up compile time by making all libraries get compiled once manually and then cached.

`endl` is similar to `"\n"` from C in that it prints out a new line, but it also clears the output buffer, forcing everything that was printed to get displayed on the screen immediately if it hasn't already.

How to run C++ programs

Compiling and running the C++20 program. This method requires the setup from A0 to be done first.

```
// Compile system headers. This method reduces unnecessary resource waste
from recompiling
g++20h iostream
```

```
// Run the program
g++20m hello.cc [-o myprogram] // Not using -o will make the output default
to a.out
```

C++ imports without C++20 modules

```
#include <iostream> // Instead of the import syntax. Slower but more stable
```

```
g++20i hello.cc [-o myprogram] // Not using -o will make the output default
to a.out
```

Input / Output

Recall that there are 3 different streams, **stdin**, **stdout**, and **stderr**.

There are different functions to interact with each stream.

- `cout << x;` will print out the contents of `x` to the **stdout** stream.
- `cin >> x;` will read from the **stdin** stream into the variable `x`.
- `cerr << x;` will print out the contents of `x` to the **stderr** stream.

Since C++ is 100% backwards compatible with C, we can use all the C functions, like `printf`, `scanf`, `"\0"`, etc. in C++. However, it is strongly recommended to only use the C++ functions whenever possible since they are usually faster, more efficient, easier to use, and safer.

Example: Reading 2 `int`s, adding them up, and printing it out again.

```
import <iostream>;
using namespace std;

int main() {
    int x, y; // Declaring 2 ints
    cin >> x >> y; // Read into x and then into y
    cout << x + y << endl; // cout can directly print integers without
formatters
    // A return statement is optional. main defaults to returning 0
}
```

Operators:

- `cin >> x;` - "get from"

- `cout << x;` - "put to" The arrows point in the direction of information flow. For example, in `cout`, the arrows point from `x` and to `cout` (which can be thought of as the **stdout** stream)

How could this go wrong? There are 3 potential errors:

1. User gives us non-integer input.
2. User supplies EOF (via *Ctrl + D*).
3. What if the integers are too big / small?

In order to make it possible to check if any of these has occurred, `cin` contains a **fail bit** and an **EOF bit**.

- **Fail bit**: `cin.fail()` - Set to `true` when the read fails (incorrect type or out of bounds)
- **EOF bit**: `cin.eof()` - Set to `true` when the EOF is reached (program runs out of input)

By using implicit conversion to `bool`, we can check for either an input failure or EOF:

```
if (cin)
```

The above statement will return true if there are no errors, and returns false if either Fail or EOF bits are true.

It is important to note that once **Fail bit** has been set as *true*, it will remain that way until the program finishes executing. The only way to reset it is to manually execute the following statement:

```
cin.clear();
```

Example: Read integers from **stdin** and print them out until failure.

```
// V1 - Basic statements
import <iostream>;
using namespace std;

int main() {
    int i;
    while (true) { // Iterate forever
        cin >> i; // Read input
        if (cin.fail()) break; // Exit out of the loop if there is an error
        cout << i << endl; // Print out the input that was read
    }
}
```

```
// V2 - Using implicit conversion to bool
import <iostream>;
```

```
using namespace std;

int main() {
    int i;
    while (true) {
        cin >> i;
        if (cin) break; // Implicit conversion
        cout << i << endl;
    }
}
```

```
// V3 - Simplification
import <iostream>;
using namespace std;

int main() {
    int i;
    while (true) {
        if (!(cin >> i)) break; // Thus is possible since cin >> i returns
cin
        cout << i << endl;
    }
}
```

```
// V4 - Shortest, Best
import <iostream>;
using namespace std;

int main() {
    int i;
    while (cin >> i) { // Keep iterating as long as there is valid input
        cout << i << endl;
    }
}
```

Example: Read and print all integers from **stdin**, skipping invalid input.

```
import <iostream>;
using namespace std;

int main() {
    int i;
    while (true) {
        if (cin >> i) cout << i << endl; // Read input and print it out if it
is successful
        else {
            if (cin.eof()) break; // Break out of the loop when EOF is
reached
            cin.clear(); // Clear the failure bit
            cin.ignore(); // Discard the current (invalid) character being
read and go to the next position
        }
    }
}
```

C++ Features

Overloading

In C, the `<<` and `>>` are bit shift operators.

In C++, the functionality of these operators depends on the context that they are used in.

- `x >> 3`: Has an `int` on the LHS, and thus functions as the bit shift operator.
- `cin >> x`: Has `cin` on the LHS, and thus reads input from **stdin** to `x`.

This is known as overloading.

Another effect of overloading is that the return value can also be customized. For example, `cout << x;` returns `cout`. This might seem weird at first, but it means that we can simplify our print statements. For example:

The code below

```
cout << "The answer is: ";
cout << x;
cout << endl;
```

Can be simplified to:

```
cout << "The answer is: " << x << endl;
```

The statement is read from the left to the right, and `cout << "The answer is: ";` executes first. That would then return `cout`, which would replace the protion of the statement that was just executed. The statement would then become `cout << x << endl;`. Similarly, the variable `x` gets printed first, and finally, `endl`.

Strings

In C, strings are sequences of characters terminated by `\0`. C strings offer very low level control, but come with many disadvantages:

- We need to manage out own memory.
- We need to save space for a null terminator (`\0`).
- We should make sure that we don't overwrite the null terminator.

In C++, there is a `string` type, `std::string`. It is found in the library `<string>`. It has numerous advantages.

- You no longer need to manage your own memory.
- No need to worry about allocating space for `\0`.

The syntax to declare a C++ string is:

```
string s = "Hello"; // "Hello" is a string literal from C, but the C++
compiler converts it into a C++ string
```

Since C++ supports overloading (changing behaviour of functions based on their input values), many operations can be applied to C++ strings:

- **Comparison:** `s1 == s2`, `s1 != s2`
- **Sorting:** (In lexicographical order) `s1 < s2` or `s1 > s2`
- **Length:** `s.length()` [This function runs with constant **O(1)** time since the length is cached]
- **Concatenation:** `s1 + s2`, `s3 += s2`
- **Accessing characters:** `s[1]`, `s[1]`, `s[2]` [These represent the individual characters on the string]

Reading and printing `string`s:

```
import <iostream>;
inport <string>;
using namespace std;

int main() {
    string s; // Declare a string variable
    cin >> s; // Read input from stdin into s
    cout << s << endl; // Print out contents of s
}
```

The `cin` statement reads one *word* from **stdin** when it is supposed to read to a `string`. A *word* is a sequence of characters that are not separated by a whitespace.

If we want to read whitespaces, we need to use the `getline` command. The syntax for it is:

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

Streams and Abstraction

Both `cin` and `cout` are examples of **streams**.

- `cin` is of type `std::istream`
- `cout` is of type `std::ostream`

These are examples of **abstractions**. The **streams** act as "*interfaces*" to perform different functionality.

- `istream` provides `>>` as an interface to "read from"
- `ostream` provides `<<` as an interface to "write to"

There are different extensions of this basic **stream** functionality with different use cases. For example, for files:

- `ifstream`: reads from a file
- `ofstream`: writes / prints to a file

The `ifstream` and `ofstream` are available in the `<fstream>` library.

Anything that can be done with an `istream` / `ostream` can be done with an `ifstream` / `ofstream`

Example: File access in C - Program to open a file and print out it's contents.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char s[256]; // We have to set an arbitrary maximum line length
    FILE *f = fopen("file.txt", "r"); // We need to open the file before
reading
    while (true) {
        fscanf(f, "%255s", &s); // We need to add complicated formatters
        if (feof(f)) break;
        printf("%s\n", s);
    }
    fclose(d); // We need to remember to close the file before the program
ends
}
```

This seems unnecessarily complicated for what it does. C++ on the other hand, has a much simpler solution.

```
import <iostream>;
import <fstream>;
import <string>;
using namespace std;

int main() {
    string s;
    ifstream f{"file.txt"}; // File opening and reading is handled for us
    while (f >> s) cout << s << endl; // We use the exact same systex as
reading from stdin
} // The file is closed automatically when main ends
```

This syntax is much easier to both write and understand.

Note the initialization syntax: `ifstream f{"file.txt"}`. This is valid syntax for initializing any C++ variable. For example: `int i{5}`.

Another application of `istream` / `ostream` abstraction is `string`s. These functions can be found in the library `<sstream>`

- `std::ostringstream`: Allows us to print to a string (with formatting).
- `std::istringstream`: Allows us to "read from" a string.

Example: Convert an `int` to a `string`.

```
string intToString(int n) {
    ostringstream oss;
    oss << n;
    return oss.str();
}
```

Example: Revisit printing all valid integer input from **stdin**.

```
import <iostream>;
import <sstream>;
import <string>;
using namespace std;

int main() {
    string s;
    while (cin >> s) {
        int i;
        // The variable iss is being declared inside the if statement
        if (istringstream iss{s}; iss >> i) cout << i << endl; // The if
statement will return true when iss could convert to i without any issues
    }
}
```

Note that this example has slightly different behaviour than the previous example. If provided with `"ab12cd"` as input:

- The original version would print `12`.
- This new version would not print anything.

Another thing to note is that there is no need to clear iss's fail bits since it is reinitialized with each iteration of the loop.

Command Line Arguments

These are provided at runtimr and are used to customize the program's behaviour. For example:

```
./program abc 123
```

They can be accessed in both C and C++ in the following way:

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

Here, `argc` represents the number of arguments given to the program (**INCLUDING the program name itself**). `argv` is an array of strings each of which represents an individual argument.

`argv[0]` stores the name of the program that is being run. `argv[1]` to `argv[argc - 1]` store the command line arguments given to the program. `argv[argc]` is refined to be the `NULL` pointer.

Note: All command-line arguments in `argv` are **C-style** strings (char pointers). In order to perform C++ exclusive functions like string comparison, concatenation, and easy modification, they need to be converted to C++ strings first.

Example: Converting all arguments to C-strings before use.

```
import <iostream>;
import <string>;
using namespace std;

int main(int argc, char **argv) {
    for (int i = 0; i < argc; i++) {
        string arg = argv[i];
        ... // Do whatever is required with the arguments
    }
}
```

Example: Sum all integers given on the command line.

```
import <iostream>;
import <sstream>;
import <string>;
using namespace std;

int main(int argc, char *argv[]) {
    int total = 0;
    for (int i = 0; i < argc; i++) {
        string arg = argv[i];
        int n;
        if (istringstream iss{arg}; iss >> n) total += n;
    }
    cout << total << endl;
}
```

Default Function Parameters

Example: A function to print out the contents of a file, with a default parameter

```
void printFile(string name="file.txt") {
    ifstream file{name};
    string line;
    while(getline(file, line)) cout << line << endl;
}
```

Running `printFile("abc.txt");` will read and print from **abc.txt**. One the other hand, running `printFile();` will read and print from the default file, which is **file.txt**.

Optional parameters **MUST** come last in the function's arguments, in order to avoid any ambiguity. Furthermore, if a function has more than one default argument, then if any of those arguments need to be provided a value, then all the arguments (both ones with and without default values) before it also need to be provided values in order to prevent ambiguity.

Since it is impossible for the function itself to determine whether the memory is unitialized or it has been initialized to a "strange" value, only the function caller has all the information to be able to set up the default arguments. Because of this, the caller of a function always sets up it's default arguments, just like it does for resular arguments.

For similar reasons, default parameters are specified only in the interface (header) file.

Overloading Revisited

Example: A function that can negate both `int`s amd `bool`s.

```
int neg(int x) { return -x; } // neg(100) -> -100
bool neg(bool b) { return !b; } // neg(true) -> false
```

By providing multiple definitions of a function with a different number and/or types of arguments, we can have different versions of the same function that work with different types of arguments and (potentially) return different things.

The compiler chooses the correct overload (version of the function) during compile time based on the number and types of arguments. The compiler can not overload based on return type. It is also important to note that there can only be one version of a function with a certain type of arguments (for example, there can only be one version of a function that accepts two `int`s, regardless of the function contents or return type).

This has already been done with default C++ operators, like `==`, `<<`, `>>`, and `+=`.

Structs

Similar to C, C++ has structures, that allows for storing different types of data together as one unit. While the C syntax for structures still works with C++, there are some changes that make using structures more convenient.

Example: Structure in C++ for the node of a linked list.

```
struct Node {
    int data;
    Node *next; // In C++, we do not have to prepend with struct
}; // Remember to end with a semicolon
```

Constants

Constants have values that can not be changed during the course of the program's execution. The syntax to declare a constant is:

```
const int MAX_GRADE = 100;
```

Any changes to `MAX_GRADE` will result in a compilation error.

We can do something similar for structures.

```
const Node{5, nullptr};
```

Here, neither the `data` field, nor the `next` field can be changed.

NULL pointers

In C, the null pointer is written as `NULL` or `0`.

In many C libraries, there is actually a line like

```
#define NULL 0
```

In C++, the term `nullptr` is used instead in order to prevent ambiguity with overloaded functions. For example

```
void f(Node *p);
void f(int x);
```

Using `NULL` would be converted to `0`, and then it is not possible to determine which version of the function needs to be called. On the other hand, the `nullptr` term is automatically converted to any pointer type, and does not cause any any similar ambiguity.

References

Why does `getline(cin, s)` edit `s`, even though it is not given a pointer? What about `cin >> x`?

C++ has another pointer-like type called a reference. **Example:** Declaring a reference.

```
int x = 5;
int& z = x; // z is an "lvalue reference" to x
```

After running the above code, performing *most* things to `z` will have the same result as doing the same thing to `x`.

- Both `x` and `z` refer to the same thing. `z` is an alias for `x`.
- Any changes in `z` are also reflected in `x`.
- The way references are implemented depends on the compiler:
  - Sometimes (usually in simple cases), the variable name is simply replaced.
  - Other times, a a hidden pointer is used that is automotically dereferenced.
- References are like constant pointers with automatic dereferencing.
- References use a similar syntax to the "address of" operator, but the exact meaning depends on the context:
  - If the `&` appears in a type, then is is a **reference**. Ex. `void f(int &n);`

- If the `&` is part of an expression, then is is the **address of** operator. Ex. `g(&x);`

Lvalue vs Rvalue: An Lvalue is anything that always has a defined region of memory where it is stored, like a variable or a constnt. On the other hand, an Rvalue is something that does not always have a defined region of memory. For example, when running `func(5)`, the value `5` is an Rvalue since it is not stored on a defined region of memory.

What you cannot do with references:

- Cannot be left unitialized
  `int& z;` Causes compilation error
- Cannot assign a temporary value (Rvalue) to an Lvalue reference
  `int& z = 5;` Causes compilation error
  Similarly, Lvalue references can only be initialized to Lvalues, like variables
  `int& z = x + y;` or `int& z = f();` will cause a compilation error since both values on the right are temporaries with no long-term memory resolution.
- Cannot create a pointer to a reference
  `int*& x = ...;` This is allowed, since it is a reference to a pointer
  `int&* x = ...;` This is not allowed, since it is a pointer to a reference
  References are not always stored in memory, so pointers to references aren't allowed.
- Cannot create a reference of a reference
  `int&& z = x;` is not allowed
  There are 2 reasons for this not being allowed. One is that it is completely pointless and not useful. The other reason is that the `&&` syntax is reserved for something different (Rvalue references).
- Cannot create an array of references
  `int& arr[3] = {x, y, z};` will cause a compilation error
  Just like for some of the other prohibited actions, this is not possible since references are not always stored in memory.

References are most useful for function arguments.

Examples: A function using a reference to get it's input value.

```
void inc(int& x) { ++x; } // The function takes a reference of the variable
passed in and stores it in x

int x = 5;
inc(x); // Changes will be reflected here directly
// There is no need to overwrite the variable value or to use pointers
```

`getline(cin, x)` and `cin >> x` work because `s` and `x` are passed by reference.

Methods for passing parameters

Suppose we have need to pass in a lot of data into a function, in the following form:

```
struct ReallyBig { ... };
```

There are 4 main ways to do this:

```
// Pass by value - very simple, but extremely slow, since a copy of the
variable needs to be made
void f(ReallyBig rb);

// Pass by pointer - Fast, no copy is made, but can be confusing, and changes
are reflected
void g(ReallyBig* rb);

// Pass by reference - Fast (similar to speed of a pointer), no copy is made,
and changes are reflected
void h(ReallyBig& rb);

// Pass by const reference - Fast, no copies, no changes, easy to reason
about
void i(const ReallyBig& rb);
```

How to take in args when designing a function:

- Go to: Constant Lvalue reference - Fast, easy to reason about
- If changes should be reflected: Pass by reference
- Pass-by-pointer: Changes reflected, and ability to pass `nullptr`.
- Pass-by-value: Useful 2 situations
    - Small arguments, < 8 bytes
    - Need temporary version of the argument to work with

Constant Lvalue references may also bind to temporaries

```
void g(const int& y);

g(z); // Works
g(5); // Also works. Compiler will reserve a temporary memory location for 5
```

All streams have a similar signature (stream << out OR stream >> in), and use references because streams have copying disabled in order to prevent ambiguity.

Dynamic Memory

Recall, Dynamic memory in C:

```
int* arr = malloc(n * sizeof(int)); // Allocates memory
... // Code that uses arr
free(arr); // Returns memory back to the OS
```

This works, but has some issues, namely:

- It is possible to store the wrong type of value.
- Calculating the size of the structure and passing it in is not very elegant miscalculating can cause errors.

Dynamic memory in C++:

```
Node* p = new Node{5, nullptr}; // Allocated memory
... // Code that uses p
delete p // Returns memory back to the OS
```

All local variables are on the stack. Stack-allocated variables are deallocated when they go out of scope.

Memory allocated with `new` is on the heap. Heap memory persists untill `delete` is called. Failure to delete will cause memory leaks, which could result in the program crashing.

Example: C++ arrays on the heap

```
int* arr = new int[10];
...
delete[] arr;
```

C++ operators `new` and `delete` are incompatible with the C operators `malloc` and `free`. Memory allocated with `new` needs to be freed with `delete`, and memory allocated with `malloc` needs to be freed with `free`.

C++ doesn't have an equivalent operator for `realloc`. The data stored in the array needs to be manually copied over.

Always pair `new` with `delete` and `new[]` with `delete[]`. Mixing and matching causes undefined behaviour.

Methods for returning values

There are many different ways to return values in C++, which all vary in many ways.

