Beginning C++

2023 Guide

Content

Chapter 1: Structure of a C++ Program

Programming Language:

A Programming language is a specific set of Keywords, Identifiers, Punctuation, Operators, and Syntax that are put together to create the building blocks of a program.

- **Keywords** are part of the vocabulary of a programming language. These words are reserved, meaning programmers cannot redefine their meaning or use them in ways to which they are not intended

- **Identifiers** are names created by programmers to represent something meaningful to them

- **Punctuation** is the symbols used to break up sections of a programming language for readability

- **Operators** are symbols that represent actions useable on certain data

- **Syntax** is the set of rules that govern the Symbols, Punctuation, and words for a programming language and how it is written out

C++ Language Examples:

`endl`	**endl** is the identifier for a standard function to insert a newline character
`int`	**int** is the keyword in C++ to represent an integer value
`+`	**+** is the mathematical operator for addition
`;`	**Semicolons** are the punctuation marker to denote the end of a statement

Data Types:

Fundamental data types:
Contain a different number of bits in memory for each type. Each data type has different rules for operations that can be done to them.

Integers - Whole Numbers
- Short
- Int
- Long
- Long Long

Unsigned Integers
- Only positive numbers
- Gives twice as many positive numbers

Floating point numbers - Fractional Numbers
- Float
- Double
- Long double

Characters - Letters
- english alphabet of letters
- differentiate between upper and lower case

Boolean Values
- True or False values

Comments:

Comments are human readable parts of code that are ignored by the preprocessor. These are usually descriptions of how a part of code works and helps with the readability of your code. Commenting is good practice so that other people looking at your code can easily figure out how it works and how to fix potential errors.

Comments in C++	
`// Comment Text`	- single line comment starting with the `//` operator

`/*`
`* Multi`
`* Line`
`* Comment`
`*/`
	- multi line comments are started with the `/*` operator and ends with a `*/` operator. each line in the comment will start with a *

Preprocessor and Preprocessor Directives:

The C++ preprocessor is a program that processes your source code before the compiler receives it. It first removes all comments from the source file, replacing them with a single space. Then it looks for preprocessor directives and executes them.

Compiler:
a program that converts source code into machine readable code (Binary)

Preprocessor Directives:
Start with the # symbol. These directives tell the preprocessor to do specific action before compiling the program

Preprocessor Directives:	
`#include`	- the preprocessor directive to include another file in your program
`#include <iostream>`	- uses the **include** directive to include the file **<iostream>** which has functionality for inputting and outputting data.

Basic Input/Output (I/O):

The basis of programming is taking in information from the user and printing out information to the user.

C++ I/O:

```
11
10
9
8
7
6
5
4
3
2
1
1
2
3
4
5
6
7
8
9
10
```

Summary

Outline

Beginning C++

Content

Accessing array elements:

Accessing an array:

Multi-Dimensional Arrays:

Multi-Dimensional Arrays:

Accessing Multi-dimensional arrays:

Vectors:

Declaring a Vector:

Accessing Vector Elements

Array syntax

vector syntax

Changing Vector Elements

Assignment Operator

Vector Pushback

Out of bounds

Chapter 4: Statements and Operators

Expressions:

Expression Examples

Statements:

Statement Examples

Operators:

Assignment Operator

Selection:

Selection statements

Nested-if statements

Switch statements

Conditional Operator

Conditional Operator Example

Iteration:

For Loop

For Loop Initialization

For Loop using an array

Comma operator

Comma operator Example

Using a vector

Range-based for loop

Range-based for loop Example

initialize a list within a for loop

With strings and characters

While loop

While loop Example

While loop Example 2

Input validation Example

Assignment

Concatenation - the building of a string from other strings

Accessing Characters in strings

Comparing strings

Substrings

Removing characters

Length of a string

Searching

input >> and getline()

Chapter 7: Functions

Functions:

Using Functions example:

Modularization:

Writing a function:

Function Example:

User Defined Function

User Defined Function 2

Random Numbers:

Defining Functions

Function Syntax

Chaining Functions

Calling functions

Function call Error

Function prototype

Function order

Pass by Value

Function Return Statement

Default Argument Values

No Default Arguments

Default Arguments

Multiple Default Arguments

Overloading Functions

Return type is not considered for overloaded functions so the function must take in some differentiating arguments or the compiler won't be able to tell the difference.

Overloading Example

Passing an Array - Error

Passing an Array

Changing an array

Pass by Reference Example

Swap Example

Vector Example

Scope Rules

Pointers

Implementing Member Methods:

Header and Implementation Files

Include Guard

Constructors and Destructors:

Constructor Example

Destructor

Default Constructor:

Overloading Constructors:

Overloading Example

Constructor Initialization Lists:

Delegating Constructors:

Delegating Example

Default Constructor Parameters:

Copy Constructor:

Implementing Copy Constructor

Shallow Copy

Shallow copy Example

Deep Copy:

Deep Copy Example

Move Constructor:

r-value references:

Reference Example

'this' pointer:

Using constants with classes:

Static Class Members:

Player class - static members

Struct vs Class:

Friends of a class:

friendship - functions

Chapter 10: Overloading

Operator Overloading:

Overloading example

Mystring class example - models a string using raw c-style pointer

Overloading Assignment operator:

Overloading move operator:

Overloading operators as member methods:

Mystring operator- function

Overloading binary operators:

Mystring operator+ concatenation

Overloading operators as global functions:

Overloading insertion and extraction operators:

overload stream insertion operator

overload stream extraction operator

Chapter 11: Inheritance

Inheritance:

Class Hierarchies

Hierarchy example

Composition Example

Derivation Syntax

Protected Members:

Constructors and Destructors - Inheritance:

Copy/Move constructors and overloaded assignment

Using and redesigning Base class methods:

Static Binding of method calls:

Multiple Inheritance:

Chapter 12: Polymorphism

Polymorphism:

Non-Polymorphic example - static binding

Static Binding

Polymorphic example - dynamic binding

Dynamic Binding

Base Class Pointer

Declaring Virtual Functions

Virtual destructor

Override specifier - redefinition example

Override specifier example

Final specifier:

Final Specifier Syntax

Final Specifier Example

Base class references:

Base class reference example

Pure virtual functions and Abstract classes:

Shape class example

Abstract classes as interfaces:

Printable Interface Example

Shape Example

Shape Example - Pointers

Chapter 13: Smart Pointers

Issues with Raw Pointers:

C++ Smart Pointers

Smart pointer - simple example

Resource Acquisition is Initialization (RAII):

Unique Pointer:

Files, Streams, and I/O:

Common Header Files

Common Stream Classes

global stream objects

Stream Manipulators:

Common Stream Manipulators:

Stream Manipulators - Boolean:

Boolean Example

Stream Manipulators - Integers:

Formatting Integers

Display Hex in Uppercase

Displaying positive sign

Stream Manipulators - Floating point:

Precision

Fixed

Scientific

Trailing Zeros

Stream Manipulators - Field width, Align and Fill:

Defaults

Field Width

Left Justify

Reading Input From Files:

Opening a File

ifstream

Open a File for Reading

Check if File Opened Successfully

Closing a File

Reading From Files Syntax

Reading From Files Example

Output files:

Opening Output File

Checking if Output File is Open

Writing to an Output File

Copying Files

String streams:

Writing to a Stringstream

Data Validation

Chapter 16: Standard Template Library

Subsection Title:

Non-Polymorphic example - static binding

Non-Polymorphic example - static binding

Non-Polymorphic example - static binding

Extra Chapter 1: Lambda Expressions

Subsection Title:

Code Example Title

Extra Chapter 2: Enumerations

Subsection Title:

Code Example Title

Chapter #: Chapter Title Template

Subsection Title:

Code Example Title

Definition Glossary

`cout`	- The basic object representing a stream of information being **output to the console**
`<<`	**-** The insertion operator to be used with ***cout***. information following this operator is output to the console
`cin`	**-** The basic object representing a stream of information being **input from the user**
`>>`	**-** The insertion operator to be used with ***cin***. information following this operator is input into the program

Example of basic C++ code:

`#include <iostream>`	- **#include** is used to include functions from C++ libraries.
`using namespace std;`	- **using namespace** is a way to tell the compiler to use a specific set of identifiers. (not recommended to use **std;**)
`int main( )`	- All C++ programs start with a main function. written as: ***(return data type) main( )***
`{`	- **function blocks** are denoted by curly braces. all code inside is run when a function is called
`int Age;`	- a **statement** in a function. - storing an integer inside a variable called ***Age*** with no value inside. - statements are separated by a semicolon
`(std::)cout << Age <<(std::)endl;`	- calling the function ***cout*** which is in the ***std*** namespace. - using the output operator **<<** to tell the compiler we want to output the value stored at ***Age*** to the console - ***endl;*** is an endline function in the ***std*** namespace used to print a newline
`}`	- close the function in a final curly brace to end it

Chapter 2: Variables and Constants

What is a Variable:

To understand what a variable in programming is you should understand the basic concept of memory and data transfer in a CPU(Central Processing Unit)

- Random Access Memory (RAM) is a contiguous block of storage used by the computer to store information. This information can be instructions or data. The basic layout of RAM is a database of cells that store information along with a location identifier for that information.

Move item 21 to location 1001

question:
What is 21?

Memory	
Information	**Memory Location**
Null	**1000**
21	**1001**
Null	**1002**
Null	**1003**
Null	**1004**
Null	**1005**
Null	**1006**
...	**...**

Null is the representation of **no data** in a computer.

Using a Variable name to associate a Memory location

Assign 21 to Age (location 1001)

or

The value at Age (location 1001) is 21

Memory

Variable	Information	Memory Location
Null	Null	1000
Age	21	1001
Null	Null	1002
Null	Null	1003
Null	Null	1004
Null	Null	1005
Null	Null	1006
...	...	...

This is called **binding**, where **the variable name is bound to the memory location** and **not to the information itself**. This lets programmers change the information stored in these memory locations using meaningful names to represent data.

Declaring a Variable:

Variables have:

- **Type** - their category or data type (integer, real number, string...)
- **Value** - the contents (10, 3.14, "Frank"...)

When declaring a variable we provide a name and data type

- We can also provide a starting value as well (initializing a variable)
- Names should be meaningful to the information being stored in that variable. This helps when people have to read your code to fix potential errors.
- Variable names cannot begin with a number, only a letter or underscore
- Variable names cannot use Keywords
- Variable names in C++ are case sensitive

If you wanted to store the value of the mass of planet earth:	
`int MOE;`	- a variable called **`MOE`** that stores integers. Not very descriptive.
`int MassOfEarth;`	- a variable called **`MassOfEarth`** that stores integers. Easily understood by people reading your code

Note the way variables are commonly written: The way you write variables is up to you, I will be using Pascal Case in the rest of this guide.	
Camel Case	**`variableName`**
Snake Case	**`variable_name`**
Pascal Case	**`VariableName`**

Initializing a Variable:

When you initialize a variable, you are setting a specific starting value to that memory location. A variable can also be uninitialized by not setting a starting value, this means the computer does not know what is in that memory location.

Initializing a Variable Examples:	
`int Age;`	• An **uninitialized** variable called **Age** • The value of age is unknown by the computer
`int Age = 21;`	• Using C-like initialization for a variable called **Age**
`int Age (21);`	• Using Constructor initialization for a variable called **Age**
`int Age {21};`	• Using C++ initialization for a variable called **Age**
The equal sign **(=)** is called an operator in programming. Specifically an assignment operator. More information in later chapters.	

Global Variables:

- Declared outside a function and can be accessed anywhere in the program

Local Variables:

- Declared within a specific function and can only be accessed by that function

Integer Overflow:

If an **integer** value takes more or less bits than the allocated number of bits, then we may encounter an **overflow** or **underflow**.

- The integer **overflow** occurs when a number is greater than the maximum value the data type can hold
- The integer **underflow** occurs when a number is smaller than the minimum value the data type can hold.

Constants:

Constants are Variables with a set value that is read-only in memory, meaning it can't be changed after being initialized.

How to initialize a constant:	

`const datatype VariableName;`
 	
`const int Age;`
	- same as initializing a variable, just starting with the keyword **`const`**
	- we make the **`Age`** variable a read only value by initializing it as a constant

When we define a constant we provide the data type, name, and value

- The value isn't optional for a constant because after the variable is initialized, it becomes read-only.

(sizeof) operator C++

`sizeof (VariableName)`	- calling the ***sizeof*** function for a specific variable
`sizeof (int)`	- Calling ***sizeof*** the variable that stores an ***integer***

The ***sizeof*** operator gets information from the ***<climit> and <cfloat>*** header files. These include files give us some defined constants like:

- ***INT_MAX -*** a constant representing the maximum value we can store in an integer on the machine.

- **String literals** are enclosed in double quotes ***"StringLiteral"***
- **Character literals** are enclosed in single quotes ***'CharacterLiteral'***
- **Integer literals** are enclosed in curly brackets ***{IntegerLiteral}***

Chapter 3: Arrays and Vectors

Arrays:

An array is a compound data type that is a collection of elements:

- All elements in an array **must** be of the same type
- each element can be accessed directly by their position or **index**
- arrays are fixed size when defined (created)
- the **first** element in an array is at **index 0**
- last element in the array is at **index-1**
- **No** checking to see if you are out of bounds

Declaring an array:	
`int  Numbers[100];`	- we start with the data type the array will store **(int)** - followed by the variable name **(Numbers)** - then the number of elements in the array **(100)**
This will give you an array of **100 integers** in the variable **Numbers.** This collection of numbers can then be passed to other functions, making accessing large amounts of contiguous information easy.	

- Adding an initializer list after the declaration lets you set the elements in an array

Initializing an array:	
`Int TestScores[5]{100,95,99,90,83};`	- an array of **integers** called **TestScores** with a length of **5** and the values are **initiated** to **(100,95,99,90,83)**
`Int HighScorePerLevel[10] {3,5};`	- an array of **integers** called **HighScorePerLevel** with a length of **10** and the **first 2** values are **initiated** to **(3,5)** while the rest are initiated to **(0)**
`Int LastArray [] {1,2,3,4,5};`	- Without specifying the array length, it will automatically be calculated based on given array elements

Accessing array elements:

Accessing an array:	
`TestScores[2] = 99;`	- in the array **TestScores** at the **3rd** index location, the value there is set to **99**
`HighScorePerLevel[0] = 3;`	- in the array **HighScorePerLevel** at the **first** index location, the value there is set to **3**

- The name of the array represents the **location in memory** of the first element in the array **(index 0)**
- The index value you give the compiler tells it the offset from the beginning of the array to look for
    - **TestScores[2]** tells the compiler "go **2** spots starting from **0**" since the first item is actually located at **index 0**, the item at **index 2** actually holds **item #3** in the array. so there is an offset of **2.**
- **C++** performs a calculation to find the correct input. Since there is no bounds checking, you need to make sure the index you are asking for is within the bounds of the array.
    - You **can** ask for **TestScores[10]** even if **TestScores** only has a length of **[7]** but you will get **junk data** since you don't know what is being stored outside of the array in memory.

Multi-Dimensional Arrays:

- An array of arrays. It stores data in a table-like format (rows/columns)

Multi-Dimensional Arrays:	
`MultiArray[5][10];`	- An array called **MultiArray** is called with **5 rows** and **10 columns**

`Const int rows {3};`
`Const int cols {4};`
`int MovieRating [rows] [cols];`
	- 2 **constant** integer are named and initialized to **3** and **4**
	- an array called **MovieRating** is initialized with the values of the constants stated above, giving it **3 rows** and **4 columns**

Accessing Multi-dimensional arrays:	
`std::cout >> MovieRating [1] [2];`	- prints out the value at **row 1, column 2**

`Int MovieRating [3] [4]`
`{`
`  { 0, 4, 3, 5 },`
`  { 2, 3, 3, 5 },`
`  { 1, 4, 4, 5 }`
`};`
	- Visual way to represent a 2D array
	- **MovieRating** is an **integer** array composed of **[3]** different arrays with a length of **[4] each**

Vectors:

Vectors are **dynamic arrays** with the ability to **resize itself automatically** when an element is **inserted or deleted**, with their storage being handled automatically by the container.

- Vector elements are placed in contiguous storage
- Data is **inserted** at the **end**
- Members are **automatically set to 0** when created
- **Can provide bounds checking**
- Has some useful functions for use on the elements inside the Vector
- Are **considered objects in C++**

Declaring a Vector:	
`#include <vector>`	- must include the header file **<vector>** to use vectors
std::vector <char> Vowels;	- if you do not include **using namespace std;** in your program, you will need to use the standard prefix when declaring vectors
std::vector <char> Vowels{‘a’,‘e’, ‘i’,‘o’,‘u’};	- declares a vector of **5 characters** called **Vowels** with the **5 values set to a,e,i,o,u,** each member separated by commas and **single quotes** (for **characters**)
std::vector<int> TestScores (10);	- declares a vector of **integers** called **TestScores** with all **10 members** set to **0**
std::vector <double> HiTemps (365, 80.0)	- declares a vector of **doubles** called **HiTemps** with **365 members** and **each member** has a value of **80.0**

Accessing Vector Elements

Array syntax	
`VectorName [element index]`	- similar to accessing the elements of an array
`TestScores [1];`	- access the item at index **[1]** in the vector **TestScores**.

vector syntax	
`VectorName.at(element index)`	- start with the name of the vector you want to search, followed by the **.at** operator and **(index number)**
`TestScores.at(1)`	- access the item at index **(1)** in the vector **TestScores**.

Changing Vector Elements

Assignment Operator	
`TestScores.at(0) = 90;`	- Assign 90 to index **[0]** in the vector **TestScores**.

Vector Pushback	
`VectorName.push_back(element)`	- start with the name of the vector you want to add to, followed by the **.push_back** operator and **(element to add)**
`Vector <int> TestScores {100, 95, 90};`	- initiate a vector of **integers** called **TestScores** initiated to **(100,95,90)**
`TestScores.push_back(80);`	- in the vector **TestScores**, add **(.puch_back) 80** to the **end** of the vector

The vector will automatically **increase** the **size** and **add** the **element** to the **end** of the vector.

Out of bounds

`Vector <int> TestScores {100, 95};`	- initiate a vector of **integers** called **TestScores** initiated to **(100,95)**
`Cin >> TestScores.at(5);`	- trying to call item at **index (5)**.
`ERROR (std::out of range)`	- Exception error generated

If you go out of bounds when working with vectors, an exception error message will be generated.

Chapter 4: Statements and Operators

Expressions:

An expression is one of the most basic parts of any program. It is a certain sequence that **is** a value or **equates to** a value.

Expression Examples	
`34`	**literal expression**
`FavouriteNumber`	**variable expression**
`1.5 + 2.8`	**addition expression**
`A > B`	**relational expression**
`A = B`	**assignment expression**

Statements:

A statement is a complete line of code that performs an action.

- **Terminated with a semicolon**
- **usually contains expressions**
- **different types of statements in C++**

Statement Examples	
`Int x;`	declaration statement
`FavouriteNumber = 12;`	assignment statement
`1.5 + 2.8;`	expression statement
`X = 2 * 5;`	assignment statement
`If (a < b) cout << b;`	if statement (boolean)

Operators:

An operator is a symbol in a programming language that represents a mathematical or logical function which when executed will equate to a value.

- **Common operators:**
  - **Assignment**
  - **Arithmetic**
  - **Incremental / Decremental**
  - **Relational**
  - **Logical**
  - **Bitwise (more info in later chapter)**

Assignment Operator	
`LHS = RHS`	**RHS** is an **expression** that is **evaluated** to a certain value. That **value** is **assigned** to the memory location of the variable **LHS**

- The value of **RHS** must be compatible with the type of **LHS**
- **LHS** must be **assignable** (a place in memory)
- More than 1 variable can be assigned in a single statement
- The **left side** of the operation must have an **lvalue** (location in memory) so it **can't be a literal** (eg. 100 = num1)

Arithmetic Operators	
+	**Addition operator**
–	**Subtraction operator**
*****	**Multiplication operator**
/	**Division operator**
%	**Modulo operator (remainder of division)**
• Variables can be assigned with arithmetic operators	
Result = 5 * 10;	- The memory location at **Result** is **assigned** to the **value** the right hand side **equates** to **(50)**

Increment / Decrement Operators	
`++`	**Increase variable by 1**
`--`	**Decrease variable by 1**

- Can be used with **Integers, floats, and pointers**
- Can be **Prefixed** or **Postfixed**
  - can have different effects depending on use
  - **Prefix -** increase the value by 1 **then** return the value
  - **Postfix -** return the value **then** increase the value by 1

`++Num`	**Prefix increment**
`Num++`	**Postfix increment**

- Don't overuse this operator, and never use it twice for the same variable in the same statement

Increment Example

`int Counter {10};`	- Initiate a variable **Counter** to **10**
`Counter++;` `Cout >> counter >> endl;`	- Increment **Counter** and return the **value**
`++Counter;` `Cout >> counter >> endl;`	- Increment **Counter** and return the **value**

Prefix Example	

`int Counter {10};`
`int Result {0};`
 	
`Result = ++counter;`
	- Initiate a variable **Counter** to **10**
	- Initiate a variable **Result** to 0
	
	- increment **Counter** (10 + 1) then store the **value** in **Result**

Postfix Example	

`int Counter {10};`
`int Result {0};`
 	
`Result = ++Counter;`
	- Initiate a variable **Counter** to **10**
	- Initiate a variable **Result** to 0
	
	- **Store** value of **Counter** in **Result** then increment counter

Adding other Operators	

`int counter {10};`
`int result {0};`
 	
`Result = ++counter + 10;`
	- Initiate a variable **Counter** to **10**
	- Initiate a variable **Result** to 0
	
	- increment **Counter** by **1**, then calculate the **+ 10** and store that **value** into the memory at **Result**

`int counter {10};`
`int result {0};`
 	
`Result = counter++ + 10;`
	- Initiate a variable **Counter** to **10**
	- Initiate a variable **Result** to
	
	- **Counter + 10** is stored into memory at **Result**, then **Counter** is incremented by **1**

Mixed Type Expressions:

C++ operations occur on same type operands

- If operands are of different types, C++ will convert one
- If it can't convert, a compiler error will occur

Conversions

`Long double, double, float, unsigned long, long, unsigned int, int`

`short` and **`char`** types are always converted to **`int`**

Higher vs Lower types are based on the size of the values that type can hold

Type Coercion - Conversion of one operand to another data type

- **Promotion** - conversion from lower to higher type
- **Demotion** - conversion from higher to lower type

`lower type (operator) higher type`	**the lower is promoted to a higher type**
`2 * 5.2`	**2 is promoted to 2.0**
`lower = higher;`	**the higher type is demoted to a lower type**
`Int num {0}; Num = 100.2;`	**Num** is set to an **int**, but given a **double value** it is read as **100** (double demoted to int)

Conversions

```
int TotalAmount {100};
int TotalNumber {8};
double Average {0.0};
```

- Initialize an integer **TotalAmount** to **100**
- Initialize an integer **TotalNumber** to **8**
- Initialize a double **Average** to **0.0**

```
Average = TotalAmount /
TotalNumber;
```

- **assign** the memory spot at **Average**(higher type) to the **value** of **100/8**(lower type)

```
Cout << Average << endl;
```

- prints out **12** (double demoted to int)

Equality Operators

`LHS == RHS`	**LHS** is the same value as **RHS**
`LHS != RHS`	**LHS** is not the same value as **RHS**

- Compares the **values** of **2 expressions**
- evaluates to a **boolean** value (**True or False / 1 or 0**)
- commonly used in control statements

Equality Examples	

`bool Result {false};`
 	
`Result = (100 == 50+50)`
	- initiates a boolean variable called **Result** with a default value of **false(0)**
	- **Assigns** the **Result** to the **value** of the equation **(100 is the same as 50+50) true**

`Result = (Num1 != Num2);`	- we compare if **Num1 is not** the same as **Num2,** and save that **true/false** answer to **Result**

`std::boolalpha;`
 	
`Cout<<(Num1 == Num2)<< endl;`
	- Using the standard function **boolalpha;** we can change the **1 or 0 output** from a boolean result to a **true or false output**

Relational Operators	
LHS>RHS	**LHS** is greater than **RHS**
LHS>=RHS	**LHS** is greater than or equal to **RHS**
LHS<RHS	**LHS** is less than **RHS**
LHS<=RHS	**LHS** is less than or equal to **RHS**
LHS<=>RHS	- In C++20 the 3-way operator is included - comparison between **2 expressions**, - **returns 0 if both are equal** **- returns less than 0 if the left side is greater than right** - **returns greater than 0 if the right side is greater than left**

Logical Operators

!	**(not) - negation**
&&	**(and) - logical AND**
||	**(or) - logical OR**

NOT operator

- If ***Expr1*** is true then ***!Expr1*** is false
- And if ***Expr1*** is false then ***!Expr1*** is true

AND operator

- If ***Expr1*** is true and ***Expr2*** is true, then ***Expr1*** && ***Expr2*** is true
- If ***Expr1*** is true and ***Expr2*** is false, then ***Expr1*** && ***Expr2*** is false
- If ***Expr1*** is false and ***Expr2*** is true, then ***Expr1*** && ***Expr2*** is false
- If ***Expr1*** is false and ***Expr2*** is false, then ***Expr1*** && ***Expr2*** is false

Will only return true if BOTH expressions are true.

