/**~~~~~~*/

CH. 1 – Basic Ideas (22)

- A C++ program consists of one or more **functions**, one of which is called **main()**. Execution always starts with main().
- The executable part of a function is made up of statements contained between braces. "{}"
- A pair of **curly braces** is used to enclose a **statement block**.
- A statement is **terminated** by a **semicolon**.
- **Keywords** are reserved words with specific meanings in C++. No entity in a program can have a name that is a keyword.
- A C++ program will be contained in one or more files.

Source files -> executable code, & **Header files** -> definitions used by the executable code.

- The source files that contain the code defining functions typically have the **extension .cpp**
- Header files that contain definitions that are used by a source file typically have the **extension .h**
- **Preprocessor directives** specify operations to be performed on the code in a file. All preprocessor directives execute before the code in a file is compiled.
- The contents of a header file are added into a source file by an "**#include**" preprocessor directive.
- The **Standard Library** provides an extensive range of capabilities that supports and extends the C++ language.
- Access to Standard Library functions and definitions is enabled through including Standard Library header files in a source file.
- **Input** and **output** are performed using **streams** and involve the use of the **insertion** and **extraction operators, << and >>** . std::cin is a standard input stream that corresponds to the keyboard. std::cout is a standard output stream for writing text to the screen. Both are defined in the **iostream** Standard Library header.
- **Object-oriented programming** involves defining new **data types** that are specific to your problem. Once you've defined the data types that you need, a program can be written in terms of the new data types.
- **Unicode** defines unique integer code values that represent characters for virtually all of the languages in the world as well as many specialized character sets. Code values are referred to as **code points**. Unicode also defines how these code points may be encoded as **byte sequences**. ('a' == 97)

/**~~~~~~*/

CH. 2 – Fundamental Data types (62)

- **Constants** of any kind are called **literals**. All literals have a type.
- You can define integer literals as decimal, hexadecimal, octal, or binary values.
- A **floating-point** literal must contain a decimal point or an exponent or both. If there is neither, it's an integer.
- The fundamental types that store **integers** are **short, int, long,** and **long long**. These store **signed** integers, but you can also use the type modifier unsigned preceding any of these type names to produce a type that occupies the same number of bytes but stores unsigned integers.
- The **floating-point** data types are **float**, **double**, and **long double**.
- Uninitilized variables generally contain garbage values. Variables may be given initial values when they're defined, and it's good programming practice to do so. A braced initializer is the preferred way of specifying initial values.
- A variable of type **char** can store a single character and occupies one byte. Type char may be signed or unsigned, depending on your compiler. You can also use variables of the types signed char and unsigned char to store integers. Types char, signed char, and unsigned char are different types.
- Type **wchar_t** stores a wide character and occupies either two or four bytes, depending on your compiler. Types **char16_t** and **char32_t** may be better for handling Unicode characters in a cross-platform manner.
- You can fix the value of a variable by using the **const** modifier. The compiler will check for any attempts within the program source file to modify a variable defined as const.
- The four main mathematic operations correspond to the binary +, -, *, and / operators.

For integers, the modulus operator **%** gives you the remainder after integer division.

- **++** and **-- operators** are special shorthand for adding or subtracting one in numeric variables. Both exist in postfix / prefix forms.
- You can mix different types of variables and constants in an expression. The compiler will arrange for one operand in a binary operation to be automatically converted to the type of the other operand when they differ.

•Compiler automatically converts the type of the result of an expression on the right of an assignment to the type of the variable on the left where these are different. This can cause loss of information when the left-side type isn't able to contain the same information as the right-side type—double converted to int, for example, or long converted to short.

- You can explicitly convert a value of one type to another using the **static_cast< >() operator**.

All manipulators declared by ios are automatically available if you include the familiar **#include<iostream>** header. Unlike those of the **#include<iomanip>** header, these stream manipulators do not require an argument:

Stream Manipulator	Description
std::fixed	Output: Floating-point data in a **fixed-point notation**
std::scientific	Output: All Subsequent Floating-point data in **Scientific Notation**, which always includes an exponent and one digit before the decimal point.
std::defaultfloat	Revert to **default floating-point** data presentation.
std::dec	All subsequent int outputs are **Decimal**.
std::hex	All subsequent int outputs are **Hexadecimal**.
std::oct	All subsequent int outputs are **Octal**.
std::showbase	Outputs the **base prefix for hexadecimal and octal integer values**. Inserting **std::noshowbase** in a stream will switch this off.
std::left	Output **justified to the left**.
std::right	Default. Output **justified to the right**.
#include<iomanip> (header)->Stream Manipulators	
std::setprecision(n)	Sets the floating-point precision or the number of decimal places to n digits. **OR** If the default floating-point output presentation is in effect, n specifies the number of digits in the output value. **OR** If fixed or scientific format has been set, n is the number of digits following the decimal point. The default precision is 6.
std::setw(n)	Sets the output field width to n characters, but only for the next output data item. Subsequent output reverts to the default where the field width is set to the number of output character needed to accommodate the data.
std::setfill(ch)	When the field width has more characters than the output value, excess characters in the field will be the default fill character, which is a space. This sets the fill character to be ch for all subsequent output.

