**Beginning C++ Programming by Frank J. Mitropoulos**My Course Notes

Troubleshooting

* CMake/NMake errors:
* Ensure that your antivirus isn’t blocking the build process by quarantining files.
* Restart computer.
* “Error: could not load cache”:
* Tools > CMake > Reload CMake Project.

Introduction

* There are multiple versions of C++: C++98, C++03, **C++11**, **C++14**, and **C++17**. The digits represent the year that version was released. The former two versions are referred to as classic C++, while the latter three are referred to as modern C++.
* C++98 was the first official standard. C++11 added many new features to the language. The other versions mainly corrected issues with the language or simplified pre-existing features.
* The process to create run an application is as follows:
* The developer inputs C++ code into header (.h) and source (.cpp) files in an IDE.
* The pre-processor looks for pre-processor directives such as *#include* and processes them.
* The compiler converts the high-level C++ code into low-level machine/binary/object code (.obj).
* The linker combines our code with other libraries and outputs an executable (.exe).
* The course will use CodeLite since it’s free, but the instructor stated that CLion by JetBrains is his IDE of choice. I have opted to use CLion as I have access to it and think highly of JetBrains.

Installation and Setup

* You can execute C++ code through an IDE, the CLI (command line interface), or via a website such as ‘repl.it’.

Curriculum Overview

* No notes taken.

Getting Started

* Code completion aids the developer by predicting what they will input and suggesting it to save time.
* Pre-processor directives don’t end in a semi-colon.
* *cout* is tied to the console and is used to output data.
* *<<* is the insertion operator that outputs the following data.
* Text between quotation marks represents a string literal: e.g. “Hello world!”.
* Statements end in a semicolon.
* *‘return 0;*’ in *main()* to indicate that there weren’t any problems.
* *cin* is also tied to the console and is used to input data from the user.
* *>>* is the extraction operator that stores the input.
* **Variables** store data.
* To declare a variable state its type and give it a name, e.g. ‘*int favourite\_number;*’.
* *#include <iostream>* includes the input/output library where *cout* and *cin* are defined.
* *endl* prints a new line and flushes the buffer.
* To build means to compile it and link it. This results in object files. The build process saves time by only building the files that it has to.
* The clean process removes the object files, but then you must build your program again.
* **Compiler errors** occur when the code doesn’t follow programming rules. It does this by identifying syntax and semantic errors.
* **Syntax error** refers to when the structure of the code is incorrect, e.g. ‘*cout << “Errors << endl*’ in this case the trailing quotation character is missing.
* **Semantic error** refers to when the structure is correct, but the code is undefined, e.g. ‘*int a = b + c’* if *a* and *b* are *int*s and *c* is a person then it may not make sense to add them.
* Making one error will lead the compiler to detect many errors. So fixing one error will resolve many compiler errors.
* **Compiler warnings** occur when code can be compiled, but has potential issues, e.g. printing an uninitialized variable ‘*int data; cout << data;*’.
* Both warnings and errors should be avoided whenever possible.
* **Linker errors** occur when libraries or object files are missing.
* **Runtime errors** occur when the program is running, e.g. dividing by zero, file not found, out of memory, etc. These can crash the program. To crash means that the program ended abruptly. Exception handling is used to deal with runtime errors.
* **Logic errors** occur when the code is technically correct, but the logic behind it is incorrect thus allowing the program to do something it shouldn’t do.

Structure of a C++ Program

* **Keywords** are reserved terms that hold special meaning in programming languages. Their meaning can’t be redefined in any way, e.g. *return, int,* etc. C++ has around 90.
* **Identifiers** are names given by the programmer, e.g. *main, include, my\_variable, cout*, etc.
* **Operators**accept one or more operands and perform an action with them, e.g. *+, -, <<, %, /, ^, &&, ::,* etc.
* **Pre-processor directives**tell the pre-processor program what to do, e.g. *#include <iostream>* tells the pre-processor to place the contents of that source file in its place. It also replaces the comments with a space.
* **Comments** allow the programmer to describe meaning or explain themselves next to the code. *//* is used to place a single-line comment, everything on that line is ignored by the compiler. */\* … \*/* is used to place a multi-line comment, everything in between is ignored.
* Ideally code is self-documenting and easy to read. Avoid unnecessary comments as it makes the code harder to read.
* All C++ programs must have one *main* function. It’s where execution begins in the program. Returning 0 indicates that the program terminated successfully.
* There are two acceptable function signatures for *main*:

|  |  |
| --- | --- |
| int main() {  return 0;  }  Program.exe | int main(int argc, char \*argv[]) {  return 0;  }  Program.exe argument1, argument2 |

The first signature is for when the program doesn’t accept any arguments, the second signature is for when the program does.

* *argc* counts the number of arguments provided, *argv* (argument vector) contains the value of each argument. These can be provided from the command-line. A vector refers to data in a one dimensional array.
* *main* is a function that is called by the operating system. A **function** is a name that refers to a block of code.
* **Namespaces** allow grouping code to avoid naming conflicts.
* Example: A library imported from a company might have their own definition of *cout*, however with the use of namespaces you’ll be able to specify exactly which one you want to use: *std::cout* or *company::cout*.
* *std* is the name of the standard namespace which contains code from the STL.
* If you don’t foresee any naming conflicts, then you can use a **usingdirective**, e.g. ‘*using namespace std;*’ now you won’t have to type ‘*std::*’ before using identifiers declared in the *std* namespace.
* This isn’t recommended in large programs due to the potential increase in naming conflicts. Instead use a **using declaration** in which you specify the identifier directly, e.g. *using std::cout*.
* Using directives and declarations can be defined at global scope or block scope.
* *cout, cin, cerr,* and *clog* are objects representing streams. A stream is a sequence of a data type, e.g. a string is a sequence of characters.
* To print a new line either insert *endl* or *“\n”* escape sequence. The *endl* stream manipulator also flushes the buffer.
* *cout, cin, cerr,* and *clog* can be chained so that multiple data can be input or extracted.
* White space is ignored by *cin*’s extraction operator. The data input must match the type of variable the data is being stored in.
* *double* contains a real number, such as 2.5, 5, -1, etc.
* Data entered into the command line is stored in a buffer. Data exists in the buffer until it is read so it may be read unpredictably unless the buffer is periodically cleared.

Variables and Constants

* RAM is a contiguous block of memory that stores program instructions and data. Each memory cell has an associated location to reference it. Low level languages work directly with these locations and move data around. Higher level languages let you use variables to associate useful names to these locations.
* **Variable**is an abstraction for a memory location. They have a type (int, string, person, etc), name (age, name, bob, etc), and content (21, “bob”, etc). They must be declared before the are used, e.g. ‘*int age; age = 12*’, or *int age = 12*. A variable’s content can vary/change.
* The type tells the compiler what data can be stored in the variable. C++ is **statically typed** meaning that the type is checked at compile time. Some languages are **dynamically typed** meaning that the type is checked at runtime.
* Variables names can contain letters, numbers, and underscores. Can’t begin with numbers. Cannot use C++ keywords. Cannot redeclare a name in the same scope. C++ is case sensitive.
* Be consistent with naming conventions, e.g. camelCase vs. my\_variable. Use meaningful names that are not too short, not too long. Never use variables before initialising them. Declare variables close to when you actually need them in code to make it clear.
* There are three ways to declare and initialize variables. C-like initialisation: *int age = 21*. Constructor initialisation: *int age (21)*. C++11 list initialisation: *int age {21}*.
* Using uninitialised variables is dangerous because the variable just has the value/content that was already at the given address. There’s no way to predict what a previous program set it too.
* **Local variables** are variables defined within a code block as their scope/visibility is limited to the statements within that block.
* **Global variables** are variables defined outside of any code block. They are called this because they have global scope/visibility and can be accessed from any part of the program.
* Global variables are automatically initialised to zero, unlike local variables.
* Global variables should be avoided as they make code difficult to debug.
* The compiler first looks locally to find the variable, it then goes up in scope until it ultimately checks the global scope for the variable.
* Variables in different scopes can have the same identifier. To specify a variable from global scope use :: without specifying a namespace.
* Primitive/fundamental data types are defined by the language itself.
* The size and precision of these data types depends on the operating system and compiler.
* The *climits* include contains information about the size and precision of various datatypes for the given compiler.
* Type sizes are expressed in bits, the more bits the more values can be stored and the more storage is required.
* One bit can represent 2 values, 8 bits can represent 28 values, 16 bits > 216, etc.
* 8 bits is 1 byte, 16 bits is 2 bytes, etc.
* Character types are used to represent single characters, e.g. ‘A’, ‘X’, ‘@’, etc. Wider types are used to represent wide character sets. Single quotes are used for characters.
* *char* is one byte, c*har16\_t­* is at least 2 bytes, *char32\_t* is at least 4 bytes, and *wchar\_t* can represent the largest available character set.
* Integer types are used to represent whole numbers, e.g. 5, 112, -23, 0, etc. There are signed and unsigned versions. Integers are *signed* by default so you don’t need to type that. *short* and *long* are *int* by default so you don’t need to type that either.
* *short* and *int* are at least 2 bytes, *long* is at least 4 bytes, and *long long* is at least 8 bytes.
* Types: *short, int, long, long long, unsigned short, unsigned, unsigned long, unsigned long long.* Unsigned types move their negative range to their positive range, meaning you can’t represent negative numbers, and get double the positive range.
* Floating types are used to represent real numbers: whole and fractional, e.g. 5, -2.5, 3.14159, etc.
* *float* is typically accurate to 7 decimal digits / 4 bytes, *double* to 15 / 8 bytes, and *long double* to 19 / 12 bytes.
* Boolean type represents true and false, zero is false, and none-zero is true. *bool* is usually 1 byte. *true* and *false* are generally used in place of numbers.
* You can use single quotes to split a large number, e.g. *long value = 123’456’789*. The quotes can be anywhere inside the number and not outside or adjacent to another quote.
* If you go over the range of a type, e.g. *short value = 70’000* this results in an **overflow**. The resulting value is the overflow amount. You can overflow by going over the maximum or minimum values for the data type.
* Storing a floating type in an integral type results in **truncation**. Only the integer is kept, the decimals are truncated. In effect, it always rounds down.
* A narrowing conversion is when a large type is stored in a smaller type, e.g. *long* to *short*, or *float* to *int*. The list initialiser syntax prevents this at compile time. It also prevents overflow and truncation errors. The opposite of a narrowing conversion, is widening.
* You can express a literal using scientific notation, e.g. *2.7e2* is 2.7x102.
* The *sizeof* operator determines the size, in bytes, of a variable/type/array/object/etc. Examples: *sizeof(int), sizeof(double), sizeof(my\_variable),* and *sizeof my\_variable*. It gets this information from *<climits>* and *<cfloat>*.
* **Constants** are mostly identical to variables except that their value/content can’t be changed once it’s set.
* Constants make it clear to programmers that the content should never be changed, e.g. months in a year are always 12. Reassigning a declared constant is a compile time error.
* There are many types of constants: literal constants, declared constants via *const*, contact expressions via *constexpr*, enumerated constants via *enum*, and defined constants via *#define*.
* Literal constants: 5 (int), 6U (uint), 20L (long), 55LL (long long), -3F (float), 5.5 (double), 10.0L (long double), ‘Z’ (character), “hello” (string), etc.
* Declared constants: *const double pi {3.14159}, const int months\_in\_year {12},* etc.
* Defined constants were used in older code and should be avoided now as it doesn’t support type checking and makes it difficult to debug, e.g. *#define pi 3.14*. The pre-processor replaces any use of the identifier pi with 3.14.
* **Pseudocode** breaks down the algorithm/steps in easy to read English rather than actual code.
* **Escape sequences** are special characters that perform a unique action when output to the console, e.g. *\n* prints a new line, *\t* prints a tab, *\\* prints a \, *\”* prints a “, \’prints a ‘, etc. They are often found in string literals.