```
// Return-by-value
Node getMeAMode(int x) {
    Node n{x, nullptr};
    return n; // n is copied from getMeANode's stack frame to caller's frame
} // This works, but copying is slow

// Return-by-pointer (broken)
Node* getMeANodePtr(int x) { // WARNING: Does not work
    Node n{n, nullptr};
    return n; // Crashes at runtime if caller uses the return value
} // n is destroyed when the function ends, returning a pointer to
inaccessible memory

// Return-by-pointer (working)
Node* getMeANodePtr(int x) { // Memory lives past the end of the function
call
    Node *p = new Node{x, nullptr};
    return p;
} // Fast, works, user much delete the pointer manually

// Return-by-reference (broken)
Node& getMeANodeRef(int x) {
    Node n{x, nullptr};
    return n; // Dangling reference, n is destroyed on return
}
```

Return-by-reference is rare, and is only used when returning non-local values, like with `cin` and `cout`.

Advise: Use return-by-value for most scenarios. Usually, the compiler can optimize the copy away from the `return` (elision, move semantics).

Operator Overloading

Recall from earlier how the `<<` and `>>` operators were overloaded to be used for input/output. We can actually overload any operator to have customized behaviour for certain input types. Most notably, this includes custom objects (`struct`s and `class`es).

The template for a function that overloads the operator `op` is:

```
<return_type> operator<op>(...<input(s)>...) {
    ... // Perform computation
    return <return_value>
}
```

Let's consider a custom structure

```
struct vec {
    int x, y;
};
```

```
vec x{1, 2};
vec y{3, 4};
vec z = x + y; // We want the plus sign to do addition

vec w = 2 * x; // We want multiplication to work as expected
vec v = x * 2;
```

This can be achieved by overloading the plus sign and asterisk operator

```
// Operloading addition
vec operator+(const vec& v1, const vec& v2) {
    vec r{v1.x + v2.x, v1.y + v2.y};
    return r;
}

// Operloading multiplication
vec operator*(int k, const vec& v) {
    return {k * v.x, k * v.y} // Not necessary to save data to variable since
it is automatically assumed based on the return type
}

// Overloading multiplication again in order to make the other type of
multiplication work
vec operator*(const vec& v, int k) {
    return k * v; // Call the multiplication operator we defined earlier
}
```

Example: Overloading `>>` and `<<` operators

```
struct Grade {
    int n;
};

ostream& operator<<(ostream& out, const grade& g) {
    return out << g.n << "%";
} // This code will work not just for cout but also for writing to files

istream& operator>>(istream& in, grade& g) {
    in >> g.n;
    if (g.n < 0) g.n = 0;
    if (g.n > 100) g.n = 100;
    return in;
} // Similarly, this code also works for files
```

Separate compilation

Speed up compilation by only compiling what is necessary.

Interface files (`.h`) - Provide declarations for functions Implementation files (`.c`, `.cc`) - Provide definitions for functions writtenE

Interface file (`vec.cc`) This is new syntax for C++20 and is NOT backwards-compatible with `#include`.

```
export module vec; // This tells the compiler that this file is an interface
file

export struct vec { // Perfixed with "export" makes it availabe to clients
(importers)
    int x, y;
}
export vec operator+(const vc& v1, const vec& v2);
```

Client code (`main.cc`)

```
import vec; // Import the module

int main() {
    vec v{1, 2};
    vec w = v + v;
}
```

Implementation file (`vec-impl.cc`)

```
module vec;

vec operator+(const vec& v1, const vec& v2) {
    return {v1.x + v2.x, v1.y, v2.y};
}
```

Importing

We generate an object file (`.o`) from a source (`.cc`) file using the following command:

```
g++20m file.cc -c # Produces file.o
```

Files are compiled in dependency order:

1. Interface files first
2. Implementation / client code second

The order for the previous example would be:

1. `vec.cc`
2. `vec-impl.cc`
3. `main.cc`

The command to link all the object files and produce an executable is:

```
g++20m vec.o vec-impl.o main.o -o program
```

Now, the executable can be run:

```
./program
```

There are no tools like **make** currently available to automate separate compilation for C++20 imports. Instead, we first run the command to generate object files twice. For the first time, some will work correctly while others will fail due to dependency issues. For the second time, all the dependencies should've been compiled by the previous command, and so any remaining files will compile. After this, the files can be linked and an executable can be generated. If we only change one file, then we just need to recompile that file and relink.

Classes

Classes are the main difference between C and C++

At it's simplest form, a `class` is a `struct` with functions inside it.

Example: An example of a `class`.

```
// student.cc
export struct Student {
    int assns, mt, final;
    float grade(); // Function inside struct
};

// student-impl.cc
float Student::grade( {
    return 0.4 * assns + 0.2 * mt + 0.4 * final;
})
```

```
// main.cc
Student s{60 70, 80};
cout << s.grade << endl;
```

The implementation of class function can be written in either the interface (inside the structure) or implementation file (like a regular function). However, writing it in the implementation file is encouraged since it is considered better syntax.

`Student` is a **class**. (there is a `class` keyword, but that isn't used here and will be discussed later)

`s` is an **object** - a particular instance of a class.

`::` - "scope resolution operator". One use is to define a function inside a class. It is also used to provide the namespace.

`Student::grade` is a "member function" or "method".

When we call a member function (like `s.grade()`), the variables in the function (`assns`, `mt`, and `final`) are bound to the variables in the class with corresponding names (`s.assns`, `s.mt`, and `s.final`).

The compiler accesses the correct fields using an implicit parameter called `this`. It is a pointer to the object that the method was called on.

Inside the function, `<var>` and `this -> <var>` mean the same thing, where `<var>` is any field in the class/struct.

Big Five

Initializing Objects

To control how objects are created, write a constructor (ctor). Constructors have no return type and have the same name as the class/struct.

Exmaple: A class that has a constructor.

```
// student.cc
export struct Student {
    int assns, mt, final;
    float grade();
    Student(int assns, int mt, int final);
}

// student-impl.cc
Student::Student(int assns, int mt, int final) {
    this -> assns = assns;
    this -> mt = mt;
    this -> final = final;
    // Note that "this" is used here becuse the arguments and struct fields
have the same names, and we need to disambiguate the field and argument.
}

// main.cc
Student s{60, 70, 80};
Student s = Student{60, 70, 80}; // This is equivalent to the line of code
abve
Student *p = new Student{60, 70, 80}; // This represents heap initialization
```

If you do not use a field, the compiler will give field-by-field initialization like in C. Meanwhile, if you write your own constructor, then the compiler will use it instead of field-by-field initialization.

The major benefit of constructors is that they are functions and can thus be customized as such:

- Argument bounds checking
- Overloading
- Default parameters

Any constructor that can be called with 0 arguments is a "default constructor". This can be either the constructor has no arguments or because it has default parameters. If we do not write any constructors, then the compiler provides the default constructor.

Compiler-provided default constructor:

- Primitive fields (`bool`, `int`, `char`, pointers) - left uninitialized
- Object fields (`class`, `struct`) - calls the object's default constructor

As soon as you write a constructor, the compiler-provided default constructor goes away. Having no default constructor and initializing the class without any arguments causes a compilation error.

You can have as many constructors as you need, as long as each one has a unique set of parameters.

Example: Class with default constructor

```
struct Student [
    int assns, mt, final;
    Student(int assns = 0, int mt = 0, int final = 0) {
        this -> assns = assns;
        this -> mt = mt;
        this -> final = final;
    }
]

Student s{30, 60, 90}; // Works same a previous example
Student newKind{}; // 0-argument ctor invoked
Student newkid; // 0-argument ctor invoked
Student toby{30, 60}; // "final" is left as 0 by the ctor
```

If we do not have a default contructor for a class, that can cause issues with any classes that uses that class inside it. Remember that this is only an issue if we provide our own custom constructor since the compiler provides a defalt constructor by default.

```
struct Basis {
    vec v1, v2;
}

Basis b; // This will cause an error, since it calls the default constructor
for Basis, which calls the (inexistant) default constructor for vec.
```

Providing a default constructor might seem to work but it actually does not solve the issue

```
struct Basis {
    vec v1, v2; // v1 and v2 are already first constructed here

    Basis() {
        v1 = vec{0, 1};
        v2 = vec{1, 0};
    }
}
```

This does not work, since `v1` and `v2` are first constructed before the constructor for `Basis` is even called, and thus the same error occurs. This is because of the way C++ creates objects.

NOTE: Whenever we have a usable object in C++, it has been constructed at some point.

Steps for object creation:

1. Enough space is allocted
2. ***Later*** - beyond the scope of current content
3. Fields are initialized - calling default constructor
4. Constructor body runs

We can fix this issue with a Member Initialization List (MIL). The MIL provides default values to initialize the fields with in step 3 instead of using the default constructor.

```
struct Basis {
    vec v1, v2;
    Basis(): v1{0, 1}, v2{1, 0} {
        // Constructor body
    }
}
```

The code `v1{0, 1}, v2{1, 0}` is the MIL, and is initialized in Step 3.

NOTE: The MIL is provided **ONLY** in the interface file.

For something like the `Basis` class, we generally want 2 different constructors:

1. Default Constructor
2. Custom Constructor (user provides all fields)

```
// Option 1: Providing default values in the default constructor
struct Basis {
    vec v1, v2;
    Basis(): v1{0, 1}, v2{1, 0} {}
    Basis(const vec& v1, const vec& v2): v1{v1}, v2{v2} {}
};

// Option 2: Providing default values in declaration
struct Basis {
    vec v1{1, 0}, v2{0, 1};
    Basis() {}
    Basis(const vec& v1, const vec& v2): v1{v1}, v2{v2} {}
}
```

If there is no MIL provided, then the default values provided during declaration are used. Fields are initialized based on the order they are declared in inside the class. The order of the fields in the MIL does not matter.

Using the MIL is oftem more efficient than setting values in the constructor body, since for the latter, each object needs to be default constructed first before it's overwritten is reset to the provided value. This causes an extra memory write cycle. Because of this, it is recommended to always use an MIL wherever possible.

An MIL **must** be used for any fields that satisfy any of these conditions:

1. Object fields that are not default constructable.
2. Const fields and reference fields.

It is possible to use an MIL for some fields and to leave the rest default constructable.

Example: Copying objects

```
struct Student {
    int assns, mt, final;
};
```

```
Student s{60, 70, 80};
Student r{s}; // We are creating a Student object by giving another Student
object
Student t = s; // Same here too
```

The 2 examples shown above invoke the copy constructor, which creates one object from another of the same type. By default, the compiler-provided copy constructor copies each field from the original object into the object being constructed.

The compiler provides the following for all classes:

1. Default constructor (goes away when ANY other constructor is written)
2. Destructor (frees up memory when the object is deleted)
3. Copy constructor
4. Copy assignment constructor
5. Move constructor
6. Move assignment constructor

Example: Copy constructor for `Student` class

```
struct Student {
    int assns, mt, final;
    Student(const Student& other): assns{other.assns}, mt{other.mt},
final{other.final} {}
};
```

It is important to note that adding ANY constructor (including a copy or more constructor) will remove the compiler-provided default constructor.

Example: Situation where we can not use the compiler provided copy constructor

```
// Linked list node
struct Node {
    int data;
    Node *next;
};

Node *n = new Node{1, new Node{2, new Node{3, nullptr}}};
Node p = *n;
Node *q = new Node{*n};
// Note: Node *q = n will simply copy over n's address
```

In this case, the copy constructor will make a copy of the first node (storing data 1), but will simply set `next` to point to the same location that `n` used to point to. This means that, despite `n` and `p` and `q` being different objects, after the first node, all 3 point to the same node, which causes data to be shared among the lists. Updating any node that isn't the first one, and addign or deleting nodes will affect all 3 linked lists at the same time

This is an example of a shallow copy, where only the first layer is copied over. Instead, we want a deep copy, where we will end up with 3 identical but still independent lists.

Aside: Ternary operator

```
// Original, naive approach
cin >> x;
string a;
if (x == 0) a = "Zero";
else a = "Nonzero";

// This can be drastically simplified with the ternary comparion operator
cin >> x;
string a = x == 0 ? "Zero" ? "Nonzero"

// Basic syntax of ternary operator
<variable> = <condition> ? <true_value> : <false_value>

// Note that even C has this operator
```

Example: Deep copy operator for linked list node

```
struct Node {
    int data;
    Node *next;
    Node(const Node& other): data{other.data}, next{other.next ? new
Node(*other.next) : nullptr} {}
};
```

The above example will perform a recursive deep copy, and when called on a linked list, will produce another that is identical but completely separate.

The copy constructor (copy ctor) is called in the following situations:

- Constructing one object from another of the same type
- Pass-by-value
- Return-by-value

There are subtleties to be discussed regarding moving.

Explicit / Implicit Constructors

Once again, let's consider the `Node` class with a copy constructor:

```
struct Node {
    int data;
    Node *next;
    Node (int data, Node *next=nullptr): data{data}, next{next} {}
    Node (const Node& other): data{other.data}, next{other.next ? new
Node{*other.next} : nullptr} {}
};
```

We have to use a `const` reference because simply calling the function with a `Node` causes the copy constructor to run, which itself results in the same copy constructor running, and so on, resulting in infinite recursion.

Single arg constructors allow implicit conversions. For example with the `Node` class shown above, we can construct it by just providing a single integer. Thus, the following behaviour makes sense:

```
Node n{4};
Node m = 4;
```

However, implicit conversions also allow for some unintuitive behaviour:

```
void f(Node n) {
    // Do something with n
}

f(Node{4}); // This makes sense
f(4); // This does not make sense, but still works
```

In the example shown above, the constructor for `Node` is called, which takes in an `int`. Once a `Node` is constructed, it is passed to `f`.

We have experienced this with C++ string initializations:

```
string s = "Hello";
// A variable of type std::string is being initialized with a const char*
value.
```

This is allowed because there is a single argument constructor for `std::string` that tales in a `const char*`.

Implicit conversions can be dangerous:

- Conversions like `int -> Node` can be unintuitive can cause confusion.
- Conversion is silent with no compiler warning.
- There is a high potential for errors.

You can disable implicit conversion by prefixing a constructor with the `explicit` keyword:

```
struct Node {
    int data;
    Node *next;
    explicit Node (int data, Node *next=nullptr): data{data}, next{next} {}
// Explicit constructor
};

Node n{4}; // Works
Node m = 4; // Won't compile

void f(Node n) {
    // Do something with n
}

f(Node{4}); // Works
f(4); // Won't compile
```

Destructors (dtors)

- Called when a stack-allocated object is popped off the stack, or when `delete` is called on heap allocated memory.
- Special method that runs when an object is destroyed.

The default compiler-provided destructor calls the destructor on all constituent objects, and does nothing to non-objects.

Steps of C++ object destruction:

1. Destructor body runs
2. Fields are de-initialized (destructor runs for all object fields) [In reverse declaration order]
3. ***Later*** - beyond the scope of current content
4. Space is de-allocated

Reasons for writing custom destructors:

Similar to with constructors, custom classes for data structures, especially those that reference themselves will need a custom constructor. Let's see why using the `Node` class as an example:

```
struct Node {
    int data;
    Node *next;
};
```

Here, `next` is a `Node*` or `Node` pointer. Thus it is not an object and no destructor will run for it. When we call `delete` on a node in a linked list, only that node will be deleted, and all the subsequent nodes will remain in memory but inacccessible, causing a memory leak.

Example: `Node` class with destructor.

```
struct Node {
    ... // Fields and constructors
    ~Node() { delete next; } // Recursively calls destructor
    // Name for dtor is ~<name>, and has no args or return type
};
```

NOTE: Running `delete nullptr;` is safe and does nothing.

Copy Assignment Operator

Let's consider an example of where we might call this operator using the `Student` class:

```
Student s{60, 70, 80};
Student r{100, 100, 100};

Student t = s; // Copy ctor is called
t = r; // Copy assignment operator is called
```

The copy assignment operator runs when we assign a value (that already exists) to another of the same type. The compiler-provided version assigns each field of the object. Similar to constructors and destructors, this would not work for many custom classes, especially those that have recursive pointers and/or are needed for data storage.

Example: Copy Assignment Operator for `Node` class (V1).

```
struct Node {
    ... // Fields, constructors, and destructors
    Node& operator=(const Node& other) { // The Node& return type is for
chaining (a = b = c)
        delete next; // We need to delete current objects to avoid memory
leaks
```

```
        data = other.data;
        next = other.next ? new Node(*other.next) : nullptr;
        return *this; // For chaining
    }
};
```

The above code works in almost all cases, but will cause a runtime error when you assign a node to itself (`n = n`). This is also known as self assignment, and we should not do anything if this happens. So, let's cosider an updated version.

Example: Copy Assignment Operator for `Node` class (V2).

```
struct Node {
    ... // Fields, constructors, and destructors
    Node& operator=(const Node& other) { // The Node& return type is for
chaining (a = b = c)
        if (this == &other) return *this; // Exit if self assignment occurs
        data = other.data;
        delete next; // We need to delete current objects to avoid memory
leaks
        next = other.next ? new Node(*other.next) : nullptr;
        return *this; // For chaining
    }
}
```

This version is almost perfect, but we can make one more improvement. Currently, we are requesting memory after we destroy old data. In case memory can't be allocated, this causes `next` to be deleted. We don't want anything to happen if memory can't be allocated. Let's consider the final version.

Example: Copy Assignment Operator for `Node` class (V3 / Final).

```
struct Node {
    ... // Fields, constructors, and destructors
    Node& operator=(const Node& other) { // The Node& return type is for
chaining (a = b = c)
        if (this == &other) return *this; // Exit if self assignment occurs
        Node *tmp = other.next ? new Node(*other.next) : nullptr;
        // If the above line fails, the function exits without any changes to
the data
        delete next; // We need to delete current objects to avoid memory
leaks
        data = other.data;
        next = tmp;
        return *this; // For chaining
    }
};
```

Copy/Swap Idiom

This is another method to implement copy assignment.

There is a function called `std::swap` in the `<utility>` library. Running `swap(a, b);` will swap the values of `a` and `b`.

Exmaple: Copy assignment operator for the `Node` class using the copy/swap idiom.

```
struct Node {
    ... // Fields, constructors, and destructors
    void swap(Node& other) {
        std::swap(data, other.data);
        std::swap(next, other.next);
        // std::swap is used here insted of swap to avoid ambiguity
    }

    Node& operator=(const Node& other) {
        Node tmp{other};
        swap(tmp);
        return *this;
    } // Dtor and copy ctor need to be defined for this to work
}; // This method is slightly slower than the previous approach
```

Move constructor

Recall, the references that we have seen are "lvalue references". They do NOT bind to temporary values.

```
Node oddsOrEvens() {
    Node o{1, new Node{3, new Node{5, ... , new Node{99, nullptr}}}};
    Node e{2, new Node{4, new Node{6, ... , new Node{100, nullptr}}}};

    char c;
    cin >> c;
    return (c == 'o') ? o : e;
}

Node l = oddsOrEvens(); // Return by value
```

When the function exits, either `o` or `e` will be copied from `oddsOrEvens`' stack frame into `l`. Once the function goes out of scope, both `o` and `e` are deleted.

We just copied 50 Nodes and then deleted the originals. This is a waste of resources. What if we instead just reused the originals?

This is only allowed if we are certain that nobody else will use the originals.

Example: Different examples of copying / moving data.