/**~~~~~~*/

CH. 3 – Working With Fundamental Data Types (85)

Operator	Conditions / Functions
x & y	AND Condition Produces 1 bit when **both** are 1. FINDS SIMILARITIES
x | y (a OR b OR Both)	OR Condition Produces 1 bit when **either** is 1 FINDS ALL BITS.
x ^ y (Either a OR b, NOT Both)	Exclusive OR Condition contains a 1 if and only if one of the corresponding input bits is equal to 1, while the other equals 0. Whenever both input bits are equal, even if both are 1, the resulting bit is 0. **FINDS DIFFERENCES**
~x	Complement Turn off condition. **INVERT BITS.**
x << y	(Multiply by Powers of 2) **Shift x's Bits Left by y (2^y)**
x >> y	(Divide by Powers of 2) **Shift x's Bits Right by y (2^y)**

•Don't memorize operator precedence and associativity for all operators, but be conscious of it when writing code. Always use parentheses if you are unsure about precedence.

•The type-safe **enumerations type** are useful for representing fixed sets of values, especially w/ names, such as days of the week or suits in a pack of playing cards.

•**Bitwise operators** are necessary when working with **flags**—single bits that signify a state. These arise surprisingly often—when dealing with file input and output, for example. The bitwise operators are also essential when you are working with values packed into a single variable. One extremely common example thereof is RGB-like encodings, where three to four components of a given color are packed into one 32-bit integer value.

• **using** keyword allows you to define aliases for other types. In legacy code, may still encounter *typedef* being used.

•By default, a variable defined within a block is **automatic**, it exists only from the point at which it is defined to the end of the block where it appears, indicated by the closing brace of the block.

•Variables can be defined outside of all the blocks in a program, in which case they have global namespace scope and static storage duration by default. Variables with global scope are accessible from anywhere within the program file that contains them, following the point at which they're defined, except where a local variable exists with the same name as the global variable. Even then, they can still be reachedby using the **scope resolution operator** (**::**)

/**~~~~~~*/

CH. 4 – Making Decisions (120)

•**comparison operators**- compares two values. This will result in a value of type bool, which can be **either true or false.**

• You can convert a **bool** value to an integer type—true will convert to 1, and false will convert to 0.

• Numerical values can be converted to type bool—a zero value converts to false, and any nonzero value converts to true. When a numerical value appears where a bool value is expected—such as in an if condition—the compiler will insert an implicit conversion of the numerical value to type bool.

•The **if statement** executes a statement or a block of statements depending on the value of a condition expression. If the condition is true, the statement or block executes. If the condition is false, it doesn't.

•The **if-else statement** executes a statement or block of statements when the condition is true and executes another statement or block when the condition is false.

•if and if-else statements can be nested.

•The **logical operators &&**, **||**, and **!** are used to string together more complex logical expressions. The arguments to these operators must either be Booleans or values that are convertible to Booleans (such as integral values).

•The conditional operator selects between two values depending on the value of an expression.

•The switch statement provides a way to select one from a fixed set of options, depending on the value of an expression of integral or enumeration type.

Comparing Data Values 89

Relational Operators

Operator	Meaning
<	Less Than
<=	Less Than OR Equal to
>	Greater Than
>=	Greater Than OR Equal to
==	Equal to
!=	Not Equal to

std::cout << std::boolalpha; // Shows True or False ----rather than 1 or 0

std::cout << std::noboolalpha; // Shows 1/0 ---rather than T/F

• **locale:** a set of parameters that defines the user's language and regional preferences, including the national or cultural character set and the formatting rules for currency and dates.