Arrays and Vectors

* **Compound data types** are types that are made up of other primitive types.
* **Arrays** contain data in which element is of the same type, are fixed size, and stored contiguously in memory. They are also known as **raw arrays** or **built-in arrays**.
* It’s convenient because a set of data could be contained within a single variable name.
* Once the array size is set, it can’t be changed.
* First index is 0 (zero based index), last index is array\_size – 1. You must ensure that you don’t access an element that’s out of bounds. The program has undefined behaviour and can crash.
* Arrays are normally looped through.
* Array declaration syntax: *int scores [3]* or ‘*const int num = 3; int scores [num]*’ – stores three integers. Note that the size must be defined via a constant.
* There are four ways to initialise arrays:

1) *int scores [5] {}* – all set to 0.

2) *int scores* *[5] {3, 4}* – first two set, rest set to 0.

3) *int scores [5] {5, 3, 4, 2, 1}* – all elements set.

4) *int scores [] {4, 5, 5, 2, 3}* – size automatically calculated.

* Each element can be accessed directly through its index, *e.g. scores[0]* gets first element, *scores[2]* gets third element, etc. *[]* is called the subscript operator.
* The name of the array represents the memory address of the first element. The [index] represents the offset from the first element.
* If you don’t initialise an array and make use of it, you will get undefined output.
* **Multidimensional** arrays represent tables/spreadsheets. To declare them use two square brackets to state number of rows, then columns, etc: e.g. *int movie\_ratings [3][4]*. You access the elements in the same way: e.g.*movie\_ratings[1][2]* returns the integer in the first row in the third column.
* These types of arrays aren’t used frequently in modern C++ as they are error prone. Instead the preference is vectors.
* A **vector** is a dynamic array. It can be resized as required. It’s ideal when you don’t know ahead of time how many elements will be contained within the array.
* The Standard Template Library (STL) has many containers, algorithms, functions, etc. that allow the programmer to focus on the task rather than reinventing code. Vector is defined within the STL. To use it *#include <vector>*.
* Vectors work similarly to built-in arrays, but can provide bounds checking, and has many methods such as sort, reverse, find, etc.
* To declare a *vector* object: *vector <char> vowels*, or *vector <int> test\_scores*, etc. You must specify the element type in the angles brackets as *vector* is a template class.
* To initialise a *vector* object there are multiple ways:

1. *vector <int> scores* – Constructor initialisation. Empty vector, no elements.
2. *vector <int> scores (10)* – Constructor initialisation. All elements set to 0.
3. *vector <int> scores (10, 20)* – Constructor initialisation. 10 elements set to 20.
4. *vector <int> scores {5, 2, 4}* – List initialisation. 3 elements set.

* Vectors are based on built-in arrays so the same logic applies regarding direct indexed access to elements, contiguous in memory, zero based index, etc.
* You can access elements using the subscript operator, but no bounds checking will be done. Use the *at()* method: e.g. *scores.at(1)* vs. *scores[1]*. If you go out of bounds, then the method will throw an exception to indicate this.
* You can add a new element: *scores.push\_back(5)*. This will add 5 to the end of the vector.
* The vector automatically resizes if there isn’t enough space in the vector.
* Call the *size()* method to determine the current size of the vector.
* 2D vector: *vector <vector<int>> ratings {…}*. Element access: *ratings[1][2]* or *ratings.at(1).at(2)* – the first *at()* returns the row, the second *at()* returns a single element.

Statements and Operators

* An **expression** computes a value from a number of operands, e.g. *34, my\_variable, 4 + 5, 2 \* 3, a > b, a = 5,* etc.
* A **statement** is a complete line of code that performs an action. It’s usually terminated with a semi-colon, and usually contains expressions, e.g. *int x; age = 21; 3 + 8; x = 2 \* 3; if (a > b) cout << “a is greater”;* null statement, etc.
* A null statement is just a semi-colon.
* There are unary, binary, and ternary operators that work on 1, 2, or 3 operands.
* Operators can be grouped as: assignment, arithmetic, increment/decrement, relational, logical, member access, etc.
* You can assign multiple variables in a row, the process occurs right to left: *e.g. var1 = var2 = 100*.
* When doing division keep in mind that for an integer variable, the fractional part is dropped.
* The modulus or remainder operation (%) only works with integers.
* The increment and decrement operators just add or minus 1 respectively from the variable, e.g. ‘*int a = 5; a++*’ *a* is now 6. It can be used with integer/floating/pointer types. Don’t overuse the operator and don’t use it more than once within the same statement as it is undefined.
* The increment operator can be applied as prefixes or postfixes. The difference is that prefix first increments, then returns the new value. Postfix returns the value, then increments. The same logic applies to the decrement operator.
* Operations occur on the same type of operands. If one is different then the compiler will attempt to convert it. With differing primitive types in most operations, the smaller type is converted to a larger type, e.g. *int + double = double* as this is a widening conversion and retains data. However, with assignment a narrowing conversion can occur, e.g. *int val = 100.2*, this is *double* to *int*.
* A **type** **cast** is when one type is changed to another. The examples above were implicit casts since the compiler does them. You can tell the compiler what type to convert to using an explicit cast, e.g. *double value = static\_cast<double>(intVariable)*.
* The C-style cast equivalent would have been *double value = (double) intVariable*, however this should be avoided as unlike *static­\_cast*, the C-style cast doesn’t check to see if it’s safe to convert a type. *static\_cast* will error at compile time if the operation is invalid.
* Equality operators include == and !=. They evaluate an expression to *true* (1) or *false* (0). They’re commonly used in control flow statements.
* Boolean expressions are output as 1 or 0 to the console. To change this and output true and false instead, use the *std::boolalpha* stream manipulator.
* Boolean operations can be strange with floating types, i.e. 12.0 == 11.99999999999999999999 may return true. This is because of how floating types are represented in the computer. For this level of precision, you will need to use a specialised library.
* Relational operators include >, >=, <, <=, and <=>. They return *true* if the relationship is true, and *false* if it isn’t, e.g. *5 > 10* returns *false*, *10 >= 10* returns *true*, etc.
* Logical operators include !, &&, and ||. They work with other Boolean types to return a result. ! flips the state, && is only *true* if both operands are, || is true if either operand is. Alternatively, you can use the operators *not, and, or*, but this is not common practice.
* **Short-circuit evaluation** means that compiler stops evaluating the expression as soon as it becomes impossible for it to be anything else: *e.g.* ‘*expr1 && expr2 && expr3*’ stops evaluating the whole expression if a sub-expression is *false*, and ‘*expr1 || expr2 || expr3*’ stops evaluating the whole expression if a sub-expression is *true*.
* **Compound assignment operators** simplify repetitive assignment operations, e.g. *a = a + 5* becomes *a += 5*, *a = a % 3* becomes *a %= 3*, etc. It works with arithmetic and bitwise operators.
* **Operator precedence** refers to the order in which sequential operators are processed. You can use parenthesis in complex expressions to prioritise certain sub-expressions. This works because parenthesis have a high precedence.