OR operator

- If ***Expr1*** is true and ***Expr2*** is true, then ***Expr1*** && ***Expr2*** is true
- If ***Expr1*** is true and ***Expr2*** is false, then ***Expr1*** && ***Expr2*** is true
- If ***Expr1*** is false and ***Expr2*** is true, then ***Expr1*** && ***Expr2*** is true
- If ***Expr1*** is false and ***Expr2*** is false, then ***Expr1*** && ***Expr2*** is false

Will only return false if BOTH expressions are false.

Logical Operator Precedence

- **NOT** has higher precedence than **AND**
- **AND** has higher precedence than OR

- **NOT** is a unary operator
- **AND / OR** are binary operators

`Num1 >= 10 && Num1 < 20`	will come back true if **Num1** is both **greater to or equal to 10 AND less than 20**
`Num1 <= 10 || Num1 <= 20`	will come back **true** if **either side** of the expressions is **true**
`Expr1 && Expr2 || Expr3`	Use Precedence rules when adjacent operators are different
`Expr1 && Expr2 && Expr3`	Associativity is usually Left to Right

Short-circuit Evaluation	
When evaluating a logical expression C++ stops as soon as the result is known	
`Expr1 && Expr2 && Expr3`	if **Expr1** is **false**, then it **won't calculate Expr2 or Expr3** for efficiency
`Expr1 || Expr2 || Expr3`	if **Expr1** is **true**, then it **won't calculate Expr2 or Expr3**

Compound Operators	
`A += B`	**increment A by B** and store **value in A**
`A -= B`	**Decrease A by B** and store **value in A**
`A *= B`	**Multiply A by B** and store **value in A**
`A /= B`	**Divide A by B** and store **value in A**
`A %= B`	**Divide A by B** and store the **remainder in A**

Chapter 5: Program Flow

Controlling Program Flow:

- **Sequence**
  - Ordering statements sequentially
- **Selection**
  - Making Decisions
- **Iteration**
  - Looping or repeating

Selection:

Selection statements	
`if (Expr1) {statement};`	Execute the **statement** code if **Expr1** is **true**, **skip if false**
`if (Expr1) {statement}; else {statement2}`	Execute the **statement** code if **Expr1** is **true**, **skip if false** Execute the **statement2** code if **statement** was skipped

Nested-if statements

```
if (expr1)
  if(expr2)
     {statement1};
  else
     {statement2};
```

A **Nested-if** statement is **nested inside another if statement**. This allows testing of **multiple conditions**. **Else belongs to the closest nested-if**

```
if (score > 90)
{
  if(score > 95)
    {std::cout<<“A+”;};
  else
    {std::cout<<“A”;};
}
else
{
   std::cout<<"Sorry, No A";
}
```

if score is **greater than 90**, run code block below

if score is **greater than 95**, print out "A+"

if **score is less than 95**, print out "A"

if **score is less than 90**, print out "Sorry, no A"

Switch statements

```
switch (integerControlExpr)
{
     Case Expr1: Statement1;
     Break;
     Case Expr2: Statement2;
     Break;
     Case Expr3: Statement3;
     Break;
     Case ExprN: StatementN;
     Break;
     Default: StatementDefault;
}
```

control statement must evaluate to int

the **case statements** must evaluate to **int or int literals**

you should **include break** statements for **each case statement**

You can have **many cases and statements**

Default statement is run if **no cases match** the **control statement**

Once a **match** is made **between the control expression and a case statement**, all the code following the **semicolon** will be **executed** until a **Break;** is reached.

Fall-through is when you **don't include breaks** after each case statement, meaning **if the first case statement matches** the control, **all the following statements** will run as well.

Conditional Operator	
?	**Conditional Operator Symbol**
`(CondExpr) ? Expr1 : Expr2`	**Conditional Expression** is evaluated **first** and must be evaluated to a **bool**. ● **If true** the **CondExpr** returns the value of **Expr1** ● **if false** it returns the value of **Expr2**

- **Is similar to `if-else` construct**
- **Ternary Operator**
- **Useful when used inline**
- **Easy to abuse**

Conditional Operator Example	
`Result =(a>b) ? a:b;`	if a is greater than b (conditional expression is true) then a is passed as the value of Result
`Result =(a<b) ? (b-a):(a-b);`	if a is less than b then the value of (b-a) is passed as the value of Result
`Result =(b!=0) ? (a/b):0;`	if b is not equal to 0 then the value of (a/b) is passed as the value of Result

Iteration:

The act of repetition or looping

- Basic building block of programming
- Allows execution of a statement or block of statements repeatedly
- Loops are made up of a loop condition and the body which contains the statements to repeat

Typical use cases:

- Loop a specific number of times
- Loop for each element in a collection
- Loop while a specific condition is true or false
- Loop until we reach the end of some input stream
- Infinite loop

For Loop

```
for (initialization; condition;
increment) statement;
```

for loop **starts with the for keyword**

followed by an **initialization expression**

then a **conditional expression** that evaluates to a **boolean**

then an **increment expression.**

1. The **initialization expression** is executed once
2. then the **conditional expression** is checked and **if true** the body of the loop is executed.
3. Then the **increment expression** is executed and the **conditional expression is checked again**.
4. When the **conditional expression is false the loop stops**.

```
int i {0};
```

- initialize an **int** variable **i** set to **0**

```
for (i = 1; i <=5 ; ++i)
      Cout << i << endl;
```

- **i begins at 1**
- **as long as i is less than or equal to 5**
    - **print out i**
    - **then increment i**
    - **then check if i is still less than 5**
    - **repeat until i > 5**

For Loop Initialization

```
for (int i {1} ; i <=5 ; ++i)

Cout << i << endl;
```

You can **declare and initialize** the **looping variable** in the **looping statement (initialization style)**

```
for (int i = 1 ; i <=5 ; ++i)
     Cout << i << endl;
```

Assignment style

For Loop using an array

```
Int Scores [ ] {100, 90, 87};
```

```
for (int i {0} ; i < 3 : ++i)
  {
       cout << Scores[i] <<
  endl;
  }
```

```
for (int i {0} ; i <= 2 : ++i)
  {
      cout << Scores[i] << endl;
    }
```

initialize array **Scores** with **3 values set**

---

sets **i to 0**

then **checks** if **i is less than 3**

then **prints** the corresponding **value** at that **index location in scores**

---

same as above but sets the condition to **check** if the **value is less than or equal to 2**

(not recommended, should always be less than your array length)

Comma operator

```
for (int i {1}, j {5} ; i <= 5 ;
++i, ++j)
```

You can set **more than one loop variable** by separating with a **comma**

Sets **i to 1**, and **j to 5**

then checks if **i is less than or equal to 5**

then **increments each value by 1** with each iteration.

Comma operator Example

`for (int i{1}; i <= 100; i++)`	sets **i to 1** checks if **i is less than or equal to 100** **if true** executes the following code
`{` `  cout << i;`	Prints out i

`  if (i % 10 == 0)`
`  cout << endl;`
`  Else`
`  Cout << " ";`
`};`
	Checks if the **remainder** of **dividing i by 10** is **0** (**every 10 items**)
	
	if true prints out a new line character
	
	If false it prints a space

`cout << endl;` `Return 0;`	End of program

`for (int i{1}; i <= 100; i++)`
`{`
`cout<<i<<((i%10==0) ? "\n":" ";`
`}`
	You can make the previous code more compact by using a **conditional operator**
	
	this code does the same as the above

Using a vector

```
vector <int> Nums {10, 20, 30
,40, 50};
```

initiate a vector of **ints** with **set values**

```
for (int i{0}; i < Nums.size();
++i)
  cout << Nums[i] << endl;
```

Sets **i to 0**

then **checks if i is less than the total size** of the vector

if it is it will **print out the value** at that vector at index **i**

then **increment i** and **loop** until the last entry in the vector is reached.

(note: nums.size() returns an unsigned int because vector size can't be negative.)

Range-based for loop

```
for (VarType VarName: sequence)
  statement;

for (VarType VarName: sequence)
{ statements; }
```

Using a range based for loop lets you easily iterate through a collection of a certain type of item

Range-based for loop Example

`Int scores [ ] {100, 90, 92};`	defines an array of **scores** with **set values**
`for (int score: scores)`	initialize an int called **score** and checks the type against each **entry in the array scores**
`cout << score << endl;`	print each **value of score** (each member of the array)
`for (auto score: scores)`	**optional** to use. This will let the compiler look at the array and match the type

initialize a list within a for loop

```
Double AverageTemp {};
Double RunningSum {};
Int Size {0};
```

define 2 **double** variables called **AverageTemp** and **RunningSum**

define an **integer** variable called **Size** and **initiate** it to **0**

```
For (auto Temp: {60.2, 80.1,
90.0, 78.2} )
```

The **list** is automatically **initiated** within the for loop and these **values** are then **passed to Temp.**

```
{
  RunningSum += Temp;
  ++Size;
};
```

for each **value** in the **list**, pass it to **Temp** then **add the value to RunningSum** each time

---

```
AverageTemp = RunningSum / Size;
```

AverageTemp = (308.5) / (4) = 77.1

With strings and characters

```
for (auto c: “Frank”)
  cout << c << endl;
```

automatically defines **c** as a **character** from the **string "Frank"** provided.

prints out each [character] in the string

While loop

```
While (expression)
        statement;


While (expression)
     { statement;} ;
```

Example of a pretest loop.

The expression must evaluate to a **boolean** value, and **if true** the **statement will be executed until it returns false.**

While loop Example

```
int i {1};
```

initiate an **integer i** set to **{1}**

```
while (i <= 5)
```

start with the keyword while (**i** is less than **5**) execute the following code

---

```
{
  cout << i << endl;
  ++i;
}
```

print the value of **i** and then **increment i** and **check condition** again

While loop Example 2

```
int Scores [ ] {100, 90, 87};
int i { 0 };
```

```
While (i < 3)
```

```
{
  cout << Scores[i] << endl;
  ++ i;
}
```

initiate an **array** called **Scores** with **set values**
initiate an **integer i** set to **{0}** (first index in an array

---

while (i is less than 3)

---

print the value at index **[i] in Scores** and then **increment i** and **check condition** again

Input validation Example

`int Num { };`	initiate an **integer variable** to hold user input
`cout << “enter an integer less than 100: ”;` `cin >> Num;`	ask the user to **enter a number** less than 100 and store that in **Num**
`while (Num >= 100)`	while the value in Num is greater than 100 execute the following code until condition is false

`{`
`  cout<<“enter an integer less than 100:”;`
`  cin >> Num;`
`}`
	asks the user again for input until a valid input is given

`while (Num <= 1 || Num >=5)`
`{`
`  cout << “enter an integer between 1 and 5: ”;`
`  cin >> Num;`
`}`
	can also check for multiple conditions by using the conditional OR operator
	
	while (**Num is less** than or equal to **1 OR** while **num is greater** than or equal to **5**) execute the following code until condition is false:

Input validation using boolean flags

```
bool Done {false};
int Num { 0 };
```

initiate a **boolean** variable called **Done** and set it to false
initiate an integer called Num and set it to 0

```
While (!Done)
```

While **!Done** is the same as **Done is = false.** run the following code:

```
{
  cout << “enter an integer
between 1 and 5: ”;
  cin >> Num;
```

ask the user for input **between 1 and 5**

```
  If (Num <= 1 || Num >=5)
     cout << “Invalid Input,Try
Again:”;
```

if the number given is **less than 1** or **greater than 5**, give them an **error** and ask to try again.

```
     else
       {
         cout <<“Thanks!”<< endl;
         Done = true;
       }
};
```

if the number given **is between 1 and 5**, **accept** the input and **set the Done flag** to **true** to exit the loop

Do-while loop

```
Do
{
Statements;
} while (expression);
```

In a do-while loop you execute the block statements while the conditional expression is true.

The **condition** is **checked** at the **end** of each iteration.

This is a **post test loop**, and this **guarantees the loop body will execute at least once.**

Input using Do-while

```
int Num { };
```

initiate an **int** called **Num**

```
Do {
```

run this code:

```
   cout << “enter an integer
between 1 and 5: ”;
   cin >> Num;
```

ask the user to enter an **integer between 1 and 5** and store value at **Num**

```
} while(Num <= 1 || Num >= 5);
```

Check if value is **less than one or greater than 5**, if **true** then run loop again

Menu system using Do-while loop

```
char Selection { };
do {
      cout >> “\n-----------------” << endl;
      cout >> “1. Selection 1 << endl;
      cout >> “2. Selection 2 << endl;
      cout >> “3. Selection 3 << endl;
      cout >> “4. Selection 4 << endl;
      cout >> “Q. “EXIT” << endl;
      cout >> “\nEnter your selection: ”;
      cin << Selection;
     switch (Selection)
{
        case '1':
            std::cout << "You are playing the game" << std::endl;
            break;

        case '2':
            std::cout << "You are loading the game" << std::endl;
            break;

        case '3':
            std::cout << "You are playing multiplayer" <<
std::endl;
            break;

        case '4':
            std::cout << "You are in settings" << std::endl;
            break;
        }
}  while (Selection != ‘q’ && Selection != ‘Q’);
cout << endl;
return 0;
```

1	declare a character called **Selection**
2	the do-while loop starts by printing out the menu options for the user and ask the **user** to select an option with an **input** character and save that to **Selection**
3	we nest a switch function inside our do-while loop to **check for input** setting the variable **Selection** for the switch expression will try to match the input to one of the case values if it matches a case it will **execute in the block** after the colon until it reaches a **break;**
4	check if the input (**Selection**) is not **q** or **Q** **if true** print the menu again and ask for input

Continue and Break

Continue

- No further statements in the body of the loop are executed
- Control immediately goes directly to the beginning of the loop for the next iteration

Break

- No further statements in the body of the loop are executed
- Loop is immediately terminated
- Control immediately goes to the statement following the loop construct

Continue and Break Example

```
std::vector<int>Values
{1,2,-1,3,-1,99,7,8,10};
```

Initiate a **vector** of **integers** called **Values** with members already set

---

```
for (auto Val: Values)
```

For each iteration of **Val** (which is automatically detected to be an **int** based on **Values type**) run the code below:

```
    {
     if (Val == -99)
     break;
     else if (Val == -1)
     continue;
     else
     cout << Val << endl;
}
```

if Val is -99 break the **for loop**

if Val is -1 continue to the next item in the vector

if Val is any other number, print it out

Infinite Loops

```
for(;;) {}

while(1){} / while(true){}

do{}while(1) / do{}whil(true)
```

- Loops whose condition expression always evaluate to true
- Usually this is unintended and a programmer error
- Sometimes programmers will use infinite loops and include break statements in the body to control them
- Some infinite loops are useful
  - Event loop in an event-driven program
  - Operating system

Nested Loop Example

```
for (OuterVal {1}; OuterVal <= 2; ++OuterVal)

      for (InnerVal {1}; InnerVal <= 3;++InnerVal)

      cout<< OuterVal << "," <<  InnerVal << endl;
```

The loop begins by checking if **OuterVal is less than or equal to 2**.

since it is set to 1 the inner loop runs. The inner loop checks if **InnerVal is less than 3**.

since it is set to 1 the code block under it runs and the **values are printed.**

The **inner loop runs again** until **InnerVal** reaches 3,

Then the **loop terminates** and the **outer loop increments OuterVal** and then checks to see if the value is less than or equal to 2 again.

since it is now 2 the inner loop runs again giving the following output:

1, 1
1, 2
1, 3
2, 1
2, 2
2, 3

This is why programmers say the inner loop runs faster than the outer loop, as you will get more repetitions with each nested loop

Multiplication Table

```
for (int Num1 {1}; Num1 <= 10; ++Num1)
{
      for (int Num2 {1}; Num2  <= 10; ++Num2)
        {
          cout << Num1 << " * " << Num2 << "=" << Num1 * Num2 <<endl;
        }
       cout << "---------------" << endl;
}
```

first for loop initiates **Num1 to 1**

then checks if **Num1** is less than or equal to **10.**

Since it is true the next for loop runs, initiating **Num2 to 1** and checking if that is **less than or equal to 10.**

since it is true **both values are printed out** and the resulting **value of their multiplication.**

After the values are printed **Num2 is incremented** and the next set of numbers and values of multiplication are printed. This **continues for all values in a 10 x 10** multiplication table.

Display Vector Elements

```
std::vector<vector<int>> Vector2D
{
    {1, 2, 3},
    {10, 20, 30, 40},
    {100, 200, 300, 400, 500}
};

for (auto Vec: Vector2D)
{
   for (auto Val: Vec)
   {
     cout << Val << " ";
   }
   cout << endl;
}
```

We use nested range based for loops to iterate through a 2d vector of integers.

The first for loop uses auto to understand that **Vector2D** is a **vector of integer vectors** and iterates **through each set** in that vector **starting with {1, 2, 3}.**

The inner loop takes that **first vector** passed and uses **auto** again to determine that the vector members are **integers** and **passes that value to Val.**

The inner loop then **prints out each value in the vector** and when it reaches the end it **terminates the loop** and goes **back to the outer loop**. This continues for each vector.

Chapter 6: Characters and Strings

Character Functions:

`#include <cctype>`	includes functions to test for characters for different properties and conversion of character cases
`isalpha(c)`	true if **c** is a letter
`isalnum(c)`	true if **c** is a letter or digit
`isdigit(c)`	true if **c** is a digit
`islower(c)`	true if **c** is lowercase
`isprint(c)`	true if **c** is printable
`ispunct(c)`	true if **c** is a punctuation character
`isupper(c)`	true if **c** is upper case
`isspace(c)`	true if **c** is whitespace

Converting Characters	
`tolower(c)`	returns lowercase of **c**
`toupper(c)`	returns uppercase of **c**

C-style strings:

Note: These are not recommended to be used in C++, but it is possible and may be beneficial in certain situations.

- **Sequence of characters**
  - contiguous in memory
  - implemented as an array of characters
  - terminated by a null character (null)
    - **null** - character with a value of zero
  - referred to as zero or null terminated strings

- **String literal**
  - sequence of characters in double quotes "Frank"
  - constant
  - terminated with null character

Example of a string literal in memory									
"Hello Tod"									
characters are stored in separate memory slots contiguously followed by a null terminator (**\0**) with each character being accessible **like an index.**									
H	**e**	**l**	**l**	**o**		**T**	**o**	**d**	**\0**

Declaring variables:

```
char MyName[ ] {"Frank"};
```

Declaring an array of characters, initializing it to "Frank"

The compiler adds the **null** terminator to the end of the array to denote the end of the string. The array is fixed and no new items can be added but items can be changed

F	r	a	n	k	\0

Declaring length:

```
char MyName[8] {"Frank"};
```

Declaring an array of characters, with a set length of 8, initializing it to "Frank"

Since the length of the array is longer than the number of characters, the free slots are filled with **null** terminators. You are able to add 2 more characters to the end of this array, as long as there is a single **null** terminator left at the end.

