Table of Contents

[**1 : Basic Data Types** 2](#_Toc160131915)

[**Arrays** 2](#_Toc160131916)

[**Questions** 2](#_Toc160131917)

[**2: Functions** 2](#_Toc160131918)

[**Switch Statement** 2](#_Toc160131919)

[**Enumerations (ENUMS)** 2](#_Toc160131920)

[**Random Number Generation** 2](#_Toc160131921)

[**Questions** 2](#_Toc160131922)

[**3 : Memory** 2](#_Toc160131923)

[**Heap v Stack** 2](#_Toc160131924)

[**Dynamic Memory:** 2](#_Toc160131925)

[**Garbage Collection** 2](#_Toc160131926)

[**Reference Counting** 2](#_Toc160131927)

[**Tracing** 2](#_Toc160131928)

[**4 : Pointers and References** 2](#_Toc160131929)

[**Pointers** 2](#_Toc160131930)

[**References** 2](#_Toc160131931)

[**Pointers v References** 2](#_Toc160131932)

[**5 : Input and Output** 2](#_Toc160131933)

[**6 : CStrings** 2](#_Toc160131934)

[**Char Arrays (C-Strings) | #include <cstring>** 2](#_Toc160131935)

[**Char Pointers (C-Strings):** 2](#_Toc160131936)

[**7 : STL Strings** 2](#_Toc160131937)

[**8 : Constructors** 2](#_Toc160131938)

[**9 : Initializer Lists** 2](#_Toc160131939)

[**10 : Preprocessor Directives** 2](#_Toc160131940)

[**11 : Templates** 2](#_Toc160131941)

[**12 : Function Pointers** 2](#_Toc160131942)

[**13 : Polymorphism** 2](#_Toc160131943)

[**14 : Big ‘O’** 2](#_Toc160131944)

[**15 : Recursion** 3](#_Toc160131945)

[**16 : lValues & rValues** 3](#_Toc160131946)

[**17 : Questions** 3](#_Toc160131947)

# **1 : Basic Data Types**

**Primitives**

double num = 1.5;

double num(1.5);

float num = 0.0f;

char c = ‘c’;

**Booleans**

bool isBool = true;

bool isBool = false;

**Loops**

do {

// do-while Loops, execute code at least once

} while (false);

**Iterating a string with auto**

string str = “Ryan”;

for (auto c : str) { // code | foreach loop for strings }

## **Arrays**

int myArr[4]; // YES! Array declaration but not initialization

int myArr2[4] = {0, 1, 2, 3}; // YES! Declared and initialized.

int myArr2[] = {0, 1, 2, 3}; // YES! Declared and initialized. Implicit size.

int myArr2[]; // NO! Need to declare size implicitly or explicitly.

~~int~~ myArr2[2] = 3; // NO! Can’t define a new array and assign a single element

myArr[0] = 3; // YES! Array initialization of first element.

int myArr2[4] = {0, 1}; // YES! Don’t need to initialize all elements.

Find array Max Size:

int num = (sizeof(arr) / sizeof(int)); // num is the max size of array.

Find array Current Size:

NO, C++ array function to calculate number of current elements used in array.

Define an external int variable and increment each time an element is added.

***2-Dimensional Arrays***

char caArray[3][5]; // Declared, not initialized.

char caArray[3][2] = {{'a', 'b'}, {'c', 'd'}, {'e','f'}};// Declared and initialized

char caArray[][2] = {{'a', 'b'}, {'c', 'd'}, {'e','f'}}; // Declared and initialized

***2-Dimensional Arrays created with pointers***

int\*\* ippArray = new int\* [3]; // Pointer of pointers | an array of int ptrs (heap)

## **Questions**

What are the different int types?

* signed int: -32767 to 32767
* unsigned int: 0 to 65535
* signed long: -2147483647 to 2147483647
* unsigned long: 0 to 4294967295
* signed long long: -9223372036854775807 to 9223372036854775807
* unsigned long long: 0 to 18446744073709551615
* size\_t (unsigned integer type)

What's size\_t?