Character classification functions provided by the **#include<cctype>** header

Character Classification Functions with cctype Header

Function(std::)	Operation ----Tests where (c) =
isupper(c)	Uppercase letter, by default 'A' to 'Z'.
islower(c)	Lowercase letter, by default 'a' to 'z'.
isalpha(c)	Uppercase or Lowercase letter (or any alphabetic character, neither uppercase nor lowercase, should the locale's alphabet contain such characters).
isdigit(c)	Digit, '0' to '9'.
isxdigit(c)	Hexadecimal digit, either '0' to '9', 'a' to 'f', or 'A' to 'F'.
isalnum(c)	Alphanumeric character; same as isalpha(c) || isdigit(c).
isspace(c)	Whitespace, by default, or: space (' '), newline ('\n'), carriage return ('\r'), form feed ('\f'), horizontal ('\t'), vertical tab ('\v').
isblank(c)	Space character used to separate words within a line of text. By default either a space (' ') or a horizontal tab ('\t').
ispunct(c)	Punctuation character. By default, this will be either a space or one of the following: _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
isprint(c)	Printable character ::: Uppercase || Lowercase letters, digits, punctuation chars, and spaces.
iscntrl(c)	Control character, the opposite of a printable character.
isgraph(c)	Has a graphical representation, which is true for any printable character other than a space.

Converting Character's with cctype (Upper/Lower)

Function	**Operation**
tolower(c)	If c uppercase, lowercase equivalent is returned; otherwise, c is returned.
toupper(c)	If c lowercase, uppercase equivalent is returned; otherwise, c is returned.

Result = type int, so you need to explicitly cast it if you want to store it as type char.

short-circuit operators: if first operand in binary logical expression determines outcome, the compiler will make sure no time is wasted evaluating the second operand. Properties of: **&&** and **||**. NOT Bitwise operators & and |.

c = a > b? a : b; // Set c to the higher of a and b

- **switch, case, default,** and **break** are all keywords

Multiple Cases:

```
case 'a': case 'e': case 'i': case 'o': case 'u':
        std::cout << "You entered a vowel." << std::endl;
        break;
```

- **fallthrough:** Remove break statements from a switch statement, the code beneath the case label directly following the case without a break statement then gets executed as well. in a way we "fall through" into the next case.

CH. 5 – Arrays & Loops(174)

- An **array** stores a fixed number of values of a given type.
- You access elements in a one-dimensional array using an **index value** between **square brackets**. Index values start at 0, so in a one-dimensional array, an index is the offset from the first element.
- An array can have more than one dimension. Each dimension requires a separate index value to reference an element. Accessing elements in an array with two or more dimensions requires an index between square brackets for each array dimension.
- A **loop** is a mechanism for repeating a block of statements.
- There are four kinds of loop that you can use: the **while loop**, the **do-while loop**, the **for loop**, and the **range-based for loop**.
- The **while loop** repeats for as long as a specified condition is true.
- The **do-while loop** always performs at least one iteration and continues for as long as a specified condition is true.

•**for loop** is used to repeat a given number of times and has three control expressions. The first is an ***initialization expression***, executed once at the beginning of the loop. The second is a ***loop condition***, executed before each iteration, which must evaluate to true for the loop to continue. The third is executed at the end of each iteration and is usually used to ***increment a loop counter***.

•**range-based for loop** iterates over all elements within a range. An array is a range of elements, and a string is a range of chars. Array & Vector containers define a range so you can use the range-based for loop to iterate over the elements they contain.

• Any kind of loop may be nested within any other kind of loop to any depth.

•Executing a **continue statement** within a loop skips the remainder of the current iteration and goes straight to the *next iteration*, as long as the loop control condition allows it.

•Executing a **break statement** within a loop causes an *immediate exit* from the loop.

•A loop defines a scope so that variables declared within a loop are not accessible outside the loop. In particular, variables declared in the initialization expression of a for loop are not accessible outside the loop.

• The **array<T,N> container** stores a sequence of **N elements** of **type T**. array<> containers is a good alternative to primitive arrays.

•The **vector<T> container** stores a sequence of elements of type T that increases dynamically in size as required when you add elements. *Use a vector<> container instead of a standard array when the number of elements cannot be determined in advance.*

/**~~~~~~*/

CH. 6 – Pointers and References

• pointer = variable that contains an address. A basic pointer is referred to as a **raw pointer**.

•You obtain the address of a variable using the **address-of** operator, **&**.

•**Dereference(indirection) operator**: To refer to the value of the pointer, use the *****.

•You access a **member**(variable) of an object through a pointer or smart pointer using the **indirect member selection operator, ->**.

•Can add or subtract integer values from the address stored in a raw pointer. It's as though the pointer refers to an array & pointer is altered by the number of array elements specified by the integer value. You cannot perform arithmetic with a smart pointer.

•The **new** and **new[] operators** allocate a block of memory in the free store—holding a single variable and an array, respectively and return the address of the memory allocated.