```
Node n{...};
Node p{n}; // We must make a copy of n
// n is still defined and we might want to use it later

Node l = oddsOrEvens(); // Temporary value - nobody else can use it!
// We can reuse the data rather than copying and deleting
```

We need to determine whether the value we are working with is a long lasting value (lvalue) [and perform a deep copy], or whether we are using a temporary [and reuse the data].

`Node&&` - rvalue reverence. Can bind to temporary values. Compiler reserves a temporary memory address to put the value in.

Example: Move constructor for the `Node` class.

```
struct Node {
    ... // Fields, constructors, and destructors
    Node(Node&& other): data{other.data}, next{other.next} {
        other.next = nullptr; // Remove other's access to the remaining nodes
so the destructor does not delete the rest of the linked list when other gets
destroyed
    }
}
```

Move Assignment Operator

This is similar to the move constructor, except that it is used when reassigning the value of a variable that has already been initialized.

```
struct Node {
    Node& operator=(Node&& other) { // Move Assignment Operator
        swap(other); // Assumes the swap function has been defined for Node,
as shown above for copy/swap idiom
        return *this;
    }
};
```

If other is a temporary (rvalue), the move constructor or move assignment oerator is called instead of copying resources. If you only write the copy constructor / assignment operator, only these will be used.

Rule of Big 5 (Suggestion)

If you write one of the Big 5 (destructor, copy contructor, copy assign operator, move constructor, move assign operator): then you should probably write them all.

Do not reinvent the wheel and use compiler-provided versions if possible. The Big 5 is usually necessary only for classes that manage a resource (memory).

Elision

Let's consider this example:

```
vec getVec() { return {0, 0}; }
vec v = getVec();
```

When we run this code, you might expect the constructor and the copy / move constructor to be called, but only the basic constructor is run.

This occurs due to copy / move elision. Rather than making a vec in getVec() and moving it into main, the compiler writes the value directly to main itself.

"Compiler is Boss": The compiler can provide this feature even if it changes the program output.

You are not expected to know all the places where elision may occur, just that it is possible.

Features of Objects

Member operators

We previously wrote vec operator+ as a standalone function, but vec operator= was a method. What is the difference?

```
struct vec {
    int x, y;
    vec operator+(const vec& other) {
        return {x + other.x, y + other.y};
    }
    vec operator*(int k) { // This function works for v * 5 but not for 5 * v
        return {x * k, y * k};
    } // In order to be able to use 5 * v, we need to write that as a
standalone function
};
```

Any function that is defined as a class method automatically takes `this` as it's first argument. If any functions need to be written that don't do so, then they need to be written as standalone functions.

For this reason, I/O operators like `operator<<` and `operator>>` need to be provided as standalone functions, since they are not necessarily always run with an instance of the object on the left side of the operator.

Advice: For arithmetic assignments, reuse logic from arithmetic overloads.

```
vec& operator+=(vec& v1, const vec& v2) {
    v1.x += v2.x;
    v1.y += y2.y;
    return v1;
}

vec operator+(vec v1, const vec& v2) { // v1 is copied (pass by value)
    v1 += v2;
    return v1;
}
```

Some operator overloads must be methods:

- `operator=`
- `operator[]`
- `operator->`
- `operator()`
- `operatorT` (where T is a type)

Arrays of Objects

Let's consider the `vec` object again:

```
struct vec {
    int x, y;
    vec(int x, int y): x{x}, y{y} {}
}; // Note that there is no default constructor
```

What if we wanted to make an array of `vec` objects:

```
vec *vp = new vec[15];
vec sArraray[10];
```

You might think that the above lines of code will work correctly, but they actually cause a compilation error. This is because in C++, any object needs to be initialized as it is created. When making an array of `vec`s, they all need to be initialized. This is done by running the default constructor for each object in the array. Since there is no default constructor here, it causes a compilation error.

Solutions - to get an array of objects:

1. Add a default constructor to the object.
   This might make sense for the `vec` object, but might not work with other objects, like for example the `student` class but with "name" and "id".
2. If the array is stack allocated, we can provide values during initialization
   `vec sArray[3] = {{0, 1}, {2, 3}, {4, 5}};`
   This method works, but has limited use cases where it can be used.
3. Use an array of pointers to objects.
   `vec **vp = new vec *[15];`
   Pointers are primitve classes (not objects) and thus don't need constructors. After declaring an array of pointers, we can initialize them one-by-one.
   `vp[0] = new vec{0, 1};`
   `vp[1] = new vec{2, 3}; ...`
   However, when we are done using this array, we need to manually free all the memory, since it is allocated in the heap.
   `for (int i = 0; i < 15; ++i) delete vp[i];`
   `delete[] vp;`

Const objects

Objects can be constant:

```
const student s{60, 70, 80};
s.assns = 100; // This will cause a compilation error
```

The fields of constant objects can not be changed after they are initialized.

However, we can not run any of the object's inbuilt functions:

```
const student s{60, 70, 80};
cout << s.grade() << endl; // This will cause a compilation error
```

Issue: The compiler does not know whether `s.grade()` will modify `assns`, `mt`, or `final`. The function call is prevented because we do not know whether `s.grade()` will respect the `const`ness of `s`.

If we know that `grade` won't modify any of the fields, we must declare it to use `grade` with a `const` object.

Example: Struct functions that work with `const` obejcts.

```
// Interface
struct Student {
    int assns, mt, final;
    float grade() const; // Note the const suffix
};

// Implementation
float Student::grade() const { // Note the const suffix
    return assns * 0.4 + mt * 0.2 + final * 0.4;
} // Modifying any of the values will cause a compilation error
```

Note: The `const` suffix must appear in both the interface and implementation files, and trying to edit any variable inside a `const` function will cause a compilation error.

With the above code, we can run the `grade` function on a `const` object:

```
const student s{60, 70, 80};
cout << s.grade() << endl; // This will work now since grade is a const
method
```

`const` objects may only have `const` methods called on them. Non-`const` objects may have either `const` or non-`const` methods called on them,

What if we wanetd to collect stats on our objects?

```
struct Student {
    int assns, mt, final;
    int calls = 0;
    float grade() const {
        ++calls;
        return assns * 0.4 + mt * 0.2 + final * 0.4;
    }
};
```

Ths would not work because the function increments `calls`.

Issue: Difference between physical vs logical `const`ness:

- Physical: Did the bits of memory that make up the object change?
- Logical: Did the "essence" of the object change? Do I consider this object different after running the method?

We can make an exception to what the compiler checks be declaring a field to be `mutable`.

```
struct Student {
    int assns, mt, finals;
    mutable int calls; // Can be changed in const mathods (for both const and
non-const objects)
}
```

The `mutable` keyword needs to be used sparingly, since it decreases the usefulness of `const`.

Static fields and methods

What if I want to record the number of calls for ALL `Student` objects, and not just one. Or keep track of the number of `Student`s created?

We use `static` fields:

```
struct Student {
    int assns, mt, final;
    inline static int numInstances = 0;

    Student(int assns, int mt, int final): assns{assns}, mt{mt}, final{final}
{
        ++numInstances;
    }
};

Student s{60, 70, 80};
Student r{100, 100, 100};
cout << Student::numInstances << endl; // Will print out 2
// s.numInstances also works, but is not recommended
```

Any `static` fields are defined for the object rather than for each instance, and the value is shared across all instances of the object. The `inline` keyword is used to allow static fields to be initialized directly in the interface rather than in the implementation file.

Static methods, just like static fields, are defined for the class rather than for any one particular object. Static methods can only access static fields.

```
struct Student {
    ...
    static void howMany() {
        cout << numInstances << endl;
    }
};
```

In order to call this function we call `Student::howMany();`. `s.howMany()` won't work.

Three-Way Comparison

Let's go back over how string comparison in C works: `strcmp(s1, s2)`:

- `< 0` if `s1 < s2` (lexicographically)
- `== 0` if `s1 == s2`
- `> 0` if `s1 > s2`

In order to compare 2 strings, we would do the following:

```
// In C
int n = strcmp(s1, s2);
if (n < 0) { ... }
else if (n == 0) { ... }
else (n > 0) { ... }
```

```
// In C++
if (s1 < s2) { ... }
else if (s1 == s2) { ... }
else { ... }
```

Notice that in C, there is only one string comparison being done (by running `strcmp`). On the other hand, in C++, there are 2 different string conparisons being done. This causes an unnecessary waste of resources since the exact samme comparison is being done twice, even though it only needs to be done once.

As of C++20, there is a more efficient way:

```
// 3-way comparison operator: (Spaceship operator)
s1 <=> s2;
```

Using the 3-way comparison operator is identical to using `strcmp` in C. Note that in order to use it we need to import the `<compare>` library.

```
std::strong_ordering n = (s1 <=> s2);
// std::strong_ordering is the return value of <=>
if (n < 0) { ... }
```

```
else if (n == 0) { ... }
else (n > 0) { ... }
```

`std::strong_ordering` is a lot to type. Instead, we can use automatic type deduction:

```
auto n = (s1 <=> s2);
// n's type is the return type of the expression on the RHS
```

We may also overload the spaceship operator for our own data types. If we define `<=>` for a type, we also automatically get all the other comparison operators automatically:

- `v1 == v2`
- `v1 != v2`
- `v1 <= v2`
- `v1 >= v2`
- `v1 < v2`
- `v1 > v2`

Even after we define the spaceship operator, we can overload `operator==` again, since it is sometimes possible to check equality much more efficiently than comparion.

Example: Operloading the spaceship operator.

```
struct vec {
    int x, y;
    auto operator<=>(const vec& other) const {
        auto n = x <=> other.x;
        return (n == 0) ? (y <=> other.y) : n;
    }
};
```

Here, we are simply comparing the fields in declaration order.. If that is the case, we can use the default version of `operator<=>`. This is not provided by default and we need to specify that we are using default behaviour.

```
struct vec {
    int x, y;
    auto opertor<=>(const vec& other) const = default;
}
```

Note: `= default` also works for all constructors, destructors, and operattors that the compiler provides by default.

Consider a case where `= default` is nott appropriate for `<=>`. Ex. Linked lists.

```
struct Node {
    auto operator<=>(const Node &other) const {
        auto n = (data <=> other.data);
        if ((n != 0) || (!next && !other.next)) return n;
        if (!next && other.next) return std::strong_ordering::less;
        if (next && !other.next) return std::strong_ordering::greater;
        return *next <=> *other.next;
    }
};
```

Encapsulation

Let's consider our linked list example again:

```
struct Node {
    Node *next;
    int data;
    // Big 5
};
```

We can cause errors by incorrectly setting values for `Node` objects.

```
Node n1{1, new Node{2, nullptr}};
Node n2{3, nullptr};
Node n3{4, &n2};
```

When this program finished running, the following happens:

- The destructor runs on `n1`, and frees the `Node` that is on the heap.
- The destructor runs on `n3`, but fails because it tries to free `n2`, which is stack allocated.

Even if we did these with variables that were all heap-allocated, we would end up with double-free errors.

We did not account for this happening when we wrote the big 5, because we always assumed:

1. `next` is always `nullptr`, or pointing to heap-allocated memory.
2. No sharing of data between lists.

These are **invariants** - properties of a data structure that must always be true.

However, these invariants can easily be violated by the user.

Encapsulation provides a solution to users violating invariants.
Users should treat our objects like "capsules" or "black boxes", where they have no access to underlying data, but rather interact with the objects by calling methods.

This is where access specifiers are useful. There are 2 different types:

- `public` fields / methods: can be accessed anywhere.
- `private` fields / methods: can only be accessed / called within class methods.

Example: `vec` struct using access specifiers

```
struct vec {
    private:
        inr x, y;
    public:
        vec(int x, int y);
        vec operator+(const vec& other) const;
};
```

It is preferred to keep fields `private` by default.

C++ has a `class` keyword, where the default visibility of fields / methods is `private`, unlike `struct`s, where it is `public`.
Note that this is the only difference between `class` and `struct`.

Example: The `vec` object being implemented as a `class`,.

```
class vec {
    int x, y; // Private by default, can not be called outside vec methods
    public:
        vec(int x, int y);
        vec operator+(const vec& other) const;
}; // Semicolon at the end, like with structs
```

Example: Revisit linked lists - add encapsulation, protect invariants

```
// List.cc
export module list;
export class List {
    struct Node; // Private nested class
    Node *head = nullptr;
    public:
        ~List();
        void addToFront(int n);
        int ith(int i);
};

// List-impl.cc
struct List::Node {
    int data;
    Node *next;
    ~Node { delete next; }
    // ... Rest of big 5, if needed
};

List::~List() { delete head; }

void List::addToFront(int n) {
    head = new Node{n, head};
}

int List::ith(int i) {
    Node *cur = head;
    for (int j = 0; j < i; ++j) cur = cur->next;
    return cur->data;
}
```

Issue: Looping through a list takes $O(n^2)$ time.

How can we have fast iteration while maintaining encapsulation?

Solution: Iterator pattern

Iterator pattern is a design pattern - effective solution to a common problem.

We solve the issue by creating an interator class, which is an abstraction of a pointer. We keep track of how far we have reached, and allow the user to access data but not modify anything.

Example: `List` data structure with Iterator pattern.

```
class List {
    struct Node;
    Node *head = nullptr;

    public:
        class Iterator {
            Node *cur;
            public:
                Iterator(Node *cur): cur{cur} {}

                Iterator& operator++() {
                    cur = cur->next;
                    return *this;
                }

                bool operator!=(const Iterator& other) const {
                    return cur != other.cur;
                }

                int& operator*() const {
                    return cur->data;
                }
        }; // End iterator

        // List class
        Iterator begin() const {
            return Iterator{head};
        }

        Iterator end() const {
            return Iterator{nullptr};
        }

        // Also add addToFront, ith, and Big 5 [like in previous example]
}; // End List class
```

```
// Example code for iterator class
int main() {
    List l = addToFront(1);
    for (List::Iterator it = l.begin(); it != l.end(); ++it) {
        cout << *it << endl;
    } // This loop runs in O(n) time.
}
```

If you have a class with the following:

1. `begin` and `end` methods returning some iterator type.
2. This iterator type has `++`, `!=`, and `*`.

- You can use a range-based `for` loop.

Exmaple: Range-based `for` loop with iterator class

```
// If you want to get a copy of the data stored in the class
for (int n : l) {
    cout << n << endl; // n here is a copy
}

// If you want to use the original data
for (int& n : l) {
    ++n; // This will presist
}
```

End of Midterm Content

This method of Encapsulation still has a small issue: `auto it - List::Iterator{nullptr};`

- This violates the idea that all ierators are created by calling `begin()` or `end()`. While `List::Iterator{nullptr}` is the same thing that is returned by `end()` in this case, it is not necessarily the case with other data structures.

Consider makinf `List::Iterator`'s contructor private.

- Being a `private` method, it can not be called by the user.
- However, even `List` would not be able to make iterators in `begin()` or `end()`.

Solution: Make an exception for the `List` class so that it gets previleged access to the normally `private` constructor for `Iterator`.

"We use the power of friendship"

Example: `Iterator` with friend classes.

```
class List {
    struct Node;
    Node *head = nullptr;
    public:
        class Iterator {
            Node *cur;
            Iterator(Node *cur): cur{cur} {}
            public:
                // Operator overloads
                friend class List; // This can be anywhere in Iterator
        }; // End iterator
        Iterator begin() { return Iterator{head}; }
        Iterator end() { return Iterator{nullptr}; }
}; // End List
```

If class `A` declares class `B` as a friend, then `B` can access all the private fields / methods of `A`.

Now, a user can now create an `Iterator`, but `List` still can. Thus, we can be sure that all `Iterator`s can be created via `begin()` or `end()`.

Advice: *Limit your friendships* - More friendships means that it is more difficult to reason about private fields / methods.

Consider using accessor / mutator methods instead (getters / setters).

Example: Accessors and mutators for the `vec` class.

```
class Vec {
    int x, y;
    public:
        int getX() const { return x; }
        int setX(int a) { x = a; }
        int getY() const { return y; }
        int setY(int b) { y = b; }
}
```

We can also declare friend methods.

How can we create a private variable that can still be printed out by an ostream?

- Must be a standalone function.
  We can't declare a friend class here.
- We could use getters in some cases.
  But sometimes, we might want a variable that is normally private.

We can declare friend functions just like how we declare friend classes. Instead of providing a class name, we provide a function signature.

Example: Declaring output operator as a friend function.

```
ostream& operator<<(ostream& out, const vec& v) {
    return out << "(" << v.x << ", " << v.y << ")";
}

class vec {
    int x, y;
    public:
        // This standalone function has access to x and y, even though they
are private fields
        friend ostream& operator<<(ostream&, const vec&);
        // ... (Rest of class definition)
}
```

Equality Revisited

We now have an encapsulated `List` class.
We already have `ith` and `addToFront` methods. We could also add a length method?
There are 2 ways we could go about doing this:

- Loop through, count the number of `Node`s, and return the length once we have reached the end: $O(n)$ time.
- Keep a length field andupdate it whenever `addToFront` is called. Return the value stored in this field when the length method is called: $O(1)$ time.

The second option is generally preferred, as it allows us to optimize equality.
Previously, to check equality, `l1 == l2` translated to `(l1 <=> l2) == 0`

Most lists will have different lengths, and so we could optimize the `==` operator by checking the length first, and only compare the individual values if the lengths differ. If the lengths differ, then the equality is calculated in $O(1)$ time.

Example: Optimized equality operator for a `List` class.

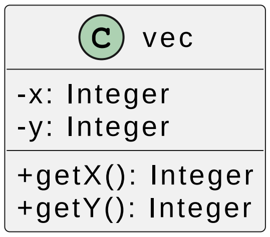
```cpp
class List {
    struct Node;
    Node* head;
    int length;
    public:
        bool operator==(const List& other) const {
            if (length != other.length) return false; // Compare lengths
            return (*this <=> other) == 0; // Compare values with <=>
        }

        bool operator<=>(const List &other) const {
            // Check if any List is empty
            if(!head && !other.head) return std::strong_ordering::equal;
            if (!head) return std::strong_ordering::less;
            if (!other.head) return std::strong_ordering::greater;
            return *head <=> *other.head; // Compare Nodes with <=>
        }
}
```

Spaceship operator provides `>=`, `<=`, `>`, `<` for `List`. We have overloaded the `==` operator, for which the compiler will use the custom optimized version, as well as for `!=`, which is just the nagation of `==`.

System Modelling

We want to graphically display the classes of a program at a high level.
We will use the popular standard language **UML (Unified Modelling Language)**.

Example: UML diagram for the `vec` class.



Here, there are 2 different sections. The upper section is for variables and the bottom section is for functions. Anything prepended by a `+` is `public`, anything prepended by `-` is `private`.

Relationships Between Classes

There are many different ways that different classes can interact with each other.

Composition

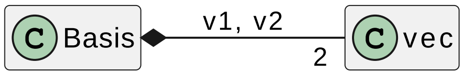
Composition is one possible relationship between classes, where one classes is embedded within the other.

Example: Composition relationship.