Controlling Program Flow

* ***if* statements** allow code to run if a certain condition is met. The expression provided in the bracket must evaluate to *true*, e.g. *if (num < 10) ++num;*.
* To execute more than one statement use a code block, e.g. *if (num < 10) { ++num; cout << “num incrememented.”; }*. The code block has its own scope and can declare other variables local to that block. The variables declared within a block are only visible within that block.
* ***if-else* statement** are similar to an *if* statement, the difference is that if the expression evalues to *false* then code in the *else* block is executed, e.g. *if (num < 10) cout << “num < 10”; else cout << “num >= 10”;*.
* ***if-else if* statements** can check for various conditions from which only the first to evaluate to *true* is executed, the rest are skipped. You can have as many *else if* statements after the *if* statement as required. You can place an *else* block at the end to run some code no matter what. Example:

if (expr1) {

…

} else if (expr2) {

…

} else if (expr3) {

…

} else {

…

}

* **Nested ­*if* statements**check a condition only if another condition has already evaluated to *true*, e.g. *if (expr1) { … if (expr2) {… } … }*.
* **Dangling else** refers to nested *if* statements followed by an *else*block. It’s not clear which *if* statement the *else* block belongs to. In C++ it belongs to the closest *if* statement. It’s recommended to make it explicit by using code blocks. Examples of both:

|  |  |
| --- | --- |
| if (expr1) // Bad: implicit.  if(expr2)  statement1;  else  statement2; | if (expr1) { // Good: explicit.  if(expr2) {  statement1;  } else {  statement2;  }  } |

* ***switch* statements** are very similar to *if* statements, but they check against the value of one constant in a more streamlined syntax. The expression must evaluate to an integral type.
* The *case* statements must contain constant integral expressions. The *break* keyword instructs the compiler to exit the *switch* statement – similar to *if-else if ­*statements. Without the *break* keyword, multiple *case* statements can be run in succession – similar to separate *if* statements.
* The *default* block is similar to an *else* block.
* A *case* statement can have multiple statements without a block, but if a local variable needs to be declared then it must be done within a block.
* The syntax is as follows:

switch (integral\_type) {

case const\_expr1: statement1; break;  
 case const\_expr2: statement2; break;  
 …  
 default: statement\_default;   
}

* Enumeration types are a custom integral type that limit what the type can be assigned to. They also provide more context in the process.
* Example: *int day\_of\_week* could be assigned 1-7 for different days of the week. However, this is prone to error as 1-7 could mean anything and the programmer is not prevented from accidentally typing another number. A better solution is to use an enumerator:

|  |  |
| --- | --- |
| enum DayOfWeek { // Bad: Unscoped  Monday,  Tuesday,  Wednesday,  Thursday,  Friday,  Saturday,  Sunday  }  DayOfWeek day = Monday; | enum class DayOfWeek { // Good: Scoped  Monday,  Tuesday,  Wednesday,  Thursday,  Friday,  Saturday,  Sunday  }  DayOfWeek day = DayOfWeek::Monday; |

It’s good practice to use scoped *enum*s to prevent namespace pollution.

* It works by using integers behind the scenes, Monday is assigned the value 0 by default, Tuesday is 1, etc. You can specify your own values and even repeat them. Since the identifiers are basically constant integers, they can be used in *switch* statements too.
* The conditional operator is a shortcut for short *if-else* statements. It’s a ternary operator that returns one expression or the other, based on the condition. It’s of the form: *(cond\_expr) ? expr\_if\_true : expr\_if\_false*.
* **Looping/iterating** allows executing the same code block repetitively.
* ***for* loops** are used to loop a specific number of times, e.g. *for(int i {0}; i < 5; i++) { … }*.
* The semi-colons are required, but the expressions are all optional. There are three parts to the syntax: initialisation, condition, and increment. The loop initialises local variables, then checks to see if the Boolean condition is true, then executes the loop once. At the end of the loop it increments, then checks to see if the condition is still true before executing the code. It’s conventional to name the loop counter *i*.
* It’s best practice to write loops without missing or complicated expressions.
* **Range-based *for* loops** are used to loop through a collection, e.g. *for(int i : array) { … }*. Can use list initialisation to loop through a range, e.g. *for(int i : {1, 2, 3, 4, 5}) { … }*. You can also provide a string literal as an argument and the loop will go through each character.
* ***while* loops** are used to as long as an expression is true, e.g. *while(is\_true) { … }*. The condition is evaluating initially, and then at the end of every loop. A common mistake is an infinite loop in which the condition never becomes *false* and thus execution never breaks out of the loop.
* ***do-while* loops**are the same as *while* loops, but execute the body at least once, e.g. *do { … } while (expr);*.
* A code block isn’t required if the loop only has a single statement in the body. It’s good practice however to include code blocks regardless as making future changes to code is simpler.
* Variables declared in the loop are local to the loop.
* To loop through an array using a basic *for* loop, provide the loop counter as an argument to the subscript operator for the array.
* Rather than specifying the type yourself, you can make the compiler deduce the type by using the *auto* keyword, e.g. *auto number = 5* – *number* is an *int*.
* The ***continue*** **statement** skips the next iteration of the loop.
* The ***break*** **statement** exits the loop altogether. The *break* statement can be combined with an infinite loop to exit the loop once a condition is met.
* **Nested loops** allow you to loop within a loop, this can be useful when dealing with multidimensional arrays as the outer loop can go through the rows, and the inner loop goes through the columns.

Characters and Strings

* The ***cctype*** **library**, *#include <cctype>*, contains functions that test the nature of a character, or convert between uppercase and lowercase characters, e.g. *isalpha(), isalnum(), isdigit(), islower(), ispunct(), isupper(), isspace(), etc*. You provide a character and they return *true* or *false*. The conversion functions include *tolower()* and *toupper()*.
* **C-style strings** are an array of characters in memory. The last character is ‘\0’ – null character. That’s how you can tell when you have reached the end of a C-style string, because it is always null terminated. Thus C-style strings are also called null-terminated strings.
* A string literal returns a C-style string, the null terminator is put at the end automatically, e.g. “Hello world!” becomes “Hello world!\0”. The size of the string literal is always +1 the characters in the quotes.
* Similar to how *int data [3] {}* results in *0 0 0*, *char str [3] {}* results in *0 0 0*. However, integers for the *char* type are converted via the ASCII table. The character equivalent of *0* is *\0* – null character. So *char str [5] {“hi”}* is equivalent to “hi\0\0\0”.
* You can assign to a character array, instead you must use the *strcpy\_s()* function to do so, e.g. *char name [10] {}; strcpy\_s(name, “Timmy”);*. You can use the *strlen()* function to count the number of characters. *strcmp()* compares strings together – if it returns 0 then they are equivalent.
* The *cstring* library, *#include <cstring>*, contains many functions that work with C-style strings. The functions are used for copying, concatenation, comparison, searching, etc.
* The *cstdlib* library, *#include <cstdlib>*, contains many functions to covert C-style strings to integers, floats, long, etc.
* Can call *cin.getline()* as opposed to *cin >> var*. It will save the entire line rather than skip whitespace.
* *size\_t* is generally used in place of *int* or *unsigned* when dealing with the size of a collection. *size\_t* is automatically assigned to the correct *unsigned* type for the system, e.g. in my environment *size\_t* is *unsigned int*.
* *std::string* is the *vector* equivalent to *raw arrays*, but for C-style strings.
* It’s defined in the string library, e.g. *#include <string>*.
* *string* is a dynamic data structure that grows in accordance with the length of the new string.
* It supports operators for easy manipulation, e.g. assignment, arithmetic, compound assignment, and Boolean operators.
* It uses a variable to keep track of the string length, so it doesn’t need to be null-terminated.
* It supports bounds checking just like a vector.
* Can use list or constructor initialisation syntax to initialise a string, e.g. *string s1, string s2 {“Frank”}, string s3 {s2}, string s4 (“Frank”, 0, 3), string s5 (3, ‘X’),* etc.
* An expression can be concatenated as a string if at least one of the operands is a string, e.g. *string + string = string, string + char [] = string,* or *char [] + string = string*.
* You can use the subscript operator, or the *at()* method to access character elements in the *string*.
* *substr()* returns the substring from the given start index to the provided offset.
* *find()* returns the index of where the first instance of a substring is found. If the substring is not found then *std::npos* is returned.
* *rfind()* – reverse find, does the same, but starts from the end.
* *erase()* deletes a substring, while the *clear()* method deletes the entire string.
* *length()* returns the number of characters in the *string*.
* *cin >> stringVariable* extracts user input, but stops at whitespace. Call *cin.getline()* or *getline()* to extract more. These methods stop at the newline character ‘\n’.
* *insert()* inserts a substring within another string.
* *swap()* swaps the contents of two *string*s.

Functions

* Functions modularise code. The goal is to maximise code re-use and minimise code-duplication, to reduce errors and code bloat. Functions often accept arguments and return data.
* The caller doesn’t have to know how the function works to use it, this is referred to as abstraction.
* C++ has a math library called *cmath*. It contains many math related functions such as *sqrt()* or *pow()*. You simply provide arguments, and use the returned value without knowing how the calculation was done. In other words the implementation is abstracted away.
* You must define functions outside of other functions, example:

|  |  |
| --- | --- |
| int add\_numbers(int a, int b) {  if(a < 0 || b < 0) {  return 0;  } else {  return a + b;  }  } | int main() {  cout << add\_numbers(3, 5); // 8  } |

* The return type is defined first followed by the name of the function and its parameters. The name of a function should be a verb and has the same rules as declaring a variable. You can have zero or more parameters, separate them by a comma.
* The *return* statement is used to return data to the caller. The type returned must match the return type. You can have zero or more *return* statements based on the return type, but only one will be executed.
* A **procedure** is a function that returns nothing and has the return type ***void***. You can still use a *return* statement to end a procedure at a specific point.
* To use a random number generate, include the *cstdlib* and *ctime* libraries. Then call *srand(time(nullptr))* to seed the generator. Then call *rand()* to return a value between 0 and *RAND\_MAX*. To limit the range use the % and + operators, e.g. *rand() % 5 + 1* limits the output between 1 and 5 inclusive.
* The random number algorithm produces the same output given a certain initial value – the seed. Hence by using the current time it’s very difficult to predict what the output will be. This is called a pseudo-random number generator.
* Functions must either be defined before they are actually used, or you can use function prototypes and define the function after it is used. **Function prototypes**, or **forward declarations**, are when the function is declared in one place, but then defined in another. Example:

|  |  |
| --- | --- |
| #include <iostream>  #include <string>  using namespace std;  void outputText(string text) {  cout << text << endl;  }  int main() {  outputGreeting(“Hello, world!”);  } | #include <iostream>  #include <string>  using namespace std;  void outputTest(string);  int main() {  outputGreeting(“Hello, world!”);  }  void outputText(string text) {  cout << text << endl;  } |