F	r	a	n	k	\0	\0	\0

cstring Library:

```
#include <cstring>
```

- **functions that work with c style strings, and all follow the rule that strings are null terminated.**
  - **copying**
  - **concatenation**
  - **comparison**
  - **searching**
  - **and more...**

cstdlib Library:

- **includes functions to convert c-style strings to:**
  - **integer**
  - **float**
  - **long**
  - **etc.**

C-style string examples	
`char str[80];`	**declare**
`strcpy(str, "hello");`	**copy**
`strcat(str, "there");`	**concatenate**
`cout << strelen(str);`	**prints out string length**
`strcmp(str, "another");`	compares **str** to "**another**", going character by character. **if strings are the same, returns 0**

C++ strings:

`Std:string` is a class in the Standard Template Library (more information later)

we must make sure to `#include <string>` to use these functions

C++ Strings:

- use the **std namespace**
- contiguous in memory
- **dynamic size**
- work with input and output streams
- lots of useful member functions
- familiar operators can be used
- can be easily converted to c-style strings if needed
- **safer than c-style**

Declaring and initializing C++ Strings:

`#include <string>` `using namspace std;`	**don't get in the habit of using the std namespace**
`string s1;`	**Empty** string
`string s2; {"frank"};`	**"frank"**
`string s3; {s2};`	**"frank"** - can assign to variables too
`string s4; {"frank", 3};`	**"fra"** (string, first 3 index locations)
`string s5; {s3, 0, 2};`	**"fr"** (string, starting at index 0, move 2 index locations)
`string s6; (3, 'X');`	**"XXX"** - using parenthesis (string length of 3, all populated with 'X')

Assignment

`string s1;` `s1 = "C++ Rocks!";`	initialize an empty string and **assign the string literal** to **s1**
`string s2 {"Hello"};` `s2 = s1`	initialize a string to "Hello", then change that string to the value of **s1**

Concatenation - the building of a string from other strings

```
string p1 {"C++"};
string p2{"is a powerful"};
```

declare 2 separate strings assigned to literals

```
string sentence;
```

declare an empty string to hold the new string

```
sentence = p1 + " " + p2 + " language";
```

concat p1 and p2 together with spaces and add another string, then assign it to the string variable sentence

Accessing Characters in strings

```
string s1;
string s2 {"frank"};

cout << s2[0] << endl;
cout << s2.at(0) << endl;
```

can use either **[index]** or **.at(index)** methods like **arrays and vectors**

Print out the **item at index 0** in the **string s2**

```
s2[3] = 'e';
s2.at(3) = 'e';
```

Assign the character 'e' to the memory address at index (3), in the string s2

Comparing strings

`string s1 {"Apple"};`
`string s2 {"Banana"};`
`string s3 {"Kiwi"};`
`string s4 {"apple"};`
`string s5 {s1};`
	Assigning strings to different string liters

`s1 == s5`	True - strings are the same
`s1 == s2`	False - Apple is not Banana
`s1 != s2`	True - Apple is not Banana
`s1 < s2`	True - Apple comes before lexically
`s2 > s1`	True - Banana comes after Apple
`s4 < s5`	False - s4 uses lowercase 'a' which comes after uppercase
`s1 == "Apple";`	True

Uses same comparison operators as usual (== | != | > | <= | etc.)

objects in the string are compared character by character lexically

can compare:

- 2 **std::string** objects
- **std::string** object and c-style string literal
- **std::string** object and c-style string variable

Substrings

`object.substr(startIndex, length)`	**Extracts a substring from a std::string**
`string s1 {"This is a test"};`	declare string s1

`cout << s1.substr(0,4);`
`cout << s1.substr(5,2);`
`cout << s1.substr(10,4);`
	start at index 0 and output 4 characters
	start at index 5 and output 2 characters
	start at index 10 and output 4 characters

`output:`
`This`
`is`
`Test`	

Removing characters

`object.erase(startIndex, length)`	**removes a substring of characters from a std::string**
`string s1 {"This is a test"};`	declare string s1
`cout << s1.erase(0,5);`	delete 5 characters starting from index 0 but prints the rest of the string after the delete
`cout << s1.erase(5,4):`	delete 4 characters starting from index 5
`s1.clear();`	Empties string s1

Length of a string

`object.length()`	**Returns length of a string**
`string s1 {"frank"};`	declare string
`cout << s1.length() << endl;`	prints length of string (5)
`s1 += "james";`	Adds James to the end of string and saves that change to s1
`cout << s1 << endl;`	Prints frank james

Searching

`object.find(searchString)`	**takes a string or char and returns the index of the first character in that string (or char) that matches the original string**
`string s1 {"This is a test"};`	declare string s1
`cout << s1.find("This");`	returns 0, since 'This' starts at index 0 in s1
`cout << s1.find("is");`	returns 2, since 'is' starts at index 2 in s1
`cout << s1.find("test");`	returns 10
`cout << s1.find('e');`	returns 11, since the first 'e' is at index 11
`cout << s1.find("is", 4);`	returns 5, but starting at index 4
`cout << s1.find("XX");`	if string is not found returns **string::npos**

input >> and getline()

`cin >>` `getline(input stream,string name,delimiter)`	**input operator** **getline function, reads an entire string. delimiter can be used to denote a value to stop reading input at**
`string s1;`	declare string s1
`cin >> s1;{hello there}` `cout << s1 << endl;`	taking user input and storing value in s1 cin only reads up to first space so it only saves - **"hello"**
`getline(cin, s1);` `cout << s1 << end;`	read entire line until line \n **"hello there"**
`getline (cin, s1, 'x');` `cout << s1 endl;`	**- "this isx" -** stop reading input at 'x' **- " "this is" -** output stops and doesn't include delimiter

Chapter 7: Functions

Functions:

C++ programs

- C++ standard libraries (a set of functions and classes already set for us to use)
- third party libraries (open source, commercial, etc)
- our own functions

Functions allow the modularization of a program

- separate code into logical self-contained units(building blocks)
- these units can be reused

Using Functions example:

```
int main ()
{
     ReadInput();

     ProcessInput();


     ProvideOutput();

     return 0;
}
```

function that reads user input

function that processes that input

function that provides output

As programs get larger and more complex, using modularized code by creating functions and classes is key to being efficient at coding. **Less detail, more extraction.**

Modularization:

```
ReadInput();
{
      Statement1;
      Statement2;
      Statement3;
}

ProcessInput();
{
      Statement4;
      Statement5;
      Statement6;
}

ProvideOutput();
{
      Statement7;
      Statement8;
      Statement9;
}
```

splitting up functions into more readable blocks of code. Eventually programs will be separated into organized files where these blocks of code will be implemented.

Writing a function:

Write code to the function specifications

- understand what the function does
- understand what information the function needs
- understand what the function returns
- understand what errors the function may produce
- understand any performance constraints

You should understand **HOW** the function works internally **IF** you are the one writing it.

Function Example:

`(return data type) FunctionName (argument);`	to call a function you need the **name of the function** and **what arguments it accepts** as input
`FunctionName (argument1, argument2, ...);`	some functions may accept or require **multiple arguments**
`cout << sqrt(400.0) << endl; double result;`	calling the square root function, with an **argument** set to **400.0**
`result = pow(2.0, 3.0);`	assigning the **value of using the pow function on values (2.0 and 3.0)**, to **result**

<cmath> is a common library for mathematical functions, and are global functions.

User Defined Function

```
int AddNumbers(int a, int b)
{
    return a + b;
}

cout << AddNumbers(20,40);
```

example function that expects **2 integers (a and b)**

It calculates the **sum of a and b** and **returns it to the caller.**

We also specify **what datatype the function returns** (in this case an integer)

We can print the value by **calling the function** in the **output stream**

User Defined Function 2

```
Int AddNumbers(int a, int b)
{
    if (a < 0 || b < 0)
        return 0;
    else
        return a + b;
}
```

an edited version of the previous function that checks **if either argument entered was negative**, if so it returns 0.

Random Numbers:

```
#include <iostream>
#include <cstdlib>      - required for rand()
#include <ctime>        - required for time()

int main( )
{
    int RandomNumber {};
    (std::)size_t count {10};
    int min{1};
    int mac {6};

cout << "RAND_MAX on my system is: " << RAND_MAX << endl;
srand(time(nullptr));                    -

for (size_t i{1}; i <= count; ++i)
{
    RandomNumber = rand() % max + min;
       cout << RandomNumbers << endl;          - prints out 10 random numbers
                                                 between 1 and 6
}
cout << endl;
return 0;
}
```

consider using <random> header file

seed random number generator with system time. (calling time func(passing 0)); (look at C++docs for more info)

start from index **i = 1** , up to **max size(10)**, **by 1 each time**

In each iteration, call rand(), which returns a number between 0 and RAND_MAX

Prints out the 10 random numbers

Defining Functions

Name of a function

- same rules as for variables
- should be meaningful
- usually verb or verb phrase

Parameter list

- variables passed into the function
- types must be specified

return type

- what type of data is returned from the function

Body of the function

- statements that are executed when the function is called. enclosed in curly braces **{ }**

Function Syntax

`ReturnType FunctionName (InputArguments)`
`{`
`  Statements;`
`  return ReturnType;`
`}`

Return statements are optional
int Addition (int a, int b) ***{*** *** int Add = a + b;*** *** return Add;*** ***}*** ***Addition(10, 20) = 30***
Function call - invoking the function on the input variables you put in parenthesis

Chaining Functions

```
void SayHello( )
{
    cout << "Hello" << endl;
    SayWorld( );
}

void SayWorld ( ) ;
{
    cout << " World" << endl;
}

int main( )
{
    SayHello( );
    return 0;
}
```

main function executes **SayHello** function, which inside it calls the **SayWorld** function, so the output will be "**Hello World**". Functions can be chained together but **each time a function is called it will execute all code in the body until another function is called or it returns a value.**

Calling functions

- functions can call other functions
- compiler must know the function details **before** it is called

Function call Error

```
int main ( )
{
    SayHello;      - called before it is defined = ERROR
    return 0;
}

void SayHello()
{
    cout  << "Hello" << endl;
}
```

All functions must be defined before being called in your main file. This is usually done in different .cpp and .h files (more information later)

Function prototypes

C++ compiler must know about a function before use

- **Define Functions before calling them**
  - OK for small programs
  - Not a practical solution for larger programs
- **Use Function Prototypes**
  - Tells the compiler what it needs to know without a full function definition
  - placed at beginning of program
  - also used in our own header files

Function prototype

```
int Function1( int );
or
int Function1( int a);
```

- both are fine for prototype

```
int Function1(int a);
{
     statements;
     return 0;
}
```

Function Parameters

- When we call a function we can pass in data to that function
- in the function call they are called arguments
- in the function definition they are called parameters
- the must match in number, order, and data type

Function order

```
int AddNumber(int, int);                    -prototype

int main ( )
{
    int result {0};
    result = AddNumbers(100,200);           - Function call
    return 0;
}

int AddNumbers(int a, int b)                -  definition
{
     return a + b;
}
```

We must provide the compiler with function information before it is called to be used. As long as we provide our **prototype** of the function before our main, we can put the definition anywhere.

Pass by Value

When you pass data into a function it is passed-by-value

- a **copy** of the data is passed to the function
- whatever changes you make to the parameter in the function does **not** affect the argument that was passed

- **Formal vs Actual parameters**
    - **formal -** parameters defined in the function header
    - **actual -** parameters used in the function call, the arguments

Function Return Statement

- if a function returns a value then it must use a return statement that returns a value
- if a function doesn't return a value (**void**) then the return statement is optional
- return statement can occur anywhere in the body of the function
- return statement immediately exits the function
- we can have multiple return statements in a function
    - avoid too many return statements
- the return value is the result of the function call

Default Argument Values

- When a function is called, all arguments must be supplied
- Some of the arguments may have the same values most of the time
- we can tell the compiler to use default values if the arguments are not supplied
- default values can be in the prototype or definition, not both
    - **best practice is to put in the prototype**
    - **must appear at the tail end of the parameter list**
- can have multiple default values
    - must appear consecutively at the tail end of the parameter list

No Default Arguments

```
double CalcCost(double BaseCost, double TaxRate);

double CalcCost(double BaseCost, double TaxRate)
{
        return BaseCost += (BaseCost * TaxRate);
}
```

Function prototype expects two different arguments. Value for base cost and a tax rate.

What if the default tax rate charged to every customer is 6%? it would be easier to set that value instead of having the computer fill it in every time

Default Arguments

```
double CalcCost(double BaseCost, double TaxRate = 0.06);
```

Now we set the default value in the prototype.

```
int main ( );
{
      double Cost{0};
      Cost = CalcCost(200.0);
      Cost = CalcCost(500.0, 0.08)
      return 0;
}
```

When we call the function with **only the BaseCost argument**, the default value will automatically be used.

If we sell to a customer in a different area that has a tax rate of 8%, we can still include a value for TaxRate and it will **override the default value**.

Multiple Default Arguments

```
double CalcCost
(double BaseCost, double TaxRate = 0.06, double Shipping = 5.50);
```

```
int main ( );
{
      double Cost{0};
      Cost = CalcCost(200.0);
      Cost = CalcCost(500.0, 0.08)
      Cost = CalcCost(1250.0, 0.12, 3.5);
      return 0;
}
```

Multiple default arguments can be used as long as they are all on the **tail end of the parameter list.**

You can't have a default set for the first argument and then not include a default for the preceding ones

when the function is called the default arguments will be supplied to the function as long as no other matching arguments are included

Overloading Functions

- We can have functions that have different parameter lists that have the same name
- Abstraction mechanism since we can just think 'print' for example
- A type of polymorphism
    - we can have the same name work with different data types to execute similar behavior
- The compiler must be able to tell the functions apart based on the parameter lists and argument supplied

Return type is not considered for overloaded functions so the function must take in some differentiating arguments or the compiler won't be able to tell the difference.

Overloading Example

```
int AddNumbers(int, int)
double AddNumbers(double, double)

int AddNumbers(int a, int b)
{
    return a + b;
}

int AddNumbers(double a, double b)
{
    returns a + b;
}

int main ( )
{
    cout << AddNumbers(5, 10) << endl;          - prints 15
    cout << AddNumbers(5.0, 25.0) << endl;      - prints 25.0
}
```

The functions are differentiated by the arguments they expect when being called (int vs double)

Passing Arrays to Functions:

We can pass an array to a function by providing square brackets in the formal parameter description

```
Void PrintArray(int Num [ ]);
```

- The array elements are **NOT** copied
- Since the array name evaluates to the location of the array in memory, this address is what is copied
- The function has no idea how many elements are in the array since all it knows is the location of the first element (the name of the array)

Passing an Array - Error

```
Void PrintArray(int Num [ ]);

int main ( )
{
     int MyNumbers[ ]  {1,2,3,4,5};
     PrintArray(MyNumbers);
     return 0;
}

void PrintArray(int Num [ ])
{
      - function doesn't know how many elements in the array
      - must include size parameter
}
```

Passing an Array

```
Void PrintArray(int Num [ ], size_t Size);

int main ( )
{
     int MyNumbers[ ]  {1,2,3,4,5};
     PrintArray(MyNumbers, 5);
     return 0;
}

void PrintArray(int Num [ ], size_t Size)
{
     for (size_t i{0}; i < Size; ++i)
        cout << MyNumbers[i] << endl;
}
```

Including the size parameter lets us tell the compiler how big to set the array

We also need to include the size parameter when using our array in functions

We can then loop through each item in the array and print out the values

Changing an array

```
void ZeroArray(int Num [ ], size_t Size)
{
     for (size_t i{0}; i < Size; ++i)
           Num[i] = 0;
}

int main ( )
{
     int MyNumbers[ ]  {1,2,3,4,5};
     ZeroArray(MyNumbers, 5);
     PrintArray(MyNumbers, 5);
     return 0;
}
```

- set all of **MyNumbers** to 0
- "0 0 0 0 0"

Since we are **passing the location of the array** (pass by reference), the function can modify the actual array values in memory (not a copy)

Constant Parameters:

- we can tell the compiler that function parameters are constant(read only)
- this could be useful in the **PrintArray** function used above since it should **NOT** modify the array

```
void PrintArray(const int Num [ ], size_t Size)
{
     for (size_t i{0}; i < Size; ++i)
           cout << MyNumbers[i] << endl;
    Num[i] = 0;
}
```

If you don't want a function to edit values, make it a constant so the values are marked read-only

Attempting to modify the values will result in a compiler error

Pass by Reference:

- Sometimes we want to be able to change the actual parameter from within the function body
- in order to achieve this we need the location or address of the actual parameter
- we can use reference parameters to tell the compiler to pass in a reference to the actual parameter
- the formal parameter will now be an alias for the actual parameter

Reference (refer):

pass a matter to (another body, typically one with more authority or expertise) for a decision.

"disagreement arose and the issue was referred back to the Executive Committee"

Pass by Reference Example

```
void ScaleNum(int &num);                    - prototype

int main ( )
{
     int Number{1000};
     ScaleNum(Number);                      - call
     cout << Number << endl;
     return 0;
}

void ScaleNum(int &num)                     - definition
{
    if (num > 100)
       num = 100;
}
```

using the **& operator** lets us pass a reference to an integer named **num.** and since the **ScaleNum** function accepts a reference to the integer **num**, it is able to change the original value at the memory location of **num.**

the value of num is referred to ScaleNum which can change it and pass it back.

Swap Example

```
void Swap(int &a, int &b);

int main ( )
{
     int x{10}, y {20};
     cout  << x << " " << y << endl;
     Swap(x, y);
     cout << x << " " << y << endl;
     return 0;
}

void Swap(int &a, int &b)
{
     int temp = a;
     a = b;
     b = temp;
}
```

We pass a **reference** to value a and b

then call Swap function using **reference** to x and y

&a is a **reference** to value at '**x**', **&b** is a **reference** to value at '**y**'

We create a **temporary integer variable** to hold value at '**x**'

Change value at '**x**' to value at '**y**'

Change value at '**y**' to value of '**temp**'

Vector Example

```
void Print(std::vector<int> &Vec);

int main ( )
{
     std:vector<int> Data {1,2,3,4,5};
     Print(Data);
     return 0;
}

void Print(std:vector<int> &Vec)
{
     for (auto Num: Vec)
          cout << Num << endl;
}
```

We pass a reference to a vector of integers called **Vec**

Call Print function for the vector **Data**

&Vec is a reference to the values in the vector Data

for each item in the vector **Vec** (reference to Data), print each value

Scope Rules

- C++ uses scope rules to determine where an identifier can be used
- C++ uses static or lexical scoping
    - **Local or Block** scope
    - **Global** scope

Local/Block scope

- Identifiers declared in a block of code (surrounded by curly braces)
- function parameters also have block scope and are only visible within the block they are declared
- Local variables are **not** preserved between function calls
- with nested blocks, inner blocks can see out, but outer blocks cannot see in.

To remember:
Think of a nested block of code like a bird nest. Built in trees so birds can see outside the nest, but other animals cannot get into the nest.

Note - Static Local Variables:

- declared with the **static** qualifier
- only initialized the first time the function is called
- Value **is** preserved between function calls

```
static int value {10};
```

Global Scope

- Identifier declared outside any function or class
- Is visible to all parts of the program after the global identifier has been declared
- global constants are **OK**
- best practice is to **not** use global variables unless specifically needed

Global Scope Example

```
{
    int Num1 {100};
    int Num2 {500};

    cout << "Local num is: " << Num1 << " in main" << endl;        -100
    {
    int Num1 {200};

    cout << "Local num is: " <<Num1<< " in inner block" << endl;   -200
    cout << "Inner block can see outer, Num2 is: "<< Num2 << endl;-500
    }
    cout << "Local num is: " << Num1 << " in main" << endl;        -100
}
```

Num1 and **Num2** are both local to the main program.

When we introduce a new block into the code, scope rules apply

Num1 is then redefined to 200 in the local block

The local block can still see **Num2** in the outer block

But the outer block can't see the inner block so it outputs the outer value of **Num1**

Function Calls:

Functions use the "function call stack"

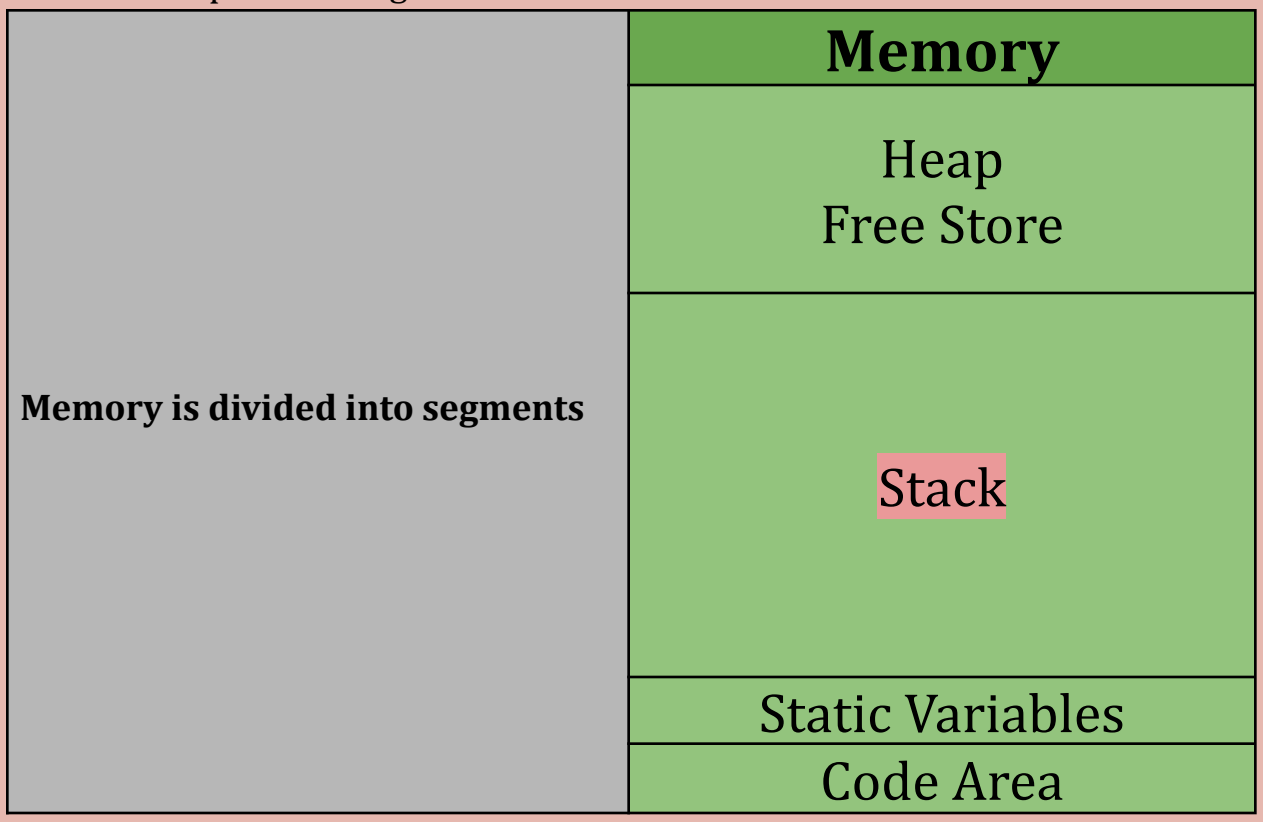
- like a stack of books
- **LIFO** - Last in First out
- **Push and Pop**
    - **push** an item to **add** to stack
    - **pop** an item to **remove** from stack

Stack Frame or Activation Record

- Functions must return control to function that called it
- each time a function is called we create a new activation record and push it onto the stack
- when a function terminates we pop the activation record and return
- local variables and function parameters are allocated on the stack

Stack size is **finite - Overflow can occur**

- if you activate too many functions on the stack, you might run out of stack space leading to an overflow error



Stack Example

```
void Func2(int &x, int &y,
int z)

{
    x+= y + z;
}

int Func1(int a, int b)
{
    int Result{ };

    Result = a + b;

    Func2(Result, a , b);

    return Result;
}

int main ( )
{
   int x {10};
   int y {20};
   int z { };

   z = Func1(x,y);

   std::cout << z << endl;
   return 0;
}
```

main is where a program starts, so it is at the top of the stack to initialize variables **(x,y,z)**

Func1 activation record is pushed to stack and variables **(a, b, Result)** are allocated in memory

Pass-by-value is used to make a copy of **x and y**
a is a copy of x = 10
b is a copy of y = 20

Result is then assigned to the value of **(a + b)**
Result = 30

Func2 activation record is pushed to stack and variables (x,y,z) are allocated in memory

Pass-by-value is used to make a copy of **a and b**
y is a copy of a = 10
z is a copy of b = 20

Pass-by-reference is used to create an alias for **Result**
Result = 30

---

x+= y + z; is executed

x = (x) + (10+20)
x = 30 + 30
Result is updated to **60**

Func2 is terminated and popped from the stack then we return to **Func1**

Func1 finishes and returns the new Result (60)

---

z is assigned to the value **Func1** returned ***z = 60***

what happens when main calls Func1 (or any function calls another)?