•Use the **delete** or **delete[] operator** to release a block of memory that you've allocated previously using new or, respectively, the new[] operator. Don't need these operators when address of free store memory is stored in a smart pointer.

•Low-level dynamic memory manipulation is synonymous for a wide range of serious hazards such as dangling pointers, multiple deallocations, deallocation mismatches, memory leaks, and so on. **Never use the low-level new/new[] and delete/delete[] operators directly**. Containers(**std::vector<>** in particular) and smart pointers are nearly always the smarter choice!

•**Smart pointer:** Object that can be used like a raw pointer. By default, used only to store free store memory addresses.

•2 Common Smart pointers. There can only ever be one type **unique_ptr<T>** pointer in existence that points to a given object of type T, but there can be multiple **shared_ptr<T>** objects containing the address of a given object
of type T. The object will then be destroyed when there are no shared_ptr<T> objects containing its address.

•**Reference:** Alias for a variable that represents a permanent storage location.

•Use reference type for the loop variable in a range-based for loop to allow the values of the elements in the range to be modified.

/**~~~~~~*/

CH. 7 – Working with Strings (254)

• The **std::string** type stores a character string.

• Like std::vector<char>, it is a **dynamic array**—meaning it will allocate more memory when necessary.

• Internally, the **terminating null character** is still present in the array managed by a std::string object, but only for compatibility with legacy and/or C functions. As a user of std::string, you normally do not need to know that it even exists. All string functionality transparently deals with this legacy character for you.

•You can store string objects in an array or, better still, in a **sequence container,** such as a **vector**.

•You can access and modify individual characters in a string object using an index between square brackets. Index values for characters in a string object start at 0.

•Use the **+ operator** to **concatenate** a string object with a string literal, a character, or another string object.

•Concatenation of numeric types must have the same data type. Easiest (least flexible) Cast is **std::to_string()** function template defined in the string header.

• Objects of type string have functions to search, modify, and extract substrings.

• string header offers functions: **std::stoi()** and **std::stod()** to convert strings to int and double.

- A powerful option to Read/Write numbers to a string is **std::stringstream**. You can use string streams in exactly the same manner as you would std::cout and std::cin.
- Objects of type **wstring** contain strings of characters of type wchar_t.
- Objects of type **u16string** contain strings of characters of type char16_t.
- Objects of type **u32string** contain strings of characters of type char32_t.

/**~~~~~~*/

CH. 8 – Defining Functions (314)

- **Functions:** self-contained compact units of code with a well-defined purpose. Create a large number of small functions, not a small number of large functions.
- A **function definition** consists of the **function header** that specifies the **function name**, the **parameters**, and the **return type**, followed by the **function body** containing the executable code for the function.
- **Function prototype** enables the compiler to process calls to a function w/o a definition defined yet.
- The **Pass-by-Value** passes Copies of the original values to a function, & original values aren't accessible from within the function.
- **Pass-by-Pointer** or **Pass-by-Reference** allows a function to change the Original Value.
- **const:** Prevents modification of the original value.
- Can pass the address of an array to a function as a pointer. If you do, you should generally pass the array's length along as well.
- Specifying a **parameter as a reference avoids the copying** that is implicit in the pass-by-value mechanism.

 - A reference parameter that is not modified within a function should be specified as const.
- **Input parameters** should be reference-to-const, except for smaller values such as those of fundamental types. Returning values is preferred over output parameters, except for very large values such as std::array<>.
- Specifying default values for function parameters allows arguments to be optionally omitted.
- Default values can be combined with **std::optional<>** to make signatures more self-documenting. std::optional<> can be used for optional return values as well.
- **Static Variables**: Created and Holds the initialized Value until the Program terminates. Carries values over like a Global Var, but only accessible and manipulatable through particular functions.
- Returning a reference from a function allows the function to be used on the left of an assignment operator. Specifying the return type as a reference-to-const prevents this.
- The **signature** of a function is defined by the function name together with the number and types of its parameters.
- **Overloaded functions**: Functions with the same name but with different signatures and therefore different parameter lists. Overloaded functions cannot be differentiated by the return type.
- **Recursive** function:A function that calls itself. Implementing algorithms recursively can result in elegant and concise code, but certainly not always, at the expense of execution time when compared to other methods of implementing the same algorithm.