* You can provide the parameter name in the declaration, but it’s ignored anyway.
* Function prototypes are usually stored in header (.h) files.
* Parameters refer to the data a function expects. Arguments are the actual data that is passed to the function. Example: *string name* could be a parameter, “bob” could be the actual argument. Arguments passed to a function must match in number, order and type.
* Data/variables passed to a function are **passed-by-value**, i.e. a copy is created. Changing the parameter won’t affect the original variable that was passed in.
* Default arguments can be supplied for parameters by initialising them within the parenthesis. This means that it’s optional for the caller to provide values for the default argument parameters. The default arguments can either be provided in the prototype or definition, but not both. It’s best practice to provide it in the prototype. Multiple parameters can have default values, but they must all be at the end of the parameter list.
* **Function overloading** allows defining multiple functions with the same name that work slightly differently based on the number or type of arguments provided. Example: can write one method to add integers, and another method with the same name to add doubles, etc.
* Overloaded functions can have different return types, but that’s not enough to differentiate them as the call will be ambiguous.
* If an overload doesn’t exist that exactly matches the argument type, then the compiler will allow one conversion to match it, e.g. *addDoubles(5, 10)* will cast *int* to *double* for each parameter, or *printString(“hello”)* will convert the C-style string into a C++ *string*.
* When raw arrays are passed into functions, only the address of the first element is copied rather than the entire array. So the changes made to the array can be seen in the original array.
* As a consequence, the function doesn’t know the size of the array. The size must be passed in separately.
* To change the original variable through a parameter, you can **pass-by-reference­**. This way the variable isn’t copied, instead the parameter is just another name for the original variable itself as it directly references it in memory. This is recommended when dealing with large objects or data structures such as a *vector* as the data won’t be copied.
* To create a reference variable use an ampersand, e.g. *int num = 5; int &ref = num;* - reading/writing either *num* or *ref* has the same effect.
* Normally variables are deleted from memory when they fall out of scope at the end of a code block. However, *static* local variables are initialised the first time a variable is called, then persist until the program closes. Example: *static int count = 0*.
* Best practice: It’s okay to place constants in global scope, but not variables.
* C++ uses static scoping. Some languages make use of dynamic scoping and they can be harder to follow.
* When functions are called they are pushed (added) onto a **function call stack**. When the execution reaches the end of a function code block, that function is popped off the stack (removed). Stacks are a LIFO – last in first out, data structure. Like a stack of plates.
* Local variables and parameters are allocated in stack memory. When the function is popped off the stack, everything in stack memory for that function is deleted which makes room for further calls.
* Stack size is limited and if too many variables are allocated this leads to a **stack overflow** which crashes the program. Infinite loops and infinite recursions commonly cause this.
* **Heap/freestore** is where dynamic memory is allocated.
* ***inline*** **functions** avoid function call overhead by inlining simple function calls into the calling function. However, compilers are advanced enough to do this without this. The downside is that it can lead to code bloat and increase the binary size.
* A **recursive function**is a function that calls itself. Many problems are better implemented with recursion, e.g. factorials, Fibonacci, fractals, binary search, search trees, etc.
* Recursions are a form of iteration, an algorithm can be implemented using both.
* Recursive functions have two components: the base case which decides when to stop the recursion, and the recursive case that calls the function again.

Pointers and References

* Pointers store the address of another variable or function. To use the data that the pointer is pointing to, you must know its type.
* To declare a pointer, append a \* after the type, e.g. ‘*int \*int\_ptr; double\* double\_ptr; char \*char\_ptr;*’. The asterisk can be next to the type or next to the name. The same is true in references for ampersands.
* By default, pointers aren’t initialised and contain garbage data. Use the ***nullptr*** keyword to initialise them to address 0. This indicates that the pointer doesn’t point to anything of interest.
* Using an ampersand outside of a declaration and before a regular variable name returns the address of a variable. Hence it is called the address operator, e.g. ‘*int age = 5; cout << “Address: “ << &age;*’.
* Using an asterisk before a pointer variable dereferences the pointer and returns the data that it points to. Hence it is called the dereferencing operator e.g. ‘*int \*age\_ptr = &age; cout << “Value: “ << \*age;*’.
* To dereference an object pointer and call its method the syntax is: *(\*object).method()*. The following syntax is short for this and is recommended: *object->method()*.
* You can have pointers to pointers, and get addresses of pointers, etc.
* Pointers can point to small or large objects, but all pointers occupy the same memory. In my environment its 4 bytes. The reason is because a pointer is just a number.
* A pointer of one type cannot point to a variable of another type.
* The ***new*** keyword allocates/uses memory on the heap returns its address, e.g. *int \*data {new int {5}}*.
* The ***delete*** keyword deallocates/frees memory from the heap when you’re done with it, e.g. *delete data*.
* You can use *new* and *delete* with arrays, e.g. ‘*int \*data {new int[3]{1, 2, 3}}; delete [] data;*’.
* An array name by itself returns the address of the first element, e.g. ‘*int data [] {1, 2, 3}; int \*ptr = data;*’. Dereferencing *ptr* will return the first element, to access the second element you must use **pointer arithmetic**, e.g. *\*ptr* returns 1, *\*(ptr+1)* moves to next address/element and returns 2, *ptr[2]* is the simplified syntax which returns 3.This is why arrays have a zero-based index.
* *\*(data+1)* is offset notation, *data[1]* is subscript notation. Both notations work with arrays and pointers.
* When pointers are incremented by 1 the address they jump to is the initial\_address + pointer\_type\_size, e.g. an *int* pointer pointing to address ABC2D0 will point to ABC2D4 when incremented by 1, ABC2D8 when incremented by 2, ABC2DC when incremented by 3, etc. The reverse is true for decrementing.
* Subtracting two pointers results in the number of elements between them.
* Pointers support equality operators and relational operators.
* A pointer declaration can use the *const* keyword twice: once to declare the underlying type as the constant and once to declare the pointer as a constant. Examples:

1. *int \*data* is a mutable pointer pointing to a mutable integer.
2. *const int \*data* is a mutable pointer pointing to a constant integer.
3. *int \* const data* is a constant pointer pointing to a mutable integer.
4. *const int \*const data* is a constant pointer to a constant integer.

Remember that the first *const* is before the type so it applies to the underlying type.

* Never return a pointer that points to a local variable in a function since that variable will be deleted from the stack when the function returns. This pointer is said to be a **dangling/wild/stray pointer** since the memory it points to is now invalid. This can also happen if two pointers point to the same address, but *delete* is called on one pointer while the other is used assuming it’s pointing to valid memory.
* Forgetting to *delete* memory allocated on the heap leads to **memory leaks** which is considered very bad. In other words, for every *new* there should be a corresponding *delete*. This can happen when a pointer falls out of scope in which case only the pointer is deleted, but the memory it points to is still in use but can’t be accessed anymore.
* **References** can be thought of as constant pointers that are automatically dereferences when used.
* They cannot be null, must be initialised upon declaration, and can’t refer to another variable afterwards.
* **l-values** refer to values that have names and are addressable. They are mutable if they aren’t marked *const*. Examples: *int x {100}, string name {“Bob”}, string &name\_ref = name, int ages [3] {},* etc. These identifiers are all l-values.
* **r-values** refer to values that aren’t l-values; they’re non-addressable and non-assignable. This includes literals and temporary variables (generated by expressions and function returns), e.g. *5, (random\_variable + 20), max(20, 30),* etc.

OOP - Classes and Objects

* **Procedural programming** contains a collection of functions to which data is passed and processed, i.e. what I’ve been doing so far. As these programs get larger they become difficult to understand, maintain, extend, debug and easier to break. This is because the relationship between all the functions is unclear.
* **Object oriented programming** contains classes and objects that model real-world entities and allow developers to think at a higher level of abstraction. Objects group data and operations that relate to that data which makes the relationship clear. Implementation specific data and logic can be hidden, this allows more abstraction and makes it easier to test, debug, maintain, and extend the program. The classes can easily be reused in other applications to speed up development and prevent reinventing the wheel.
* **Classes** are a blueprint from which objects are created. They contain attributes (data), and methods (functions/procedures). They can make data and methods private, and simultaneously provide a public interface. Examples: *Account, Employee, Image, std::vector, std::string*, etc.
* **Objects** are created from classes and represent a specific instance. Each object has its own unique identity in memory and operators independently of other instances. Many objects of the same type can be created, e.g. multiple *string*s, *vector*s, etc.
* To declare a class, use the *class* keyword followed by the class name, and then by a set of attributes and methods contained within a code block. **Attributes** define the state of the object, and methods define the behaviour of the object.
* Attributes and methods are also called class members.
* Classes can be declared within a function, but this is usually not recommended. You should generally declare classes within global scope so that every part of the program has access to it.
* Primitive attributes contain garbage data if they’re not initialised. You can directly initialise an attribute as you would with a regular local variable.
* Methods in a class definition are usually function prototypes. The code will compile even without definitions for a prototype, but this leads to a linker issue if they are called.
* Use the dot operator to access class members such as attributes and methods, e.g. *frank\_account.balance, frank\_account.deposit(100.00), my\_string.length(), (\*frank\_account\_ptr).deposit(100.00),* etc.
* Use the arrow / member-of-pointer operator to access class members for pointers, e.g. *frank\_account\_ptr->deposit(100.00), my\_string->length(), etc*.
* Methods accessing attributes within the same class do not need to use the dot/arrow operators and can just use the attributes directly as if they were local variables. This means you can write methods that use less parameters.
* **Access modifiers** determine the data hiding level for parts of a class.
* Each of the following modifiers is more restricting than the last:
* ***public*** members are accessible anywhere - within the class, friends, inheritance, or dot/arrow operators.
* ***protected*** members are accessible within the class, friends, or inheritance – derived classes.
* ***private*** members are only access within the class, or by friends of the class.
* To use an access modifier, state the level followed by a colon. Members beyond that point will have the chosen access level. Attempting to access a member without the correct access level will cause a compiler error.
* **Encapsulation** enables data hiding and protecting the design of a class. It’s important to set the correct access level as it limits what could change the state/attributes of an object. Attributes are commonly marked *private* while methods are commonly marked *private/public.* This makes it easier to debug any errors since the attribute can only be modified from certain methods. It also allows validation as the getter/setter methods can check/modify the input before assigning it to the attribute. It also decouples the getter/setter identifier from the attribute identifier.
* Methods can be defined within the class declaration or outside of it. Defining it within the declaration makes the method *inline* implicitly. This is okay for small methods, but not recommended otherwise.
* To define a method outside a class you define it as if it’s a regular function, except you must include the class name and use the scope resolution operator to make it clear that the method is from that class.