Multiple way to achieve same results:

main:

- Push space for the return value
- Push space for the parameters
- Push the return address
- Transfer control to Func1 (jmp - assembly instruction)

Func1:

- Push the address of the previous activation record
- push any register values that will need to be restored before returning to the caller
- perform code in Func1
- restore the register values
- restore teh previous activate record (move stack pointer)
- transfer control to the return address (jmp)

main:

- Pop the parameters
- Pop the return value

Inline Functions:

- **function calls have certain amount of overhead**

- we can **suggest** to the compiler to compile simple functions 'inline'
  - avoids function call overhead
  - generate inline assembly code
  - faster
  - could cause code bloat

- Compiler optimizations are very sophisticated
  - **will likely inline even without your suggestion**

Inline example

```
inline int AddNumbers(int a, int b) { return a + b;}              -
Inline function definition

int main ( )
{
    int Result {0};
    Result = AddNumbers(100,200);
- Function call
}
```

Recursive Functions:

- A function that calls itself
  - either directly or indirectly through another function

- Recursive problem solving
  - base case
  - divide the rest of problem into subproblem and do recursive call

- Many problems that lend themselves to recursive solutions
  - Mathematical - factorial, Fibonacci, Fractals
  - Searching and Sorting - binary search, search trees

Factorials

0! = 1

*n! = n * (n-1) !*

Base case
- *factorial(0) = 1*

Recursive case
- *factorial(n) = n * factorial(n-1)*

Factorial Example

```
unsigned long long Factorial(unsigned long long n)
{
     if (n == 0)
         return 1;
     return n * Factorial(n-1);
}

int main ( )
{
     cout << Factorial(8) << endl;      - 40320
     return 0;
}
```

Factorials can generate huge numbers so we use an **unsigned** (only positives) **long long** to try to prevent overflow

we check our base case first, **is n == 0,** and if true, we return 1 and exit function

if **n != 0** we run the function again with **n-1**

we are returning the base **n** multiplied by the value of **Factorial(n-1)**

starting at **8**, the function will **run until n == 0**

Fibonacci Example

```
unsigned long long Fibonacci(unsigned long long n)
{
     if (n <= 1)
         return n;

     return Fibonacci(n-1) + Fibonacci(n-2);
}

int main ( )
{
     cout << Fibonacci(30) << endl;          - 832040
     return 0;
}
```

Fibonacci can also generate huge numbers so again we use an **unsigned** (only positives) **long long** to try to prevent overflow

we check **both base cases** in a compound bit of code (checking for either **1 or 0**)

if the value is greater than 1, return **Fibonacci(n-1) + Fibonacci(n-2)**

Important Notes for Recursion:

- if recursion doesn't eventually stop you will have infinite recursion
- recursion can be resource intensive
- remember base case(s)
    - it is how a recursive function is terminated

- only use recursive solution when it makes sense
- anything that can be done recursively can be done iteratively
    - stack overflow error

Memory Example

```
unsigned long long
Factorial(unsigned long long n)
{
     if (n == 0)
         return 1;
     return n * Factorial(n-1);
}



int main ( )
{
     cout << Factorial(3) <<
endl;
     return 0;
}
```

More than 1 function call of the same function in the stack - This is recursion

each time a function returns, the function is terminated and Popped off the stack

Stack

Function call	Data
Factorial(0)	n = 0 0 == 0 return 1 to Factorial(1)
Factorial(1)	n = 1 1 != 0 n = 1 * Factorial(0) return n to Factorial(2)
Factorial(2)	n = 2 2 != 0 n = 2 * Factorial(1) return n to Factorial(3)
Factorial(3)	n = 3 3 != 0 n = 3 * Factorial(2) return n to Factorial(3)
Main	Factorial(3)

Adding Digits of an Integer Example:

```
int sum_of_digits(int n)
{
    if (n == 0)
       return 0;
   return (n % 10 + sum_of_digits(n/10));
}
```

```
(n = 123)
(123 != 0)
(123 % 10) = 3 +sum_of_digits(123 / 10)
```

```
sum_of_digits(12)
(12 != 0)
(12 % 10) = 2 +sum_of_digits(12 / 10)
```

```
sum_of_digits(1)
(1 != 0)
(1 % 10) = 1 + sum_of_digits(1 / 10)
```

```
sum_of_digits(0)
(0 != 0)
return 0;
sum_of_digits(123) = 6
```

to print the sum of the digits of a number **(n)**

we take the remainder of dividing **n by 10**

then we run the function again with the new value

The function is continued to be called until **n == 0**, then the function unwinds and the final result is given

Chapter 8: Pointers and References

Pointers:

A variable whose **value** is an **address**. A function or another variable can be at that address.

If I have an integer called **Num** with a value of **10**, I can declare a pointer to that **memory location.** To use the data the pointer is pointing to you must know the type.

- Inside functions, pointers can be used to access data that are outside the function. Those variables may not be in scope so you cant access them by name
- Pointers can be used to operate on arrays very efficiently
- We can allocate memory dynamically on the heap or free store
    - this memory doesn't even have a variable name
    - the only way to get to it is via a pointer
- Pointers can access specific addresses in memory
    - useful in embedded and systems applications

Declaring Pointers	
`VariableName *PointerName`	when declaring a variable we denote a pointer with the asterisk symbol in front of the **PointerName**
`int *IntPtr;`	a pointer to a **memory address** which stores an **integer** variable
`double* DoublePtr`	a pointer to a **memory address** which stores a **double** variable
`char *char_ptr;`	a pointer to a **memory address** which stores a **character** variable
`std::string *StringPtr;`	a pointer to a **memory address** which stores a **string** object

Initializing a Pointer

```
VariableName *PointerName {nullptr};

int *IntPtr { };

double* DoublePtr {nullptr};

char *char_ptr; {nullptr};

std::string *StringPtr; {nullptr};
```

Initializing a pointer variable is important so you know what memory address that pointer is pointing to.

we use {nullptr} to tell the pointer variable to point to 'nowhere'

- we should always initialize pointers
- uninitialized pointers contain garbage data (they could be pointing anywhere)
- initializing to {0} or {nullptr} represents address zero
- either initialize your pointer to a variable, constant, or zero.

Accessing Pointer Address

```
int Num{10};

cout << "Value of num is: " << Num << endl;
cout << "sizeof of num is: " << sizeof Num << endl;
cout << "Address of num is: " << &Num << endl;
```

& operator is used to access pointer addresses

Variables are stored in unique addresses

when used on the left side of an operand, the **&** operator evaluates to the address of its operand

- **operand cannot be a constant or expression that evaluates to temp values**

Accessing Pointer Address Example

```
int *P;

cout << "Value of P is: " << P << endl;            - garbage
cout << "Address of P is: " << &P << endl;         - correct address
cout << "sizeof of P is: " << sizeof P << endl;    - shows size of address

p = nullptr;                                       - set p to point nowhere

cout << "Value of P is: " << P << endl;    - 0
```

sizeof a Pointer Variable

```
int *P1 {nullptr};

double *P2 {nullptr};

unsigned long long *P3 {nullptr};

vector<string> *P4 {nullptr};

string *P5 {nullptr};
```

- Don't confuse the size of a pointer and the size of what it points to
- All pointers in a program have the same size, but they may be pointing to larger or smaller types

Storing an Address in Pointer Variables

```
int Score{10};
double HighTemp{100.7};

int *ScorePtr {nullptr};

ScorePtr = &Score;          - OK

ScorePtr = &HighTemp;       - Compiler error
```

The compiler will make sure that the address stored in a pointer variable is of the correct type

Storing an Address in Pointer Variables 2

```
double LowTemp {37.2};
double HighTemp{100.7};

int *TempPtr {nullptr};

TempPtr = &HighTemp;        - points to HighTemp

TempPtr = &LowTemp;         - changes to points to LowTemp

TempPtr = nullptr;          - changes to point to nowhere
```

- Pointers are variables so they can change
- Pointers can be null
- Pointers can be uninitialized **(don't do this)**

Dereferencing a Pointer

```
int Score {100};
int *ScorePtr {&Score};

cout << *ScorePtr << endl;

*ScorePtr = 200;

cout << *ScorePtr << endl;     - 200
cout << Score << endl;         - 200
```

To access the data a pointer is pointing to, we need to dereference it

if ***ScorePtr*** is a pointer and has a valid address

Then you can access the data at the address contained in the ***ScorePtr*** using the dereference operator **(*) (asterisk)**

Dereferencing a Pointer 2

```
double LowTemp {37.2{;
double HighTemp{100.7};

double *TempPtr {&HighTemp};   - pointer to value at HighTemp

cout << *TempPtr << endl;      - 100.7

TempPtr = &LowTemp;            - pointer to value at LowTemp

cout << *TempPtr << endl;      - 37.2
```

Dereferencing a Pointer 3

```
std::string Name {"Frank"};

std::string *StringPtr {&Name};

cout << *StringPtr << endl;          - Frank

Name = "James";

cout << *StringPtr << endl;          - James
```

Dynamic Memory Allocation:

Allocating storage from the heap at runtime

- we often don't know how much storage we need until we need it
- we can allocate storage for a variable at run time
- recall C++ arrays
    - we had to explicitly provide the size and it was fixed
    - but vectors grow and shrink dynamically
    - we can use pointers to access newly allocated heap storage

Using 'new' to Allocate Storage

```
int *IntPtr {nullptr};

IntPtr = new int;
```
- allocate an integer on the heap

```
cout << IntPtr << endl;
```
- address of new integer

```
cout << *IntPtr << endl;
```
- garbage data because the integer wasn't initialized

```
*IntPtr = 100;
```
- dereference pointer to get to the integer and then store 100 as the value

```
cout << *in
```

- if you allocate storage this way the storage is on the heap
- the allocated storage contains garbage data until initialized
- the allocated storage has no name, and can only be accessed by pointer
- if you lose pointer, you lose the way to that storage location and a memory leak occurs.
- when done using storage, you need to deallocate storage

Deallocate storage

```
int *IntPtr {nullptr};

IntPtr = new int;
```
- allocate integer on the heap

```
delete IntPtr;
```
- frees allocated storage when finished

Allocate Storage for an Array

```
int *ArrayPtr {nullptr};
int Size{ };

cout << "Enter size of array: ";
cin >> Size;

ArrayPtr = new int[size];          - allocate array on the heap

delete [ ] ArrayPtr;               - free allocated storage (square brackets must be empty)
```

Relationship Between Arrays / Pointers:

- the value of an array name is the address of the first element in the array
- the value of a pointer variable is an address
- if the pointer points to the same data type as the array element then the pointer and array name can be used interchangeably (almost)

Arrays and Pointers Example

```
int Scores[ ] {100,95,89};

cout << Scores << endl;
cout << *Scores << endl;

int *ScorePtr {Scores};

cout << ScorePtr << endl;
cout << *ScorePtr << endl;
```

- address of first item in array (value of array name)
- dereference to get value at address

- declare **ScorePtr** is a pointer to an integer and initialize it to **Scores**

- address of **ScorePtr** is same as **Scores**
- ***value is the same***

```
int Scores[ ] {100,95,89};

int *ScorePtr {Scores};

cout << ScorePtr[0] << endl;
cout << ScorePtr[1] << endl;
cout << ScorePtr[2] << endl;
```

- using array subscripting on a pointer

Pointers in Expressions

```
int Scores[ ] {100,95,89};
int *ScorePtr {Scores};

cout << ScorePtr << endl;
cout << (ScorePtr+1) << endl;

cout << (ScorePtr+2) << endl;
```

- 0x61ff10 memory address
- 0x61ff14 value of memory address of first item plus 1 (adding address of next integer value)
- 0x61ff18

Remember **sizeof** a pointer is **4 bytes(in this example)**, so **ScorePtr+1** is **adding 4** to the **address value**

```
int Scores[ ] {100,95,89};
int *ScorePtr {Scores};
cout << Scores[0] << endl;
cout << ScorePtr[0] << endl;
```

- both ways are valid and will return same value (item in array Scores at first address)

Pointer and Array Notation

```
int Scores[ ] {100,95,89};
int *ScorePtr {Scores};

cout << Scores[1] << endl;

cout << ScorePtr[1] << endl;

cout << *(ScorePtr+1) << endl;

cout <<  *(Scores+1) << endl;
```

- Array subscript notation
- Pointer subscript notation
- Pointer offset notation
- Array offset notation

Pointer Arithmetic:

Pointers can be used in

- assignment expressions
- arithmetic expressions
- comparison expressions

C++ allows pointer arithmetic
Pointer arithmetic only makes sense with raw arrays

Increment and Decrement

`IntPtr++;`	- increments pointer to next element
`IntPtr--;`	- decrements pointer to previous element
`IntPtr+= n; or IntPtr = IntPtr + n`	- increments pointer to by **n * sizeof(type)**
`IntPtr-= n; or IntPtr = IntPtr - n`	- decrement pointer to by **n* sizeof (type)**

```
cout << *IntPtr << endl;
ScorePtr++
```

```
cout << *ScorePtr++ << endl;
```

- You can condense the above code to this (dereference the pointer, then increment)

Subtracting Two Pointers

```
int n = IntPtr2 - IntPtr1;
```

Determine the number of elements between the pointers.

Both pointers must point to the same data type.

Comparing Two Pointers

```
std::string S1 {"Frank"};
std::string S2 {"Frank"};
std::string S3 {"James"};

std::string *P1 {"&S1"};
std::string *P2 {"&S2"};
std::string *P3 {"&S1"};
std:boolalpha

cout << (P1 == P2) << endl;  - False
cout << (P1 == P3) << endl;  - True

cout << (*P1 == *P3) << endl;  - True
cout << (*P1 == *P3) << endl;  - True

P3 = &S3;                      - Change where P3 is pointing to
cout << (*P1 == *P3) << endl;  - False
```

determine if two pointers point to the same location. This does not compare the data where they point.

Comparing addresses not the values

Comparing values by dereferencing pointer

Constants and pointers:

- **there are several ways to qualify pointers using const**
  - pointers to constants
  - constant pointers
  - constant pointer to constants

Pointer to Constants

```
int HighScore{100};
Int LowScore{65};

const int *ScorePtr {&HighScore};

*ScorePtr = 86;          - can't change values
ScorePtr = &LowScore     - can change where pointer is point to
```

The data pointed to by the pointer is constant and cannot be changed but the pointer itself can change and point somewhere else

Constant Pointers to Constants

```
int HighScore{100};
Int LowScore{65};

const int *const ScorePtr {&HighScore};

*ScorePtr = 86;          - can't change values
ScorePtr = &LowScore     - can't change where pointer is pointing
```

The data pointed to by the pointer is constant and cannot be changed and the pointer itself cannot be changed to point somewhere else

Passing pointers to functions:

- we can use pointers and dereference operator to achieve pass by reference
- the function parameter is a pointer
- the actual parameter can be a pointer or address of a variable

Pass by Reference with Pointers

```
void DoubleData(int *IntPtr);      - declaring function prototype

void DoubleData(int *IntPtr);      - definition
{
    *IntPtr *= 2;                  - using the compound assignment statement
    *intPtr = *IntPtr * 2;         - same as above
}

int main ( )
{
   int Value {10};

   cout << Value << endl;          - 10
   DoubeData(&Value);              - pass in pointer using & operator
   cout << Value << endl;          - 20
}
```

Pointers and Vectors Example 1

```
void Display(std::vector<string> * V)
{
    for(auto Str: *V)
        cout << Str << " ";
    cout << endl;
    (*V).at(0) = "Funny";
}

int main( )
{
    std::vector<string> Stooges {"larry","moe","curly"";
    Display(&Stooges);
}
```

Function expects an address of a vector of strings as input

dereference V to get the actual values

Using vector syntax to access and change elements that a pointer is pointing to

Pass address of Stooges to **Display** function, printing out each string in vector

Constant Vector Example

```
void Display(const std::vector<string> * V) -
(*V).at(0) = "Funny";   - Error

void Display(const std::vector<string> *const V) - s
V = nullptr;            - Error
```

Making the definition a constant vector for functions that shouldn't change values (display function shouldn't need to change data)

making the pointer constant will let you avoid it pointing to an unknown address

Pointers and vectors Example

```
void Display(int *array, int sentinel)
{
    while Display(*array != int sentinel)
cout << *array++ << " ";
    cout <, endl;
}
int main( )
{
    int Scores[ ] {100,98,95,89,80,-1};
    Display(Scores, -1);
}
```

Adding a sentinel when creating arrays can be used to include breakpoints in your array you can check for.

while the array value is not the sentinel, print element and increment to next one

Return a Pointer From a Function:

functions can return pointers

```
Type *Function( );
```

Should return pointers to memory dynamically allocated in the function or to data that was passed in. Never return a pointer to a local function variable.

Returning a pointer example

```
int *LargestInt(int *IntPtr1, int *IntPtr2)  -
{
     if (*IntPtr1 > *IntPtr2)
        return IntPtr1;
    else
      return IntPtr2;
}
int main ( )
{
     int a{100};
     int b{200};

    int *LargestPtr {nullptr};
    LargestPtr = LargestInt(&a, &b);
    cout << *LargestPtr << endl;  -
   return 0;
}
```

returning a pointer to an int

comparing values by dereferencing pointers

Function returns a pointer

declaring pointer to integer

calling LargestInt function and pass in addresses of both a and b

dereference pointer to display which pointer value was larger

Returning dynamically allocated memory

```
int *CreateArray(size_t Size, int InitValue = 0)
{
     int *NewStorage {nullptr};
     NewStorage = new int[Size]

     for (size_t i{0}; i < Size; ++i);
           *(NewStorage + i) = InitValue;
    return NewStorage;
}

int main ( )
{
    int *MyArray;
    MyArray = CreateArray(100, 20);
    delete [ ] MyArray;
}
```

Function called **CreateArray** takes an unsigned integer (containing **size of array**), and a **default parameter** (initialize value) to set the value of each item

Declare an **integer pointer variable**

Create the storage

Starting at **index 0** in the array, **if i is less than the size of the array...**

Change value to **default**

Return the address of first integer in created array

Function call, create an array with 100 items all set to 20

Remember to **free storage** after!

```
int *Func1( )
{
    int Size{ };
    return &Size;
}                                   - don't do this

int *Func2( )
{
  int Size{};
  int *IntPtr {&Size};
}                                   - or this
```

NOTE: Don't return a pointer to a local variable! local variables are lost after function is used then terminated.

Pointer Issues:

- uninitialized pointers
- dangling pointers
- not checking if 'new' failed to allocate memory
- leaking memory

Dangling pointer

Pointer that is pointing to released memory

- 2 pointers point to the same data, 1 pointer releases the data with delete while the other pointer tries to access that released data.

Pointer that points to memory that is invalid

- returning a pointer to a local variable

'new' Fail

- If 'new' fails, an exception is thrown
- dereferencing a null pointer will cause the program to crash

leaking memory

- forgetting to release allocated memory with delete
- if you lose your pointer to the storage allocated on the heap you have no way to get to that storage again
- the memory is considered "leaked"
- one of the most common problems with pointers

References:

- A reference is an alias for a variable
- it must be initialized to a variable when declared
- cannot be null
- once initialized, it cannot be made to refer to a different variable
- useful in function parameters
- **like a constant pointer that is automatically dereferenced**

Reference example

```
std::vector<string> Stooges {"Larry", "Moe", "Curly"};

for (auto str: Stooges)
      str = "Funny";
```

- pass by value makes a copy

```
for (auto str: Stooges)
      cout << str << endl;
```

- same data (str wasn't changed)

```
std::vector<string> Stooges {"Larry", "Moe", "Curly"};

for (auto &str: Stooges)
      str = "Funny";
```

- **reference** changes the original vector elements

```
for (auto str: Stooges)
      cout << str << endl;
```

- every element is "Funny"

```
for (auto const &str: Stooges)
      str = "Funny";
```

- constant qualifier is added in front of the loop variable
- gives an error

l-values / r-values:

l-values

- have names and are addressable
- modifiable if they are not constants
- are not literals (expressions)
- cannot be on right side of an assignment statement

```
int x{100};

string name {"Frank"};

double Percent {99.99};
```

r-values

- a value that is not an l-value
- temp values intended to be non modifiable
- literals
- can be on left or right side of an assignment statement

```
int x{100};

string name {"Frank"};

double Percent = (x + 90.99);
```

l-values references

```
int x {100};

int &ref1 = x;
ref1 = 1000;

int &ref2 = 100
```

- **ref1** is reference to **l-value**

- Error, **100** is an **r-value**

```
int square(int &n)
{
     return n*n;
}

int num {10};

square(num);

square(5);
```

- Input is reference

- OK, **can reference l-value**

- Error, **can't reference `r-value`**

Chapter 9: Object Oriented Programming

Classes and Objects

Procedural Programming:

- focus is on processes or actions that a program takes
- programs are typically a collection of functions
- data is declared separately
- data is passed as arguments into functions

Limitations:

- Functions need to know the structure of the data
    - if the structure of the data changes. functions must be changed

- As programs get larger they become difficult to understand, maintain, debug, extend, and are more prone to breaking.