```
class Basis {
    vec v1, v2;
};
```

Here, `Basis` is **composed** of 2 `vec`s. This is also called "owns-a". Here, `Basis` owns 2 `vec`s.

The following is the UML diagram for the `Basis` example:



Properties of Composition: If "`A` owns-a `B`", then:

- If `A` dies, then `B` also dies.
- If `A` is copied, then `B` is copied (deep copy).
- `B` has no independant existence outside of `A`.

Aggregation

Aggregation is similar to Composition, except with some minor differences. Instead of one class being embedded in another, it is rather linked to the other. An aggregation relationship is also called "has-a".

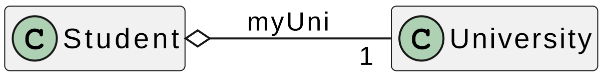
Properties of Aggregation: If "A has-a B", then:

- If A dies, then B keeps living.
- If A is copied, then B is not copied (shallow copy).
- B may have independant existence outside of A.

Example: Aggregation relationship.

```
class Student {
    int id;
    University* myUni{};
    public:
        Student(...) {}
        // Anything else
};

int main() {
    University uw { ... };
    Student s1 {1, &uw};
    Student s2 {2, &uw};
}
```

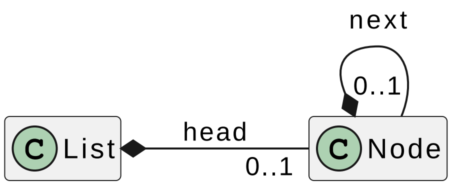
The following is the UML diagram for the `Student` example:



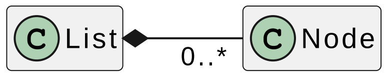
Typically, Composition uses object fields, while aggregation uses References or Pointers.

However, this is not always the case. Consider linked lists, implemented via pointers, but it is a composition relationship.

UML diagram for a linked list:



\



The first diagram suggests a recursive implementation, while the second diagram suggests an iterative implementation.

Specialization / Inheritence

Imagine a program to manage and catalogue library books. We will manage Books, Texts, and Comics.

Example: Class implementations for the library catalogue program.

```
class Book {
    string title, author;
    int length;
    public:
        Book(...) { ... }
};

class Text {
    string title, author;
    int length;
    string topic;
    public:
        Text(...) { ... }
};

class Comic {
    string title, author;
    int length;
    string hero;
    public:
        Comic(...) { ... }
};
```

Now, what if we want an array of all the books of the different types:

- Should store `Text`s, `Book`s, and `Comic`s.

1. Use `void` pointers - can point at anything.
    - Not type safe
2. Use Union type:

```
Union BookType {
    Book* b;
    Text* t;
    Comic* c;
}; // This stores ONE of the data types
```

This is also not super safe and does not have any safeguards to prevent against anything that can result in undefined behaviour.

```
BookType u;
u.b = new Book{ ... };
cout << u.t << endl; // Undefined behaviour
```

The issue is that the compiler is not aware of the relationship between these classes:

- `Comic` and `Text` are both types of `Book`.
- A `Comic` "is-a" `Book`, and a `Text` "is-a" `Book`.

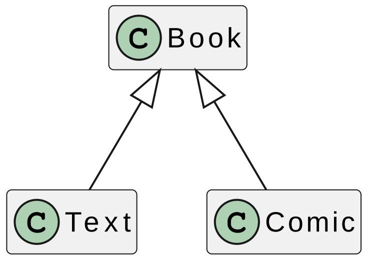
In this case, we would want to use a Specialization relationship, or "is-a" relationship. It is implemented in C++ using Inheritence.

```
class Book { // Basic class or Superclass
    string title, author;
    int length;
    public:
        Book(string title, string author, int length):
            title{title}, author{author}, length{length} {}
};

// Derived subclasses
// All fields / methods are inherited from Book
class Text: public Book {
    string topic;
    public:
        Text(...) { ... }
};

class Comic: public Book {
    string hero;
    public:
        Comic(...) { ... }
};
```

`Text` and `Comic` inherit from `Book`. This means that they have `title`, `author`, and `length` fields. A subclass inherits all fields and methods from it's superclass.

- `title`, `author`, and `length` are `private` in `Book`, and can only be acessed by `Book`, and not it's subclasses.

Specialization or "is-a" in UML:



The above syntax indicates Specialization, or public inheritence in C++.

Example: Incorrect constructor for a subclass.

```
// WARNING: This code does not work and the reason why is explained below.
class Text: public Book {
    string topic;
    public:
        Text(string title, string author, int length, string topic):
            title{title}, author{author}, length{length}, topic{topic} {}
};
```

This code does now work, since `title`, `author`, and `length` are private to `Book`, and can not be changed by `Text`. Furthermore, the `MIL` is only for the class in which it is defined, and thus, this method will not work even if we set all the fields in `Book` to be `public`.

Let's go over the **Object Creation Sequence**, this time with all the steps:

1. Space is allocated.
2. *Superclass constructor runs.*
3. Fields are initialized via MIL.
4. Constructor body runs.

In step 2, the superclass constructor is called. If there are no arguments specified for the superclass constructor, then the default constructor will be used. If the superclass does not have a default constructor, then the code won't compile.

Example: Constructor for a subclass.

```
class Text: public Book {
    string topic;
    public:
        Book(string title, string author, int length, string topic):
            Book{title, author, length}, // Step 2
            topic{topic} // Step 3
            {} // Step 4
};
```

Protected Variables and Methods

Generally, it is a good idea to keep superclass firlds private to subclasses. However, if we do want to give subclasses access but not any other class or function, then we can use the `protected` keyword. It functions similar to `private`, except that it grants access to both the class and any of it's subclasses.

`protected` variables are prepended by a `#` in UML.

Example: Different abilities for different subclasses.

```
// Consider adding multiple authors to Texts
class Book {
    protected:
        string title, author;
        int length;
    public:
        Book (...) { ... }
};

class Text: public Book {
    string topic;
    public:
        Text(...) { ... }
        void addAuthor(string a) { author += a; }
        // The above function can access and modify author since it is a
protected field
};
```

Example: Protected mutators.

```
// This approach is recommended when working with inheritence
class Book {
    string title, author;
    int length;
    public:
        Book(...) { ... }
    protected:
        void setAuthor(string a) { author += a; }
        // The above function may only be called inside Book or any of it's
    subclasses
};
```

Virtual Methods

Let's consider the library program again. Suppose we want a `Book` case, which is an array of `Book`s (of all the different specializations) that my behave differently depending on which inerited class it is (`Book`, `Text`, or `Comic`).

For example, let's create an `isHeavy()` method, which checks whether the `Book` is heavy or not. The classification for whether each type of book is "heavy" is as follows:

- `Book`: heavy if > 200 pages
- `Text`: heavy if > 500 pages
- `Comic`: heavy if > 30 pages

Example: Simple implementation of `isHeavy()` method for `Book` and `Text` classes.

```
class Book {
    protected:
        string title, author;
        int length;
    public:
        Book(...) { ... }
        bool isHeavy() const {
            return length > 200;
        }
};

class Text: public Book {
    string topic;
    public:
        Text(...) { ... }
        bool isHeavy() const {
            return length > 500;
        }
};

Book b{"...", "...", 300};
Text t{"...", "...", 300, "topic"};
b.isHeavy(); // True
t.isHeavy(); // False
```

This makes sense. While both `b` and `t` have `300` pages, `b` is a `Book` and is thus considered heavy while `t` is a `Text` and is thus not heavy.

However, what if we were to save a `Text` into a `Book`?

```
Book b = Text{"...", "...", 300, "topic"};
// This works because Text inherits from Book. This won't work the other way.

b.isHeavy(); // Returns True
// This would use the Book definition
```

After assigning `b` to a `Book`, we lose all the information that `b` was a `Text`. `b` is from that point treated as a `Book`. This is because it is running the compiler provided move constructor, which only moves `title`, `author`, `length`. The same thing also happens with copy constructors.

Object slicing: Constructor runs for a superclass and chops off the subclass fields.

What if we use pointers instead?

```
Text t{"...", "...", 300, "topic"};
Book *bp = &t;
bp->isHeavy(); // Also returns True
```

To get `isHeavy()` to use the `Text` definition even if it is pointed to by a `Book` pointer, we must use a **virtual method**.

Every object in C++ actually has 2 different types:

- **Static Type:** Given by the type in the declaration.
  `Book *pb = &t;` < In this case, the **static** type is `Book`.
- **Dynamic Type:** The type of the "underlying" object.
  `Book *pb = &t;` < In this case, the **dynamic** type is that of `t`.

How to determine which type is being used:

1. If not using a pointer or reference, any method is always called based on the static type.
2. If using a pointer or a reference:
   - Non-virtual method: Use **static type**.
   - Virtual method: Use **dynamic type**.

A **virtual method** will be called based on the **dynamic type** of the object that it is being called on.

Example: Virtual method

```
class Book { // Superclass
    ... // Fields
    public:
        virtual bool isHeavy() const { return length > 200; }
        ... // Other methods, if required
};

class text: public Book { // Subclass
    ... // Fields
    public:
        virtual bool isHeavy() const override { return length > 500; }
        ... // Other methods, if required
};
```

Assuming the above **virtual method** definition, the following code will run as explained in the comments.

```
Text t{"...", "...", 300, "topic"};
Book b{"...", "...", 300};
Text *pt = &t;
Book *bp = &t;

t.isHeavy(); // False
b.isHeavy(); // True
pt->isHeavy(); // False
pb->isHeavy(); // False
```

This is what we want to happen. Now, let's revisit the Bookcase example with the same virtual method definitions.

```
Book *myBooks[2];
myBooks[0] = new Text{...};
myBooks[1] = new Comic{...};

for (int i = 0; i < 2; ++i) {
    cout << myBooks[i]->isHeavy() << endl;
} // This code will use the dynamic type for each book
```

Note: The `override` keyword does not actually do anything during the execution of the program. However, it is useful to add because it checks during compile time in order to ensure that the method is actually `virtual` for the superclass and provides a helpful error if it isn't.

Virtual methods are an example of **polymorphism**. The word "polymorphism" is Greek for "many forms".

- Here, a superclass is being able to represent many different subclasses (forms).
- We've actually seen this before. For example, `ifstream` is an `istream`.

Destructors Revisited

Let's consider how a destructor would work for a superclass and it's subclass(es).

Example: INCORRECT destructor for superclass and subclass.

```
// Class definitions
class X {
    int *n; // Array of int's in the heap
    public:
        X(int size): n{new int[size]} {}
        ~X() { delete n; }
};

class Y: public X {
    int *m; // Another array of int's in the heap
    public:
        Y(int size1, int size2): Y{size1}, m{new int[size2]} {}
        ~Y() { delete[] m; }
};

// Example code
X *xp = new X{5};
Y *yp = new Y{5, 10};
X *xptoy = new Y{5, 10};

delete xp; // Works
delete yp; // Works
delete xptoy; // Leaks memory!
```

Let's go over the full object destruction sequence now, in order to understand why this is happening.

Object Destruction Sequence:

1. Destructor body runs.
2. Object fields have their destructors called in reverse declaration order.
3. *Superclass destructor runs.*
4. Space is reclaimed

Why does calling the destructor on `xptoy` leak memory?

- `xptoy` is a `Y` pointer to an `X` type.
- Since the destructor is not a `virtual` function, `X`'s destructor is called instead of `Y`'s, which causes a memory leak.

Solution: Make the destructor for `X` (superclass) `virtual`.

```
class X {
    ... // Fields
    public:
        virtual ~X() { delete[] n; }
};
```

If you know that a `class` may be subclassed, you must **ALWAYS** make it's destructor virtual. If you know that a `class` will **NEVER** be subclassed, you should make the compiler enforce it via `final`.

```
// This class can not be subclassed
// If done, the program will not compile
class C final {
    ... // Fields and methods
};
```

Pure Virtual Methods

Let's consider a program to manage the different students in a university.

```
class Student {
    public:
        virtual int fees() const;
};

class Regular: public Student {
    public:
        int fees() const override;
};

class Coop: public Student {
    public:
        int fees() const override;
};
```

This program has `class`es for `Regular` and `Coop` `Student`s, with each of them having different fees that they need to pay. We have declared a function `Student::fees()`, but we do not ever want this to be run, since we never intend on creating `Student` objects that aren't `Regular` or `Coop`. It is possible to enforce this using **pure virtual methods**.

Example: Pure virtual method

```
class Student {
    public:
        virtual int fees() const = 0;
        // Setting a function to 0 makes it pure virtual
};
```

Pure virtual methods do not need to have an implementation, although one can be created if needed. Any class that defines one or more pure virtual methods can not be instantiated. For example, running `Student s;` after the above example code will cause a compilation error.

While the compiler might not require an implementation for pure virtual methods, we might have to provide one in some cases. For example, if we make a pure virtual destructor, we need to implement it.

`Student` is now an **abstract class**. An abstract class can not be instantiated. Any subclasses of an abstract class are also abstract, unless they provide an implementation for all the inherited pure virtual methods. Such sublasses are referred to as **concrete**.

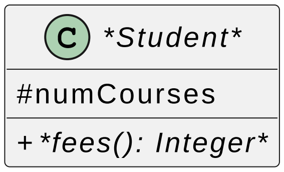
If we want to make a class abstract, but there are no methods for a class that could reasonably be made pure virtual, the best pratice is to make the destructor pure virtual. Note that we would still need to provide an implementation for it.

Abstract classes are useful for organizing subclasses.

Example: Abstract class and inherited concrete class

```
class Student { // Abstract
    protected:
        int numCourses;
    public:
        virtual int fees() const = 0;
};

class Regular { // Concrete
    public:
        int fees() const override {
            return 1600 * numCourses;
        }
}
```

In UML, we represent abstract classes and pure virtual methods with italic text. If we are drawing out a diagram, we can surround anything with asterisks (*) instead of italicizing it.



Revisiting Big 5 with Inheritence

Let's once again consider the `Book` and `Text` classes.

```
class Book {
    string title, author;
    int length;
    public:
        ... // Big 5
};

class Text: public Book {
    string topic;
    public:
        // Nothing here
        // Use compiler-provided Big 5
};

Text t1{...}, t2{...};
t1 = t2;
```

In the above example, the compiler-provided copy constructor works perfectly, and all of `t1`'s fields get set to the values of `t2`'s fields.

Let's see what the compiler-provided Big 5 look like for inherited classes.

```
// Copy constructor
Text::Text(const Text& other): Book{other}, topic{other.topic} {}

// Move constructor
Text::Text(const Text&& other):
    Book{std::move(other)}, topic{std::move(other.topic)} {}
```

The copy constructor makes sense here, but the move constructor has this new `std::move` syntax. This is because if we were to simply call `Book{other}`, it would copy all the individual fields rather than moving them. This is because `other` is considered an **Lvalue** here (even though it is actually an **Rvalue** reference) because it stays alive for the entire length of the function. Due to this, **Lvalue** copy semantics are invoked, and all the fields are copied rather than moved.

We can use `std::move` to force something to be treated as an **Rvalue**, and thus invoke move semantics. We also need to use `std::move` for non-inherited classes when initializing `string`s and objects.

Let's see how the different compiler provided assignment operators behave.

```
// Copy Assignment
Text& Text::operator=(const Text& other) {
    Book::operator=(other); // Call Book's operator
    topic = other.topic;
    return *this;
}

// Move Assignment
Text& Text::operator=(Text&& other) {
    Book::operator=(std::move(other));
}
```

However, consider this example.

```
Text t1{"Algorithms", "CLRS", 1000, "CS"};
Text t2{"Shakespeare", "Mr. English", 200, "English"};
Book& r1 = t1; Book& r2 = t2;
r1 = r2;
// operator= is non-virtual, so it uses the static type
```

This is not good, since `t1` now contains:

```
{"Shakespeare", "Mr. English", 200, "CS"}
```

All the `Book` fields are copied over, but all the `Text` fields remain the same. This is called "**partial assignment**". We can solve this by making `Book::operator=` virtual.

Example: `virtual` copy assignment operator

```
class Book {
    string title, author;
    int length;
    public:
        virtual Book& operator=(const Book& other);
        ... // Rest of Big 5
};

class Text: public Book {
    string topic;
    public:
        Text& operator=(const Book& other);
        ... // Rest of Big 5
}
```

Notice the special syntax here. The overloaded function returns a `Text` (which is allowed, since it is a subclass of the return type of the virtual function), but takes in a `Book`. This is because the arguments need to match exactly with the arguments in the virtual function. This is similar to how the appropriate override for a function is chosen by the compiler, where just the arguments are compared, and the return type is meaningless.

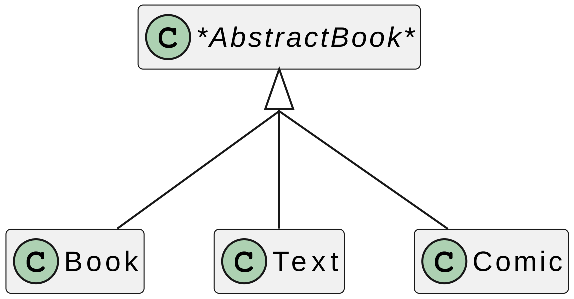
This works as we want it to, however, it allows the following:

```
Text t1{...};
t1 = Book{...};
t1 = Comic{...};
```

This is **mixed assignment**, and the operator shown in the previous example allows this to compile. Thus, we are faced with a dilemma:

- If `operator=` is non-virtual: *Partial assignment*
- If `operator=` is virtual: *Mixed assignment*

The only realistic solution to this problem (for now) is to restructure the class hierarchy. It is advised to always **make superclasses abstract**. The UML diagram of this updated class structure is as follows:



Example: Book example with abstract superclass.