|  |  |
| --- | --- |
| class Player {  public:  void greet() {  cout << “Hello, world!” << endl;  }  } | class Player { // Recommended.  public:  void greet();  }  void Player::greet() {  cout << “Hello, world!” << endl;  } |

* The class specification and implementation can be separated into header files and source files respectively. This is recommended as it makes the class easier to manage.
* Including many files in a large program with reusable components will create an error due to duplicate declarations. This can be prevented by using a header guard in each header file. This will prevent the pre-processor from including the same header multiple times.

|  |  |
| --- | --- |
| #ifndef \_FILENAME\_H\_  #define \_FILENAME\_H\_  // Class declaration  #endif | If the following symbol is not defined. Then do everything from here…  Define the symbol.  Declare the class.  … to here. |

* You can think of the end of the *#ifndef* line to *#endif* as a code block. It’s conventional to declare constants with all capitals and header specific constants with a \_ prefix and \_H\_ suffix.
* Alternatively, you can just write *#pragma one* at the top of the file, but not all compilers support this.
* The header file must then be included by any file that will make use of it including the header file’s corresponding source file.
* *#include <>* syntax is used for system files while *#include “”* syntax is used for local/project files.
* There are special methods which are automatically called by the compiler under certain conditions.
* **Constructors** are methods which are automatically called when an object is created to initialise it. They have the same name as the class, have no return type, and can also be overloaded. The parameters passed to the object upon instantiation must match one of the constructors.
* A **default constructor** is a constructor that requires no arguments. This can either be a constructor with no parameters, or a constructor with only default argument parameters.
* **Destructors** are methods which are automatically called when an object is destroyed to release memory and other resources. They have the same name as the class, but with a ~ prefix. They have no return types or parameters, and cannot be overloaded. It’s called when an object falls out of scope, or its pointer is deleted.
* If you don’t define any constructors or destructors, then the compiler will include a default constructor and destructor for you that do nothing. It’s usually best practice to define your own to set reasonable defaults for attributes, especially if you have primitive attributes as they contain garbage data.
* **Constructor initialisation lists** are a more efficient alternative to assigning values for attributes through the constructor’s body. The list directly initialises attributes in the order that they are declared in the class.

You can still write code in the constructor body.

* You can **delegate constructors** which means that one constructor calls another constructor. This is useful in situations where various constructors share duplicate code. Constructors can call other methods, but those methods can’t call constructors since constructors are designed to run during initialisation.

|  |  |
| --- | --- |
| class Player {  string name;  int health;  public:  Player() {  name = “None”;  health = 0;  }  Player(string n, int h) {  name = n;  health = h;  }  } | class Player { // Recommended.  string name;  int health;  public:  Player() : Player{“None”, 0} {}  Player(string n, int h) :  name {n}, health {h} {}  } |

* Objects often need to be copied. This is achieved through the **copy constructor** which copies attributes from a pre-existing object into the new object, thus creating an identical copy. Copies are created anytime an object is passed by value.
* If you don’t specify your own copy constructor then the compiler will provide one by default that creates a memberwise copy. This means that each corresponding attribute is made equal to the other. This works fine for any class that doesn’t have raw pointer attributes. The reason is that the pointer is directly copied (**shallow copy**) while the underlying object is what needs to be copied (**deep copy**). The copy constructor’s parameter should be a *const T&* as the original object should be unaffected.
* The issue with not creating a deep copy is that the destructor from the original object will release the memory, while the new object will assume it’s still valid and attempt to access it. This is undefined.
* In addition to a copy constructor, there are also **move constructors** which allow initialising an object from a temporary variable – an r-value. If a move constructor isn’t provided the compiler just uses the copy constructor, this can be inefficient. The move constructor takes an r-value reference of an object with the same type as the class, e.g. *MyClass(MyClass &&source)*.
* When moving one object into another create a shallow copy of all members including pointers, and set pointers in the original object to *nullptr*. Thus when the original object attempts to *delete* the pointers it won’t do anything.
* RVO or **return value optimisation** is an efficiency technique used by compilers in which they automatically call the move constructor to avoid copying a temporary variable when returning a local variable.
* R-value references (T&&) are used by move semantics and perfect forwarding as they represent temporary variables.
* L-value references can only initialise with l-values, while r-value references can only initialise with r-values. The same rules apply when passing l-values or r-values to functions with l-value or r-value references.
* ***this*** is a keyword that’s used within class scope to obtain a pointer for the current object. All normal pointer rules apply to it. It can be used to access data members and methods, and it can be used to determine if two objects are the same.
* *this* is often used implicitly, e.g. when calling class methods from within that class, or using attributes. In some cases you must explicitly use it, e.g. when the parameter in a method has the same name as an attribute and you want to access the attribute, or for polymorphism, or to compare the current object to an object that was passed as an argument.
* Only *const* methods can be called for *const* objects as this tells the compiler that the method won’t modify the state of the object (its attributes).
* To define a *const* method, append *const* at the end of the method signature, e.g. *void my\_method(args) const*.
* Getters are a good example of methods that should be marked *const*.
* Marking the correct aspects of a program as *const* is referred to as ***const* correctness**.
* By default, attributes and methods are **instance attributes** and **instance methods** as they affect each instance independently.
* You can also create **class attributes** or **class methods** by using the *static* keyword in their declaration. This ties the attribute or method to the class rather than an instance. This means that attributes changed from one object are also changed in other objects as they refer to the same single attribute, and that methods can be called without creating an instance of the class. Access modifier rules still apply.
* Class members example: *static int count, static int get\_count(),* etc.
* Instance members can use class members, but class members can’t use instance members.
* *const static* attributes can be initialised in the class declaration, but non-*const* *static* attributes must be initialised outside of the declaration via the scope resolution operation, e.g. *int My\_Class::My\_Var = 0*. An attribute/method only needs to be marked *static* in the declaration, not the definition.
* To access class/static members use the scope resolution operator, e.g. *My\_Class::method()*.
* ***struct*** or structures exist from C, they are the same as classes except members are *public*by default.
* Structures tend to be used to create passive objects with public attributes and without methods, so basic objects that keep track of data. They don’t add anything to the language that a class can do.
* A ***friend*** of a class is an external class or function that has access to private members within this class. The friends can be other classes, global scope functions, or methods defined within other classes.
* This is a controversial feature as some think it increases encapsulation, and others think it reduces it.
* A class itself must declare if other classes/functions are friends, so inside-out rather than outside-in.
* Access modifiers don’t affect *friend* declarations.
* Both classes must declare each other as friends if they both want access to each other’s private members.
* Use friendship sparingly to avoid making the program too complex.

Operator Overloading

* Operator overloading allows defining how an operator works with your class. Almost all operators can be defined this way, e.g. ::, :?, ., and *sizeof* cannot be overloaded.
* Only the assignment operator is defined automatically by the compiler if you don’t provide a definition. This is because assignment is a very common operation. It works the same as the default copy constructor and does a shallow/memberwise copy. This is fine if there’re no raw pointers.
* It’s recommended to only provide a definition for an operator if it makes a lot of sense to. Don’t try to force every operator to work with your class.
* The first time an object is created, either through {} or =, an appropriate constructor will be called. Every time after that when = is used the copy/move assignment operator will be called.
* To overload an operator declare it like a normal method, but using the *operator* keyword followed by the operator, e.g. *Class &operator=(const Class &rhs)*. *rhs* refers to right hand side since the source object will be on the right-hand side of the operator. It’s recommended to check if *this* and *rhs* point to the same object in which case you should return from the method, this can happen via self-assignment. Since the method returns the same type, this allows for method chaining, e.g. *Obj1 = Obj2 = Obj3 = etc*.
* Using the operator actually calls the operator overload method, e.g. *Obj1 + Obj2* is the same as *Obj1.operator+(Obj2)*. The issue with this is that the object to be converted to must always be on the left hand side. Member operator overloads always have one parameter less than what the operator requires, because *this* automatically refers to the left-hand side operand.
* Operator overloads can also be non-member methods. They are often friends of the class to allow access to *private* members. This isn’t required if you use getters and setters. They require the same number of parameters as the operator requires as *this* isn’t used outside classes. Most operators can be defined as either member methods or non-member functions. With global variants the object could be on either side of the operator and the compiler will still be able to apply the operator. It first checks to see if the rhs operand can be casted into the lhs operand, if not then it’ll try the opposite.
* The compiler first checks to see if the operator has been overloaded as a member before checking global scope. So, don’t create definitions in both class scope and global scope. This is why the assignment operator must be overloaded as a member, because if you don’t define it the compiler will.
* To support output and input streams with your class you must overload << and >> operators respectively. They should be defined as non-member functions to avoid having to write awkward syntax when chaining multiple insertion/extraction operations. For the former the first parameter is *std::ostream&*, and for the latter this is *std::istream&* as these are the types of *cout* and *cin* respectively. Remember to return references to these arguments to allow operator chaining.