/**~~~~~~*/

CH. 9 – Function Templates

- **Function template:** a parameterized recipe to generate overloaded functions.
- The **parameters** in a function template can be **type parameters** or **nontype parameters**. The compiler creates an instance of a function template for each function call that corresponds to a unique set of template parameter arguments.
- A function template can be **overloaded** with other functions or function templates.
- **auto** and **decltype**(auto) can be used to deduce the return type of a function. Powerful in the context of templates because their return type depends on the value of one or more template type arguments. The **typename** keyword identifies T as a type.

template T larger(T a, T b); // Prototype for function template

template <typename T> T larger(T a, T b) { return a>b ? a : b; } //Function Template Definition

/**~~~~~~*/

CH. 10 – Program Files and Preprocessing Directives

•Each **Entity** in a **Translation Unit** one definition. Multiple definitions are allowed throughout a program, if they're identical.

•TYPES LINKAGE: **Internal link.**: Name accessible through translation unit; **external link.**: name accessible from any translation unit; _or **No linkage**: name accessible only in block in which it is defined.

• **Header files** contain **definitions and declarations** required by your **source files**. A header file can contain template and type definitions, enumerations, constants, function declarations, inline function and variable definitions, and named namespaces. By convention, header file names use the **extension: .h**

•Your **Source Files** contain definitions for all **non-inline functions** & variables declared in corresponding header.

- A C++ source file usually has the file name **extension: .cpp**

• Insert **header file** contents into a .cpp file by using **#include "HeaderName.h"** directive.

•A **.cpp file** is the basis for a **translation unit** that is processed by the compiler to generate an **object file**.

• **Namespace** defines a **scope**; all names within a scope have namespace attached to them.

- All declarations of names that are not in an explicit namespace scope are in the global namespace.

•Single Namespace: Made up of several separate **namespace declarations** with the same name.

•Identical names that are declared within different namespaces are distinct.

•To refer to an **identifier** that is **declared** outside the needed namespace, specify the namespace name / identifier, separated by the **scope resolution operator**, **::** . Example::Namespace

• Names declared within a namespace can be used without qualification from inside the namespace.

•The **preprocessing phase** executes **directives** to transform the **source code** in a **translation unit** prior to compilation. When all directives have been processed, the translation unit will only contain C++ code, with no preprocessing directives remaining.

•Use **conditional preprocessing directives:** Ensure the contents of a header file are never duplicated within a translation unit.

•Use **conditional preprocessing directives:** To control whether **trace** or other **diagnostic debug code** is included in your program.

•**assert() macro** enables you to test logical conditions during execution and issue a message and abort if condition is false.

• **static_assert** = check type arguments for template parameters in instance so type argument is consistent w/template definition.

- Main debugging tools- **Tracing Program Flow / Setting Breakpoints / Setting Watches / Inspecting Program Elements**

-Random Number Generator

369 - This Seeds a Random Number (Allows it to be generated)

std::srand(static_cast<unsigned>(std::time(nullptr))); // Seed random generator

This Creates a Random Number from Range: 0 to HighestNumber

size_t random(size_t count) { return **static_cast(std::rand() / (RAND_MAX / HighestNumber));** **}**

Standard Preprocessing Macros

365 There are several standard predefined preprocessing macros, and the most useful are listed. They all have **2 Underscores before & also usually after the name**

MACRO	DESCRIPTION
__LINE__	The line number of the current source line as a decimal integer literal.
__FILE__	The name of the source file as a character string literal.
__DATE__	The date when the source file was preprocessed as a character string literal in the form Mmm dd yyyy. Here, Mmm is the month in characters, (Jan, Feb, etc.); dd is the day in the form of a pair of characters 1 to 31, where single-digit days are preceded by a blank; and yyyy is the year as four digits (such as 2014).
__TIME__	The time at which the source file was compiled, as a character string literal in the form hh:mm:ss, which is a string containing the pairs of digits for hours, minutes, and seconds separated by colons.
__cplusplus	A number of type long that corresponds to the highest version of the C++ standard that your compiler supports. This number is of the form yyyymm, where yyyy and mm represent the year and month in which that version of the standard was approved. At the time of writing, possible values are 199711 for nonmodern C++, 201103 for C++11, 201402 for C++14, and 201703 for C++17. Compilers may use intermediate numbers to signal support for earlier drafts of the standard as well.