* std::size\_t is the **unsigned integer type** result of size\_of operator
* std::size\_t represents the maximum total memory size of an object of any type
* std::size\_t are included in the stdlib.h and stddef.h header files
* When indexing C++ containers, such as std::string, std::vector, the appropriate type is the member **size\_type** ( defined as a synonym for **std::size\_t** ).
* std::vector<int>::size\_type MyVectorSize;
* auto vector\_size = myVect.size();

Why define a vector’s size as size\_type or size\_t instead of just using int?

* The size\_type alias helps to shield *your* code from such details, so it'll work properly regardless of what actual type should be used to represent a vector's size.
* C++ allows the author to be as general or flexible as possible (key strength of C++).
* Allows for larger values than an int can hold.
* Works great for templates.

NOTE: When passing in an array, don’t include the bracket symbols []

* Arrays are always passed by reference by default, local changes affect values outside of function
* Outputting an array variable, prints the memory address of the array.
* Arrays don't have bounds checks.

Do you need to specify a data type in a function call?

* No, only pass in the name of variable when passing an argument to function.

Are all arrays’ pointers and are all pointers’ arrays?

* Arrays are not pointers. Array types and pointer types are treated differently by the compiler.
* Arrays and pointers are synonymous in terms of how they use to access memory.
* The important difference between them is that, a pointer variable can take different addresses as value whereas, an array’s memory address is always fixed.
* In C , name of the array always points to the first element of an array.
* The symbol to an array will decay to a ptr (of the array’s data type) when accessed.
* This decay is why you can write:
  + char somearray[123];
  + char \*p = somearray; // Doesn’t need ‘&’ because the array auto decays to it’s address.

If first element of array initialized, is entire array considered initialized?

* Yes, If no explicit initializer list is specified for all elements, the non initialized values are 0.

# **2: Functions**

* Functions assign an individual code block.
* **Function prototypes** on top of file or in header are known as **function declarations**.
* Functions that contain not only **a header and a block of code** are called **function definitions**.
* Function prototypes **don’t** need a variable name listed in the parameters.
* **Function overloading** occurs when function have the **same name** but **different parameter** types.
* Function overloading requires the **return types to be the same**.
* Function templates are typically preferred to function overloading.
* Parameters in function definitions can be assigned a default value.
* Arrays are always passed by reference and array size is typically included in the parameters.
* Array parameters in a function definition and prototype show [] symbol, but calling the function and passing in an array argument does not include the [] symbol as part of function syntax, just name.

## **Switch Statement**

switch (num) // strings are not supported in C++ for switch statements

{

case 0: // '' - single quotes for chars

{

// code

}

break;

case 1:

{

// code

}

break;

}

*Can switch statements switch on strings/char arrays in C++?* ***No***.

## **Enumerations (ENUMS)**

* enum ECardSuit // User-defined type

{

keClubs = 4,

keDiamonds, // int value of 5 is implied

keHearts = 10,

keSpades // int value of 11 is implied

};

* ENUMs are a set of named integral constants.
* ENUMs behave like const int arrays.
* ENUMs have no built-in function for accessing the named valued, need map or switch.
* ENUMs are good for readability and a cleaner substitute for using a bunch of #defines <int value>

## **Random Number Generation**

time\_t t; // variable t of type time\_t

time(&t); // function call time() passes in a time\_t reference ‘t’.

srand(t); // random seed function | srand(time(0) does the same thing.

rand(); // actual random function

// Modulus for a smaller subset or range values

cout << rand() % 4 << endl; // range 1 to 4

## **Questions**

Why is Function overloading not possible with different return types?

* Function overloading occurs when two functions share a name but have different signatures.
* Signatures are based on # of parameters and types of parameters.
* *A different return type doesn’t affect the compiler’s signature overload check*,
* Changing only the return type does not differentiate the function signature and will cause error.

Why define a function with a const string& reference parameter?

* Makes the string being passed in unchangeable and at the same time prevents a copy operation.
* Const ref variable parameters’ logic applies to any data type parameter.

What is the difference between macros and inline functions in C++ ?

* Inline functions are typesafe and macros are not.
* Macros are simple text replacements in the preprocessor.
* Inline functions are like regular functions but unlike macros, they are handled by the compiler.
* From the compiler’s perspective, when it encounters an inline function, it writes a copy of the compiled function definition instead of generating a function call.
* Inline functions that are compiled are not visible in the source, making debugging very difficult.