Inheritance

* Inheritance allows creating a new class based on attributes and methods in another class. The new class can then introduce more attributes and methods to define its behaviour. This is recommended when there is a close parent-child relationship between the existing class and the new class.
* **Base**/super/parent class refers to the existing class. **Derived**/sub/child class refers to the new class. **Root** class refers to a class that isn’t inheriting from a base class.
* An example of this is a banking program in which an *Account* class contains members relevant to all accounts, and then specialised types of accounts such as *Savings\_Account* and *Current\_Account* which inherit from *Account* and specialise it to their requirements. The classes could all be independent, but then this would lead to duplicated error-prone code.
* **Single inheritance** is when the derived class inherits from one base class. **Multiple inheritance** is when the derived class inherits from multiple base classes.
* Public inheritance models an ‘is-a’ relationship, e.g. a *Savings\_Account* is an *Account, Circle* is a *Shape*, etc. This is only true going from a derived class and any of its base classes. It’s not true going from a base class to a derived class, and it’s not true with classes that aren’t in the same line of the hierarchy.
* Base classes are by design more general, re-usable, simple, and abstract.
* Derived classes are more specific and complex.
* **Class diagrams** visually show the relationships between classes via a class hierarchy. Primitive attributes usually aren’t included.
* To use inheritance, follow the class’s declaration a list of the classes it inherits from, e.g. *class Savings\_Account : public Account {…}*. If you don’t specify the access specifier then it is assumed *private*. Structs do the opposite.
* **Composition** models a ‘has-a’ relationship between classes. Example: A *Person* is not an *Account* and an *Account* is not a *Person*, however, a *Person* has an *Account*.
* To use composition, declare the composite class as an attribute.
* It’s good practice to prefer composition over inheritance when appropriate.
* *private* and *protected* inheritance models a ‘derived class has a base class’ relationship. Public inheritance is most commonly used.
* *protected* members are visible within the same class and derived classes. So, unless inheritance is involved *protected* acts the same as *private*. However, *protected* members are considered bad practice as they allow for similar issues to using global scope variables, but to a smaller scale.
* With *public* class inheritance, the derived class has access to *public* and *protected* members. It can’t access *private* members.
* With *protected* class inheritance, the derived class can still access *public* and *protected* members, but inherited *public* members will be considered *protected* from the derived class onwards.
* With *private* class inheritance, the derived class can still access *public* and *protected* members, but the inherited *public* and *protected* members will be considered *private* from the derived class onwards.
* In derived classes, the base classes are initialised before the derived classes, e.g. for *Savings\_Account*, *Account* is initialised first, followed by *Savings\_Account*. This means that an *Account* constructor is called followed by a *Savings\_Account* constructor. Constructors from a derived class must call a constructor from a base class, this can be the default constructor (implicit) or any.
* Destructors are called in the opposite order. So constructors are called from the root class to the most derived class, while destructors are called from the most derived class to the root class. Each destructor should only release resources created in that specific class.
* Regardless of access level, a derived class does not inherit constructors, destructor, overloaded assignment operators, or friends. The derived class versions of these methods can call the base class ones. Unless the derived class explicitly states which base class constructor to use, the compiler will attempt to use the default constructor. To specify a base class constructor, specify the base class first in the constructor initialiser list along with the required arguments, then follow it up with the required initialisations for the derived class, e.g. *Derived(T1 arg1, T2 arg2) : Base {arg1, 5}, attribute {arg2} {…}.*
* C++11 allows explicit inheritance of base non-special constructors via *using Base::Base*. However, there are many rules involved and it’s often better to define constructors explicitly. Special constructors refer to default, copy, and move constructors.
* The derived class can use operators and other methods defined in the base class. However, a lot of the base class methods may expect a *const Base &other* argument while the derived class equivalent will be *const Derived &other*. However, when a reference/pointer of a derived object is assigned to a base class reference/pointer, the variable is **sliced** thus allowing the base class reference/pointer to access the base part of the object from the derived object. Example: ‘*Derived d\_obj {}; Base &b\_obj = d\_obj*’ *d\_obj* is sliced and so *b\_obj* can access members defined in the *Base* portion of *Derived*.
* If you do not define a copy/move constructor/assignment-operator in the derived class, then the compiler will automatically create them and then call the base class’ version of that method. If you do define your own version then you must ensure that you call the base class’ version yourself as the compiler will not do so. If the class doesn’t deal with raw pointers then you most likely do not need to define them.
* Inherited members can be redefined or overridden in the derived class. The derived class versions of these methods can also call the base class versions via the scope resolution operator, e.g. *Base::method(args)*.
* By default, variables make use of **static binding** in which the compiler decides at compile time which method needs to be called, e.g. ‘*Base base {}; base.method()*’ will call *Base::method()*.
* References and pointers make use of **dynamic binding** in which the most overridden version of a method is called, e.g. ‘*Derived derived {}; Base &ref = derived; ref.method()*’ will call *Derived::method()*. However, for this to work *method()* must be defined in *Base*, and then overridden in *Derived*. If it’s not defined in *Base* then the compiler won’t see it at compile time and will display an error, if it’s not overridden by *Derived* then *Base::method()* will be called.
* In multiple inheritance, a class inherits from multiple classes rather than one. The inherited classes can belong to unrelated class hierarchies.
* There are some compelling use cases for this, but it’s best practice to refactor the design to make use of single inheritance to reduce complexity of the code.
* ***constexpr***, or constant expression, is a compile time constant. *const* is a runtime expression. If the data contained by a variable is determinable at compile time, it’s far more efficient to use *constexpr*. Examples include initialising variables with literals, or other *constexpr* expressions.

Polymorphism

* Polymorphism means to have many forms – in this case referring to functions. Compile-time / early binding / static binding refers to when a specific function call is hardcoded by the compiler. Runtime / late binding / dynamic binding refers to when a function call depends on the type of object.
* To make use of compile-time polymorphism, you overload functions. The compiler chooses which overload to call based on the number and type of arguments provided. If it can’t choose one it’ll throw a compile-time error.
* To make use of runtime polymorphism, you must create an object that’s addressed by a base class reference or pointer. The class must also make use of *virtual* functions.
* ***virtual*** tells the compiler not to bind the function call at runtime for reference/pointer types. Instead the program checks at runtime to see specifically what type of object it is, and then calls its specific implementation of that function. The keyword only needs to be applied to a base class version of a function. Every subsequent re-definition or override will have it implicitly applied.
* Any class that first declares virtual methods in its hierarchy must declare its destructor virtual. This is to ensure that the correct destructor is called if a base class pointer is deleted.
* Function overloads are versions of a function which are bound at compile time, while function overrides are versions of a function which are bound at runtime.
* ***override*** tells the compiler that a method is supposed to override a base class method. This isn’t necessary at all for polymorphism to work, but it prevents bugs by making sure that the method signature and return type matches the base class method signature. As such it’s best practice to use it as otherwise you may accidentally overload instead of override.
* The debugger can be used to step through the code. You can see which version of a method is called during runtime. This can clarify any polymorphic call confusion.
* ***final*** before a class or a method tells the compiler that the entity can’t be derived further. A *final* class can’t be inherited from, and a *final* virtual method can’t be overridden. In both cases this allows the compiler to optimise. These are often used when a class design has to be protected from any modification to specific methods or to the class itself. Examples: *class Derived final: public Base {…}, void my\_method() final*.
* An **abstract class** is a class that cannot be instantiated. These classes are designed to be inherited from and provided with specific functionality. As such base classes tend to be abstract classes and they are referred to as abstract base classes. They are useful when the base class itself doesn’t describe a concept enough to be of practical use. An example of this is the Account class. The account class can contain attributes and methods that are common to all accounts, but perhaps Account itself isn’t specific enough to be of use and should be inherited from. In this case it would make sense to mark it abstract.
* To mark a class abstract, it must contain at least one **pure virtual function**. A pure virtual function is a function which does not have a definition, thus it must be overridden in a derived class. Otherwise the derived class will also be an abstract class. To declare a pure virtual function: *virtual bool deposit() = 0*. These methods can still be provided with a definition, but they don’t usually have one.
* The opposite of an abstract class is a **concrete class**. All of the methods in these classes have been defined and as such the class can be instantiated.
* An **interface** is an abstract class with only pure virtual functions. This means that every method must be overridden by the derived class. This is useful in situations where unrelated class hierarchies need something in common to execute code. An example of this could be a *Printable* interface in which the derived classes will all be able to print their contents generically.
* Sometimes interfaces are prefixed with I\_ to differentiate them from regular classes.
* To tell the compiler to generate a default special method use *= default* after the declaration, e.g. *virtual ~Base() = default*’.