```
class AbstractBook {
    string title, author;
    int length;
    protected:
        AbstractBook& operator=(const AbstractBook& other) = default;
    public:
        AbstractBook(...) { ... }
        virtual ~AbstractBook() = 0;
        // Use the destructor to make a class abstract
        // If no other method makes sense
        ... // Rest of Big 5
};

class Regular: public AbstractBook {
    public:
        Regular& operator=(const Regular& other) {
            AbstractBook::operator=(other);
            return *this;
        }
        ... // Rest of Big 5
};

... // Similar code for other 2 subclasses (Text and Comic)
```

The above code will not suffer from either mixed or partial assignment:

- **Mixed Assignment:** `Text` can only be set to `Text`, `Comic` only to `Comic`, etc. So setting one tyoe of book to another will raise a compilation error, because there is no operator overload for it.
- **Partial Assignment:** We can not perform assignment to `AbstractBook` objects because it's constructor is `protected` and can thus only be called from inside it or any of it's subclasses.

Templates

Let's consider the linked list that was implemented in the past.

```
class List {
    struct Node {
        int data;
        Node *next;
    };
    Node *head;
    public:
        class Iterator {
            int& operator*() const;
            ... // Other iterator methods
        }; // End iterator
        int ith(int i) const;
        void addToFront(int n);
};
```

While this implementation works, it only works with `int`s. If we wanted to make a `List` of any other type (like `string`, `bool`, or `Student`), then this implementation will not work. There are multiple options for how to solve this issue:

- Copy-paste class contents and change the type of data stored: This works, but is not effective, since every minor change needs to be reflected in every `List` implementation.
- Use `void` pointers: We would lose the ability to use overloaded operators on the contents of the list, and also run the risk of memory leaks if the client code is not implemented correctly.
- Use a **template**: This is the best option available, and involves creating a "template" for the `List` class, which is used to automatically generate classes for each type that is being stored.

Example: List class implemented with templates, along with example client code.

```
template<typename T> class List {
    struct Node {
        T data;
        Node *next;
    };
    Node *head;
    public:
        class Iterator {
            T& operator*() const;
            ... // Other iterator methods
        }; // End iterator
        T ith(int i) const;
        void addToFront(const T& n);
};

// Client code
List<int> l;
l.addToFront(2);
l.addToFront(3);

List<string> ls;
ls.addToFront("hello");

for (List<int>::Iterator it = l.begin(); it != l.end(); ++it) {
    cout << *it << endl; // Iterator works as expected
}

List<List<int>> l3; // Can make a List of Lists
l3.addToFront(l);
```

Templates are just as fast during runtime as making a class for each data type, although compile time might slightly increase. This is because the compiler creates a copy of the template class for each type T that is used.

Note: C++ templates are actually Turing complete, and many calculations that don't depend on user input can be calculated during compile time.

Vectors (Standard Template Library)

C++ has a **Standard Template Library**, which contains a collection of useful templated classes.

One of these classes is std::vector. It is found in <vector> and is an automatically resizing array.

Example: Client code for std::vector

```
vector<int> v{4, 5}; // Contains {4, 5}
v.emplace_back(6); // Contains {4, 5, 6}
v.emplace_back(7); // Contains {4, 5, 6, 7}
```

All memory management is handled by the `vector` class, and there is no need to allocate or free memory when using it.

Note: When initializing `vector`s, using `{}` and `()` cause different behaviour.

```
vector<int> v{4, 5}; // Contains {4, 5}
vector<int> v(4, 5); // Contains {5, 5, 5, 5}
```

We can also use automatic type deduction for `vector`s, altuough it is not recommended.

```
vector w{1, 2, 3}; // int type is inferred
```

Example: Looping through `vector`s in different ways.

```
// Regular loop (like for arrays)
for (int i = 0; i < v.size(); ++i) {
    cout << v[i] << endl; // Constant time
    // vector class has overloaded [] operator
}

// Using an iterator
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
} // Notice iterator here has a lowercase 'i'

// Iterator with range-based for loop
for (int n : v) cout << n << endl;

// Reverse iterator
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
    cout << *it << endl;
} // We can not make a range-based for loop for the reverse iterator
```

More `vector` functions:

- `v.pop_back();` Removes the final element
- `v.erase(it);` Removes the element pointed to by `it`

Note: Since `v.erase(it);` deletes the current element, it means it updates the position of the iterator, and this there is a specific syntax that needs to be used when calling it in a for loop.

Example: Code that iterates through a `vector` and removes all elements that are ewual to `5`.

```
// Notice that we can use automatic type deduction
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it == 5) v.erase(it);
    else ++it;
}
```

It is recommensed to use `vector`s instead of `new[]` and `delete[]` since they are safer. It is also guaranteed that `vector`s are implemented using arrays (and thus have constant access times) regardless of the compiler, platform, and flags used.

Design Patterns

In general, we would like to program for interfaces instead of implementations. For this, we use abstract classes to provide method names and signatures, which we can inherit from custmize the behaviour of the functions.

Iterator Pattern

We know that the `Iterator` is a design pattern. Since a similar implmentation could be used in many places, the C++ language has the range-based for-loop, in order to simplify the use of `Iterator`s.

We can simplify `Iterator`s even further, by manking an Abstract iterator class, and inheriting from it to make more iterators, which all share the same basic behaviour.

Example: Abstract iterator class.

```
// Abstract class
class AbstractIterator {
    public:
        virtual int operator*() const = 0;
        virtual AbstractIterator& operator++() = 0;
        virtual bool operator!=(const AbstractIterator& other) const = 0;
        virtual ~AbstractIterator() {}
};

// We can now inherit from this class to make other iterators
class List {
    ... // Fields and methods
    public:
```

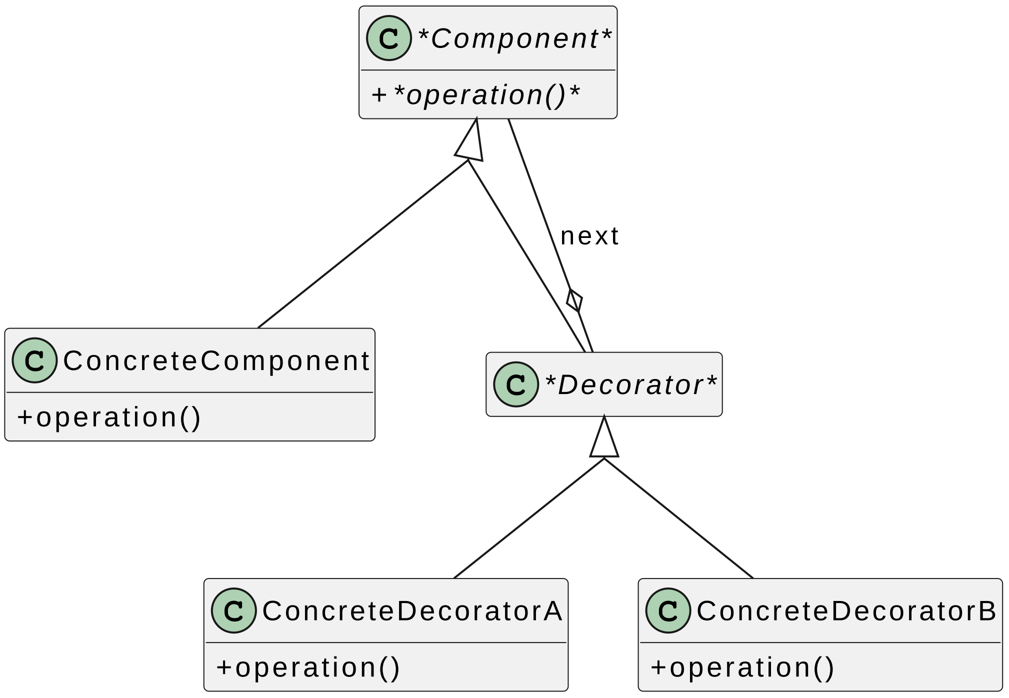
```cpp
        class Iterator: public AbstractIterator {
            ... // Implement virtual methods
        };
};

class Tree {
    ... // Fields and methods
    public:
        class Iterator: public AbstractIterator {
            ... // Implement virtual methods
        };
};

// We can also create functions that work on an AbstractIterator. These
functions can be called with it's subclasses without changing the functions
themselves
void foreach(AbstractIterator& start, AbstractIterator& end, void (*f) (int))
{
    while (start != end) {
        f(*start);
        ++start;
    }
}
```

Decorator Pattern

We want to be able to add / remove functionality at runtime. For example, let's consider making a GUI windowing system. We might want a basic window, as well as some toggleable features, like *Tabs*, *Scrollbar*, and *Bookmarks*.

If we were to create a superclass, and make a subclass for every permutation of features being enabled / disabled, we would have to make 2^n subclasses (where n is the number of toggleable features). This causes exponential growth in the amount of code needed, which is undesirable.

Inetead, we could use a decorator pattern. A UML diagram is given below.

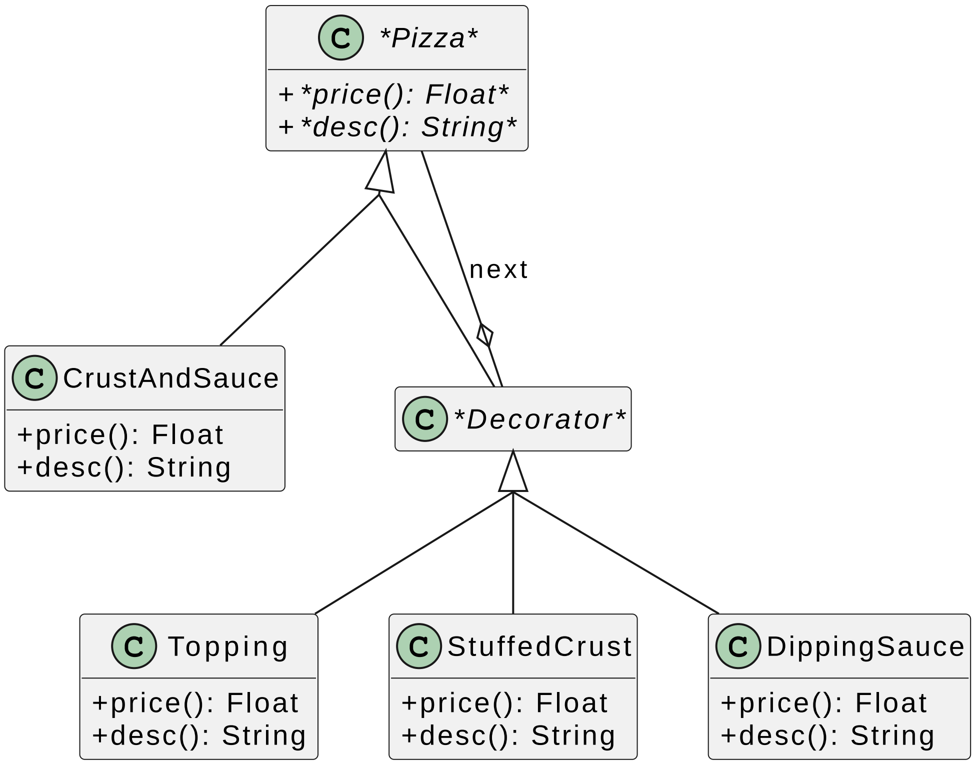


This acts like a linked list of functionality. Here, `ConcreteComponent` describes the default behaviour without any added features, and `ConcreteDecorator`s add on to the functionality of the rest of the list.

Example: Pizza Ordering App.

Imagine we wanted to make an app to order pizza, and wanted to be able to add different toppings, sauces, and crusts.

This is the UML diagram:



This is the code implementation:

```
// Base class
class Pizza {
    public:
        // Pure virtual methods
        virtual float price() const = 0;
        virtual string desc() const = 0;
        virtual ~Pizza() {} // Virtual destructor
};

class CrustAndSauce: public Pizza {
    public:
        // Overriden methods
        float price() const override { return 7.99; }
        string desc() const override { return "pizza"; }
};

class Decorator: public Pizza {
    protected:
        // Linked list implementation
        // Nothing else is required
        // Subclasses add other fields
        Pizza *next;
    public:
        // Constructor and destructor
        Decorator(Pizza *p): next{p} {}
        ~Decorator() { delete next; }
}; // This is abstract since price() and desc() are not implemented

class Topping: public Decorator {
    // Name of topping
    string name;
    public:
        // Constructor
        Topping(string name, Pizza *p): Decorator{p}, name{name} {}
        // Overriden price() and desc() methods
        float price() const override {
            return 0.99 + next->price();
        }
        string desc() const override {
            return next->desc() + " with " + name;
        }
};

class StuffedCrust: public Decorator {
    public:
        // Constructor (only needs to call superclass ctor)
        StuffedCrust(Pizza *p): Decorator{p} {}
        // Overriden price() and desc() methods
        float price() const override {
            return 1.50 + next->price();
        }
        string desc() const override {
            return next->desc() + " with stuffed crust";
        }
};

// Example client code
int main() {
    Pizza *newPizza = new CrustAndSauce{}; // Basic Pizza
    myPizza = new Topping{"cheese", myPizza}; // Add toppings
    myPizza = new Topping{"olives", myPizza};
    myPizza = new StuffedCrust{myPizza}; // Add stuffed crust
}
```

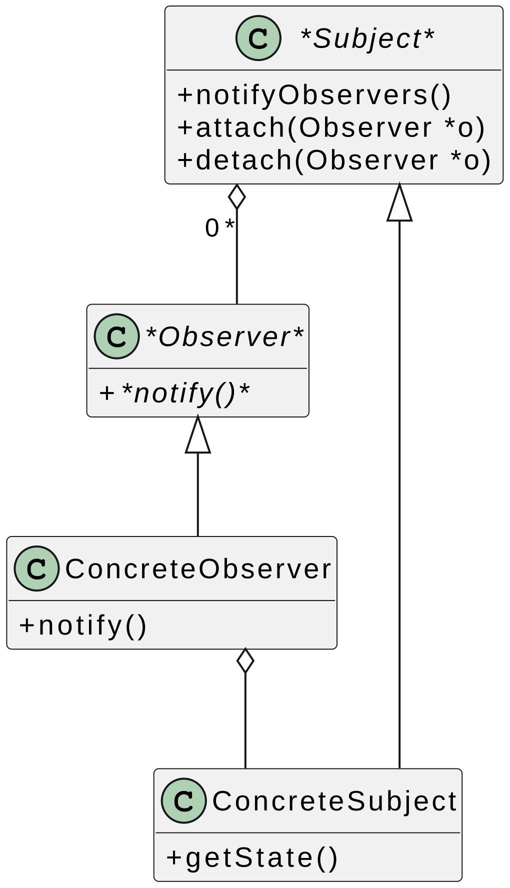
Here, the decorators act as a linked list, with the stuffed crust being the first in the list, which points to olives, which points to cheese, which finally points to the crust and sauce. Any functions that are called to the pizza firsr call the stuffed crust, which then calls the toppings and then the crust and sauce by using `next`, recursively generating the price or description.

Note: We could also refactor this implementation to make it so that the `CrustAndSauce` is also a `Decorator`, and the linked list of `Decorator`s ends when `next` for any `Decorator` is `nullptr`. However, this implementation requires a lot more testing and sanity checks.

Observer Pattern

Consider the case where we want some class with data to dispatch updates to other objects whenever it's data changes. For example: graphs in spreadsheets, social media notifications.

Here is the UML diagram for the Observer Pattern:



This is the steps of execution when the `Subject` is changed:

1. Concrete `Subject` is changed
2. `notifyObservers` is called (either inside or outside the class)
3. `notify` is called for each `Observer` in the `Subject`'s list
4. Each `ConcreteObserver` can call `getState` and update accordingly

Example: Twitter, or similar social media app.

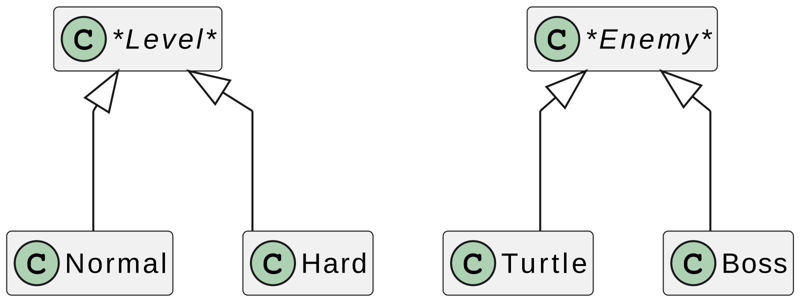
Concrete Subject: `Tweeter`, publishes tweets to followers *Concrete Observer*: `Follower`, reacts to updates to the one `Tweeter` they follow.

```
// Abstract Subject class
class Subject {
    // Observers vector
    vector<Observer *> observers;
    public:
        // Notify observers
        void notifyObservers() const {
            for (auto p: observers) p->notify();
        }
        // Add new observer
        void attach(Observer *o) { observers.emplace_back(o); }
        // Remove observer
        void detach(Observer *o) {
            // Iterate through vector
            for (auto p: observers) {
                // Remove and break out if o matches
                if (*p == o) {
                    observers.erase(p);
                    break;
                }
            }
        }
        // Pure virtual destructor
        // Since we don't have any other function
        // to make pure virtual
        virtual ~Subject() = 0;
};

// External definition for destructor
Subject::~Subject() {}

// Abstract Observer class
class Observer {
    public:
        // Pure virtual notify function and destructor
        virtual void notify() = 0;
        virtual ~Observer() {}
};

// Tweeter
class Tweeter: public Subject {
    string lastTweet; // Most recent tweet
    ifstream file; // File to read from
    public:
        // Constructor
        Tweeter(const string& filename):
            file{filename} {}
        // tweet function
        bool tweet() {
            // set lastTweet to line from file
            getline(file, lastTweet);
            // Check whether file still has lines
            return file.good();
        }
        // getState - getter for most recent tweet
        string getState() const { return lastTweet; }
};

// Follower
class Follower: public Observer {
    string name; // Name
    Tweeter *iFollow; // Tweeter being followed
    public:
    // Constructor
        Follower(string name, Tweeter *t):
            name{name}, iFollow{t} {
                iFollow->attach(this);
        }
        // notify function
        void notify() override {
            // Get latest tweet
            string tweet = iFollow->getState();
            // Check if name is found in tweet
            // and print a message accordingly
            if (tweet.find(name) != string::npos) {
                cout << name << "says: Yay!" << endl;
            } else {
                cout << name << "says: Boo!" << endl;
            }
        }
};

// Example client code
int main() {
    // Create Tweeter
    Tweeter elon{"elon.txt"};
    // Create Followers
    Follower joe{"joe", &elon};
    Follower mary{"mary", &elon};
    // Tweet and notify Followers
    while(elon.tweet()) {
        elon.notifyObservers();
    }
}
```

Factory Method Pattern (Virtual constructor pattern)

Problem: Want to be able to create diferent versions of an object based on policies that should be easily customizable.

Example: Enemy and Level generation.



Let's say that `Normal` levels generate mostly `Turtle`s, and some `Boss`es, while `Hard` leveles generate mostly `Boss`es, and some `Turtle`s.