You can use the date and time macros to record when your program was last compiled with a statement such as this:

```
std::cout << "Program last compiled at " << __TIME__ << " on "<< __DATE__ << std::endl;
```

/**~~~~~~*/

CH. 11 – Defining Your Own Data Types (pg.445)

- **Class** = way to define your own data types. Classes represent whatever objects your particular problem requires.
- Classes contain: **member variables** and **member functions**. The member functions of a class always have free access to the member variables of the same class, such as protected and private.
- **Objects**- (Instances)- created and initialized using member functions called **constructors**, called automatically when an object declaration is encountered. Can be overloaded to provide different ways of initializing an object.
- **Copy Constructor**- Constructor for an object, initialized with an existing object of the same class. The compiler generates a default copy constructor for a class if you don't define one.
- **Members** of a class can be specified as **1. public:** freely accessible in a program. **2. private:** accessed only by *member functions*, **friend functions**, or members of **nested classes**.
- Member variables of a class can be **static**. Only one instance of each static member variable of a class exists.
- **Static member variables** aren't part of the object and don't contribute to its size.
- All **non-static** member function contains the pointer **this->** ,or **(*this).** , points to the current object, which the function called.
- **Static member funcs** can be called even w/no objects of class are created. Doesn't contain the pointer *this*.
- **const member functions** can't modify, unless the member variables have been declared as **mutable**.
- Using references to class objects as args to func calls avoids alot of overhead in passing complex objects to a function.
- **Destructor-** member function, called object is destroyed. If not defined, the compiler supplies a default destructor.
- **Nested Class-** A class defined inside another class definition.

/**~~~~~~*/

CH. 12 – Operator Overloading (pg.488)

Make sure that your implementation of an overloaded operator doesn't conflict with what the operator does in its standard form.

You need to decide the nature and scope of the facilities each class should provide. Always keep in mind when defining a data type—a coherent entity—and that the class needs to reflect its nature and characteristics.

- You can overload any number of operators within a class for class-specific behavior. Do so to make code easier to read and write.
- **Overloaded operators** should mimic their built-in counterparts as much as possible.

Popular exceptions to this rule are the << and >> operators for Standard Library streams and the + operator to concatenate strings.

- **Operator functions**= members of a class || global operator funcs. Use member functions whenever possible. Resort to global operator functions if there is no other way or if implicit conversions are desirable for the first operand.
- For a **unary operator** defined as a class member function, the operand is the class object.

For a unary operator defined as a global operator function, the operand is the function parameter.

- For **binary operator functions** as a member of a class, the left operand for class object, the right operand for function parameter.

Binary operators defined by a global operator function, first parameter specifies left operand, second parameter is right operand.

- Functions implementing overloading of **+= operator** can be used to implement the **+ function**. This is true for all op= operators.
- To overload the increment or the decrement operator, you need 2 functions to provide prefix and postfix form.

The func to implement postfix operator has an extra parameter, type int, to distinguish func from prefix version.

- Customized type conversions, 2 choices: conversion operators || combination of conversion constructors and assignment operators.

/**~~~~~~*/

CH. 13 – Inheritance (pg.522)

Inheritance: derive a class based on one or more existing classes & is fundamental characteristic of OOP & polymorphism.

- A **class** may be **derived** from 1+ **base classes**, in which case the derived class **inherits** members from all of its bases.
- **Single inheritance-** deriving a class from a single base class. **Multiple inheritance**- deriving a class from 2+ base classes.
- **Access** inherited members 2 factors: **access specifier** in base class & **access specifier** of base class in derived class declaration.
- A constructor for a derived class is responsible for initializing all members of the class, including the inherited members.
- Creating derived class objects involves the constructors of all direct/indirect base classes, called in sequence (from the most base to the most direct) prior to the execution of the **derived class constructor**.
- Derived class constructors can & should explicitly call constructors for its direct bases in initialization list for constructor. If not called explicitly, base class's default constructor is called. Copy constructor in derived class should always call the copy constructor of all direct base classes.
- A member name declared in a derived class, which is the same as an inherited member name, will hide the inherited member. To access the hidden member, use the scope resolution operator to qualify the member name with its class name.
- Can use **using** for type aliases & inherit constructors (always same access as base class), modify the access specifications inherited members, or to inherit functions, hidden by a derived class's function with the same name but different signature.

•A derived class w/ 2+ direct base classes contain 2+ inherited subobjects w/same class, duplication is prevented by declaring the duplicated class as a **virtual base class**.

/**~~~~~~*/

CH. 14 – Polymorphism (pg.567)

•**Polymorphism-** calling a virtual member function of a class through pointer OR reference & the call resolved dynamically. The particular function to be called is determined by the object that is pointed to OR referenced during execution.
•A function in a base class can be declared **virtual**. All functions in classes, derived from the base, will then be virtual too.
•Always declare the destructor as virtual of classes as intended to be used as a base class as virtual (often done in combination with **= default**). Ensures correct selection of a destructor for dynamically created derived class objects.
•Should use the **override** qualifier for each derived class member that overrides a virtual base class member.
This causes the compiler to verify that the functions signatures in the base and derived classes are the same.
• **final** qualifier – can be used on a virtual function override to signal that it may NOT be overridden any further.
If an entire class is specified to be **final**, no derived classes can be defined for it anymore.
•Default argument values for parameters in virtual functions are assigned statically. If default values for a virtual function exist, default values specified in a derived class will be ignored for dynamically resolved function calls.
• **dynamic_cast<>() =** Cast from pointer-to-**polymorphic**-**base**-class to pointer-to-a-**derived**-class. If pointer does not point to an object of the given derived class type, **dynamic_cast<>** evaluates to **nullptr**. This type check is performed dynamically, at runtime.
•**Pure Virtual Function** – has a member value set to 0. Used as an Interface. Derived Virtual Functions must redefine the Base's.
• **Abstract Class** - A class with 1+ pure virtual functions.