Smart Pointers

* The idea behind smart pointers is to remove the need of allocating and deallocating memory on the heap via *new* and *delete* since it’s a consistent source of bugs in code. Smart pointers handle allocation and deallocation internally so that the programmer no longer has to.
* Raw pointers have the following issues which smart pointers aim to resolve: uninitialised/wild pointers, memory leaks, dangling pointers, and not exception safe. It’s also not clear who owns the raw pointer, as the owner should be the one to delete it.
* There are four types of smart pointers: unique pointers (*unique\_ptr*), shared pointers (*shared\_ptr*), weak pointers (*weak\_ptr*), and auto pointers (*­auto\_ptr*). Auto pointers have been deprecated and shouldn’t be used anymore.
* All of them wrap raw pointers and provide additional functionality. They’re used similar to raw pointers since they overload the dereference (\*) and pointer-to-method (->) operators. You don’t have to worry about allocating or deallocating memory since that’s managed by the smart pointer. They don’t support pointer arithmetic. They can have custom deleters.
* To use a smart pointer include the *memory* library, e.g. *#include <memory>*. Then declare and initialise the smart pointer, e.g. *std::shared\_pointer<Base> ptr {new Derived {}}*. Use it like a regular raw pointer, e.g. *ptr->method(),* or *std::cout << \*ptr << std::endl*. The smart pointer will automatically be dealt with when you’re done.
* **RAII**, or resource acquisition is initialisation, is a common design pattern for managing a resource. ‘Resource acquisition’ refers to opening a file, allocating memory, acquiring a lock, etc. ‘Is initialisation’ refers to acquiring the resource in the constructor. The resource is freed by the destructor, e.g. closing the file, deallocating memory, or releasing the lock. Smart pointers make use of this concept since the memory is allocated via the constructor and then automatically deallocated via the destructor.
* **Unique pointers** point to an object of type *T* on the heap. You must specify the type via the angled brackets. It’s called unique because it owns the resource.
* They can only be moved, they can’t be copied. This ensures that only one object at a time has access to a resource. Once the unique pointer falls out of scope, the resource is automatically freed.
* It’s bad practice to initialise a raw pointer then to pass in its variable into the unique pointer’s constructor. This is because the unique pointer will assume it’s the owner and turn the pointer into a dangling pointer once it falls out of scope.
* The *get()* method returns the raw pointer itself.
* Unique pointers can auto cast to bool. The pointer returns true if it’s pointing to valid memory, and false if it’s pointing to *nullptr*, e.g. *if(my\_unique\_ptr) {…}*.
* The *reset()* method frees the resource and sets the raw pointer to *nullptr*.
* To move a unique pointer use *std::move()*. This can be used to move any type that supports move semantics.
* *make\_unique<T>(args)* is a better way to create a unique pointer as you don’t have to use the *new* keyword. It’s also more efficient.
* Unique pointers should be the preferred choice, followed by shared pointers.
* **Shared pointers** also point to objects of type *T* on the heap. It’s called shared because multiple objects share the resource – shared ownership.
* They can be copied or moved. Once all of the shared pointers that share the same resource fall out of scope, the resource is automatically freed. An attribute in the shared pointer keeps track of the number of objects sharing the resource, if that reference count is 0 in the destructor, the resource is freed.
* Unique pointers can create arrays on the heap, while shared pointers can’t.
* Initialising a shared pointer sets its counter to 1, copying it increments the counter in the old object and copies the counter to the new object, moving it sets the counter in the new object to the current count then sets it to 0 in the old object, *reset()* decrements the counter in all of the objects and sets the counter to 0 in the current object, the destructor decrements the counter in all of the objects.
* The counter should be the same between all of the shared pointers that are associated with the same resources.
* *make\_shared<T>(args)* works the same way for shared pointers as *make\_unique* does for unique pointers.
* **Weak pointers** provide a non-owning/weak reference to a resource owned by a shared pointer. As such creating one doesn’t affect the reference count in a shared pointer.
* They are always created from a *shared\_ptr*.
* They are used in situations where two classes have shared pointers that point to the opposite class. This circular ownership stops the resource from being freed even when the shared pointers fall out of scope – memory leak. However, by replacing one shared pointer with a weak pointer, it stops the circular ownership.
* Custom deleters allow running additional code when the resource is being freed.
* Can’t use *make\_unique* or *make\_shared* if you want to use customer deleters.

Exception Handling

* Exceptions indicate that an unusual situation has occurred. They then allow the program to deal with it. A common example is dividing by zero as this is undefined.
* Exceptions can be caused by insufficient resources, missing resources, invalid operations, range violations, underflows, overflows, illegal data, etc.
* **Exception safe** code is code that can handle exceptions correctly. This can be difficult to do in practice.
* The developer community is divided over when to use exceptions. Some developers barely use them due to the performance cost, while other developers use them even if the situation isn’t so exceptional.
* Exceptions should only be used for synchronous code, not asynchronous code.
* An exception is an object or primitive type that signals that an error has occurred.
* When the code detects that an error has occurred or is about to occur, it can **throw/raise** an exception. Usually the place where the error occurs does not know how to handle the error. The code can then throws the exception to another part of the program that does.
* **Catching/handling** the exception means to deal with the exception as appropriate. If the exception signals a major problem, then the program may terminate, e.g. out of memory, storage, network disconnected, etc.
* ***throw*** keyword throws the exception and is followed by an argument, e.g. *throw Exception()*. If a matching *catch* statement is not found in the current function, then the exception propagates to the calling method, and then its calling method, etc until the program terminates. This is called **stack unwinding**.
* ***try*** blocks contain the code that may throw an exception. If a exception is thrown the *try* block is exited, and an appropriate *catch* statement deals with it, e.g. *try {…}*.
* ***catch*** statements handle the exception. A *try* block can be followed by multiple *catch* statements, e.g. *catch(Exception ex) {…}*. The type of the thrown exception must match the *catch* type exactly, or auto cast to it. A catch-all statement can be used as the last statement to catch any exceptions not caught, e.g. *catch(…) {…}*.
* It’s best practice to throw an object type rather than primitive, to throw the object by value, and to catch by reference or *const* reference.
* Since any object can be thrown, it’s best to create a class whose name describes the exception, e.g. *DivideByZeroException*, or *NegativeValueException*, etc. The class can be empty and contain nothing since the name of the class can describe a lot.
* Constructors cannot return data directly to indicate that something has gone wrong, instead you can use exceptions. Do not throw exceptions from a destructor as they are marked *noexcept* by default.
* *noexcept* tells the compiler that the function won’t throw an exception. If it does then the program will terminate without handling the exception. The keyword must appear in the declaration and definition.
* *std::exception* is the base class of the C++ standard library exception class hierarchy. All subclasses must override *what()* to define the cause of the exception as a character array. The hierarchy is thorough and covers the majority of exceptions that a program will run into.

I/O and Streams

* C++ uses streams as an interface between the program and input/output devices.
* It’s independent of the actual device.
* A stream is a sequence of bytes.
* The input stream provides data to the program. The output stream sends data from the program.
* *iostream* allows input/output to streams, *fstream* allows input/output to files, *iomanip* allows manipulating stream formatting.
* *fstream* allows input and output to files because it inherits *ifstream* and *ofstream* via multiple inheritance.
* *stringstream* allows input/output on memory based strings.
* *cout, cin, cerr,* and *clog* are global scope objects for some of these classes.
* Most stream formatters come in two versions: member methods, or manipulators. Example: *cout.width(10)* vs *cout << setw(10)*. Manipulators are preferred.
* Boolean values by default are displayed as 1 or 0. To display true or false instead use *boolalpha*. To turn it off use *noboolalpha*.
* Integer values can be displayed as decimal, octal, or hexadecimal values via *dec, oct,* and *hex*. The base prefix can be enabled using *showbase*. The letters in a hexadecimal value can be uppercased using *uppercase*. To display a + or – symbol depending on the value use *showpos*. Default: *dec, noshowbase, nouppercase, noshowpos*.
* Floating point values can be displayed with varying precision, *e.g. setprecision(4)* will display four significant figures, however by specifying the *fixed* manipulator, it will now display the figure to four decimal places. If the figure can be displayed to the given significant figures, it will be displayed using scientific notation. Use *scientific* to display the figure using scientific notation regardless. Use *uppercase* to display the ‘e’ in the scientific notation as ‘E’. Use *showpoint* to display trailing zeroes to match the precision. Default: *setprecision(6), noshowpoint, nouppercase, noshowpos*.
* All of these manipulators can be disabled either by calling the *no* variant manipulator, e.g. *noshowpos*, or by calling the *resetiosflags(std::ios::flag)* manipulator, or by calling the *unsetf(…)* method. All of the manipulators so far also apply to all future output unless they are disabled.
* *setw()* only applies to the next data to be inserted into the stream. It sets the field with and then right justifies the data within, e.g. *cout << setw(10) << 1234.5* will output ‘ 1234.5’. You can also left justify and fill the remaining space with another character, e.g. *cout << setw(10) << left << setfill(‘-‘) << 1234.5* will output ‘1234.5-----‘. *setfill, right,* and *left* apply to all future output.
* Files can be read in many ways, e.g. binary mode, text mode, one character at a time, one line at a time, etc. The file should be closed when dealt with.
* To open a file for reading: *std::fstream file {“../myfile.txt”, std::ios::in}*. *in* means to open the file in input mode. Alternatively: *std::ifstream file {“../myfile.txt”}*.
* To open a file in binary mode: *std::fstream file {“../myfile.txt”, std::ios::in | std::ios::binary}*. Files are opened in text mode by default. The bitwise OR operator is used to combine many flags in one pass.
* You can use the extraction operator for formatted reading. It works the same way as with *cin*. *std::getline* can be used to read an entire line at once. Use *get()* to extract one character.
* To open a file for writing: *std::fstream file {“../myfile.txt”, std::ios::out}*. Alternatively: *std::ofstream file {“../myfile.txt”}*. If the file doesn’t exist it will automatically be created. Output files are overwritten/ truncated by default: *std::ios::trunc*. You can also append: *std::ios::app*, and seek to the end: *std::ios::ate*.
* You can seek around in files using random access.
* Closing a file is recommended since it will flush any unwritten data from the buffer.
* Use *put()* to insert one character.
* For input files or output files:
* *open()* is used to open a file after construction.
* *is\_open()* is used to check if a file was successfully opened.
* *close()* is used to close the file once you’re done with it. It’s good practice to release resources.
* String streams allow reading to or writing from strings in memory, similar to how we work with files. Stream manipulators are supported here just as they are when working with files or standard I/O.
* To use string streams: *#include <sstream>*. Declare a *stringstream, istringstream,* or *ostringstream* and connect it to a *std::string*. You can then read/write using formatted I/O.