```cpp
class Level {
    ... // Private fields and methods
    public:
        virtual Enemy *getEnemy() = 0;
        virtual ~Level() {}
        ... // Other public fields and methods
};

class Normal: Public Level {
    ... // Private fields and methods
    public:
        Enemy *getEnemy() override {
            // Mostly Turtles, some Bosses
        }
        ... // Other public fields and methods
};

class Hard: Public Level {
    ... // Private fields and methods
    public:
        Enemy *getEnemy() override {
            // Mostly Bosses, some Turtles
        }
        ... // Other public fields and methods
};

// Client code
Level l = ...;
Enemy *e = l.getEnemy();
// We use e and l, and access their public methods
// This makes it easy to add new Levels, Enemies, and policies
```

Here, we are accessing the enemies in the level using a public pure virtual method. By changing the subclass that we are creating, the behaviour of the functions will get altered based on the subclass being used. In this way, we do not have to change the client code whenever we add or change levels and / or enemies.

Template Method Pattern, NVI

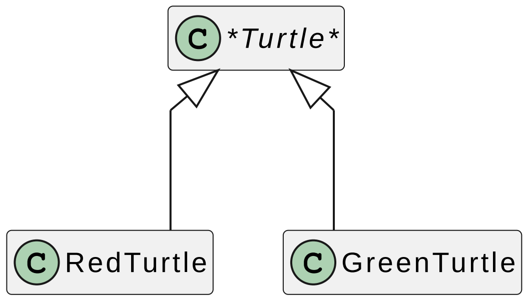
- Not related to the C++ templates, but share the same name.

Problem: What if we want some customizable behaviour in the subclasses, but not all.

Tenplate Mathod Patterm:

- Some portions of a superclass algorithm should be costomizable, whereas other parts stay the same.

Example: Code to draw a Turtle.



```
class Turtle {
    void drawHead() { /* Draw Turtle Head */ }
    void drawLegs() { /* Draw Turtle Legs */ }
    virtual void drawShell() = 0;
    public:
        void draw() {
            drawHead();
            drawShell();
            drawLegs();
        }
};

class RedTurtle: public Turtle {
    void drawShell() override {
        // Print out Red Shell
    }
};

class GreenTurtle: public Turtle {
    void drawShell() override {
        // Print out Green Shell
    }
};
```

Whenever we call `draw()` on a `Turtle*`, the `drawHead()` function is called from the `Turtle` class. Them the `drawShell()` function is called from either the `RedTurtle` or the `GreenTurtle` class. Finally, the `drawLegs()` function is called from the `Turtle` class.

Purpose of `public` methods - To provide an interface to clients, with invariants, pre-post conditions, and a description of what the method does.

Purpose of `virtual` methods - To provide an "interface" for subclasses to change the bahaviour of.

What about `public virtual` methods? The very definition of them is contradictory, since they promise behaviour to clients, while giving subclasses the power to change their behaviour. Thus, there is no guarantee that the subclasses will respect the invariants when overriding.

The **Template Method Pattern** may be generalized into **Non-Virtual Idiom (NVI)**. The **NVI** states the following:

1. Public methods should be non-`virtual`.
2. Virtual methods should be `private` or `protected`.
3. Exception: Destructor should be `public virtual`.

Example: No NVI vs. NVI.

```
class DigitalMedia { // No NVI
    public:
        virtual void play() = 0;
};

class DigitalMedia { // With NVI
    virtual void doPlay() = 0;
    public:
        void play() { doPlay(); }
}
```

The main benefit here is flexibility.

- We can add code that will always run when something is played by putting it before / after `doPlay()`, ex. logging, statistics, cover art, check ability to play.
- We can also add more "hooks" for customization by adding `private virtual` methods, like `showCoverArt()`.

We do not need to change the public interface for any of these modifications. The code will also remain as fast as before, since the compiler will optimize out the extra function call.

While it is also possible to add new features without using NVI, we would have to track down every location where `play()` is called and add the code to each location. Thus, it is easier to use NVI from the start rather than adding it later.

Advanced C++ Features

Exceptions (Error Handling)

Let's revisit vectors. Recall that `v[i]` gets the `i`-th element of the vector. If `i` is outside bounds, then there is undefined behaviour. However, there is also a different method. `v.at(i)` also gets the `i`-th element of the vector, but if `i` is outside bounds, then an error is signalled rather than the program crashing, This is an example of how error handling works in C++. Before we go into detail about C++ error handling, let's recall how we would perform error handling in **C**:

1. Reserve a sentinel value like `-1`, `INT_MIN`, etc.
   This shrinks our return space and makes it so that the reserved sentinel value can not be read in as input. Furthermore, reserving a sentinel value is much more complicated when the value to be returned is an array, `vector`, or `class`.
2. Return a `struct` with an error field.
   a. This wastes unnecessary space for most returns.
   b. The error field is very easy to ignore
3. Use a global variable, like the `errno` integer.
   This is also easy to ignore, and may be overwritten if there are multiple errors that result from the same line of code, resulting in one of them being ignored.

In order to handle errors better, C++ introduced exceptions.
In order to signal an error, an exception (exn) can be **raised** / **thrown**.. The program then goes to a **handler** / **catch block** to deal with the exception.
If no appropriate handler is found, then the program crashes using `std::terminate`, where the program instantly termainates, without freeing up any memory.

If `v.at(i)` has `i` out of bounds, then a `std::out_of_range` exception is thrown, which is found in the `<stdexcept>` library.

Another example is when we call `new`, and the OS rejects the request for more memory. In this case, a `std::bad_alloc` is thrown.

Example: Throwing and catching errors.

```
vector<int> v{1. 2. 3};

try {
    // This line throws an error
    int n = v.at(100);
    // The control flow jumps to the catch block
    // This line will never be run
    cout << n << endl;
}
catch (out_of_range r) {
    // r.what() returns a string that describes the error
    cout << "Error: " << r.what() << endl;
}
```

Notice that `out_of_range` is just an object, and `what()` is a method describing the error that occurred. All exception classes are just special types of objects, that store a string that explains the type of error.

Example: Throwing a custom error and catching it.

```
int main() {
    try {
        h();
    } catch (out_of_range r) {
        cout << r.what() << endl;
    }
}

void h() { g(); }
void g() { f(); }
void f() {
    throw out_of_range{"f threw"};
}
```

Once the exception is thrown, the control flow transfers from `f` directly to the `catch` block. In doing so, we perform **stack unwinding**, where all stack memory between the error and the `catch` block gets freed (in this case, it would be the stacks of `f`, `g`, and `h`). However, heap memory is **NOT** freed here, and thus any heap-allocated memory in the error area that was also meant to be freed withing the error area would be leaked.

However, it is possible to ensure that heap-allocated memory will also get freed in case an `exception` is thrown. More on this later.

After the exception is handled, the control flow switches to the code directly after the catch block, and the program continues running.

Sometimes. we want a catch handler to perform some work, and then raise a new `exception`.

Example: Catching an `exception` and throwing another one.

```
try { ... }
catch (out_of_range r) {
    // Do some work, any changes, etc.
    throw invalid_argument{"Reprompt Input"};
}
```

Example: Catching an `exception` and throwing it again.

```
try { ... }
catch(out_of_range r) {
    // Do some work, any changes, etc.
    throw;
}
```

We need to use `throw`, and not `throw r`, due to the chance of there being an **inherited error class**. Consider an error type `ErrorSituation`. Since an exception is just a class, and can be inherited from. Let's say that we have an error type `SpecialError` that inherits from `ErrorSituation`. Now, consider the following code:

```
try { ... }
catch (ErrorSituation& e) {
    throw e;
}
```

In this case, if `e` was an `ErrorSituation`, then there wouldn't be any problems. However, if `e` was a `SpecialError`, then **object slicing occurs**, as the `SpecialError` gets converted into an `ErrorSituation` object. However, if we just use `throw`, then no object slicing would occur.

We can also write a `catch` statement that works for any type of `exception`.

Example: Generic `catch` statement (for all error types).

```
try{ ... }
catch (...) { // 3 dots inside catch () means catch all
    ... // Error handling
}
```

In most cases, we `throw` and `catch` special objects. However, it is possible to `throw` and `catch` anything, from primitives like `int` to custom data types.

It is also possible to create custom exception types:

```
class BadInput { ... };
try { throw BadInput{}; }
catch (BadInput& b) { ... }
```

Advice: `throw` exceptions by value, and `catch` them by reference. This prevents object slicing.

Advice: Do not let destructors `throw` an exception. Dectructors have an `implicit` tag called `noexcept`, which means that if we were to throw an exception inside a destructor, then the program immediately crashes (using `std::terminate`).

However, if we were to tag our destructor as `noexcept(false)`, then it will not immediately crash, but however, it can cause an issue with stack unwinding.

A rule in C++ is that we have at most **one** exception at any given time. If an error occurs and causes stack unwinding, then the destructor runs for all the stack allocated objects. If there is an exception inside any of the destructors, then it will cause two different exceptions:

1. The original exception that caused the stack unwinding.
2. The exception that was thrown inside the destructor.

The presence of two active exceptions will cause the program to crash using `std::terminate`.

STL Maps, Structured Bindings

An **STL Map** is an Array that can be indexed with different types rather than just an integer. It is found in `<map>`.

Example: Creating and using an STL Map.

```
map<string, int> m;
m["abc"] = 2;
m["def"] = 3;
cout << m["abc"] << endl; // 2
cout << m["ghi"] << endl;
// If a key is not found, then the value is default constructed (objects) or
zero-initalized (primitives).

if (m.count("abc")) { ... } // m.count returns 1 if the key is found, and 0
otherwise
m.erase("abc"); // Removes the provided key and associated value
```

Iterating through an STL Map:

```
for (auto& p : m) {
    cout << "key: " << p.first << ", value: " << p.second << endl;
    // p.first is the key and p.second is the value
}
```

In the above example, `p` is of type `std::pair<string, int>&`. It is stored in the `<utility>` library.

`pair` is just a `struct` with two different fields, each storing a different template type. Here it is a `string` and an `int`. We are using `struct`s here because it is just a collection of data with no invariants to preserve.

Alternatively, we could also use a structured binding:

```
for (auto& [key, value] : m) {
    cout << key << ", " << value << endl;
}
```

Structured bindings are used to convert data stored somewhere, typically in a struct (or class with public fields) or array, into local variables.

```
vec v{1, 2};
auto [localX, localY] = v;
// localX will be set to 1 and localY to 2
```

We can also use structured bindings for stack-allocated arrays, if the size is known:

```
int a[3] = {1, 2, 3};
auto [x, y, z] = a;
```

Coupling / Cohesion

What should go into a module?
How to tell if code is well-structured?

So far, we have stored one `class` in each module. Larger programs contain multiple `class`es per module.

Coupling: To what extent do modules depend on each other:

- ***Low Coupling***: Simple communication via parameters / results.
- **<**   Communication via arrays or structures.
- **<**   Modules affect each others' control flow.
- **<**   Modules share global data.
- ***High Coupling***: Modules have access to each other's implementation (`friend`s).

Cohesion: How much do parts of a module relate to each other?

- ***Low Cohesion***: Module parts are unrelated (ex. `<utility>`)
- **<**   Module parts share a common theme, but otherwise unrelated (ex. `<algorithm>`)
- **<**   Module parts cooperate to manage a lifetime state (ex. read/open/write files)
- ***High Cohesion***: Module parts co-operate to perform one task.

We desire low coupling and high cohesion. The combination of both makes the program easy to understand and change.

Example: How to declare 2 classes that depend on each other.

```
class A {
    int x;
    B y;
};
```

```
class B {
    int x;
    A y;
};
```

We can not determine the size of A or B. Thus, we break the chain of dependencies via a pointer.

```
class B; // Forward declaration

class A {
    int x;
    B *y;
};

class B {
    int x;
    A *y;
};
```

We can not create forward declarations in a different module to the one where the `class` is actually declared, and thus A and B need to be in the same module.

Note: Forward declaration is not allowed for object fields or for inheritence.

Example: Let's consider applying **coupling** and **cohesion** to `TicTacToe`.

```
class Board {
    ... // Private fields and methods
    public:
        void play() { ... cout << "Your Move" << endl; }
        ... // Other public fields and methods
};
```

Here, `Board` is coupled with `cout`. We cn not reuse `Board` without getting print statements. What if we wanted to print to a file, or not print anything at all?

```
class Board {
    istream& in;
    ostream& out;
    ... // Other private fields and methods
    public:
        Board(istream& in, ostream& out): in{in}, out{out} { ... }
        void play() { ... out << "Your Move" << endl; }
        ... // Other public fields and methods
};
```

While this allows us to cusomize the stream we use, we are still coupled with streams. What if we wanted to use a graphical display or a Web API? The `Board` should not communicate with the user.

Single Responsibility Principle: Each `class` should have exactly one reason to change. If changes to two different parts of the specification both require changes to the same `class`, the the **SRP** is violated.
`Board` State and Communication are both different things.

We used to use the `main` function to perform most communication, however this is not good style, since the `main` function is not reusable like other classes are.

Instead, we should split the program into `Model`, `View`, and `Controller` for handling data, output, and input respectively.

Model View Controller (MVC)

Model, View, Controller Architecture:

- ***Model***: Keeps track of the data, and the logic surrounding the data.
  Communicates with the **Controller** via parameters / results.
  - Sometimes, there is an Observer relationship between the **Model** and **View**(s).
- ***View***: Handles output, and communication with the user.
- ***Controller***: Manages input, and control flow between the **Model** and **View**.
  May encapsulate logic for rules / turn-taking.
  - Sometimes, input is taken from the **View** (ex. if it is a window).

The **Controller** receives input from the user, passes commands to the **Model**, and receives results from it. The **Controller** also communicates with the **View** to provide information to be displayed, and the **View** passes control back to the **Controller**.

The **MVC** architecture promotes the reuse of code. For example, one **Model** (like a `TicTacToe` game) could be reused with different **Model**s and **Controller**s (ex. CLI game, AI training, web app, etc.).

Exception Safety

Consider the following code, assuming that `C` is any object:

```
void f() {
    C myC{};
    C *myCptr = new C{};
    g();
    delete myCptr;
}
```

While it might seem like this code is perfect, there is one edge case. If there was an error inside `g()`, and an exception were to be thrown, then stack unwinding occurs, and `f` will be removed from the stack since it does not `catch` any errors. While stack-allocated variables like `myC` will be freed during stack unwindong (the destructor is called), the heap allocated `myCptr` will not be freed since it's destructor is never called.

Consider the above code with the code to free heap variables in case of an error:

```
void f() {
    C myC{};
    C *myCptr = new C{};
    try{ g(); }
    catch(...) { delete myCptr; throw; }
    delete myCptr;
}
```

This works. However, it is super clunky. The code to delete heap memory is duplicated here, and we would also have to repeat it for every function we call that could `throw` an exception. Furthermore, the code we need to at to each `catch` block would only get more complicated every time we allocate more memory from the heap in the function.

We want to make sure that `delete` runs for all the heap-allocated memory regardless of whether the function terminates normally or due to an exception. Other languages have a `finally` clause, which runs regardless of whether an exception was throws or not. However, C++ does not have this.

There **IS** a guarantee that the destructor will run for all stack-allocated objects during stack unwinding.

Solution: Wrap dynamically allocated memory in a stack-allocated object.
In general, using stack-allocated objects is preferred.

RAII - Resource Acquisition is Initialization

Constructor: Acquires the resource
Destructor: Releases the resource

We have already used RAII before. Consider using an `ifstream`:

```
{ // Start of some function
    ifstream f{"file.txt"};
} // End of the function
```

Here, the file is accessed in the constructor of the `ifstream`, and closed in the destructor, which runs when the function that the `ifstream` is defined in exits.

`unique_ptr`

`std::unique_ptr<T>`, found in `<memory>` is an RAII class that manages dynamic memory.
Constructor: Takes in a `T*`
Destructor: Deletes the pointer

Example: Using RAII in a function using `unique_ptr`.

```
// Once again, let C be any object
void f() {
    C myC{};
    unique_ptr<C> myCptr{new C{}};
    g();
}
```

Whichever way `f` exits, the destructor will run on `myCptr`, and the memory will be freed.

We could also use the `std::make_unique` function, also found in `<memory>`:

```
void f() {
    C myC{};
    auto myCptr = make_unique<C>();
    g();
}
```

`make_unique` creates a `C` object in the heap using `new` and the arguments that are provided to it.

What heppens if we copy a `unique_ptr`?

```
auto p = make_unique<C>(...);
unique_ptr<C> q = p; // Invokes copy ctor for unique_ptr
```

Copying a `unique_ptr` doesn't make sense since having two different `unique_ptr` objects pointing to the same memory address would cause a double `delete`. Because of this, the copy constructor and copy assignment operators are disabled for `unique_ptr`.

`unique_ptr` also has a function called `get()`, which returns the memory address of the pointer that it contains.

Example: Implementation of `unique_ptr`.

```
template <typename T> class unique_ptr {
    T *ptr;
    public:
        // Ctor and dtor
        explicit unique_ptr(Y *ptr): ptr{ptr} {}
        ~unique_ptr() { delete ptr; }
        // Disable copy ctor and assignment operator
        unique_ptr(const unique_ptr<T>& other) = delete;
        unique_ptr<T>& operator=(const unique_ptr<T>& other) = delete;
        // Move ctor and assignment operator
        unique_ptr(const unique_ptr<T>&& other) = delete:
            ptr{other.ptr} { other.ptr = nullptr; }
        unique_ptr<T>& operator=(const unique_ptr<T>&& other) {
            delete ptr;
            ptr = other.ptr;
            other.ptr = nullptr;
            return *this;
        }
        // Functions to get pointer and reference
        T& operator*() { return *ptr; }
        T* get() { return ptr; }
};
```

If we need to copy a pointer, what should we do?
We first need to ask, "Who is in charge of ownership?"

- Whoever owns the memory gets the `unique_ptr`
- Everyone who just uses the memory can access it via raw pointers. We can use `p.get()` for this.

Our new understanding of ownership can be signalled via type.

- `unique_ptr` - Represents ownership. Associated memory is deleted when it goes out of scope.
- **Raw pointers** - Represent non-ownership. The default is that they do not free memory when they go out of scope.
- **Moving** `unique_ptr` - Transfer of ownership.

Parameters:

- `void f(unique_ptr<C> p)`: `f` takes ownership of the `unique_ptr`, which is deleted when `f` finishes running.
- `void g(*p)`: Ownership is not transferred from the caller to `g`. `g` should just use `p`, and not delete it.

Results:

- `unique_ptr<C> f()`: Returns always transfer ownership. The pointer is now owned by the caller.
- `C *g()`: Call shouldn't delete the returned value. It is either stack memory or owned by someone else.

`shared_ptr`

Rarely, we my also need true shared ownership. Consider a graph data structure.
In such a case, we can use shared pointers (`std::shared_ptr`, found in `<memory>`).

Example: Using `std::shared_pointer`.