/**~~~~~~*/

CH. 15 – Runtime Errors and Exceptions (pg.609)

- **Exceptions**: objects used to signal errors in a program.
- Code that may throw exceptions is usually contained within a **try block**, enables exceptions to be detected/processed.
- Code to handle exceptions, may be thrown in a try block is placed in one or more **catch blocks**, immediately following try block.
- A try block, along with its catch blocks, can be nested inside another try block.
- A catch block with a parameter of a base class type can catch an exception of a derived class type.
- **catch(…)** will catch an exception of any type.
- If an exception isn't caught by a catch block, **std::terminate()** function is called, immediately aborts the program execution.
- **Every resource** should always be Acquired / Released by an **RAII object**. DON'T use keywords **new** & **delete** in modern C++ code.
- Standard Library offers various RAII types you should always use including: std::unique_ptr<>, shared_ptr<>, and vector<>.
- **noexcept** indicates a function doesn't throw exceptions. If a it does, **std::terminate()** is called.
- If a destructor doesn't have an explicit noexcept specifier, the compiler will usually generate one. Never allow an exception to leave a destructor; otherwise, std::terminate() will be triggered.
- Std Library defines several exception types in **stdexcept header**, derived from **std::exception**, defined in exception header.

/**~~~~~~*/

CH. 16 – Class Templates (pg.658)

template <template parameter list> // General form of a class template looks like this: **Defining Class Templates**
class ClassName { // Template class definition... } ;

Template	**Result**
is_default_constructible_v<T>	Only True if type T is default constructible: has a no-arg constructor
is_copy_constructible_v<T>	Is True if type T is copy constructible: has a copy constructor
is_assignable_v<T>	Is True if type T is assignable: has an assignment operator function
is_pointer_v<T>	Is True if type T is a pointer type
is_null_pointer_v<T>	Is True only if type T is of type std::nullptr_t
is_class_v<T>	Is True only if type T is a class type

•**Class templates** define a family of class types.
•**Instance** of *class template* = **class definition**. Generated by compiler from template using a set of specified template args.
•**Implicit instantiation** of a class template arises out of a definition for an object of a class template type.
•**Explicit instantiation** of a class template defines a class for a given set of arguments for the template parameters.
•An *argument* for to a *type parameter* in a class template can be: fundamental type, class type, pointer type, or reference type.
•The type of a **nontype parameter** can be an integral or enumeration type, pointer type, or reference type.
•**Partial specialization** of class templates Defines a new template to be used, restricted subset of args for original template.
•**Complete specialization** of template Defines new type for a specific, complete set of parameter arguments for original template.
• A **friend** of a class template can be a function, a class, a function template, or a class template.
• An ordinary class can declare a class template or function template as a friend.

/**~~~~~~*/

CH. 17 – Move Semantics (pg.693)

Move operations- Since arguments are temporary, the function instead of copying data, it can steal it.
•**rvalue**: expression typically results in a temp val; **lvalue** is one that results in a more persistent value. Named Var.
•**std::move()**used to convert an lvalues (like variables) into rvalues. ONCE MOVED, objects shouldn't be used anymore.
•An **rvalue reference** type is declared using a **double ampersand, &&**.
•***Move Constructors*** & ***Move Assignment Operators*** have rvalue reference parameters, called for temp arguments.
•If *function copies an input*, ***passing arg by value*** is preferred, it caters lvalue & rvalue inputs w/a function definition.
This Also applies to Class Object
•Auto vars & function parameters should be *returned by value* and without adding std::move() to the return statement.
•Move members should normally be noexcept; if not, they may be invoked by STL containers and other templates.
•**Rule of five-** Declare all copy members, move members, and the destructor together, ***or none of them at all***.
•**Rule of zero-**Strives to define none at all. The means to achieve rule of zero compliance you actually already know: always manage dynamic memory and other resources using **smart pointers, containers,** & **other RAII techniques**!