# **3 : Memory**

## **Heap v Stack**

* Local variable *declarations* are stored on the stack.
  + Variables are read top to bottom by the compiler
  + As variables are read by compiler, the variables are put on the top of the stack.
  + The stack pointer moves back in memory as variables are compiled, address recorded.
* Local variables that are *initialized* with the new keyword are stored on the heap.
* Allocate on the heap only when you cannot allocate on the stack.
  + Lifetime to be longer than the scope.
  + When you need large amounts of data (like a texture).
* Performance difference between the stack and the heap comes from the allocation
  + When calling new, it has to call the free list, and book keep that data

## **Dynamic Memory:**

new operator - allocates dynamic heap memory for object.

int\* pIntPtr = new int(); // int\* pIntPtr = new int;

\*pIntPtr = 34; // deferences ptr, goes to memory address, and changes it’s value

cout << \*pIntPtr << endl; // derefernces ptr and reads value at that pointee address

delete pIntPtr; // always use delete when using a new keyword. Use [] for strings.

int arrSize; cin >> arrSize;

int\* myIntArr = new int[arrSize]; // dynamically allocated array with variable size

* If an array is created in a local function on the stack, the memory space is freed once the function returns. Returning that array and assigning it to a pointer outside of that function will return junk data.
* Creating a local dynamic array on the heap will allow the array to be returned while retaining the data. Since both the outer ptr and local dynamic array point to the same block of memory, delete frees both.

*What is a cache miss (relation to the heap) ?*

* Cache miss is an event in which a system or application makes a request to retrieve data from a cache, but that specific data is not currently in cache memory.
* Contrast this to a cache hit, in which the requested data is successfully retrieved from the cache.
* A cache miss requires the system or application to make a second attempt to locate the data, this time against the slower main database.

## **Garbage Collection**

* Garbage collection is a process by which memory that is no longer is use is reclaimed for re-use.
* GC occurs automatically in C#, Python, among many object orientated languages.
* Garbage Collection eliminates bugs (dangling pointers, memory leaks, multiple deallocations)
* Garbage collection requires overhead so it slows your program down and will occasionally over allocate memory and may not free memory at the ideal time.

## **Reference Counting**

* ***reference counting***is a type of garbage collection that tracks how many variables reference an obj.
* The count increases when the variables reference is copied and decreases when the reference changes value or goes out of scope.  When the count goes to 0, the reference is freed.
* **Reference counting** has **issues** with **Circular linked list** with no outer pointers.
* Every element in the circular list has a nonzero reference count, yet memory isn’t referenced by any object outside the list.  The memory should be deallocated, but reference counting won’t do it.

**Weak References** – references that are not included in an object’s reference count.

When every cycle of references contains a weak reference, then you can reclaim the structure when you lose the last external reference.

For example, consider a doubly linked list, every pair of adjacent elements for a cycle, so the list isn’t reclaimed even when it’s no longer externally referenced.  If all “previous” references are defined as weak referenced, then when there are no external references to the list, the head element’s reference count becomes 0 and it is deallocated.  This causes a cascading deallocation along the list as deallocation of each element set the reference count of the next element to 0.

This is available in C++ as std::shared\_ptr and std::weak\_ptr.

## **Tracing**

* With **Tracing** garbage collection, memory that is no longer referenced remains allocated until it is identified and deallocated during a garbage collections cycle.
* **Mark and Sweep** is used for tracing garbage collection
  + Two passes happen during mark and sweep.
  + Pass one marks all objects than can be accessed by any thread in the program.
  + Pass two deallocates all unmarked objects.
  + All execution threads are suspended during this process, unpredictable pauses

# **4 : Pointers and References**

## **Pointers**

* Pointers are variables that store a memory address.
* Pointers variable names and their values (memory address) both exist on the stack.
* Pointers don’t store values directly.
* Pointers can point to data on the stack or the heap.
* Pointers variables have their own memory address (via variable name) in addition to their memory address value that point to a chunk of allocated memory.

int\* myIntPtr = NULL; // int\* myIntPtr = 0; // 1 memory address for name.