```
{ // Start of some function
    auto p = make_shared<C>{...};
    if(...) {
        auto q = p;
    }
} // End of the function
```

Once the `if` statement finishes executing, `q` gets deleted but `p` still exists and uses the pointer. Once the function ends, `p` goes out of scope, and the `shared_ptr` is deleted since nothing uses the pointer anymore.

`shared_ptr`s work by maintaining a **reference count**.
On creation, it is incremented, and on deletion, it is decremented.
When it reashed `0`, the underlying object is deleted.

Exception Safety Revisited, PImpl

Exception safety **does not** mean that a function never throws, or that it handles all `exception`s internally.
If is about the **guarantees** we are given if we call a function and it throws an `exception` at us.

1. **Basic Guarantee**: If an `exception` is thrown from a function `f`, then the program is in a valid but unspecified state. `Class` invariants are maintained, and there is no memory leaks or data corruption. But that is all that can be guaranteed. Some data could've changed.
2. **Strong Guarantee**: In an `exception` is thrown from `f`, then the program state is returned to before the function call. Thus, it will be like the function was never called in the first place.
3. **NoThrow Guarantee**: If we call `f`, then it will never throw an `exception` and it will always do it's job.

Consider the following code. What is the exception safety of `c::f` below?
Assume that `A::g` and `B::h` both provide the strong guarantee.

```
class C {
    A a;
    B b;
    public:
        void f() {
            a.g();
            b.h();
        }
};
```

If `a,g()` throws, then it will undo any side effects since it provides the strong guarantee. So it is as if the function was never called.

If `b.h()` throws, then it will undo any of it's own side effects, but it can not undo any of `a.g()`'s work, since it has already been successfully run.

Therefore, `C::f` can only provide the basic guarantee. This is not optimal, and it would be recommended to modify the function to make it provide the strong guarantee.

Idea: Use temporary values. This way, if `b.h()` throws, then `a` and `b` aren't changed. This will work only if `a.g()` and `b.h()` have **only local side effects**. Consider the following code:

```
class C {
    A a;
    B b;
    public:
        void f() {
            A aTemp = a;
            B bTemp = b;
            aTemp.g();
            bTemp.h();
            a = aTemp;
            b = bTemp;
        }
};
```

While this code looks like it provides the strong guarantee, there is one case where it would not. If an error is thrown in the copy constructor for `B` in the line `b = bTemp`, then `a` has already been modified. Thus, this implementation also only provides the basic guarantee.

We need to use the **PImpl (Pointer to Implementation)** idiom.
With this method, we can simply swap pointers at the end, which is guaranteed to never throw an exception.

Example: Exception safety with PImpl.

```
struct CImpl {
    A a;
    B b;
};

class C {
    unique_ptr<CImpl> pImpl;
    public:
        void f() {
            auto temp = make_unique<CImpl> pImpl;
            temp->a.g();
            temp->b.h();
            std::swap(temp, pImpl);
        }
};
```

This implementation provides the **strong guarantee**.

In general, if `A::g` or `B::h` do not provide any exception safety, then neither does `C::f`.

Vectors - RAII and Exception Safety:

- `vector<C> v` - Represents ownership of `C` objects. When the destructor runs for `v` when it goes out of scope, the `C` objects are also deleted. Does not support polymorphism since any subclasses inserted into `v` will be sliced.
- `vector<C*> w` - Represents non-ownership. When `w` goes out of scope, the destructor will not delete what is pointed to by the `C*`s. Supports polymorphism since we can put subclass pointers in the vector.
- `vector<unique_ptr<C>> u` - When `u` goes out of scope, the destructor runs and frees memory. Indicates ownership and allows polymorphism.

Vectors and Exception Safety:

`vector<T>::emplace_back(...)` - Supports the strong guarantee. If it throws, then the dynamically allocated array is unchanged.

Pseudocode steps to implement the code for `emplace_back` when the array is filled up and needs to double in size, with strong guarantee:

- Slow approach, but works everywhere:
    1. Allocate a new array with 2 times the current capacity.
    2. Copy each object from the old array to the new array. If any of the copy constructors throw an error, then delete the new array and rethrow.
    3. Point the old array pointer to the new array.
- Faster approach, but requires `T`'s move constructor to provide the nothrow guaranee:
    1. Allocate a new array with 2 times the current capacity.
    2. Use the move constructor to move each object to a new array.
    3. Point the old array pointer to the new array.

As briefly mentioned in the title, the second approach does not always provide the strong guarantee, since it any of the move constructors throws an error, then the original array has been modified since all the objects before the erraneous object have been moved over to the new array.

In the real `emplace_back` implementation, if it is guaranteed that the move constructor will not throw, then the compiler will move elements in the vector. However, if this guarantee isn't present then the compiier will resize vectors via copying.

For the compiler to use the more optimal version of `emplace_back`, all moves and swaps must provide the nothrow guarantee. If we tag a function as `noexcept`, then the compiler will perform the optimizations.

Example: Using the `noexcept` tag.

```
class C{
    ...
    C(C&& other) noexcept;
    C& operator=(C&& other) noexcept;
};
```

Casting

Casting is the process of converting a variable of one type to a different type.

Recall, C casting:

```
Node n;
int *ip = (int *) &n;
```

C-style casting is not recommended in C++.

C++ instead provides 4 casting functions for various situations:

`static_cast`

Sensible casts with well definded semantics.

Ex. `float -> int`

```
float f;   void g(int x);   void g(float f);
g(static_cast<int>(f)); // Calls the int version of g
```

`static_cast` allows us to down cast: `Superclass * -> Subclass *`

```
Book *pb = ...;
Text *pt = static_cast<Text *>(pb);
cout << pt->topic << endl;
```

Here, `pb` must point at a `Text`, otherwise there is undefined behaviour.
The compiler trusts the client code to only run correct code.

`const_cast`

Allows you to remove `const` from a type.

```
void f(int *p); // Assume f doesn't modify whatever p points to
void g(const int *p) {
    f(p); // Won't compile
    f(const_cast<int *>(p)); // Will compile
}
```

If `f` changes `p`, then we get undefined behaviour.

`reinterpret_cast`

Take memory and reinterpret the bits stored there as a different type.

```
Turtle t;
Student *s = reinterpret_cast<Student *>(&t);
```

- The bahaviour depends on the compiler and object layouts.
- Casting one object to another only works if they have the same size.
- This is generally unsafe.

Example: Modifying private data with `reinterpret_cast`.

```
// Class definitions
class C {
    int x;
    public:
        explicit C(int xval): x{xval} {}
        int getX const { return x; }
};

class RogueC {
    public:
        int x;
};

// Client code
C c{10};
cout << c.getX() << endl; // Prints out "10"

RogueC *p = reinterpret_cast<RogueC *>(&c);
p->x = 20;
cout << c.getX() << endl; // Prints out "20"
```

In the above example, we were able to modify a private field in an object by accessing it as another object that has the sane fields which are public.

`dynamic_cast`

Allows checking which subclass a superclass pointer is pointing to.

```
Book *bp = ...;
Text *tp = dynamic_cast<Text *>(bp);

if (tp) cout << "Text" << endl;
else cout << "Not Text" << endl;
```

If `bp` points at a `Text`, then `tp` will point at the same `Text` object.
Otherwise, `tp` is set to `nullptr`

A caveat is that `dynamic_cast` only works if you have at least one virtual method.

Casting `shared_ptr`s

We can also cast from `shared_ptr`s to other `shared_ptr`s.

The following functions are available in `<memory>`:

- `static_pointer_cast`
- `dynamic_pointer_cast`
- `const_pointer_cast`
- `reinterpret_pointer_cast`

We can also `dynamic_cast` references:

```
Book& br = ...;
Text& tr = dynamic_cast<Text&>(br);
```

Since there is no such thing as a `NULL` reference, a `std::bad_cast` exception is thrown.

Polymorphic Assignment Problem Revisited

```
Text t1{...}, t2{...};
Book& r1 = t1; Book& r2 = t2;
r1 = r2;
```

- If `operator=` is non-virtual, we get *partial assignment*.

- If `operator=` is virtual, we get *mixed assignment*.

Example: Polymorphic assignment with `dynamic_cast`.

```
Text& Text::operator=(const Book& other) {
    if (this == &other) return *this;
    const Text& tother = dynamic_cast<const Text *>(other);
    Book::operator=(tother);
    topic = tother.topic;
    return *this;
}
```

If we provide anything that isn't a `Text` into this operator, an exception is thrown and partial or mixed assignment is avoided.

Is `dynamic_cast` good style?
With `dynamic_cast`, we can make decisions based on the runtile information (RTTI) of an object. For example:

```
void whatIsIt(shared_ptr<Book> b) {
    if (dynamic_pointer_cast<Text>(b)) cout << "Text" << endl;
    else if (dynamic_pointer_cast<Comic>(b)) cout << "Comic" << endl;
    else cout << "Regular Book" << endl;
}
```

This is very tightly coupled to the `Book` hierarchy. If a new subclass is added, then another `else if` statement must be added to the function. Furthermore, this must be repeated everywhere where this loop is used. In case any of them is missed, then there will be a bug.

In conclusion, whether `dynamic_cast` is good style or not depends entirely on it's use. The use in `operator=` doesn't need changing if we add more subclass types, and thus it is good style. However, the use in `whatIsIt` needs to change whener the subclass structure changes and it is thus typically considered bad style. There is a better way to implement `whatIsIt`, using virtual methods:

```
class Book {
    ...
    public:
        virtual void identify() { cout << "Book" << endl; }
};

class Text: public Book {
    ...
    public:
        void identify() override { cout << "Text" << endl; }
};
```

```
void WhatIsIt(Book *b) {
    if (b) b->identify();
    else cout << "Nothing" << endl;
}
```

We can use this solution in the following cases:

1. There are a large number of subclasses of `Book` and we want to identify all of them with ease.
2. `Book` and it's subclasses have a uniform interface, and the subclass behaviour doesn't deviate too much.

Whenever we add a new subclass, we also only need to add one function (`identify`), and we don't need to change the client code.

Consider the opposite case:

1. We know the subclasses in advance and we are fine with addding new subclasses requiring extensive code changes.
2. The subclasses may not conform to a uniform interface, and each my have significantly differing behaviour.

Consider this example:

```
class Enemy {
    ...
};

class Turtle: public Enemy {
    void stealShell();
};

class Boss: public Enemy {
    void epicBossBattle();
}
```

Here, adding new enemines will require large changes to the codebase anyway since each ma y have unique behaviour. In this case, using `dynamic_cast` isn't so bad.
Furthermore, inheritance might not be the best choice for relationship between the classes. Instead, we might want to use `std::variant`.

Variants

`std::variant`, found in the `<variant>` library, acts as a type safe union.

```cpp
// Type alias: Enemy now means variant<Turtle, Boss>
using Enemy = variant<Turtle, Boss>;

Enemy e{Turtle{...}}; // Or {Boss{...}}

// Finding the underlying type
if (holds_alternative<Boss>(e)) cout << "Boss" << endl;
else cout << "Turtle" << endl;

// Accessing value
try {
    Turtle t = get<Turtle>(e);
}
catch (bad_variant_access e) { ... }
```

If a `variant` is left uninitialized (ex. `Enemy e;`), it is set to the default construction of the first type in the variant list. If the first type is not default constructable, then there will be a compilation error.

If the first type it not default constructable, then we have the following options:

1. Add a default constructor.
2. Reorder the types so that the first one has a default constructor.
3. Use `std::monostate` - represents empty (defined as a `struct` with no fields or functions)
   Ex. `variant<monostate, Turtle, Boss>`

`std::optional<T>` is equivalent to `variant<monostate, T>`

Virtual Pointers, Vtables

How do virtual methods actually work?
Consider this code:

```cpp
struct Vec {
    int x, y;
    void f();
};

struct Vec2 {
    int x, y;
    virtual void f();
};

Vec v; Vec2 w;
cout << sizeof(int) << " " << sizeof(v) << " " << sizeof(w) << endl;
```

The above code would print out `4 8 16`. This shows that an integer is `4` bytes in size. A `Vec` object is `8` bytes in size, which makes sense since it is twice the size of an integer and it contains 2 integers and nothing else. However, a `Vec2` object is `16` bytes in size despite it containing the exact same fields. The only difference is that `Vec2` has a virtual method while `Vec` does not.

`w`, which is a `Vec2` uses up `8` bytes more space than `v`, which is a `Vec`.
If we call a regular method, then the compiler will make the correct function get called based on the static type, which can be determined during compile time. However, `virtual` methods are called based on the dynamic type, which can only be determined during runtime. Thus, the compiler needs a way to allow the object to access it's `virtual` methods based on it's dynamic type during runtime.

Any object that corresponds to a class with virtual methods also contains a **vptr** (virtual pointer). The vptr points to the **vtable** - which contains function pointers to the methods of a class. For `Vec2`, there is only one such function, `f`.

For the above example, a `Vec2` object is laid out as follows:

Vec2
`vptr` ---> *VTable*
`x`
`y`

VTable (for Vec2)
`void (*pf) ()` ---> *`virtual void Vec2::f()`*

For another example, consider the Library (`Book`) code from earlier:

```
Book *bp = new Text / Comic / Book; // Pick any one of the classes
bp->isHeavy();
```

The compiler generates the following instructions:

1. Follow **vptr** to **vtable**.
2. Find corresponding entry in vtable.
3. Jump to function pointer's location.

All of these happen at runtime for every virtual function call. Because of this, they are slower than non-virtual functions.

Additionally, objects corresponding to classes with virtual functions are larger (by 8 bytes) due to the presence of the vptr and thus consume more memory.

Classes with virtual methods always have their vptrs on the top, and then have the superclas fields, and finally the subclass fields. This is done because for the following reasons:

- It is not possible to determine where exactly the vptr is located at runtime if it is not on top of the class, since the size of the fields (and thus the vptr location) would depend on which superclass / subclass it is. Thus, keeping it before all the fields makes it easy to access the vptr regardless of the current dynamic type.
- By leaving the subclass objects after the superclass objects, when accessing the object as the superclass (i.e. through a superclass pointer), the subclass fields are simply ignored and only the superclass fields are accessed

Multiple Inheritence

Objects in C++ can inherit from multiple other objects.

Example: Multiple inheritence.

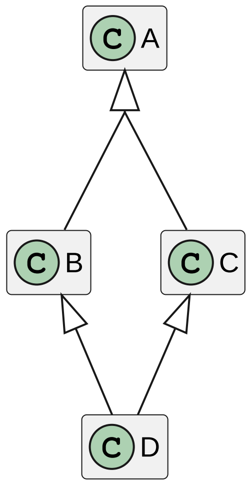
```
struct A { int a; };
struct B { int b; };

struct C: public A, public B {
    int c;
};

C cobj;
cout << cobj.a << " " << cobj.b << " " << cobj.c << endl;
```

Multiple inheritence is usually straightfoward, however, there are caveats when a class inherits from multiple classes which share an ancestor class.

Example: Multiple Inheritence with a shared ancestor.

```
struct A { int a; };
struct B: public A { int b; };
struct C: public A { int c; };

struct D: public B, public C {
    int d;
};
```

This is the UML diagram for the above code:



This is also known as the *Deadly Diamond (of Death)* because it can cause a lot of confusion.

For the above code, consider the following client code:

```
D d;
cout << d.a << endl;
```

While this seems like it will work, it actually causes a compilation error. This is because there is ambiguity, since every D object contains 2 A objects, one that is part of the B object and one that is part of the C object. Accessing the field a does not work since the compiler does not know which of the 2 a fields is supposed to be accessed.
In order to allevaiate this, the field name can be prefixed with the superclass name:

```
d.B::a; // Gets the 'a' field of the superclass of B
d.C::a; // Gets the 'a' field of the superclass of C
```

NOTE: This syntax of prefixing the inherited field name with the superclass name is actually valid syntax for all inherited objects in C++. It is not used often since it unnecessarily increases the amount of code to write.

If only one copy of the shared ancestor is needed, then this can be done with **virtual inheritence**.

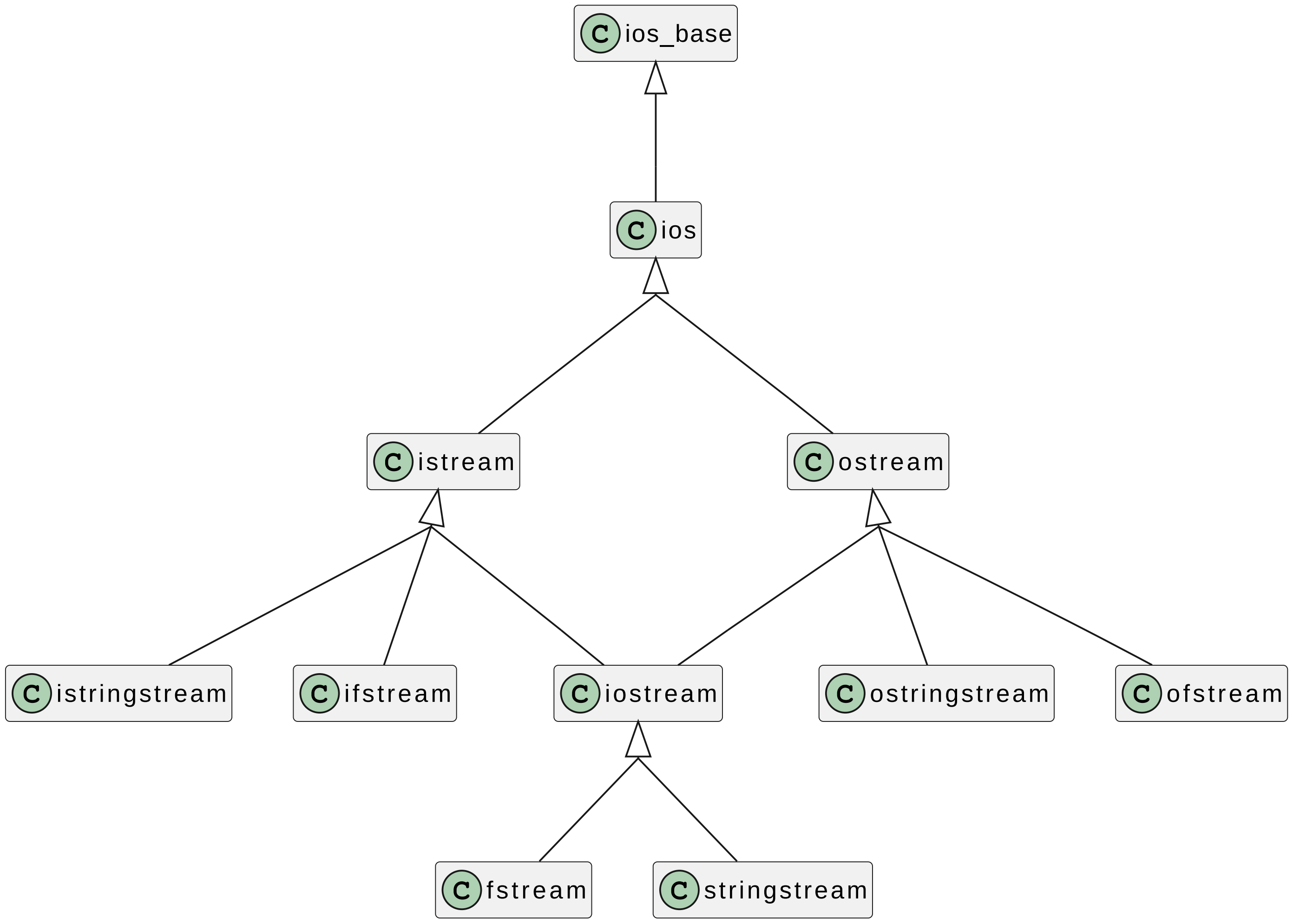
```
struct A { int a; };
struct B: public A { int b; };
struct C: public A { int c; };