Object-Oriented Programming:

Built up of classes and objects

- focus is on classes that model real-world domain entities
- allow developers to think at a higher level of abstraction
- used successfully in very large programs

Encapsulation is a key part of OOP

- Objects contain data and operations that work on that data
- Abstract Data Type (ADT)

Information Hiding

- hide implementation specific-logic
- hide users of the class code to the interface
- easier to maintain, debug, extend

Reusability

- easier to reuse classes in other applications
- faster development
- higher quality

Inheritance

- can create new classes based off of existing classes easily
- polymorphic classes (more information later)

Limitations:

OOP won't make bad code better!

- not suitable for all problems
- steeper learning curve, especially for C++
- more up-front design is necessary to create good models and hierarchies
- programs can be larger in size, slower, and more complex

Classes and Objects:

Class

- blueprint from which objects are created
- a user-defined data-type
- has **attributes(data)**
- has **methods(functions)**
- can hide data and methods
- provides a public interface

Class Examples:

`Account`
`Employee`
`Image`
`std::vector`
`std:string`

Object

- created from a class
- represents a specific instance of a class
- can create many objects
- each has its own identity
- each can use the defined class methods

Object Examples:

- **Frank's account** is an **instance** of the **class Account**
- **Jim's account** is an **instance** of the **class Account**
- **each** has its own **balance, can make deposits, withdrawls**, etc.

Class / Object Syntax:

```
int HighScore;
int LowScore;

Account FrankAccount;

Account JimAccount;

std::vector<int> Scores;

std::string Name;
```

- defining variables

- syntax for classes and objects are similar to defining variables

- vectors are considered objects so we already have some practice

Declaring a Class:

```
class ClassName
{
    declaration(s);
};
```

Class Definition:

```
class Player
{
    std::string Name;
    int Health
    int xp;

   void talk(std::string TextToSay);
   bool IsDead( );

};
```

- attributes (data)

- methods (functions)

Attributes are **data** included in this class that can be operated on

Methods are the **functions** that can be performed on this class

Creating Objects:

```
Player Frank;
Player Hero;

Player *Enemy = new Player( );

Delete [ ] enemy;
```

Declaring an object of type Player called Frank

Declaring **Enemy** as a **pointer** to a **Player object** which is **created dynamically** on the heap using **'new'** which creates new **player object** named **Enemy**

Free up storage after use

Account Class Example:

```
class Account
{
    std::string Name;
    double Balance;

   bool withdraw(double amount);
   bool deposit(double amount);
};
```

Creating more Objects:

```
Account FrankAccount;
Account JimAccount;

Account Accounts[ ] {FrankAccount, JimAccount};

std::vector<Account> Accounts1 {FrankAccount};
Acounts1.push_back(JimAccount);
```

Objects can be used like any other variable

Accessing Class Members:

- We can access both attributes and methods
- Some class members will not be accessible (public vs private)
- We need an object to access instance variables

Using Dot Operator (.):

```
Account FrankAccount;

FrankAccount.Balance;
FrankAccount.Deposit(1000.00);
```

- access data at (**Balance**)
- access function (**Deposit**)

- Object operator (attribute or method)

Pointers

```
Account *FrankAccount = new Account ( );

(*FrankAccount).Balance;
(*FrankAccount).Deposit(1000.00);
```

If we have a pointer to an object (member of pointer operator)

- dereference pointer then use dot operator

```
Account *FrankAccount = new Account ( );

FrankAccount->Balance;
FrankAccount->Deposit(1000.00);
```

or use the member of pointer operator (->)

Access Modifiers:

- **Public**
  - accessible everywhere
- **Private**
  - accessible only by members or friends of the class
- **Protected**
  - used with inheritance (more information later)

```
class ClassName
{

public:
        declaration(s);

private:
       declaration(s);
}
```

You will get a compiler error when trying to access private members of a class

by default, all items in a class are private

Implementing Member Methods:

- Similar to how we implement functions
- Member methods have access to member attributes (no need to pass as arguments)
- Can be implemented inside the class declaration (implicitly inline)
- Can be implemented outside the class declaration (need to use **ClassName::MemberName**)
- Can separate specification from implementation
  - **header for class declaration**
  - **.cpp file for class implementation**

Implementing inside class declaration

```
class Account
{
private:
   double Balance;
public:
   void SetBalance(double bal) {Balance = bal;}
   double GetBalance( ) {return Balance;}
}
```

- implementation inside the class declaration

Implementing outside class declaration

```
class Account
{
private:
   double Balance;

public:
   void SetBalance(double bal);
   double GetBalance( );
}

void Account::SetBalance(double bal)


{
      Balance = bal;
}

double Account::GetBalance( )
{
    return Balance;
}
```

- like a function prototype

- implementing outside mean you need to use the scope resolution operator (**Class::MethodName**)

Header and Implementation Files

Account.h

```
#ifndef  _ACCOUNT_H_
#define _ACCOUNT_H_

class Account
{
private:
  double Balance;
public:
  void SetBalance(double bal);
  double GetBalance( );
}
#endif
```

- include guard

Header file that outlines the specification for the Account class

Account.cpp

```
#include "Account.h"

void Account::SetBalance(double bal)
{
      Balance = bal;
}

double Account::GetBalance( )
{
    return Balance;
}
```

- to include another file **"FileName.extension"**

Source files for the implementation of the methods in our class

Main.cpp

```
#include<iostream>
#include "Account.h"

int main ( )
{
     Account FrankAccount;
     FrankAccount.SetBalance(1000.00);
     double bal = FrankAccount.GetBalance( );

     std::cout << bal << std::endl;
}
```

Main file for calling functions to run the program

always include .h files in your main program

Include Guard

```
#ifndef  _ACCOUNT_H_
#define _ACCOUNT_H_

   - account class declaration

#endif
```

#ifndef - if not defined **(file name)**
#define - then define **(file name)**

```
#pragma once
```

Pragma once is used in header files the same way as an include guard and is available on some compilers.

Constructors and Destructors:

Constructor

- is a special member method
- invoked during object creation
- used for initialization
- has the same name as the class
- no return type is specified
- it can be overloaded

Constructor Example

```
class Player
{
private:
        std::string Name;
        int Health;
       int Xp;
public:
```

- overloaded constructors (different based off of arguments)

```
       Player( );
      Player(std::string Name);
     Player(std::string Name. int Health, int Xp);
}
```

```
class Account
{
private:
        std::string name;
        double Balance;
public:
       Account( );
      Account(std::string Name, double Balance);
      Account(std::string Name);
     Account(double Balance);
};
```

Destructor

- is a special member method
- same name as the class with the **(~)** tilde prefix
- invoked automatically when an object is destroyed
- no return type and no parameters
- only 1 destructor is allowed per class and it can't be overloaded
- used to release memory and other resources
- called automatically when a local object goes out of scope, or we delete a pointer to an object

Destructor Example

```
class Player
{
private:
    std::string Name;
    int Health;
    int Xp;
public:
     - overloaded constructors (different based off of arguments)
    Player( );
    Player(std::string Name);
    Player(std::string Name. int Health, int Xp);

    ~Player( );
}
```

Class Account Example

```
class Account
{
private:
        std::string name;
        double Balance;
public:
       Account( );
      Account(std::string Name, double Balance);
      Account(std::string Name);
     Account(double Balance);

     ~Account( );
};
```

Creating objects

```
{
       Player Slayer;
      Player Frank {"Frank", 100, 4};
     Player Hero {"Hero"};
    Player Villain {Villain"};
}
```

When local objects go out of scope, their destructors are automatically called. If one is not defined, C++ will use a default(empty) destructor.

Default Constructor:

- does not expect any arguments
    - also called the no-args constructor
- if you write no constructors at all for a class C++ will generate a default constructor that does nothing
- it is called when you instantiate a new object with no arguments

```
Player Frank;
Player *Enemy = new Player;
```

```
Account FrankAccount;
Account JimAccount;

Account *MaryAccount = new Account;
delete MaryAccount;
```

- default construct for when no arguments are set

Set your own Default Constructor

```
class Account
{
private:
   std::string Name;
   double Balance;
public:
 Account( )
 {
   Name = "None";
   Balance = 0.0;
 }
     bool Withdraw(double Amount);
     bool Deposit(double Amount);
};
```

- you can set a default constructor to have any default values for your attributes

Calling Without a Default Constructor

```
class Account
{
private:
          std::string Name;
          double Balance;
public:
         Account(std::string NameVal, double Bal)
        {
              Name = NameVal;
              Balance = bal;
        }
     bool Withdraw(double Amount);
     bool Deposit(double Amount);
};
```

- we define a string that takes in a string and double and assign attributes to these values

C++ will not generate the no-args default constructor automatically, if we need it we need to explicitly define it.

if you have code that creates an object with no initialization information, you will get a compiler error since the default constructor is no longer generated.

```
Account FrankAccount;
```

- Error can't create no-arg Account objects without default constructor

```
Account *MaryAccount = new Account;
delete MaryAccount;
```

- Error

```
Account BillAccount {"Bill", 15000.0};
```

- OK, can create Account objects with provided the correct constructor (correct arguments)

Overloading Constructors:

- classes can have as many constructors as necessary
- each must have a unique signature
- default constructor is no longer automatically generated once another constructor is declared **(should always declare your own default constructor)**

Overloading Example

```
class Player
{
private:
  std::string Name;
  int Health;
  int Xp;
public:
  Player( );
  Player(std::string Name);
  Player(std::string Name. int Health, int Xp);
}
```

constructors are different based on arguments it accepts. you can have as many of these as you need as long as the compiler can tell them apart.

```
Player::Player( )
{
  name = "None";
  Health = "0;
  Xp = 0;
}
Player::Player(std::string NameVal)
{
  name = NameVal;
  Health = 0;
  Xp = 0;
}
Player::Player(std::string NameVal, int HealthVal, int XpVal)
{
  name = NameVal;
  Health = Health Val;
  Xp = XpVal;
}
```

Constructor Initialization Lists:

- More efficient
- immediately follows parameter list
- initializes data members as the object is created
- order of initialization is the order of declaration in the class

Initialization List Example

```
Player::Player( )
{
  name = "None";
  Health = "0;
  Xp = 0;
}
```

- creating a default constructor

```
Player::Player( )
    : name{"None"}, health{0}, xp{0} { }
```

- initialization list

lmmediately after the parameter list, we use a colon **(:)** followed by a list of initializers. Data members are initialized in order they are declared in the class declaration

Delegating Constructors:

- Often the code for constructors is very similar
- duplicated code can lead to errors
- C++ allows delegating constructors
    - code for once constructor can call another in the initialization list
    - avoids duplicating code

Delegating Example

```
class Player
{
private:
   std::string Name;
   int Health;
   int Xp;
public:
   Player( );
   Player(std::string Name);
   Player(std::string Name. int Health, int Xp);
}
```

using initialization lists for overloaded constructors outside class

```
Player::Player( ): Name{"None"}, Health{0}, Xp{0} { }

Player::Player(std::string NameVal): Name{NameVal}, Health{0}, Xp{0}{ }

Player::Player(std::string NameVal, int HealthVal, int XpVal):
Name{NameVal}, Health{HealthVal}, Xp{XpVal} { }
```

Delegating Constructors

```
Player::Player(std::string NameVal, int HealthVal, int XpVal):
Name{NameVal}, Health{HealthVal}, Xp{XpVal} { }
```
- write initialization list as normal

```
Player::Player( ): Player {"None", 0, 0} { }
```
- when delegating a constructor, you use the constructor name, followed by arguments to constructor

```
Player::Player(std::string NameVal ): Player {NameVal, 0, 0} { }
```
-only works in initialization list

Default Constructor Parameters:

- can often simplify our code and reduce number of overloaded constructors
- same rules apply as with non-member functions

Default Parameters Example

```
class Player
{
private:
  std::string Name;
  int Health;
  int Xp;
public:
Player(std::string NameVal = "None", int HealthVal = 0, int XpVal = 0);
};

Player::Player(std::string NameVal, int HealthVal, int XpVal):
Name{NameVal}, Health{HealthVal}, Xp{XpVal} { }
```

- creating a constructor with default parameter values

- single constructor that matches parameter list

```
Player Empty;                          - None, 0, 0
Player Frank{"Frank"};                 - Frank, 0, 0
Player Vilain{"Villain", 100, 55};     - Villain, 100, 55
Player Hero{"Hero", 100};              - Hero, 100, 0
```

no need to supply default values as they are already set in constructor parameters

Copy Constructor:

- When objects are copied C++ must create a new object from an existing object. copies are made:
    - When passing an object by value as a parameter
    - Returning an object from a function by value
    - Constructing an object based on another of the same class

- C++ provides a compiler-defined copy constructor if you don't provide a way to copy objects
- Copies the values of each data member to the new object
- Watch for pointers to data members as pointer will be copied, not what it is pointing to

Best Practices:

- Provide a copy constructor when your class has raw pointer members
- Provide the copy constructor with a const reference parameter
- Use STL classes as they already provide copy constructors
- Avoid using raw pointer data members if possible

Declaring Copy Constructor

```
Type::Type (const Type &source);

Player::Player(const Player &source);
```

Pass Object by-value

```
Player Hero {"Hero", 100, 20};

void DisplayPlayer{Player p}                    - p is a copy of Hero
{
                                - destructor for p called when out of scope
}
DisplayPlayer(Hero);
```

```
Player Enemy;

Player CreateSuperEnemey( )
{
        Player AnEnemy{"Super Enemy", 1000, 1000};
        return AnEnemy;
                                        - Copy of AnEnemy is returned
}

Enemy = CreateSuperEnemy( );
```

```
Player Hero {"Hero", 100, 100};

Player AnotherHero {Hero};          - Copy of Hero is made
```

Implementing Copy Constructor

```
Player

Player::Player(const Player &source):
     name{source.name},
     health{source.health},
     xp {source.xp} { }
```

Shallow Copy

The default behavior provided by the default copy constructor

- memberwise copy
- each data member is copied from source object
- pointer is copied NOT what it points to (shallow copy)
- when we release storage in the destructor the object still refers to released storage

Shallow copy Example

```
class Shallow
{
private:
      int *data;                          - raw pointer data member
public:
     Shallow(int d);
     Shallow(const Shallow &source);

     ~Shallow( );
}

Shallow:Shallow(int d)
{
     data = new int;                      - allocate storage
     *data = d;
}

Shallow::~Shallow( )
{
       delete data;                       - make sure to free memory when allocating storage

      std::cout << "destructor freeing data" << std::endl;
}

Shallow::Shallow(const Shallow &source)
        : data(source.data) {std::cout << "Shallow Copy" << std::endl;
}
```

Only the pointer is copied, not what it is pointer to, so **source** and the **new object** both have the same **data** area in memory. the newly created object will still point to that area in memory even after the storage is released with the destructor

Deep Copy:

- creating a copy of the pointed to data
- each copy will have a pointer to unique storage in the heap
- should deep copy when you have a raw pointer as a class data member

Deep Copy Example

```
class Deep
{
private:
      int *data;                    - raw pointer data member
public:
     Deep(int d);
     Deep(const Shallow &source);
     ~Deep( );
}
Deep:Deep(int d)
{
     data = new int;               - allocate storage (same as shallow copy) and
    *data = d;                       store input data in pointer
}
Deep::~Deep( )
{
  delete data;                     - make sure to free memory when allocating storage
  std::cout << "destructor freeing data" << std::endl;
}
Deep::Deep(const Deep &source)
{
  data = new int;          - allocate storage
  *data = *source.da       - copying source data pointer(the int data) into the new storage

  cout << Deep Copy << std::endl;
}
```

Deep copy - Create new storage, then copy values to that storage

Delegating - Deep Copy

```
Deep::Deep(const Deep &source)
    : Deep{*source.data} {
       std::cout << "Copy Constructor - Deep" << std::endl;
}
```

We can delegate object construction from a copy constructor to another constructor within the same class.

Delegate to **Deep (int)** and pass in the **int (*source.data)** source is pointing to

```
void DisplayDeep(Deep s)
{
   cout << s.GetDataValue( ) << endl;
}
```

when **s** goes out of scope the destructor is called and releases **data**

Move Constructor:

- The compiler will occasionally create unnamed temporary values

```
int Total {0};
Total = 100 + 200;
```

100 + 200 is evaluated and **300** stored in an unnamed **temp value.**

The **300** is then stored in the variable total

Then the **temp value is discarded**

There can be a large amount of overhead if copy constructors are called over and over again, making copies of these objects.

- Sometimes copy constructors are called many times automatically due to the copy semantics of C++
- Copy constructors doing deep copying can have a significant performance bottleneck
- C++11 introduced move semantics and the move constructor
- Move constructor moves an object rather than copy it
- Optional but recommended when you have a raw pointer
- Copy elision - C++ may optimize copying away completely (RVO - Return Value optimization)

r-value references:

- Used in moving semantics and perfect forwarding
- move semantics is all about r-value references
- used by move constructor and move assignment operator to efficiently move and object rather than copy it
- r-value reference operator **(&&)**

Reference Example

```
int x {100}

int &LRef = x;          - l-value reference
LRef = 10;              - change x to 10

int &&RRef = 200;       - r-value reference
RRef = 300;             - change RRef to 300 (changing the temporary variable)

int &&XRef = x;         - compiler Error, can't assign l-value to r-value reference
```

L-value reference parameters

```
int x {100};            - x is an l-value

void func(int &num);    - expects an l-value reference

func(x);                - x is an l-value
func(100);              - Error 100 is an r-value
```

R-value reference parameters

`int x {100};`	- **x** is an **l-value**
`void func(int &&num);`	- expects an **r-value** reference
`func(200);`	- **200** is an **r-value**
`func(x);`	- Error **x** is an **l-value**

R-value reference
Type::Type(Type &&source);
Move::Move(Move &&source);

Inefficient Copying - Move Class

```
class Move
{
private:
       int *data;                          - Raw pointer
public:
      void SetDataValue(int d) {*data = d;}
      int GetDataValue( ) {return *data;}
     move(int d);                          - constructor
     move(const Move &source);             - copy constructor (deep copy)
    ~Move( );                              - destructor
}
```

```
Move::Move(const Move &source)
{
         data = new int;
         *data = *source.data;
}
```

```
vector<Move> vec;

vec.push_back(Move{10});
vec.push_back(Move{20});
```

```
Output:
  1.Constructor for: 10
  2.Constructor for: 10
  3.Copy constructor - deep copy for: 10
  4.Destructor freeing data for: 10
  5.Constructor for: 20
  6.Constructor for: 20
  7.Copy constructor - deep copy for: 20
  8.Constructor for: 10
  9.Copy constructor - deep copy for: 10
  10. Destructor freeing data for: 10
  11. Destructor freeing data for: 20
```

Instead of making a deep copy of the move constructor

- it 'moves' the resource
- simply copies the address of the resource from source to the current object
- and then nulls out the pointer in the source pointer
- makes it very efficient

Efficient Copying - Move Class

```
class Move
{
private:
   int *data;
public:
   void SetDataValue(int d) {*data = d;}
   int GetDataValue( ) {return *data;}
   Move(int d);
   Move(const Move &source);
   Move(Move &&source);              - move constructor added
  ~Move( );
}
```

```
Move::Move(Move &&source)
   : data{source.data}  {           - copy of a pointer data member
       source.data = nullptr;       - null out source.data (pointer)
}
```

```
vector<Move> vec;
vec.push_back(Move{10});
vec.push_back(Move{20});
```

```
Output:        (no copy constructor calls)
  1.Constructor for: 10
  2.Move constructor - moving resource: 10
  3.Destructor freeing data for nullptr
  4.Constructor for: 20
  5.Move constructor - moving resource: 20
  6.Move constructor - moving resource: 10
  7.Constructor for: 10
  8.Copy constructor - deep copy for: 10
  9.Destructor freeing data for nullptr
  10. Destructor freeing data for nullptr
  11. Destructor freeing data for: 10
  12. Destructor freeing data for: 20
```

'this' pointer:

- **'this'** is a reserved keyword
- it contains the address of the object - so it's a pointer to the object
- can only be used in class scope
- all member access is done vie **'this'** pointer

can be used by the programmer to:

- Access data member and methods
- Determine if two objects are the same (more information later)
- Can be dereferenced(***this**) to yield the current object

```
void Account:SetBalance(double Balance)
{
        Balance = Bal;        - this->Balance is implied
}
```

'this' example

```
void Account:SetBalance(double Bal)
{
        Balance = Balance;        - What Balance?
}

void Account:SetBalance(double Balance)
{
        this->Balance = Balance   - using this->
}
```

'this' comparison

```
int Account::CompareBalance(const Account &other)
{
  if(this == &other)
    std::cout << "the objects are the same" << std::endl;
}
```

helps performance when checking many objects for equality

Using constants with classes:

- pass arguments to class member methods as **const**
- we can also create **const** objects
- what happens if we call member functions on **const** objects

```
const Player Villain {"Villain", 100, 55};
```

Villain is a constant object so attributes cannot change

Constant Attributes and Methods

```
const Player Villain {"Villain", 100, 55};

Villain.SetName("Nice Guy");
```

- Error, can't edit constant

```
const Player Villain {"Villain", 100, 55};

voice DisplayPlayerName(const Player &p)
{
              std::cout << p.GetName ( ) << std::endl;
}
DisplayPlayerName(Villain);
```

- Error

```
class Player
{
private:
. . .
public:
 std::string GetName( ) const;
. . .
};
```

- we add the const qualifier before the semicolon in the method prototype

const functions can **see** const attributes, they still cannot be changed.