int myInt = 10;

int myInt2 = 20;

myIntPtr = &myInt; // assigns memory address of myInt to myIntPtr

cout << \*myIntPtr << endl; // 10

myIntPtr = &myInt2; // re-assigns the ptrs value a different memory address (myInt2)

cout << \*myIntPtr << endl; // 20

Sets the value directly to the pointers pointee(myInt2) with dereference operator(\*) Since myIntPtr is still pointing to myInt2, the value of myInt2 gets changed

\*myIntPtr = myInt; // \*myIntPtr = 10; myInt2 = 10;

cout << myInt2 << endl; // 10

cout << \*myIntPtr << endl; // 10

char\* myCharPtr = new char[8]; // new char of size 8, asking for a block of 8 bytes

memset(myCharPtr, 0, 8); // memset() fills a block of memory with data we specify

Fills in the value of 0 starting at memory address myCharPtr for 8 bytes. {0,0,0,0,0,0,0,0}

// int array can be stored as a pointer

int\* intArrPtr = new int[5]; // int pointer with 40 blocks of memory to store 5 contigous ints.

int intArr[5]; // int array able to store 5 int values.

delete intArrPtr[]; // needs [] here because of the brace syntax indicating multiple blocks of memory

*Difference between int\* myPtr; (uninitialized) and int\* myPtr = NULL; (null pointer)?*

* A null pointer is guaranteed to compare unequal to any pointer that points to a valid object,
* An unitialized pointer is NOT guaranteed to compare correctly to another pointer.

## **References**

int a = 5;

int& b = a;

* A **reference variable** is an **alias** to another variable.
* A reference variable’s **value** is a **memory address of another variable**.
* **Reference variables** have to be initialized as they are declared.
* When accessing a **reference variable**, that variable is implicitly de-referenced, giving it the appearance of having a direct value, even though the reference variable is only a memory address.
* References allow multiple return values from a function by passing variables by reference.
* Return values from a function can also be references themselves.
* Don’t return a local variable in a function as a reference (local variable is freed on return).

*Do reference variables use memory?*

* It is allocated on stack. But you will never be able to get this address from you code because you are not required to know the address. So; a reference does occupy memory. In this case, it is the stack memory, since we have allocated it as a local variable.

*Are reference variables stored on the stack?*

* The C++ standard does not specify, it is implementation-defined.
* The usual approach is that a reference is implemented as a pointer which is allocated on stack.
* However if this reference is not passed anywhere, it may be optimized away and simply replaced by the variable it points to. It is optimized away by the compiler saying "make a note in the symbol table that b is now an alias for a."

*How are reference variables used within classes?*

* Initializing the member references are required using a member initializer list constructor.
* Member reference variables are used in inheritance hierarchies.

## **Pointers v References**

* Pointers and References get around scope.
* A reference variable does NOT contain its own accessible memory address.
* References ‘has-a’ memory address, while Pointers ‘is-a’ memory address.
* References cannot be NULL, while pointers can be NULL.
* Once a reference is initialized to an object, it’s memory address value cannot be changed, whereas a pointer’s value can be re-assigned to a different memory address that points to a different object.

# **5 : Input and Output**

**CIN and COUT**

string s;

std::cin >> s; // Take input using cin

std::cout << s; // Print outputs until first ‘space’ char **or a NULL char**

**GETLINE**

std::string name;

std::getline (std::cin, name); // Grabs all characters until newline is entered.

// **DONT** use both std::cin >> std::getline(cin, name) in same block, cin gets flushed

**Printf() |** #include<stdio.h>

int printf( const char \* format, ... ); // printf prototype signature

printf(“Ryan Smith”); // outputs Ryan Smith

%[flags][width][.precision][length]specifier – // flags to specify output’s format

%d or %i – unsigned decimal integer | %f – float | %s – string | %p – ptr address

int a = 5;

printf("%i", a); // outputs 5

**float** Self\_paced\_price = 3999.0;

**float** system\_design\_price = 10999.0;

**printf**("the value of system design price / self paced price is %.3f",

           system\_design\_price / Self\_paced\_price);

// Output: the value of system design price / self paced price is 2.750

* .3f depicts the value that should be printed within 3 decimal places. // default: 6