The Standard Template Library (STL)

* It is a library of reusable, adaptable, generic classes and functions. It implements common data structures and algorithms.
* They are implemented using C++ templates.
* **Containers/collections** are used to hold other types of data, e.g. *array, vector, deque, stack, set, map, etc*.
* There are any algorithms that produce a result from/on the collections, e.g. *find, max, count, accumulate, sort, etc.*
* **Iterators** generate a sequence of elements from the collections, e.g. forward, reverse, by value/reference, constant, etc. The range-based for loop works using iterators.
* There are also functors and allocators.
* **Generic programming** allows writing code that works with multiple types in one ago. This can be done through macros, function templates, and class templates.
* **Macros** are created using the *#define* pre-processor directive. They’re generally not recommended, especially for generic programming. It’s fine to use them for header guards. It provides no type information and is just substituted into code by the pre-processor, e.g. *#define PI 3.14* will just put *3.14* wherever it finds *PI*.
* Macros can also take parameters for generic programming, e.g. *#define MAX(a, b) ((a>b) ? a : b)*. It’s best to wrap macros in parenthesis to maintain precedence, e.g. *#define SQUARE(a) a\*a* is dangerous for *result = 100/SQUARE(5)* – outputs 100, *#define SQUARE(a) (a\*a)* produces correct result.
* Templates allow declaring a placeholder type. The compiler then generates appropriate functions/classes with the type if it doesn’t already exist. This is referred to as **meta-programming** as the compiler generates more code based on instructions from the programmer.
* To define a template function: *template <typename T> T max(T a, T b) {…}*. This defines a template function that takes in two arguments of type *T* and returns a *T*. To use the function: *max<int>(5, 10)* will return 10. You can omit the type as the compiler can tell from the arguments that *T* is *int*: *max<>(5, 10)*. Empty angled brackets are optional and can be omitted: *max(5, 10)*. You can even use a custom class as long as it supports the > operator, as this operator is used by *max* to determine which argument is higher.
* To define multiple template parameters: *template <typename T1, typename T2> void func(T1 a, T2 b) {…}*. This can be called as *func<int, double>(5, 10.1)* or as *func(5, 10.1)*. You can use *const/&* qualifiers as needed and not all parameters/return have to be a generic type, etc.
* You can also use *class* instead of *typename*, but *typename* is the more modern keyword, there’s no other difference.
* If a class contains many methods that depend on the same generic type, then the entire class can be declared as a template class, e.g. *template <typename T> class Item {…}*. You can now use *T* when declaring variables anywhere within the class. Template classes should usually be put in header files, otherwise there’ll be compilation errors.
* Variables can be passed at compile time through templates, e.g. *template <int N> …* will accept an integer.
* The *pair* template class is used to associate two pieces of data together. It’s generic so the type of data can be anything. You can either provide arguments through the constructor, or through *make\_pair()*. You can access the attributes directly through *first* and *second*.
* The *tuple* template class works like *pair* except it can associate many pieces of data together. All of them can be of a different type. You can either provide arguments through the constructor, or through *make\_tuple()*. You can access the data through *std::get<index>(tuple)*.
* There are three types of containers: sequence containers, associative containers, and container adapters.
* **Sequence containers** maintain insertion order, e.g. *array, vector, list, forward\_list, deque*.
* **Associative containers** allow fast retrieval of elements using a key, e.g. *set, multi set, map, multi map*. They’re usually implemented using balanced binary trees (red-black tree), or hashsets.
* **Container adapters** are variations of the other containers, e.g. *stack, queue, priority queue*. They do not support iterators and as such the *algorithm* doesn’t support them.
* Sequence containers:
* It’s more efficient to use arrays over vectors when the size of data is known and fixed. However, built in arrays aren’t considered safe, it’s better to use *std::array*. This is because it provides bounds checking, and keeps track of the size, etc. It doesn’t have a constructor however, so the data won’t be automatically initialised and will contain garbage/undefined values by default – you should use {}. Can use all iterators and they do not invalidate since the size of the array is constant. Elements are stored in contiguous memory. To use: *#include <array>*.
* When working with *vector*, prioritise calling *emplace\_back()* over *push\_back()*. The latter makes sense when copying or moving a pre-existing object. The former is faster when creating a new object as it directly initialised the object within the *vector*. Example: prefer *emplace\_back(“Bob”, 18)* over *push\_back(Person{“Bob”, 18})*. Elements are stored in contiguous memory.
* A *deque*, pronounced deck, is a double ended queue and can insert/delete from the back or front in constant time. Inserting or deleting elsewhere is linear. Objects can be emplaced in the back or front. All iterators are available, but may invalidate since the *deque* can change size. The elements are not stored in contiguous memory. To use *#include <deque>*. Deques are stored semi-contiguously in memory, you can think of it as a list of vectors.
* A *list* is non-contiguous in memory and has no direct access to elements, rather each element links to the next element via a pointer. Since each element is linked, links can be deleted or inserted in constant time. However, to get to a certain link requires going through previous links so access complexity is linear. All iterators are available and only invalidate when the element that it points to is deleted.
* A *list* is bi-directional, also called a doubly-linked list since each element has a pointer to the next and previous element. If you only ever need to traverse the sequence in one direction then you can use a *forward\_list*. This has reduced overhead since each element only has a pointer to the next element. The sequence can only be traversed forwards, and there’s no reverse iterator. There’s no *back()* method either to access the last element, nor is there a *size()* method.
* Associative containers:
* There are four types of sets: *set, multiset, unordered­\_set,* and *unordered\_multiset*. All of them have constant time insert, remove, and find operations. For *set* and *multiset*: *#include <set>*, for *unordered\_set* and *unordered\_multiset*: *#include <unordered\_set>*.
* A *set* is ordered by key and ignores duplicates. All iterators are available and invalidate when the corresponding element is deleted.
* A *multiset* allows duplicate elements, otherwise it’s the same as *set*.
* An *unordered\_set* isn’t ordered, elements can’t be modified – they must be deleted and replaced, and it doesn’t support reverse iterators, otherwise it’s the same as *set*.
* An *unordered\_multiset* is like an *unordered\_set* except that it allows duplicates.
* There are four types of maps: *map, multimap, unordered\_map,* and *unordered\_multimap*. In theory they are all very similar to the *set* variants. The different is that with each key there is an associated value, together they are referred to as a key-value pair. As such *pair* is commonly used when dealing with maps. Another name for a map is dictionary. Elements can be directly accessed by using the appropriate key.
* Container adapters:
* A *stack* is a LIFO (last in first out) structure. Elements are *push()*ed and *pop()*ped to/from the top. It’s implemented using other containers. Any container that can add or remove elements to at least one side can be used to implement it, e.g. *vector, list,* or *deque*. All operations occur at the top of the *stack*.
* A *queue* is a FIFO (first in first out) structure. Elements are pushed at the bottom and popped at the top. It can be implemented using a *list* or *deque* as they can push and pop from the front and back. You can access both the *front()* or *back()* elements.
* A priority queue assigns a priority to each element and *push()*es them in that order. Elements with the highest priority are *pop()*ed first. Priority is determined using the *operator<* overload – greater elements are given higher priority. Elements are stored as a *vector* by default.
* No iterators are supported by either *stack,* *queue,* or *priority queue* since you only have access to certain elements. You must *pop()* the top elements to access the other elements. As iterators aren’t supported, neither are the STL algorithms.
* There are five types of iterators: input, output, forward, bi-directional, and random access.
* **Input iterators** make container elements available to the program.
* **Output iterators** allow writing elements to the container.
* **Forward iterators** iterate in one direction over a sequence and can read/write.
* **Bi-directional iterators** are the same as forward iterators, but in both directions.
* **Random access iterators**use the subscript operator to access any element.
* The iterator type for a container can be obtained via the *iterator* type alias, e.g. *std::vector<int>::iterator*, *std::map<std::string, int>::iterator*, etc. *auto* is often used in this case to let the compiler deduce the type, e.g. *auto it = vector.begin()*.
* *begin()* is defined as the iterator to the first element. *end()* is defined as the iterator one passed the last element.
* All iterators can be pre/post-incremented and assigned. I/O iterators can be dereferenced. Input iterators support equality operators. Bidirectional iterators support pre/post-decrement operators. Random access iterators support comparison operators and increment/decrement operators.
* Incrementing an operator moves it to the next element in the sequence. Dereferencing an operator returns the data that the iterator is pointing to.
* *const* iterators only allow reading elements, you can’t write.
* Iterators can become invalid during processing, e.g. is *clear()* is called the iterators point to invalid locations.
* *sort()* sorts a collection. You must provide the beginning and end iterators and the function will sort within that range. It’s common to sort the entire collection, e.g. *sort(vector.begin(), vector.end())*.
* *reverse()* reverses a sequence.
* *accumulate()* returns the sum of the sequence. The third parameter is for the initial value, usually 0.
* Different containers use different iterators and different algorithms support different iterators. So to use an algorithm with a container, they must both support the same iterator.
* *find()* iterates through a container for an element and returns the iterator at which the element was first found. If there are no occurrences of the element, the iterator returns the *end()* iterator. This function uses the *==* operator to determine equality, so a custom class must override it.
* Many algorithms expect additional information to run which can be provided through different types of functions: functors (function objects), function pointers, and lambda expressions. A functor is a class that overloads the function call operator – (). Best practice is to use lambdas.
* *for\_each()* iterators through a container and applies a provided function to each element, e.g. *for\_each(data.begin(), data.end(), [](int x) { std::cout << x\*x << " "; })* outputs the square of each element to the console.
* *count\_if()* iterates through a container and applies a provided predicate to each element, e.g. *count\_if(data.begin(), data.end(), [](int x) { return x%2 == 0; })* returns a count of the number of elements matching the condition.
* *all\_of()* iterates through a container and checks to see if a provided predicate is valid for every element,
* A **predicate** is a function that compares input and returns a Boolean based on a condition.
* *replace()* iterates through a container and replaces matching elements with the substitute element, e.g. *replace(data.begin(), data.end(), 10, 100)* will replace all instances of 10 with 100.
* *transform()* iterates through a container and applies a transformation to the elements, e.g. *std::transform(data.begin(), data.end(), data.begin(), [](int x) { return x\*x; })* replaces each element with its square. The third argument tells the function where to save its results to. In this case the result is saved in the original container.
* *insert()* is used to insert an element or multiple elements at a certain position in the target container.

Bonus Material and Source Code

* No notes required.