Static Class Members:

- Class data member can be declared as static
  - a single data member that belongs to the class, not objects
  - **useful to store class-wide information**

- Class functions can be declared as static too
  - independent of any objects
  - can be called using class name

Player class - static members

```
class Player
{
private:
       static int NumPlayers;
public:
      static in GetPlayers( );
};
```

player.cpp

```
#include "Player.h"
int Player::NumPlayers = 0;
```

- here we initialize the static data

function definition

```
int Player::GetPlayers( )
{
   return NumPlayers;
}
```

- since **GetPlayers** is static, it has access to static attributes (**NumPlayers**)

```
Player::Player(std::string NameVal, int HealthVal, int XpVal)
      : Name{NameVal}, Health{HealthVal}, Xp{XpVal} {
            ++NumPlayers;
}
```

- increase static variable in Player constructor

```
Player::~Player( ){
     --NumPlayer;
}
```

- decrease static variable when destructor is called

We use the **static** prefix in front our functions/variables

Struct vs Class:

- in addition to classes we can also declare a **struct**
- struct comes from the C programming language
- essentially the same as a class except members are **public by default**
- everything you can do with classes, you can do with structs

```
struct Person {
     std::string name;
     std::string GetName( );
};

Person p;
p.name = "Frank";
```

- because Person is a struct, attributes and methods are set to public, not private

Struct

- use for passive objects with public access
- don't declare methods in struct

Class

- use for active objects with private access
- implement getter/setters as needed
- implement member methods as needed

Friends of a class:

Friend

- A function or class that has access to private class member
- And, that function or class is **not** a member of the class it is accessing

Function

- Can be regular non-member functions
- Can be member methods of another class

Class

- Another class can have access to private class members

Friendship is granted not taken

- declared explicitly in the class that is greanting friendship
- declared in the function prototype with the keyword **friend**

Friendship is not symmetric

- **if A is a friend of B**
- **B is not a friend of A**

Friendship is not transitive

- **if A is a friend of B and**
- **B is a friend of C**
- **then A is not a friend of C**

friendship - functions

```
class Player
{
    friend void DisplayPlayer(Player &p)
    std::string Name;
    int Health;
    int Xp;
public:
. . .
};
```

Using the **friend** prefix in front of the function prototype will mark it as a friend. **DisplayPlayer** has access to all of the **Player** class and since it is a pointer it can change those values too.

Friendship - other classes

```
class Player
{
    friend class OtherClass;
    std::string Name;
    int Health;
    int Xp;
public:
. . .
};
```

we can declare an entire seperate class as a friend. All methods in **OtherClass** will have access to the **Player** class

Chapter 10: Overloading

Operator Overloading:

- using traditional operators with user-defined types
- allows user defined types to behave similar to built-in types
- can make code more readable and easier to write
- is not done automatically, except for the assignment operator (=), so they must be explicitly defined

Given a user defined **Number class** that can model any number:

```
number result = multiply(add(a,b), divide(c,d));
```

```
number result = (a.add(b)).multiply(c.divide(d));
```

Overloading
`Number result = (a + b) * (c / d)`
you are still using member methods and functions but code looks cleaner

There are a few operators that cannot be overloaded:

- **(::)** Scope resolution operator
- **(:?)** conditional operator
- **(.*)** pointer to member operator
- **(.)** dot operator
- **(sizeof)** size of operator

Basic rules for overloading:

- precedence and associativity rules cannot be changed
- 'arity' rules cannot be changed (can't make division operator unary)
- can't overload operators for primitive types (int, char, double,etc)
- can't create new operators
- **[ ], ( ), ->, and =** must be declared as member methods
- other operators can be declared as member methods or global functions

Overloading example

int

```
a = b + c
a < b
std::cout << a
```

double

```
a = b + c
a < b
std::cout << a
```

long

```
a = b + c
a < b
std::cout << a
```

std::string - you can concat strings with the **+ operator**

```
s1 = s2 + s3
```

```
s1 < s2
```
- you can compare strings with **relational operators**

```
std::cout << s1
```

(class MyString) - You can create user defined classes that work with operators

```
s1 = s2 + s3
s1 < s2
s1 == s2
std::cout << s1
```

(class Player) - when overloading operators it should make sense as to why

```
p1 < p2
p1 == p2
std::cout << s1
```

Mystring class example - models a string using raw c-style pointer

```
class Mystring
{
private:
      char *str;          - c-style string
public:
      Mystring( );
      Mystring(const char *s);
      Mystring(const Mystring &source);
     ~Mystring( );
     void Display( ) const;
     int get_length( ) const;
     const chat *get_str( ) const;
};
```

Mystring class is used throughout this chapter

Mystring class implementation

```
Mystring::Mystring()
    : str{nullptr} {
    str = new char[1];
    *str = '\0';}
```

- create an object with a pointer called **str**
- allocate space for 1 character and assign address to **str**
- dereference pointer and assign value to **null** terminator (empty string)

```
Mystring::Mystring(const char *s)
  : str {nullptr} {
    if (s==nullptr) {
    str = new char[1];
    *str = '\0';
  } else {
     str = new char[std::strlen(s)+1];
    std::strcpy(str, s);}
}
```

- create a new object that accepts a **constant character pointer** with a pointer called **str**
- check for null pointer, if there is, create empty string
- allocate memory for the amount of characters + 1 on the heap
- copy s to str (**str points to s**)

```
Mystring::Mystring(const Mystring &source)
  : str{nullptr} {
   str = new char[std::strlen(source.str )+ 1];
   std::strcpy(str, source.str);}
```

- allocate memory for (length of source string +1)
- copy source string to str (str pointer to source string)

```
Mystring::~Mystring() {
    delete [] str;}

void Mystring::display() const {
    std::cout << str << " : " << get_length() << std::endl;}
int Mystring::get_length() const { return std::strlen(str); }

const char *Mystring::get_str() const { return str; }
```

Overloading Assignment operator:

copy assignment operator **(=)**

- **C++ provides a default assignment operator used for assigning one object to another**

`Mystring s1 {"Frank"};` **`Mystring s2 {s1};`**	**-** **not** assignment (it's initialization)
`s2 = s1;`	- assignment
default is memberwise assignment (shallow copy)	

Overloading assignment operator	
`Type &Type::operator=(const Type &rhs;`	**-** prototype
`Mystring &Mystring::operator=(const Mystring &rhs);`	
`s2 = s1;`	**-** what we write
`s2.operator=(s1)`	- **operator=** method is called

Overloading assignment implementation

```
Mystring &Mystring::operator=(const Mystring &rhs)
{
  if(this == &rhs)
   return *this;

   delete [ ] str;

str = new char[std::strlen(rhs.str) + 1];

    std::sctrcpy(str, rhs.str);
    return *this;
}
```

- if what you want to copy is the same as what you have then return a pointer to what you have

- '**this**' (str) object is overridden, so deallocate reference on heap

- assign new space on heap for **str** and assign it to the **length of the source string + 1**

- **deep copy** source string to str (str points to source)

- return ***this** to support chain assignment

Overloading move operator:

move assignment operator (=)

- **C++ will use the copy assignment operator if necessary (default)**

```
Mystring s1;

s1 = Mystring {"Frank};          - move assignment
```

if we have a raw pointer we should overload the move assignment operator for efficiency

```
Type &Type::operator=(Type &&rhs);

Mystring &Mystring::operator=(Mystring &&rhs);

s1 Mystring{"Joe"};        - move operator= called

s1 = "Frank";              - move operator= called
```

```
Mystring &Mystring::operator=(Mystring &&rhs)
{
       if (this == &rhs)          - check if self assignment
        return *this;

     delete [ ] str;              - deallocate current object storage
       str = rhs.str;             - steal pointer to source object (shallow copy of pointer)

     rhs.str = nullptr            - null out the source object pointer

    return *this;                 - return current object (now the same as source)
}
```

Overloading operators as member methods:

```
ReturnType Type::operatorOp( );

Number Number::operator-( ) const;
Number Number::operator++( );              - Pre-increment
Number Number::operator++(int);            - Post-increment
bool Number::operator!( ) const;

Number n1{100};
Number n2 = -n1;                           - n1.operator-( )
n2 = ++n1;                                 - n1.operator++( )
n2 = n1++                                  - n1.operator++(int)
```

default is memberwise assignment (shallow copy)

Mystring operator- make lowercase

```
Mystring larry1 {"LARRY"};
Mystring larry2;

lary1.display( );                          - "LARRY"

larry2 = -larry1;                          - larry1.operator-( )

larry1.display( );                         - "LARRY"
larry2.display( );                         - "larry"
```

Note: You need to ask yourself if the code makes sense from an efficiency standpoint. is it better to overload the operator to do a function, or just create a function with a more meaningful name as to its operation.

Mystring operator- function

```
Mystring Mystring::operator-( ) const
{
  char *buff = new char[std::strlen(str) + 1];
  std::strcpy(buff, str);

  for (size_t i = 0; i < std::strlen(buff); i++)
    buff[i] = std::tolower(buff[i]);

  Mystring temp {buff};
  delete [ ] buff;
  return temp;
}
```

returning a lower case object of an existing object

overloading the **operator-,** expecting no parameters, and it's a constant because it shouldn't modify the current object but create a new one based on it.

Allocate space on the heap for the lowercase copy of the current object string (copy of length of source string +1)

use **std::strcpy(TempStorage, Source)** to copy the string into that storage

for each character in that string convert it to the lower case equivalent using **std::tolower** function

Construct a temporary object and initialize it to the temp storage

Clear the temporary storage and return the Temp object you created

Overloading binary operators:

```
ReturnType Type::operatorOp(const Type *rhs)

Number Number::operator+(const Number &rhs) const;
Number Number::operator-(const Number &rhs) const;
bool Number::operator==(const Number &rhs) const;
bool  Number::operator<(const Number &rhs) const;

Number n1 {100}, n2 {200};
Number n3 = n1 + n2;                    - n1.operator+(n2)
n3 = n1 + n2;                           - n1.operator-(n2)
if (n1 == n2). . .;                     - n1.operator==(n2)
```

Equality operator

```
bool Mystring::operator==(const Mystring &rhs) const {
    if (std::strcmp(str, rhs.str) == 0)
       return true;
    else
      return false;
}
```

(std::strcmp(str, rhs.str) - will compare 2 c-style strings and **return 0 if they are identical** (is **str** in this object the same as **str** in the source object)

Mystring operator+ concatenation

```
Mystring larry {"larry"};
Mystring moe {"moe"};
Mystring stooges {" is one of the three stooges"};

Mystring result = larry + stooges;          - larry.operator+(stooges)

result = moe + " is also a stooge";         - moe.operator+(" is also a stooge:);

result = "Moe" + stooges;                   - "moe".operator(+stooges) ERROR
```

```
Mystring Mystring::operator+(const Mystring &rhs) const {
    size_t buff_size = std::strlen(str) + std::strlen(rhs.str) + 1;
    char *buff = new char[buff_size];
    std::strcpy(buff, str);
    std::strcat(buff, rhs.str);
    Mystring temp {buff};
    delete [ ] buff;
    return temp;
}
```

Allocate a character buffer large enough for both of the strings plus 1 for the terminator

Allocate the buffer on the heap

first copy the left side string with **strcpy** to buffer

then concatenate the string with **strcat**

create a new object to contain the concat string using buffer as initializer

free memory for buffer then return new string

Overloading operators as global functions:

```
ReturnType operatorOp(Type &obj);

Number operator-(const Number &obj);
Number operator++(Number &obj);          - pre increment
Number operator++(Number &obj, int);     - post increment
bool operator-(const Number &obj);

Number n1 {100};                         - operator-(n1)
Number n2 = -n1;                         - operator++(n1)
n2 = n1++;                               - operator++(n1, int)
```

Mystring operator- make lowercase global function

```
Mystring larry1 {"LARRY"};
Mystring larry2;

larry2 = -larry1;
```

```
Mystring operator-(const Mystring &obj)  {
        char *buff = new char[std::strlen(obj.str) + 1];
        std::strcpy(buff, obj.str);
        for (size_t i = 0; i < std::strlen(buff[i]);
               buff[i] = std::tolower(buff[i]);
        Mystring temp {buff};
        delete [ ] buff;
        return temp;
}
```

assumed set as a **friend function** since the function accesses **private Mystring** attribute regular global function

Overloading binary operators as global functions

```
ReturnType operatorOp(const Type &lhs, const Type &rhs);

Number operator+(const Number &lhs, const Number &rhs);
Number operator-(const Number &lhs, const Number &rhs);
Number operator==(const Number &lhs, const Number &rhs);
Number operator<(const Number &lhs, const Number &rhs);

Number n1 {100}, n2 {200};
Number n3 = n1 + n2;
n3 = n1 - n2;
if(n1 == n2). . .;
```

Overloading equality operator - global

```
bool operator==(const Mystring &lhs, const Mystring &rhs)  {
      if (std::strcmp(lhs.str, rhs.str) == 0)
           return true;
     else
          return false;
}
```

has to be declared friend to access private attributes, otherwise need to user getter methods

Mystring operator+ (concat) global

```
Mystring larry {"larry"};
Mystring moe {"moe"};
Mystring stooges {" is one of the stooges"};

Mystring result = larry + stooges;            - operator+(larry, stooges);
Mystring result = moe + " is also a stooge"; - operator+(moe, "is also a stooge");
result = "moe" + stooges;                     - ok with non-member functions
```

Mystring operator+ (concat) global function

```
Mystring operator+(const Mystring &lhs, const Mystring &rhs);
  size_t buff_size = std::strlen(lhs.str) + std::strlen(rhs.str) + 1);
  char *buff = new char[buff_size];
  std::strcpy (buff, lhs.str);
  std::strcat(buff, rhs.str);
  Mystring temp {buff};
  delete [ ] buff;
  return temp;
}
```

non-member **binary** functions requires both sides for input **(left side, right side)**

Overloading insertion and extraction operators:

we can overload these operators to read and extract data from our objects.

```
Mystring larry{"larry"};

Mystring larry;
std::cin >> larry;
std::cout << larry << std::endl;
```

- **doesn't make sense to implement as member methods**
  - left operand must be user-defined class
  - not the the way we normally use these operators
  - (**larry << cout;** - doesn't make sense)

overload stream insertion operator

```
std::ostream &operator<<(std::ostream &os, const Mystring &obj)
{
   os << obj.str;            - if friend function
   os << obj.get_str( );     - if not friend function (use getter method)
   return os;
}
```

we return a **reference** to the **ostream** so we can keep inserting, **don't return it by value**

overload stream extraction operator

```
std::ostream &operator<<(std::istream &is, Mystring &obj)
{
  char *buff = new char[1000];
  is >> buff;
  obj = Mystring{buff};        - if you have copy or move assignment
  delete [ ] buff;
  return is;
}
```

we return a **reference** to the **istream** so we can keep inserting

Chapter 11: Inheritance

Inheritance:

Inheritance provides a method for creating new classes from existing classes

- The new class contains the data and behavior of the existing class
- It allows for reuse of existing classes
- Allows us to focus on common attributes among a set of classes
- Allows new classes to modify behaviors of existing classes to make it unique, without actually modifying the original class

When designing a solution for an application, decide what classes are related, this is where inheritance is most useful.

- **Single inheritance** - a new class created from a single class
- **Multiple inheritance** - a new class created from 2 or more classes

No Inheritance Example

```
class Account
{ balance, deposit, withdraw...}

class Savings Account
{ balance, deposit, withdraw, interest rate}

class CheckingAccount
{ balance, deposit, withdraw, minimum balance, per check fee}

class TrustAccount
{ balance, deposit, withdraw, interest rate}
```

You end up with duplicate code when creating many new classes individually like this

Inheritance Example

```
class Account
{ balance, deposit, withdraw...}

class Savings Account : public Account
{ interest rate, specialized withdraw}

class CheckingAccount :public Account
{ minimum balance, per check fee,specialized withdraw}

class TrustAccount
{interest rate, specialized withdraw}
```

with an inherited class, the derived classes all receive the base attributes so we can focus on adding what is different

Base class (parent class / super class)

- class being inherited from

Derived class (child class / sub class)

- the class being created from the base class
- will inherit attributes and operations from base class

"is-a" relationship

- public inheritance
- derived classes are sub-types of their base classes
- can use a derived class object wherever we use a base class object

Generalization - combining similar classes int a single, more general class based on common attributes

Specialization - creating new classes from existing classes providing more specialized attributes or operations

Class Hierarchies

organization of our inheritance relationships

Hierarchy example

```
class Person <- Student
             <- Employee <- Staff
                         <- Faculty
                         <- Administrator
```

In UML, a class inheritance is denoted by a solid line from the derived class, with an empty arrow pointing to the parent class

Person is the root class in the hierarchy

Both **Student** and **Employee** are a **Person**

Staff, Faculty, and Administrator, are all **Employees**, and also all **Persons**

relationships are not bi-directional. A **Staff is always a person**, but a **person isn't always a Staff**

Inheritance vs Composition:

- **Public Inheritance**
  - "is-a" relationship
  - An **Employee** "is-a" **Person**
- **Composition**
  - "has-a" relationship
  - A **Person** "has-an" **Account**

Composition Example

```
Account --- class Person <- Student
                         <- Employee <- Staff
                                     <- Faculty
                                     <- Administrator
```

composition relationship is modeled as a single line between classes

All **Persons** have an **Account**

So **Student** has an **Account**, as does **Faculty, etc.**

Composition Implementation

```
class Person  {
private:
    std::string name;          - "has-a" name
    Account account;           - "has-an" Account
};
```

Derivation Syntax

Derived Class Example

```
class Base  {
. . . base members
};

class Derived: AccessSpecifier BaseClass {
. . . derived members
};
```

Access specifier can be **Public, Private, or Protected**

- **Public**
  - most common inheritance
  - establishes "is-a" relationship between derived and base class

- **Private** and **protected**
  - establishes "derived class has a base class" relationship
  - "is implemented in terms of" relationship
  - different from composition
  - not used often

Account Example

```
class Account {
. . . Account members
};

class SavingsAccount: public Account {
. . . SavingsAccount members
};
```

A savings account 'is-an' Account

Protected Members:

Protected Members and Class Access

```
class Base{
. . . protected:
          Account members
};

class Derived: protected Base{
. . . SavingsAccount members
};
```

accessible from the base class itself but not accessible from classes derived from the base class. they are not accessible by objects of the base or derived class

```
class Base
{
public:
  int a;
protected:
  int b;
private:
  int c;
};
```

```
class Derived
{
public:
  int a;        - Access
protected:
  int b;        - Access
private:
  int c;        - no Access
};
```

for a Public inheritance...

Public members are inherited and are public in the derived class

Protected member are inherited and are protected in the derived class

Private members are inherited but are not accessible by derived class

Constructors and Destructors - Inheritance:

When a derived class inherits from a base class, the base part of the derived class must be initialized before the derived class is initialized

- **when a derived object is created**
  - Base class constructor executes
  - Then derived class constructor executes

Constructors

`class Base  {`
`public:`
`  Base( ) {cout << "Base constructor" << endl; }`
`};`
 	
`class Derived : public Base {`
`public:`
`  Derived( ) {cout << "Derived constructor" << endl; }`
`};`

`Base Base;` - Base constructor called `Derived Derived;` - Base constructor called - Derived constructor called

when a derived object is destroyed:

- Class destructors are invoked in the **reverse order** as **constructors**
- Derived part of the Derived class must be destroyed before the base class destructor is invoked
  - Derived class destructor executes
  - Then Base class destructor
  - Each destructor should free resources allocated in it's own constructor

Destructors

```
class Base  {
public:
       Base( ) {cout << "Base constructor" << endl; }
       ~Base( ) {cout << "Base destructor" << endl; }
};

class Derived : public Base {
public:
       Derived( ) {cout << "Derived constructor" << endl; }
       ~Derived( ) {cout << "Derived destructor" << endl; }
};
```

```
Base Base;
```
- Base constructor called
- Base destructor called

```
Derived Derived;
```
- Base constructor called
- Derived constructor called
- Derived destructor called
- Base destructor called

A derived class does not inherit:

- base class constructors
- base class destructor
- base class overloaded assignment operators
- base class friend functions

The derived class constructors, destructors, and overloaded assignment operators can invoke the base class versions

C++ allows explicit inheritance of base 'non-special' constructors with:

- **using Base::Base;** anywhere in the derived class declaration.
- lots of rules involved and usually easier to define constructors yourself

Passing arguments to base class constructors:

- the base part of a derived class must be initialized first
- we can invoke whichever base class constructor we wish in the initialization list of the derived class

Passing arguments example

```
class Base
{
public:
  Base( );
  Base(int);
};

Derived::Derived(int x)
  : Base(x), {optional initializers for Derived}
{
  - code -
}
```

Base class example

```
class Base
{
   int value;
public:
  Base( ): value{0} {
    cout << "Base no-args constructor" << endl;
  }
  Base(int x) : value{x} {
   cout << "int Base constructor" << endl;
 }
};
```

Derived class example

```
class Derived : public Base {
   int DoubledValue;
public:
  Derived( ): Base{ }, DoubledValue{0}
    {
     cout << "Derived no-args constructor" << endl;
    }
  Derived(int x): Base{x}, DoubledValue{x*2}
   {
    cout << "int Derived constructor" << endl;
   }
};
```

Derived from Base using public inheritance

class initialization

`Base Base;`	- Base no-args constructor called
`Base Base{100};`	- int Base constructor called
`Derived Derived;`	- Base no-args constructor called - Derived no-args constructor called
`Derived Derived{100};`	- int Base constructor called - int Derived constructor called

Copy/Move constructors and overloaded assignment

- Not inherited from the base class
- you may not need to provide your own, the compiler provided version may work fine for your program
- we can explicitly invoke the base class versions from the derived class

Base Copy constructor

```
Derived::Derived(const Derived &other)
   : Base(other), {Derived initialization list}
{
   - code -
}
```

can invoke base copy constructor explicitly

- derived object **'other'** will be **sliced**

Derived Copy constructor

```
class Derived : public Base {
  int DoubledValue;
public:
   - constructors -

Derived(const Derived &other) : Base(other), DoubledValue {other.value}
{
   cout << "Derived copy constructor" << endl;
}
};
```

Base (operator=)

```
class Base {
  int value;
public
  - constructors -
  Base &operator=(const Base &rhs)
  {
       if (this != &rhs)
       {
         value = rhs.value;
       }
  return *this;
  }
};
```

Derived (operator=)

```
class Derived : public Base
{
  int DoubledValue;
public
  - constructors -
  Derived &operator=(const Derived &rhs)
  {
     if (this != &rhs) {
       Base::operator=(rhs);
       DoubledValue = rhs.DoubledValue;
     }
  return *this;
  }
};
```

You often don't need to provide your own copy/move and **overloaded operator=**

- If you **do not** define them i**n Derived class** then the compiler will create them automatically and **call the base class's version**
- If you **do** provide derived versions, then you must **invoke the base versions explicitly** yourself.
- Be careful with raw pointers
    - If Base and Derived each have raw pointers, make sure you provide deep copy semantics

Using and redesigning Base class methods:

- Derived class can directly invoke base class methods
- derived class can override or redefine base class methods
- powerful in the context of polymorphism (more information later)

Account example

```
class Account {
public:
  void deposit(double amount) {balance += amount;}
};

class SavingsAccount: public Account {
public:
  void deposit(double amount)
  {
    amount += SomeInterest          - redefine Base class method
    Account::deposit(amount);       - invoke call Base class method
  }
};
```

Static Binding of method calls:

- Binding of which method to use is done at compile time
- Default binding for c++ is static
- Derived class objects will use Derived::deposit
- We can explicitly invoke Base::deposit from Derived::deposit
- A more powerful approach is dynamic binding (more information later)

Static Binding of Method Calls	
Base b;	
b.deposit(1000.0);	- Base::deposit
Derived d;	
d.deposit(1000.0);	- Derived::deposit
Base *ptr = new Derived( );	
ptr->deposit(1000.0);	- Base::deposit ???