/**~~~~~~*/

CH. 18 – First-Class Functions (pg.718)

Table 18-1. The Function Object Class Templates Offered by the <functional> Header

Comparisons Ops	**less<>, greater<>, less_equal<>, greater_equal<>, equal_to<>, not_equal_to<>**
Arithmetic Ops	**plus<>, minus<>, multiplies<>, divides<>, modulus<>, negate<>**
Logical Operations	**logical_and<>, logical_or<>, logical_not<>**
Bitwise Operations	**bit_and<>, bit_or<>, bit_xor<>, bit_not<>**

-Lambda expressions- have versatile & expressive syntax for defining anonymous Functions & to create lambda closures capable of capturing any number of variables from their surroundings. They are much *more powerful than function pointers*;
-Unlike function objects, they do not require you to specify a complete class.
•Pointer to a function **stores the address of any function**, with the specified return type and number and types of parameters.
•Can use pointer to a function to *call function* at the address it contains. Can also pass a pointer to a *function as a function arg*.

•**Functors**(Function objects)- objects that behave precisely like a function by overloading the function call operator.
•Any number of member variables / functions can be added to a function object, making them far more versatile than plain function pointers. For one, functors can be parameterized with any number of additional local variables.
•Function objects are powerful but a lot of set up. **Lambda Expressions-** alleviates defining class for each function object.
•**Lambda expression**= anonymous functions or function objects & typically pass a function as an argument to another function.
•Lambda expressions always begin with a **lambda introducer** '**[]**' that consists of a pair of square brackets that can be empty.
•Lambda [] can contain a **capture**, which specifies which variables in the enclosing scope can be accessed from the body of the lambda expression. Variables can be **captured** by *value* or by *reference*.
•2 Default Capture Clauses: **[=]** specifies all variables are *captured by value*. **[&]** specifies all variables are *captured by reference*.
•*Capture clause* can specify variables to be captured by *value or by reference*.
•Variables captured by value *has a local copy created*. The copy is not modifiable by default.
-Adding **mutable** keyword *after* the parameter list allows local copies of variables captured by value to be modified.
•If return type isn't specified, the compiler deduces the return type from the first return statement in the lambda.
-You can specify the return type for a lambda expression using this **trailing return type syntax**.
•Can use **std::function<> template type**, Inside -> **#include <functional>** to specify the type of a function parameter that will accept any first-class function as an argument, including lambda expressions. It allows you to specify a named type for a variable—be it a function parameter, member variable, or automatic variable—that can hold a lambda closure. Which would otherwise be very hard as the name of this type is known only to the compiler.

/**~~~~~~*/

CH. 19 – Containers and Algorithms (pg.756)

- Typical container, especially sequential containers can: add, remove, and traverse elements.
- **Algorithms**: HUGE Collection of generic, higher-order function templates. More advanced ways to manipulate the data.
- Consult **Standard Library reference**— all member functions of various containers, algorithm templates, and precise semantics.
- Sequence containers store data in a straightforward user-determined linear order, one element after the other.
- **Go-To Sequential Container** == **std::vector<>**. OTHERS == list<>, forward_list<>, and deque<>.
- **3 Container Adapters**—std::stack<>, queue<>, and priority_queue<>.
- **SETS** are duplicate-free containers & good at determining whether they contain a given element.
- **MAPS** associate Keys w/ Values & quickly retrieve a value given their keys.
- Sets & Maps: **Ordered** & **Unordered**. Ordered == sorted view; Unord. == may be more efficient w/ effective hash function.
- You & Std algorithms—can use **iterators** to enumerate elements of any container, without having to know the organization.
- **Iterators** make heavy use of operator overloading (look/feel like pointers).
- Std Library offers 100+ algorithms, most in the **algorithms header**. Some are defined by **numeric header** also.
- All algorithms operate on half-open ranges of iterators, and many accept a first-class callback function. Mostly call an algorithm with a lambda expression if its default behavior does not suit.
- Algorithms to retrieve a single element from a range (**find(), find_if(), min_element(), max_element(), and ...**) OR do so by **returning** an **iterator**. The end iterator of the range is then always used to denote "not found."
- Algorithms that produce multiple output elements (**copy(), copy_if(), and ...**) should normally always be used in conjunction with the **std::back_inserter()**, **front_inserter(),** and **inserter()** utilities provided by the **iterator header**.
- To remove multiple elements from sequence containers, you should use the **remove-erase idiom**.
- Take advantage of the extensive **multiprocessing** capabilities of current hardware by passing the **std::execution::par** (execution policy) as the 1st argument to most algorithms.

INDEX (pg.761) (KEYWORDS & PG #)