struct D: public virtual B, public virtual C {
    int d;
};
```

If virtual inheritence is used for all the superclasses, then the shared ancestor is shared among the superclasses, and thus using syntax like `d.a` works.

Note that the ancestor field is only shared if *ALL* of the superclasses that have the ancestor fields are inherited virtually. If even one of the superclasses is inherited non-virtually, then every single superclass will have it's own copy of the shared ancestor class.

NOTE: Since UML is meant to be language-independent, there is no specific UML syntax for virtual inheritence. However, it is acceptable to label inheritence arrows as `virtual` for better understanding.

Consider this inheritence diagram of all the different C++ streams, which use virtual inheritance:



Notice that `iostream` inherits from both `istream` and `ostream`. `fstream` and `stringstream` allow for interaction with files and strings respectively, but allow for both reading from and writing to.

What about object layout?

Consider the deadly diamond example from earlier. The compiler can not simply arrange the fields class by class like it did for single inheritence, since there will always be a class that can not be formed correctly. Consider this example:

D
vptr ---> *VTable*
A Fields
B Fields
C Fields
D Fields

Using the above layout makes it possible for the class to be represented as an A object or a B object, but not a C object, since there will be seemingly gibberish B fields in between the A fields and the C fields. Thus, the above method **does not work**.

The compiler lays out the object in the following way.

D
vptr ---> *VTable*
B Fields
vptr ---> *VTable*
C Fields
D Fields
vptr ---> *VTable*
A Fields

To locate the fields for a specific class, the vtable is used, which stores the distance to the fields for each superclass' object. This is where virtual inheritence gets it's name.

Suppose we wanted an `A *` pointing at a D object. In this case, pointer adjustment occurs, and the pointer is made to point at the vptr that is before the A fields. This occurs with assignment and static / dynamic casting, but not with `reinterpret_cast`.

Template Functions

We have seen many functions that can operate on an arbitrary type, like `make_unique`, `static_cast`, and `dynamic_cast`, but how do they work?

They are **template functions**. Just like templated classes and structs, templated functions can operate on any type.

Example: Template function.

```
template <typename T> T min(T x, T y) {
    return x < y ? x : y;
}

// Cleint code
min(1, 2); // Calls min<int>. The type is automaticlly deduced
min<int>(1, 2); // We can specify the type of necessary
min('a', 'c'); // Calls min<char>
min(1.5, 2.3); // Calls min<float>
```

The above function will work for any types that have their `operator<` (or `operator<=>`) defined, and also have a copy / move constructor.
Just like with abstract objects, the compiler automatically generates a copy of the function for each `class` or `struct` that it is called for.

The algorithm library

`for_each`

Let's consider our `for_each` example from earlier (during inheritence). We now have the ability to implement it in a better way using template functions.

Example: `for_each` function reimplemented.

```
// Original function (from before)
void for_each(AbstractIterator& start, AbstractIterator& end, int (*f)(int))
{
    while (start != end) {
        f(*start);
        ++start;
    }
}

// Reimplemented version (using template functions)
template <typename Iter, typename Fn> void for_each(Iter start, Iter end, Fn
f) {
    while (start != end) {
        f(*start);
        ++start;
    }
}

// Client code
void print(int n) { cout << n << endl; }
int a[5] = {1, 2, 3, 4, 5};
```

```
for_each(a, a + 5, print);
// Prints out 1 2 3 4 5 (in separate lines)
```

The second function does not require creating and inheriting from an `AbstractIterator` type, and works with the iterators provided for the inbuilt objects and structures as well as pointers. It works as long as `Iter` has `++`, `!=`, and `*` operators defined, and the `*` operator's return type matches `Fn`'s argumenttype.

The `for_each` function has already been implemented and is part of the base language.

It can be found in the `<algorithm>` library. The library also has many other useful functions. Some other functions are described below.

`find`

Finds an item in between 2 iterators.

```
template <typename Iter, typename T> Iter find(Iter start, Iter finish, const
T& val) {
    // Searches in [Start, Finish) for an Iter that when dereferenced == val.
    // Returns either an iterator to the first that matched, or finish, if
not found
}
```

Allows for easily searching through any iterable for a specific value. Works as long as `Iter` has `++`, `!=`, and `*` operators defined, and the `*` operator's return type it `T`.

`count`

Similar to `find`, but rather than finding the first occurrence, returns the number of occurrences of the value.

```
template <typename Iter, typename T> int count (Iter start, Iter finish,
const T& val) {
    // Searches in [Start, Finish) for Iters that when dereferenced == val.
    // Increments the return value each time such a value is found
}
```

`count_if`

Similar to `count`, but rather than comparing values, `count_if` applies a function (that returns a boolean) to the value in each location of the container, and counts the number of such locations cause the function to evaluate to `true`.

```
template <typename Iter, typename F> int count_if(Iter start, Iter finish, Fn
f) {
    // Searches in [Start, Finish) for Iters that when dereferenced and
passed into f, return true.
    // Increments the return value each time such a value is found
}
```

`copy`

Copies values from one container to another.

```
template <typename InIter, typename OutIter> OutIter copy(InIter start,
InIter finish, OutIter result) {
    // Copies the values from [Start, Finish) and increments the result each
time
    // Returns the last location of result
}

// Example client code
vector v{1, 2, 3, 4, 5, 6, 7};
vector<int>w(4); // w = {0, 0, 0, 0}
copy(v.begin() + 1, v.begin() + 5, w.begin());
// w = {2, 3, 4, 5}
```

The contents of the original container are not modified, and the copy container is used wherever there are objects.

`copy` requires that both `InIter` and `OutIter` have all the iterator operators defined, both are iterators of containers of the same type (or the destination type can be constructed from the source type), and that if the iterable stores objects, they have a copy contructor. Furthermore, `copy` assumes that the destination container has enough space to copy and store the values. It is up to the user to verify that this is in fact the case.

`transform`

This is the C++ equivalent of the `map` function from Python or Racket.

```
template <typename InIter, typename OutIter, typename Fn> OutIter
transform(InIter start, InIter finish, OutIter result, Fn f) {
    while (start != finish) {
        *result = f(*start);
        ++start; ++result;
    }
}
```

```
// Client code
int add1(int n) { return n + 1; }
vector<int> v{2, 3, 5, 7, 11};
vector<int> w(v.size());
transform(v.begin(), v.end(), w.begin(), add1);
// w = {3, 4, 6, 8, 12}
```

`transform` has almost identical requirements to `copy`, with the only difference being that there is no constructor requirements, and the `Fn` should take one input value of the same type as the source container, and the destination container be of the same type as (or implicitly constructable from) the return value of `Fn`.

Functors

In all the previous examples, we saw `Iter` requires `++`, `!=`, `*`.

For `Fn`, the only requirements are that the argument and return types must match and that `f(*start)` must be valid syntax.

While the second requirement is usually only valis for functions, we can also overload the `operator()` of an object, which makes a function object, of `functor`.

Example: A functor.

```
class Plus {
    int m;
    public:
        Plus(int m): m{m} {}
        int operator() (int n) { return n + m; }
};

// Client code
Plus p{5};
cout << p(10) << endl; // 15

vector<int> v{2, 3, 5, 7, 11};
vector<int> w(v.size());
transform(v.begin(), v.end(), w.begin(), p);
// w = {7, 8, 10, 12, 16}
```

While this just seems like a more complicated way to define something that behaves like a function, functors also have the ability to store (and modify) their state. Furthermore, a user can instantiate multiple objects with different values to create functors that behave differently.

Example: Functor that stores and uses state.

```
class IncreasingPlus {
    int m = 0;
    public:
        int operator() (int n) { n + (m++); }
};

// Client code
IncreasingPlus ip;
cout << ip(5) << ip(5) << ip(5) << endl; // 5, 6, 7

vector v{2, 3, 5, 7, 11};
vector w(v.size());
transform(v.begin(), v.end(), w.begin(), IncreasingPlus{});
// w = {2, 4, 7, 10, 15}
```

Lambdas

Consider that we have a `vector` of `int`s, and want to find out how many are even?
We can use `count_if`

```
bool even (int n) { return n % 2 == 0; }
vector<int> v{...};
int x = count_if(v.begin(), v.end(), even);
```

While this works, it is pointless to define a function if we are only going to use it in one place. It makes more sense to define a function for one-time use.
In CS135, we would use a *lambda* instead of a full function. We can do the same here.

```
vector<int> v{...};
int x = count_if(v.begin(), v.end(), [](int n) { return n % 2 == 0; });
```

This syntax uses less space than defining a full function. Notice that there is no mention of the return value, and it is automatically deduced. The general syntax of a *lambda* is as follows:

```
[](<Input values>) { Code }
```

Notice that unlike *lambda*s in other languages like Python or Racket, C++ *lambdas* can have multiple lines of code in them, and also explicitly need a `return` statement.

Advanced Iterator Uses

C++ Iterators are a very powerful concept, and they can be applied beyond just the notion of containers. Many such functions are available in the `<iterator>` module.

`ostream_iterator`

The `ostream_iterator` acts as a container for data, which takes all the values sent to it and prints them to `stdout`.

Example: Printing out values using an `ostream_iterator`.

```
vector<int> v{1, 2, 3, 4, 5};
ostream_iterator<int>out{cout, ", "}; // ostream and delimiter
copy(v.begin(), v.end(), out); // Prints 1, 2, 3, 4, 5
```

This is a much simpler method of printing out the contents of a container rather than iterating through it and calling `cout` for each one.

`back_inserter`

Recall the `copy` requires that there is enough space in the destination container to copy the data into. With `back_inderter`, it is possible to automatically expand the destination container while copying data over into it.

Example: Using `back_inserter`.

```
vector v{1, 2, 3};
vector w{4, 5, 6};
copy(v.begin(), v.end(), back_inserter(w));
// w = {4, 5, 6, 1, 2, 3}
```

When using the `back_inserter` iterators, assignment automatically calls the `push_back` function each time, which allocates space (if necessary) and copies the contents over.

Advice: Use functions from the `<algorithm>` library whenever possible, since it can greatly simplify code and reduce bugs.

Ranges

Recall the `vector<T>::erase` function. It takes in am iterator, erases the value at the iterator's location, shaifs down (moves all the values after the deleted value one slot to fill up the freed space), and returns an iterator to the new element in that location. All of this happens in $O(n)$ time.

This is fine if we are only erasing one element. However, what if we wanted to erase `k` elements from a container?

If we were to use erase in a loop, it would run in $O(n * k)$ time. This is quite inefficient, and a better solution is necessary for sunc cases.

Instead, we can use the ranged version of erase.

Example: Ranged version of the erase function.

```
// Client code for ranged erase
erase(Iter start, Iter finish);
// Erases contents of container from [Start, Finish)
// Only shifts values over once, regardless of the number of items deleted.
```

A range is a collection of 2 iterators.

Consider that we have a vector of integers, and we want take only the odd numbers and square them.

Example: Filter odd integers and square them (using lambdas and <algorithm> functions).

```
auto odd = [](int n) { return n % 2 != 0; }
auto sqr = [](int n) { return n * n; }
vector<int> v{...};
vector<int> w, x;

copy_if(v.begin(), v.end(), back_inserter(w), odd);
transform(w.begin(), w.end(), back_insertr(x), sqr);
```

This implementation works fine, but it has some problems:

1. Functions are on separate lines rather than composed.
2. w is only used for temporary storage.

These can be fixed in the following ways:

1. Fixed if copy_if and transform return ranges instead of ending iterator.
2. Solved using lazy evaluation (values are only calculated when they are accessed).

Example: Filter odd integers and square them (using ranges).

```
#include <range>
vector<int> v{1, 2, 3, 4, 5};

auto X = ranges::views::transform(ranges::views::filter(v.odd), sqr);
// X will apply filter and then sqr when you iterate.
```

These also have versions that take a function and return a function object to take the range.

Example: Filter odd integers and square them (using range functions).

```
auto f = filter(add);
auto g = transform(sqr);

auto X = g(f(R)); // Where R is a range
```

We can actually simplify this even further.
`operator|` is defined such that `R | f == f(R)`

Example: Filter odd integers and square them (using `operator|`).

```
auto X = v | filter(odd) | transform(sqr);
```

> End of CS 246 course content.

Conclusion

What is CS 246 about?
This is **not** a course about C++.

Low level thinking: What does the compiler do? Semantics of copy/move
High level thinking: Simplicity vs abstraction, design principles

This course is about adding a new programming paradigm to your toolkit.

Every programming paradigm (so far, we know about functional, imperative, and object-oriented) has it's own pros and cons.

Additions

Exam review resources are **A WORK IN PROGESS**. Please check the URL provided in page 2 for the newest version.

- Useful commands
- Code Samples
- Course Content Review
- Practice Questions (with solutions)

Course Content Review

Streams

- **Streams** are a form of **abstraction**, and provide an *interface* to perform different functionality related to input/output.
- There are two basic types of streams:
    1. `istream` (input stream): provides `>>` as an interface to "read from"
    2. `ostream` (output stream): provides `<<` as an interface to "write to"
- These basic streams are abstracted away to provide various other streams for different use cases:
    1. `cin` (`istream`) and `cout` (`ostream`) for reading from and writing to `stdin` and `stdout` respectively. There is also `cerr` which functions like `cin` but writes to `stderr`. Found in `<iostream>` library.
    2. `ifstream` (`istream`) and `ofstream` (`ostream`) for reading from and writing to a file. Found in `<fstream>` library.
    3. `istringstream` (`istream`) and `ostringstream` (`ostream`) for reading from and writing to a string as though it is a file or IO stream. There is also `stringstream` that does both. Found in the `<stringstream>` library.

References

- A pointer-like type that is new to C++.
- Allows to alias (reference) one variable using another, such that changes to one will be reflected in the other. Can be passed into functions to create parameters that modify the original value.
- They behave like constant pointers with automatic dereferencing.
- The method of implementation depends on the compiler and optimization level. The reference is either replaced with the variable it is referencing at compile time, or a pointer is used which is automatically dereferenced.
- There are 2 types of references:
    - Lvalue references, for values that always have a defined region in memory to be stored in.
      Syntax: `int& a = b;` (provided `b` is an `int`).
    - Rvalue references, for values that are not necessarily stored in a memory region (temporary values).
      Syntax: `int&& x = 5;` (`5` here is not a variable and thus isn't stored in a defined memory region).
- References have some limitations - the following can not be done with refreences:
    - Can not be left uninitialized (no `NULL` reference).
    - Can not assign a temporary value (Rvalue) to an Lvalue reference.
    - Lvalue references can only be assigned to variables. NOTHING else.
      `int& z = x + y;` or `int& z = f();` are invalid.
    - Reference to a pointer is allowed, but pointer to a reference is NOT allowed.
    - Can not create a reference of a reference.
    - Can not create an array of references.

C++ Dynamic Memory

- Similar to how `C` uses `malloc` and `free` for allocating and freeing heap memory, `C++` uses `new` and `delete`.
    - `<Type>* p = new <Type>;` allocates space for a new object of type `<Type>` to be stored in the heap.
    - `delete p;` frees the space pointed to by `p`. Here, `p` must be a pointer to heap memory.
- `new` and `delete` are type-safe, and ensure that the allocated memory is used by the correct type.
- Memory that has been allocated with `malloc` and memory that has been allocated with `new` are incompatible with each others' freeing functions and `free` and `delete` need to be used respectively to free the memory.
- C++ does not have an equivalent to `realloc`, and thus `new` and `free` need to be used.
- For allocating arrays, the commands `new[]` and `delete[]` are used. They function similarly to `new` and `delete`, but allocate space for multiple objects.
    - `<Type>* arr = new <Type>[k]`, where `k` is an integer, allocates an array of `<Type>`s of size `k`.
    - `delete[] arr` deletes `arr`, provided that it was allocated with `new[]`.
    - Memory allocated with `new` must be freed with `delete` and memory allocated with `new[]` must be freed with `delete[]`.
- `new` and `new[]` can also be used to allocate space for custom `class`es and `struct`s.

Passing / Returning Values

- There are 4 main ways to pass data into a function (consider `struct ReallyBig {...};`):
    - **Pass by value**: Very simple but extremely slow since all data needs to be copied.
      Useful for small arguments, or if a temporary version of the variable is needed.
      `void f(ReallyBig rb);`
    - **Pass by pointer**: Fast, but can be confusing to use, and changes are reflected.
      Useful in cases where `nullptr` might need to be passed in.
      `void g(ReallyBig* rb);`
    - **Pass by reference**: Fast (similar to pointer), changes are reflected.
      Usually preferred over pass by pointer, but can not pass `nullptr`.
      `void h(ReallyBig& rb);`
    - **Pass by const reference**: Fast, no copies, easy to reason, also supports Lvalues.
      Go to method if the variable does not need to be changed.
      `void i(const ReallyBig& rb);`
- There are n different ways to return data from a function:
    - **Return by value**: The value is copied from the function's stack frame to the caller's stack frame, and it is thus slow.
      Recommended in most scenarios since it is the easiest to reason about.
    - **Return by pointer**: Much faster since data doesn't need to be copied, however it only works with heap-allocated memory, and the caller has to free the memory afterwards.
    - Return by reference is not possible, since it results in a dangling reference, as the original variable that is being referenced will be deleted when the function finishes execution.

Objects

- Just like in C, C++ has `struct`s. These can store different values in their fields.

Ctors, Dtors, Big 5

UML

Iterator Pattern

Decorator Pattern

Observer Pattern

Factory Method Pattern

Template Method Pattern

Exception Handling

Smart Pointers

MVC

Casting