Multiple Inheritance:

- A derived class inherits from two or more base classes at the same time
- the base classes may belong to unrelated class hierarchies

```
class Person <- Student
             <- Employee <- Staff
                         <- Faculty<---------------
                                                  |----Department chair
                         <- Administrator <--------
```

A **Department Chair** is a **Faculty**

and is an **Administrator**

C++ syntax

```
class DepartmentChair:
          public Faculty, public Administrator
         . . .
};
```

Multiple inheritance can be very complex in practice and is beyond the scope of this guide. it is also easily misused but there is some use cases where it can be a benefit

Chapter 12: Polymorphism

Polymorphism:

There is two main types of polymorphism

- Compile time - early binding / static binding
- run time - late binding dynamic binding
    - function overriding is runtime polymorphism

Runtime Polymorphism

- being able to assign different meaning to the function at runtime
- achieved by:
    - inheritance
    - base class pointers or references
    - virtual functions

Non-Polymorphic example - static binding

```
Account(withdraw) <- Savings(withdraw) <- Trust(withdraw)
                  <- Checking(withdraw)
```

```
Account a;
a.withdraw(1000)          - Account::withdraw( )

Savings b;
b.withdraw(1000)          - Savings::withdraw( )

Checking c;
c.withdraw(1000)          - Checking::withdraw( )

Trust d;
d.withdraw(1000)          - Trust::withdraw( )

Account *p = new Trust( );
P->withdraw(1000);        -Account::withdraw( )
                          - Should be Trust::withdraw( )
```

since **p** is a pointer to an **Account** object, when we create a new Trust account dynamically and call withdraw, The pointer just points to **'an' Account** so the compiler will call the **Account withdraw method**

Static Binding

Account(display) <- Savings(display) <- Trust(display)
<- Checking(display)

```
void display_account(const Account &acc) {
        acc.display( );
}


Account a;
display_account(a)

Savings b;
display_account(b)

Checking c;
display_account(c)

Trust d;
display_account(d)
```

- will always use **Account::display**

creates 4 accounts (a,b,c,d) and passes each object into the function

Polymorphic example - dynamic binding

Account(withdraw) <- Savings(withdraw) <- Trust(withdraw)
<- Checking(withdraw)

```
Account a;
a.withdraw(1000)          - Account::withdraw( )

Savings b;
b.withdraw(1000)          - Savings::withdraw( )

Checking c;
c.withdraw(1000)          - Checking::withdraw( )

Trust d;
d.withdraw(1000)          - Trust::withdraw( )

Account *p = new Trust( );
P->withdraw(1000);        -Trust::withdraw( )
```

withdraw method is virtual in Account

Dynamic Binding

Account(display) <- Savings(display) <- Trust(display)
<- Checking(display)

```
void display_account(const Account &acc) {
            acc.display( );
          }

Account a;
display_account(a)

Savings b;
display_account(b)

Checking c;
display_account(c)

Trust d;
display_account(d)
```

- will always call the display method depending on the object's type at runtime

Display method is virtual in Account

Base Class Pointer

```
Account *p1 = new Account( );
Account *p1 = new Savings( );
Account *p1 = new Checking( );
Account *p1 = new Trust( );

p1->withdraw(1000);          - Account::withdraw
p2->withdraw(1000);          - Savings::withdraw
p3->withdraw(1000);          - Checking::withdraw
p4->withdraw(1000);          - Trust::withdraw
```

create 4 pointers to account objects, initialized to a different type of that object

```
Account* array [ ] = {p1, p2, p3, p4};

for (auto i = 0; i < 4; ++i)
       array[i]->withdrawy(1000);
```

declaring an array that hold pointers to account objects (base class pointers)

looping through the array, the correct withdraw method will be chosen correctly based on type of account

```
vector<Account *> accounts {p1,p2,p3,p4};

for (auto acc_ptr: accounts)
       acc_ptr->withdraw(1000);          - same method works for vectors
```

Virtual Functions:

- **redefined** functions are bound **statically**
- **overridden** functions are bound **dynamically**
- **virtual functions** are a type of **overridden** functions and they allow us to treat all objects generally as objects of the base class

Declaring Virtual Functions

```
class Account {
public:
       virtual void withdraw (double amount);
       . . .
};
```

Declare the function you want to override as **virtual in the base class.** Virtual functions are virtual all the way down the hierarchy from this point.

```
class Checking : public Account {
public:
     virtual void withdraw(double amount);
      . . .
};
```

override the function in the derived classes.

The function signature and return type **must match exactly**

virtual keyword isn't required here but it is best practice. If you don't provide an overridden version it is inherited from its base class

Virtual destructors:

- problems can occur when we destroy polymorphic objects
- if a derived class is destroyed by deleting its storage via the base class pointer and the class a non-virtual destructor then the behavior is undefined in the C++ standard
- Derived objects must be destroyed in the correct order starting at the correct destructor

Virtual destructor

```
class Account {
public:
         virtual void withdraw (double amount);
         virtual ~Account( );
         . . .
};
```

If a class has virtual functions, always provide a public virtual destructor

If the base class destructor is virtual, then all derived class destructors are also virtual.

Override Specifier:

- we can override base class virtual functions
- the function signature and return must be **exactly** the same. if they are different then we have redefinition not overriding.
    - redefinition is static bind
    - overriding is dynamic bind

Override specifier - redefinition example

```
class Base {
public:
   virtual void say_hello( ) const {                    - constant
     std::cout << "hello - i'm a base class object" << std::endl;
  }
  virtual ~Base ( ) { }
};

class Derived : public Base {
public:
  virtual void say_hello( )                             - no constant (not overriding)
    std::cout << "hello i'm a derived class object" << std::endl;
  }
  virtual ~Derived ( ) { }
};
```

the compiler considers this redefinition since the signatures don't match exactly.

```
Base *p1 = new Base ( );
p1->say_hello ( );

Base *p2 = new Derived( );
p2->say_hello ( );
```

say_hello method signatures are different, so Derived redefines say_hello instead of overriding it

Override specifier example

```
class Base {
public:
  virtual void say_hello( ) const {
    std::cout << "hello - i'm a base class object" << std::endl;
  }
  virtual ~Base ( ) { }
};

class Derived : public Base {
public:
  virtual void say_hello( ) override {
    std::cout << "hello i'm a derived class object" << std::endl;
  }
  virtual ~Derived ( ) { }
};
```

- constant

- produces compiler error
(need to match signature exactly still)

just adding the override specifier in the Derived class isn't enough, you need to make sure your functions are the same first, then add the override.

Final specifier:

- **C++11 provide the final specifier**
  - when used at the class level, it prevents a class from being derived from
  - when used at the method level, it prevents virtual methods from being overridden in derived classes

Final Specifier Syntax

```
class My_class final {
        . . .
};

class Derived final: public Base {
        . . .
};
```

Adding the final specifier after the class name at declaration marks the class as final and it can not be derived from. the compiler will generate an error

Final Specifier Example

```
class A {
public:
  virtual void do_something( );
};

class B: public A {
public:
  virtual void do_something( ) final;
};

class C: public B {
public:
  virtual void do_something( );
};
```

- final specifier prevents further overriding

- compiler error

Base class references:

- we can use base class references with dynamic polymorphism
- useful when passing objects to functions by reference that expect a base class reference

Base class reference example

```
Account a;
Account &ref = a;                    - create reference to an Account
ref.withdraw(1000);                  - Account::withdraw

Trust t;
Account &ref1 = t;                   - create reference to a Trust
ref1.withdraw(1000);                 - Trust::withdraw
```

```
void do_withdraw(Account &account, double amount) {
          account.withdraw(amount);
}

Account a;
do_withdraw(a, 1000);                    - Account::withdraw

Account t;
do_withdraw(t, 1000);                    - Trust::withdraw
```

Pure virtual functions and Abstract classes:

- **Abstract class**
  - Cannot instantiate objects
  - Theses classes are used as base classes in inheritance hierarchies
  - Often referred to as abstract base classes

- **Concrete class**
  - Used to instantiate objects from
  - All their member functions are defined

Abstract base classes are too generic to create objects from (eg. Shape, Employee, Account, Player). So it serves as a parent class for derived classes that may have objects. And it contains at least one pure virtual functions

- **Pure Virtual Function**
  - Used to make a class abstract
  - Specified with the (**=0**) in its declaration
  - typically don't provide implementations (but possible to give them one)

```
virtual void function ( ) = 0;
```
- pure virtual function

The Derived classes must override the Base class. If the Derived class does not override, then the Derived class is also abstract.

Shape class example

```
class Shape {
private:
   - attributes common to all shapes -
public:
  virtual void draw ( ) = 0;              - pure virtual function
  virtual void rotate ( ) = 0;            - pure virtual function
  virtual ~Shape ( );
};
```

```
class Circle : public Shape {
private:
   - attributes for a circle -
public:
  virtual void draw ( ) override; {... }      - draw a circle
  virtual void rotate ( ) override;  {...}    - rotate a circle
  virtual ~Circle ( );
};
```

remember to mark the function as override in the derived class

Abstract classes as interfaces:

Using a class as an interface

- An abstract class that has only pure virtual functions
- These functions provide a general set of services to the user of the class
- Provided as public
- Each subclass is free to implement these services as needed
- Every service (method) must be implemented
- The service type information is strictly enforced

C++ does not provide true interfaces so we use abstract classes and pure virtual functions to achieve it

`std::cout << any_object << std::endl`
We want to be able to provide **printable** support for any object we wish without knowing its implementation at compile time. **any_object** must conform to the **printable** interface. We can create an interface class for printing that provides the service with a pure virtual function. The user defined classes can then be derived from the interface and override the function

Printable Interface Example

```
class Printable {
  friend ostream &operator<< (ostream &, const Printable &obj);
public:
  virtual void print(ostream &os) const = 0;
  virtual ~Printable( ) { };
};
ostream &operator<<(ostream &os, const Printable &obj) {
   obj.print(os);
   return os;
}
```

```
class Any_Class : public Printable {
public:
  virtual void print(ostream &os) override {    - must override Printable:print ( )
    os << "Hi from Any_Class" ;
  }
};
```

```
Any_class *ptr = new Any_Class( );
cout << *ptr << endl;

void function1 (Any_Class &obj) {
    cout <<  obj << endl;
}
void function2 (Printable &obj) {
   cout << obj << endl;
}

function1(*ptr);                          - "Hi from Any_Class"
function2(*ptr);                          - "Hi from Any_Class"
```

Functions are bound dynamically (so both **Any_Class** and **Printable** can print)

Shape Example

```
class Shape {
public:
  virtual void draw ( ) = 0;
  virtual void rotate ( ) = 0;
  virtual ~Shape ( ) { };
};
```

```
class Circle : public Shape {
public:
  virtual void draw ( ) override { code };
  virtual void rotate ( ) override { code };
  virtual ~Circle ( ) { };
};
```

must override pure virtual functions to make a concrete class, otherwise derived class is also abstract

```
class I_Shape {
public:
   virtual void draw ( ) = 0;
   virtual void rotate ( ) = 0;
   virtual ~I_Shape ( ) { };
};
```

Classes meant to be used as interface classes are usually denoted with the **I_** preceding the class name

Shape Example - Pointers

```
vector<I_Shape> *> shapes;

I_Shape *p1  = new Circle( );
I_Shape *p2  = new Line( );
I_Shape *p2  = new Square( );

for (auto const &shape: shapes) {
      shape->rotate( );
      shape->draw( );
}
```

- Free memory -

create a vector of pointers to **I_Shape** objects

then create 3 pointers, each initialized to different type of shape

Loop through the vector and call the rotate and draw functions for each shape

Chapter 13: Smart Pointers

Issues with Raw Pointers:

C++ provides absolute flexibility with memory management

- Allocation
- Deallocation
- Lifetime management

some potentially serious problems that occur

- Uninitialized (wild) pointers
- Memory leaks
- Dangling pointers
- Not exception safe

Ownership

- Who owns the pointer?
- When should a pointer be deleted?

Smart Pointers:

- Smart pointers are objects
- They can only point to heap allocated memory
- Automatically call delete when no longer needed
- Adhere to RAII principles

C++ Smart Pointers	
`unique_ptr`	Unique Pointer
`shared_ptr`	Shared Pointer
`weak_ptr`	Weak Pointer

We must include the memory header file

```
#include <memory
```

Smart pointers are implemented with class templates

- Wrapper around a raw pointer
- Dereference(*) and member selection(->) supported
- No pointer arithmetic (++)
- Can have custom deleters

Smart pointer - simple example

```
{
   std::smart_pointer<Some_Class> ptr = . . .

   ptr->method( );
   cout << (*ptr) << endl;
}
```

smart pointer type

ptr will be destroyed automatically when it is no longer needed

Resource Acquisition is Initialization (RAII):

A common idiom or pattern used in software design based on container object lifetime

RAII objects are allocated on the stack

Resource acquisition:

- Open a file
- Allocate memory
- Acquire a lock

is initialization:

- The resources is acquired in a constructor

Resource relinquishing

- Happens in the destructor
    - Close file
    - Deallocate memory
    - Release the lock

Unique Pointer:

unique_ptr

- Simple smart pointer, very efficient

unique_ptr<T>

- Points to an object of **type T** on the heap
- It is unique, there can only be one **unique_ptr<T>** pointing to the object on the heap
- Owns what it points to
- **Cannot** be assigned or copied
- **Can** be moved
- When the pointer is destroyed, what it points to is **automatically destroyed**

Unique pointer example

```
{
  std::unique_ptr<int> p1 {new int {100} };
  std::cout << *p1 << std::endl;                    - 100
  *p1 = 200;

  std::cout << *p1 << std:: endl;                   -200
}     - automatically deleted -
```

declare **p1** to be a unique pointer to an **int**

we initialize it to point to a **new int created on the heap**, which is initialized to **100**

we can then **dereference p1** to get the **int** that it points to

we can also modify the integer in the same way

Unique pointer - useful methods

```
{
  std::unique_ptr<int> p1 {new int {100} };
  std::cout << p1.get( ) << std::endl;          - Address of pointer
  p.reset;                                      - p1 is now nullptr

  if (p1)
   std::cout << *p1 << std::endl;               - won't execute
}   - automatically deleted -
```

Unique pointer - user defined classes

```
{
 std::unique_ptr<Account> p1 {new Account {"Larry"} }; -Pointer will manage
       std::cout << *p1 << std::endl;                   an Account object on
                                                        the heap
   p1->deposit(1000);
   p1->withdraw(500);

}  - automatically deleted -
```

Unique pointer - vectors and move

```
{
  std::vector<std::unique_ptr<int>> vec;        - Vector of unique pointers to ints
  std::unique_ptr<int> ptr {new int {100} };

  vec.push_back(ptr);                           - Error, no copying allowed with smart pointers

  vec.push_back(std::move(ptr));                - ptr gives up ownership of pointer and it's
                                                  given to the vector, ptr is set to nullptr

}   - automatically deleted -
```

Unique pointer - make_unique(C++14)

```
{
 std::unique_ptr<int> p1 = make_unique<int>(100);

 std::unique_ptr<Account> p2 = make_unique<Account> ("Curly", 5000);

 auto p3 - make_unique<Player> ("Hero", 100, 100);

}
```
- automatically deleted -

make_unique functions allows us to return a **unique pointer** of a specialized type, while also allowing us to pass in initialization values into the constructor of the object. we can **create the pointer, create the new object, and initialize it all in one statement.**

Efficient, no calls to **new** or **delete**

Shared Pointer:

```
shared_ptr
```

Provides shared ownership of heap objects

```
shared_ptr<T>
```

- Points to an object of **type T** on the heap
- It is not unique, there **can be many** shared pointers pointing to the same object on the heap
- Establishes **shared ownership relationship**
- **Can** be assigned and copied
- **Can** be moved
- **Does not support** managing arrays **by default**
- When the **use count is zero**, the managed object on the heap is destroyed

shared pointer - creating, initializing, using

```
{
        std::shared_ptr<int> p1 {new int {100} };
        std:: cout << *p1 << std::endl;

        *p1 = 200;

     std::cout << *p1 << std::endl;

}
```

- automatically deleted if no other object is using it -

we can use the same implementation as with a unique pointer

Shared Pointer - Useful Methods

```
{
 use_count                          - number of shared_ptr objects managing the heap object
 std::shared_ptr<int> p1 {new int {100} };
 std:: cout << p1.use_count ( ) << std::endl;       - 1

 std::shared_ptr<int> p2 {p1};                      - shared ownership
 std:: cout << p1.use_count ( ) << std::endl;       - 2

p1.reset ( );                                  - decrement use_count; p1 nulled out
 std:: cout << p1.use_count ( ) << std::endl;                  - 0
 std:: cout << p2.use_count ( ) << std::endl;                  - 1

}     - automatically deleted -
```

Shared Pointer - User Defined Types

```
{
 std::shared_ptr<Account> p1 {new Account {"Larry"} };
 std::cout << *p1 << std::endl;

 vec.push_back(ptr);                          - copy is allowed with shared pointers

 std::count << ptr.use_count( ) << std::endl;     - 2 (reference from pointer
                                                      and from shared pointer in vector
}      - automatically deleted -
```

Shared Pointer - Vectors and Move

```
{
 std::vector<std::shared_ptr<int>> vec;
 std::shared_ptr<int> ptr {new int {100} };

 vec.push_back(ptr);
 vec.push_back(std::move(ptr));

}
```
- automatically deleted -

Same implementation as unique pointer

Shared Pointer - make_unique

```
{
 std::shared_ptr<int> p1 = std::make_shared<int>(100);
 std::shared_ptr<int> p2 {p1}
 std::shared_ptr<int> p3;
 p3 = p1;

}
```
- use_count 1
- use_count 2
- use_count 3

- automatically deleted -

All 3 pointers point to same object on the heap

Weak Pointer:

```
weak_ptr
```

Provides a non-owning 'weak' reference

```
weak_ptr<T>
```

- Points to an object of **type T** on the heap
- does not participate in owning relationship
- **always** created from a shared pointer
- **does not** increment or decrement reference use count
- used to prevent strong reference cycles which could prevent objects from being deleted

Custom Deleter:

Sometimes when we destroy a smart pointer we need more than to just destroy the object on the heap. These are special use cases.

- C++ smart pointers allow you to provide custom deleters
- Can't use make_shared or make_unique when using custom deleters.

Deleter function

```
void my_deleter(Some_Class *raw_pointer) {
     -custom deleter code-
     delete raw_pointer
}
shared_ptr<Some_Class> ptr {new Some_class{ }, my_deleter };
```

function is provided with a raw pointer to the object the smart pointer is referencing

when pointer is destroyed it will call custom deleter

```
void my_delete(Test *ptr) {
   cout << "in my custom deleter" << endl;
   delete ptr;
}
share_ptr<Test> ptr {new Test { }, my_deleter};
```

Deleter lambda

```
shared_ptr<Test> ptr (new Test{100}, [ ] (Test *ptr){
  cout << "\tUsing my custom delete" << endl;
  delete ptr;
});
```

lambda expressions are anonymous functions. They have no name and can be defined in line right where you expect to use it (more information later)

Chapter 14: Exception Handling

Basic Concepts - Exceptions:

Exception handling

- Dealing with extraordinary situations
- Indicates that an extraordinary situation has been detected or has occurred
- Program can deal with the extraordinary situations in a suitable manner

Exceptions are caused by:

- Insufficient resources
- Missing resources
- Invalid operations
- Range violations
- Underflow/overflow
- Illegal data
- And more

Code is considered exception safe when it handles these conditions

An **exception** is an object or primitive type that signals an error condition has occurred.

Throwing an Exception:

- Code detects that an error has occurred or will occur
- The place where to error occurred my not know how to handle the error
- Code can throw an exception describing the error to another part o the program that knows how to handle the error

Catching the Exception:

- Code that handles the exception
- May or may not cause the program to terminate

C++ syntax for exception handling

`throw`	Throws an exception followed by an argument
`try{ }`	Place the code that may throw and exception in a try blow If the code throws an exception the try block is exited The thrown exception is handled by a catch handler. if none exists, the program terminates.
`catch( ) { }`	Code that handles the exception Can have multiple handlers May or may not cause program to terminate

Divide by zero

```
double average { };
average = sum / total;
```

what happens if **total** is zero?

Divide by zero solution

```
double average { };
try {
     if (total == 0)
        throw 0;                  - if total is 0, exit
    average = sum / total;        - now this won't execute if total = 0
}
catch (int &ex) {                 - exception handler
     std::cerr << "can't divide by zero" << std::endl;
}
std::cout << "program continues" << std::endl;
```

Throwing an exception from a function:

```
double calculate_avg(int sum, int total) {
   return static_cost<double>(sum) / total;
}
```

What do we return if total is zero?

Exception Example

```
double calculate_avg(int sum, int total) {
  if (total == 0)
    throw  0;
  return static_cost<double>(sum) / total;
}
```

we can insert a throw exception into a function to check for exceptions

```
double average { };

try {
  average = calculate_avg(sum, total);
  std::cout << average << std::endl;
}
catch (int &ex) {
  std::cerr << "You can't divide by zero" << std::endl;
}

std::cout << "Bye" << std::endl;
```

we can implement a try block to catch the exception thrown in the function and deal with it appropriately

Throwing multiple exceptions:

```
double calculate_mpg(int miles, int gallons) {
  return static_cast<double>(miles) / gallons;
}
```

```
doubles calculate_mpg(int miles, int gallons) {
  if (gallons == 0)
    throw 0;
  if (miles < 0 ||  gallons < 0)
    throw std::string{"Negative value error"};
  return static_cast<double>(miles) / gallons;
}
```

using multiple if statements with exceptions throws will let us check for multiple exceptions in the function

```
double miles_per_gallon{ };
try {
  miles_per_gallon = calculate_mpg(miles, gallons);
  std::cout << miles_per_gallon << std::endl;
}
catch (int &ex) {
  std::cout << "You can't divide by zero" << std::endl;
}
catch (std::string &ex) {
  std::cerr << ex << std::endl;
}

std::cout << "Bye" << std::endl;
```

Catch All handler

```
catch (int &ex) {
}
catch (std::string &ex) {
}
catch (. . .) {
  std::cerr << "Unknown exception" << std::endl;
}
```

Using this handler lets you run exception blocks for multiple throws in one statement. You do not have access to the object using this method.

Stack unwinding:

If an exception is thrown but not caught in the current scope, C++ tries to find a handler for the exception by unwinding the stack.

- Function in which the exception was not caught terminates and is removed from the call stack
- If a try block was used or the catch handler doesn't match, stack unwinding occurs
- If the stack is unwound back to mian and no catch handler deal with the exception, the program terminates

User Defined Exception Classes:

We can create exception classes and throw instances of those classes

Best practice:

- Throw an object not a primitive type
- Throw an object by value
- Catch an object by reference (or const ref)

```
class DivideByZeroException {
};

class NegativeValueException {
};
```

Basic example of an exception class

We can create an object of these types and throw it.

We can also implement class members and methods if we wish to provide information about the exception.

Throwing user defined exception class example

```
double calculate_mpg(int miles, int gallons) {
 if (gallons == 0)
  throw DivideByZeroException( );
 if (miles == 0)
  throw NegativeValueException( );

 return static_cast<double>(miles) / gallons;
}
```

```
try {
  miles_per_gallon = calculate_mpg(miles, gallones);
  std::cout << miles_per_gallon << std::endl;
}
catch (const DivideByZeroException &ex) {
  std::cerr << "You can't divide by zero" << std::endl;
}
catch (const NegativeValueException &ex) {
  std::cerr << "Negative values aren't allowed" << std::endl;
}

std::cout << "Bye" << std::endl;
```

Same format as a regular exception catch, the arguments should take in the class objects you created

Class-level exceptions:

Exceptions can be thrown from within a class

Method

- These exceptions work the same way as do for functions

Constructor

- Constructors may fail
- Constructors do not return any value
- Throw an exception in the constructor if you cannot initialize an object

Destructor

- Don't throw exceptions from your destructor

```
Account::Account(std::string name, double balance)
   : name{name}, balance{balance} {

  if (balance < 0.0)
    throw IllegalBalanceException{ };
}
```

```
try {
  std::unique_ptr<Account> moes_account =
    std::make_unique<Checking_Account>("Moe", -10.0);
}
catch (const IllegalBalanceException &ex) {
  std::cerr << "Couldn't create account" << std::endl;
}
```

Exception Hierarchy:

C++ standard library provide a class hierarchy of exception classes

- **std::exception** is the base class
- All subclasses implement the **what( )** virtual function
- We can create our own user defined exception subclasses

```
virtual const char *what( ) const noexcept;
```

Deriving from std::exception example

```
class IllegalBalanceException : public std::exception
{
public:
  IllegalBalanceException( ) no except = default;
  ~IllegalBalanceException( ) = default;
  virtual const char* what( ) const no except {
    return "Illegal balance exception";
  }
};
```

derived from std::exception same as any base class. We provide a default constructor and destructor

We then use the **what( )** virtual function and return a c-style string describing the exception

```
Account::Account(std::string name, double balance)
   : name{name}, balance{balance} {
     if (balance < 00)
       throw IllegalBalanceException( );
}
```

```
try {
  std::unique_ptr<Account> moes_account =
     std::make_unique<Checking_Account>("Moe", -10.0);
}
catch (const IllegalBalanceException &ex) {
   std::cerr << "Couldn't create account" << std::endl;
```

Chapter 15: Input/Output and Streams

Files, Streams, and I/O:

C++ uses streams as an interface between the program and input/out

- Independent of the actual device
- Input stream provides data to the program
- Output stream receives data from the program

Common Header Files	
`iostream`	Provides definitions for formatted input and output from/to streams
`fstream`	Provides definitions for formatted input and output from/to file streams
`iomanip`	Provides definitions for manipulators used to format stream I/O

Common Stream Classes	
`ios`	Provides basic support for formatted and unformatted I/O operations. base class of most other classes
`ifstream`	Provides for high level input operations on file based streams
`ofstream`	Provides for high level output operations on file based streams
`fstream`	Provides for high level I/O operations on file based streams derived from ***ofstream*** and ***ifstream***
`stringstream`	Provides for high level I/O operations on memory based streams derived from ***ostringstream*** and ***istringstream***

global stream objects	
`cin`	Standard input stream - default is connected to the standard input device of the system(keyboard). instance of **istream**
`cout`	Standard output stream - default is connected to the standard output device of the system(console). instance of **ostream**
`cerr`	Standard error stream - default is connected to the standard error device (console). instance of **ostream** (**unbuffered**)
`clog`	Standard logstream - default is connected to the standard log device (console). instance of **ostream** (**unbuffered**)
Global objects are initialized before main executes Best practice is to use **cerr** for error messages, and **clog** for log messages	

Stream Manipulators:

Streams have useful member functions to control formatting.

- Can be used on input and output streams
- The time of the effect on the stream varies
- Can be used as member functions or as manipulator

```
std::cout.width(10);          - member function
std::cout << std::setw(10);   - manipulator
```

Common Stream Manipulators:

- **Boolean**
  - boolalpha
  - noboolalpha
- **Integer**
  - dec
  - hex
  - oct
  - showbase
  - noshowbase
  - showpos
  - uppercase
- **Floating point**
  - fixed
  - scientific
  - setprecision
  - showpoint
  - showpos
- **Field width, justification, and Fill**
  - setw
  - left
  - right
  - internal
  - setfill
- **Others**
  - endl
  - flush
  - skipws
  - ws

Stream Manipulators - Boolean:

Default for booleans is true for 1 and 0 for false, sometimes printing out true and false are more appropriate.

Boolean Example

```
std::cout <<(10 == 10) << std::endl;          - 1
std::cout <<(10 == 20) << std::endl;          - 0
```

```
std::cout << std::boolalpha;

std::cout <<(10 == 10) << std::endl;          - true
std::cout <<(10 == 20) << std::endl;          - false
```

setting the output stream to boolalpha mode, so **all further bool output** will be formatted **true** or **false**, instead of 1 or 0

```
std::cout << std::noboolalpha;          - turn off boolalpha
```

```
std::cout.setf(std::ios::boolalpha);
std::cout.setf(std::ios::boolalpha);

std::cout << std::resetiosflags(std::ios::boolalpha);
```

we can also format the output of boolean types with the **setf** method. using flags to set and reset formatting directives

Stream Manipulators - Integers:

Integers are displayed as **dec (base 10)** by default.

- ***noshowbase*** - prefix used to show hexadecimal or octal
- ***nouppercase*** - when displaying a prefix and hex values it will be lower case
- ***noshowpos*** - the '+' is not displayed for positive numbers

These manipulators affect all further output to the stream

Formatting Integers

```
int num {255};

std::cout << std::dec << num << std::endl;       - 255
std::cout << std::hex << num << std::endl;       - ff
std::cout << std::oct << num << std::endl;       - 377
```

```
int num {255};

std::cout << std::showbase;
std::cout << std::dec << num << std::endl;       - 255
std::cout << std::hex << num << std::endl;       - 0xff
std::cout << std::oct << num << std::endl;       - 0377
```

Display Hex in Uppercase

```
int num {255};

std::cout << std::showbase << std::uppercase;

std::cout << std::hex << num << std::endl;         - 0XFF
```

Displaying positive sign

```
int num1 {255};
int num2 {-255}

std::cout << num1 << std::endl;          - 255
std::cout << num2 << std::endl;          - (-255)

std::cout << std::showpos;

std::cout << num1 << std::endl;          - (+255)
std::cout << num2 << std::endl;          - (-255)
```

Stream Manipulators - Floating point:

Default when displaying floating point values is:

- ***setprecision*** - number of digits displayed (6)
- ***fixed*** - not fixed to a specific number of digits after the decimal point
- ***noshowpoint*** - trailing zeros are not displayed
- ***nouppercase*** - when displaying scientific notation
- ***noshowpos*** - no + sign for positive numbers

These manipulators affect all further output

Precision
`double num {1234.5678};` `std::cout << num << std::endl;` -1234.57
default is 6 for precision and rounding

`double num {123456789.987654321};`
 	
`std::cout << std::setprecision(9);`
`std::cout << num << std::endl;` -123456790

with **setprecision**, rounding will still occur

Fixed

```
double num {123456789.987654321};

std::cout << std::fixed;
std::cout << num << std::endl;                    - 123456789.987654
```

The **Fixed** manipulator starts precision at the right side of the decimal place. rounding will still occur and default precision is set to 6

```
double num {123456789.987654321};

std::cout << std::setprecision(3) << std::fixed;
std::cout << num << std::endl;                    - 123456789.988
```

we can use both **setprecision** and **fixed** to specify how many digits after the decimal place we want

Scientific

```
double num {123456789.987654321};

std::cout << std::setprecision(3) << std::scientific;

std::cout << num << std::endl;                    - 1.23e+008
```

using **setprecision** with **scientific** notation will specify how many digits before the exponents

```
double num {123456789.987654321};

std::cout << std::setprecision(3) << std::scientific; <<std::uppercase

std::cout << num << std::endl;                    - 1.23E+008
```

Using the **uppercase** manipulator displays the **E** as a capital in scientific notation

Trailing Zeros

```
double num {123.34};

std::cout << num << std::endl;          - 12.34

std::cout << std::showpoint;
std::cout << num << std::endl;          - 12.3400
```

showpoint will display trailing zeros based on the level of precision

Stream Manipulators - Field width, Align and Fill:

Default when displaying floating point values is:

- ***setw*** - not set by default
- ***left*** - when no field width
- ***right*** - when using field width
- ***fill*** - not set by default, blank space is used

Some of these manipulators only affect the next data element put on the stream

Defaults

```
double num {1234.5678};
std::string hello {"Hello"};

std::cout << num << hello << std::endl;
```

- 1234.57Hello

6 digits of precision, with string immediately following

```
double num {1234.5678};
std::string hello {"Hello"};

std::cout << num << std::endl;
std::cout << hello << std::endl;
```

- 1234.57
  Hello

Field Width

```
double num {1234.5678};
std::string hello {"Hello"};
std::cout << std::setw(10) << num << hello << std::endl;
```

---1234.57Hello

setw( ) only affects the **next data item** in the stream, so only **num** is affected. setting the field width to **10** and **right justifying num**

```
double num {1234.5678};
std::string hello {"Hello"};

std::cout << std::setw(10) << num
          << std::setw(10) << hello
          << std::setw(10) << hello<< std::endl;
```

---1234.57-----Hello-----Hello----------

setting each data item to 10, each item will be right justified in a width of 10

Left Justify

```
double num {1234.5678};
std::string hello {"Hello"};

std::cout << std::setw(10)
          << std::left
          << num
          << hello << std::endl;
```

- 1234.57---Hello

using **left** or **right** will explicitly state the side to justify on. It will only affect the next data item in the stream

```
double num {1234.5678};
std::string hello {"Hello"};

std::cout << std::setw(10) << num
          << std::setw(10) << std::right << hello
          << std::setw(15) << std::right << hello<< std::endl;
```

---1234.57-----Hello----------Hello-----

Setfill

```
double num {1234.5678};
std::string hello {"Hello"};

std::cout << std::setfill('&');
std::cout << std::setw(10) << num
          << hello << std::endl;
```

&&&1234.57Hello

using **setfill** lets us specify a character to fill blank spaces when using **setw**

Reading Input From Files:

fstream and **ifstream** are commonly used for input files

```
#include <fstream>
```

1. Declare an fstream or ifstream object
2. Connect it to a file on your system (opens it for reading)
3. Read data from file vie the stream
4. Close the stream

Opening a File

```
std::fstream in_file {"file name", std::ios::in};

std::fstream in_file {"file name", std::ios::in | std::ios::binary};
```

creating an object named **in_file** of **fstream** type

It has 2 parameters, first is the name of the file (location directory usually)

The second parameter specifies the mode and other options for the file being opened

std::ios::in - open the file in input mode, meaning we can read but not write to it

std::ios::binary **-** open the file in binary mode to read binary data

The bitwise OR used in this case sets both the input and binary modes to true

ifstream

```
std::ifstream in_file {"file name", std::ios::in};

std::ifstream in_file {"file name"};
```

ifstream is used for input files only so the **std::ios::in** is optional as it is default already

Open a File for Reading

```
std::ifstream in_file;
std::string filename;
std::cin >> filename;

in_file.open(filename);

in_file.open(filename, std::ios::binary);
```

we can used the **.open** method to open a file

Check if File Opened Successfully

```
if (in_file.is_open( )) {
    -read from file-
else {
    -exception handle-
}
```

using the **is_open()** method returns a boolean true if the file being checked is open for processing

Closing a File

```
in_file.close( );
```

Always close files after finishing with them to flush out any unwritten data. This is more important for output files than input files.

Reading From Files Syntax

```
int num{};
double total{};
std::string name{};

in_file >> num;
in_file >> total >> name;
```

```
100
255.67
Larry
```

We can use the extraction operator for formatted reading. will read until it reaches whitespace

```
std::string line{};

std::getline(in_file, line);
```

```
This is a line
```

We can use getline to read entire lines of data

Reading From Files Example

```
std::ifstream in_file{"../myfile.txt"};
std::string line{};

  if(!in_file) {
     std::cerr << "File open error" << std::endl;
     return 1;

 while (!in_file.eof()) {
   std::getline(in_file, line);
   cout << line << std::endl;
}

in_file.close();
```

Open the file

Check if file opened properly

.eof returns true if we have reached the end of the file. So our while loop continues as long as we have not reached the end **(! == NOT)**

Then we store the entire line of text into the string variable **line**

```
while (std::getlin(in_file, line))
    cout << line << std::endl;
```

We can condense the while loop by including the string input statements in the condition statement. when the end of file is reached or an error is encountered the loop will terminate

reading one character at a time

```
std::ifstream in_file{"../myfile.txt"};
char c;

  if(!in_file) {
     std::cerr << "File open error" << std::endl;
     return 1;

 while (in_file.get(c))
    cout << c;
}

in_file.close();
```

Output files:

fstream and **ofstream** are commonly used for output files

```
#include <fstream>
```

1. Declare an fstream or ofstream object
2. Connect it to a file on your system (opens for writing)
3. Write data to the file via stream
4. Close the stream

- Output files will be created if they don't exist
- Output files will be overwritten by default
- Can be opened so that new writes append
- Can be opened in text or binary mode

```
std::fstream out_file {"file name", std::ios::out};

std::ofstream out_file{"file name"};
```

Using **fstream** will let us open a file to read and write

Using **ofstream** will make **std::ios::out** the default

```
std::ofstream out_file{"file name", std::ios::trunc};

std::ofstream out_file{"file name", std::ios::app};

std::ofstream out_file{"file name", std::ios::ate};
```

We can specify how we want to open the file. **truncation** is default.

We can also specify to append on each write with **std::ios:app**

std::ios::ate flag lets is open a file and set the initial position of the next write to the end of the file

Opening Output File

```
std::ofstream out_file;
std::string filename;
std::cin >> filename;

out_file.open(filename);
```

opening the output file the same way as an input file

Checking if Output File is Open

```
if (out_file.is_open()) {
    -read file-
else {
    -exception handle-
```

always need to check if our files are open before reading / writing

Writing to an Output File

```
int num {100};
double total {123.45};
std::string name {"Larry"};

out_file << num << "\n"
                << total << "\n"
                << name << std::endl;
```

100
123.45
Larry

we can use formatted output using stream insertion **(<<)**

Copying Files

```
std::ifstream in_file{"myfile.txt"};
std::ofstream out_file{"copy.txt"};

if (!in_file) {
   std::cerr << "File open error" << std::endl;
   return 1;
}
if (!out_file) {
   std::cerr << "File creator error" << std::endl;
   return 1;
}
```

to copy data from one file to another we create 2 stream objects (in and out files)

we then need to make sure they both opened correctly

```
std::string line { };

while (std::getline(in_file, line))
       out_file << line << std::endl;

in_file.close();
out_file.close();
```

we use a while loop to read each line from the **in_file** and output it to the **out_file** stream

then we close the files

```
char c;

while (in_file.get(c))
       out_file << c << std::endl;
```

- we can do the same copy by character

String streams:

Allows us to read or write from strings in memory much as we would read or write to files. very powerful and useful for data validation.

```
stringstream

istringstream

ostringstream
```

3 classes for string streams we can use, must have **#include<sstream>**

1. Declare a **stringstream**, **istringstream**, or **ostringstream** object
2. Connect it to a **std::string**
3. Read/write data from/to the string stream using formatted I/O

reading from a stringstream

```
int num { };
double total { };
std::string name { };
std::string info {"Moe 100 1234.5"};

std::istringstream iss{info};
iss >> name >> num >> total;
```

We can create an **istringstream** object, connected to the string stream you want to access.

Writing to a Stringstream

```
int num {100};
double total {1234.5};
std::string name {Moe};

std::ostringstream oss{ };
oss << name << " " << num<< " " << total;
std::cout << oss.str() << std::endl;
```

We can create an empty **ostringstream** object and insert the data into it using the input operator.

We can display the string using the built in input buffer method **.str()**

Data Validation

```
int value { };
std::string input { };

std::cout << "Enter an integer: "
std::cin >> input;

std::stringstream ss{input};
if (ss >> value) {
   std::cout << "An integer was entered";
else
   std::cout << "An integer was not entered";
```

We can read the user input into a string

If we create a **stringstream** object and initialize it to the input, we can create an if loop to check if the input is a valid integer or not

Chapter 16: Standard Template Library (STL)

Subsection Title:

Subsection Text Example

- Subsection List Text Example

Non-Polymorphic example - static binding

Non-Polymorphic example - static binding

Non-Polymorphic example - static binding

Extra Chapter 1: Lambda Expressions

Subsection Title:

Subsection Text Example

- Subsection List Text Example

Code Example Title	
`Code Example Code Text` `Code Example Code Text` `Code Example Code Text`	- Inline Explanation Text Example - Inline ExplanationText Example - Inline Explanation Text Example
Explanation of Code • More notes about Code	
Important Definitions (and link to Glossary)	

Extra Chapter 2: Enumerations

Subsection Title:

Subsection Text Example

- Subsection List Text Example

Code Example Title	
`Code Example Code Text` `Code Example Code Text` `Code Example Code Text`	- Inline Explanation Text Example - Inline ExplanationText Example - Inline Explanation Text Example
Explanation of Code • More notes about Code	
Important Definitions (and link to Glossary)	

Definition Glossary

Data Type	
Tells us how to interpret the bits at a memory location, and the valid operations for those bits	
Example:	***integer*** or ***int*** represents whole numbers ***Character*** or ***char*** represents letters ***String*** represents a contiguous selection of ***Characters***

Variable	
An abstraction for a memory location where information can be stored, read, and changed by the computer. variables have a name and data type associated with them.	
Example:	***int Age;*** - ***Age*** is a variable name for a location in memory where an ***integer*** is stored

Keyword	
A keyword in a programming language is a list of words that are reserved for set functions that cannot be changed. These words cannot be used for variable names in programs.	
Example:	***Const;*** - ***const*** is the keyword that is used when initializing a variable to a **read-only value** ***char;*** - ***char*** is the keyword that is used to represent ***character*** information

Namespace

Namespaces allow developers to group their code into grouped entities that start with their namespace (eg. 'std' is the standard library in C++ used for things like cin and cout)

Example:

using namespace std; - tells the compiler to use the standard namespace for functions within it

(prefix)::(function);
(std)::(cout); **-** you would need to use the ***namespace prefix*** with those functions if you don't include the ***using namespace*** statement in your program

Identifier

A unique name created by a programmer that is given to a function or variable. These should be easy to read and meaningful names.

Example:

cout **-** the standard identifier for the ***cout*** function to print the data to the console

cin **-** the standard identifier for the ***cin*** function to take in data from the console

Operator

A character or symbol that represents a specific mathematical or logical action or process.

Example:

+	- The addition operator
*****	- The multiplication operator
&&	- Logical operator for **AND**

Syntax

The syntax of a computer language is the rules that define the combinations of symbols that are considered to be correctly structured statements or expressions in that language.

Example:

int World;
and
int world;

-C++ Syntax is case sensitive, so variables ***World*** and ***world*** would be different

Glossary Entry Title

Description

Example:

Example Text

Expression

In programming, an expression is a value, or anything that executes and ends up being a value. This is represented by a mix of variables (and/or) constants and operator(s).

Example:	***Price;*** - variables are expressions because they **equate** to a **memory address** **(4 != 5);** - a relational expression equates to **true** or **false**, but a computer reads that as a **1** or **0**

Statement

A statement is a grouping of expressions that follow the syntax of the programming language to express actions to be carried out.

Example:	***int num = 52;*** - assignment statement using the assignment operator**(=)** to put **52** into the **memory location** of the variable **num**

Glossary Entry Title

Description

Example:	***Example Text***

Glossary Entry Title

Description

Example:	***Example Text***

Standard Library *(std)*	
the standard library namespace in C++ it includes functions like ***cout, cin, endl*** or objects like ***string*** and ***vector***	
Example:	***std::cout << VariableName << std::endl;*** - using the ***std*** prefix and ***(::)*** operator tells the compiler which instance of the ***cout*** and ***endl*** functions to use.

Glossary Entry Title	
Description	
Example:	***Example Text***

Glossary Entry Title
Description

Example:	***Example Text***

Glossary Entry Title	
Description	
Example:	***Example Text***