**Input – Read from this file (**#include <fstream>)

ifstream qFile("XoaX.txt"); // input file stream - read from this file.

if (!qFile.good()) { return 1; } // validate file opens first

string qLine;

while (!qFile.eof()) // read data until getting to end of file

{

// need to #include<string> for getline(.., ..)

getline(qFile, qLine); // read in a single line and store it in qLine

cout << qLine; // display the line of data read from getline to the screen

}

qFile.close();

**Output – writes strings to file**

ofstream qFile("XoaX.txt", ios\_base::app); // append mode – adds to eof.

string qLine1("Those who grant sympathy to guilt,");

string qLine2("grant none to innocence.");

qFile << qLine1 << endl; // << insertion operator – inserts string into file.

qFile << qLine2 << endl; // << appends the 2nd string on new line after 1rst

qFile.close();

# **6 : CStrings**

## **Char Arrays (C-Strings) | #include <cstring>**

const int MAX = 20;

char caName[MAX];

cout << "What is your name? " << endl;

cin.getline(caName, MAX);

**strlen(myString)** // C-String length

**strcpy(destination, source)** // C-String copy – deep copy

**strcat(finalString, addedString)** // C-String appends final string w/ added string.

* FinalString must be initialized to a large enough size to hold both the initial cstring and the added cstring. Strcat does NOT add a space “ “.

char myCArray[] = {'a', 'b', 'c'}; // no null character added

cout << myCArray << endl; // this will print abc (and a bunch of garbage values)

char myArray[] = "abc"; // this does add a null character

char myCArray2[] = {'a', 'b', 'c', 0};

cout << myCArray2 << endl;

* Character Arrays exist on the stack and need to specify a size either implicitly or explicitly.
* Char arrays are NOT string literals, but they can be initialized with one.
* Char arrays act as ptrs with a block of memory implicitly initialized & pointing to first char of array
* Char array’s memory address can not change, but the value it points to can (via indexing).
* When initializing a char array to a string literal, a null character will automatically get added as part of the memory allocated.
* If a char array’s size is explicitly defined (to 10 for example), then the string literal it is initialized to can be no longer than 9 characters. char caName[10]; cin >> caName // needs to be 9 chars or less
* Char arrays and non-char arrays don't share all the same syntax, only some.

## **Char Pointers (C-Strings):**

* char\* name = “Ryan”; **// error:** char pointers can **NOT** be initialized to string literals in C++ v11
* char\* pointers can only be assigned to known memory values on the stack (a char array) or given a dynamic chunk of memory that exists on the heap.
* String literals exist in undefined inaccessible memory, therefor a string literal’s direct memory address can **not** be assigned to a pointer.
* When STL strings, char[], and const char \* are initialized to a string literal, the C++ auto assigns a chunk of memory (the syntax implies the memory allocation).
* Those variables exist on the stack making memory accessible and assignable (to a string literal).
* The char\* is a pointer that points to the first character of a character array.
* Changing the value of a char pointer to a different string only changes the memory address of the char\*.
* char\* only needs to be de-allocated with [] notation ONLY if memory was allocated using new() keyword.

const char\* name = “Ryan”; // const char pointers **CAN** be initialized to string literals.

char\* myName = new char[5];

strcpy(myName, "Ryan"); // closest thing to assigning a string literal to a char\* without const

cout << myName << endl; // cout, printf(), and indexing[] auto dereference value

delete[] myName; // Once initialized, char ptr is a char array, dynamic needs [] when deleting

**String Literals**

* String literals are any value in double quotes(“ “).
* String literals are stored in memory at an undefined location.
* String literals are immutable, therefor, a string literal value cannot be modified directly.
* String literals do not exist on the stack, they are undefined, hence cannot be modified.
* String literals are different than char arrays, char pointers, const char pointers, and STL String objects.
* char\* (not const char\*) are unable to assign a string literal to it directly because a string literal itself DOES NOT represent a block of memory somewhere in memory. Char\* need to point to valid memory.
* Any string literal that is assigned to string variable automatically adds a null character (‘/0’) – 1 byte

**Char Pointer of Pointers – Array of char pointers**

char\*\* cppStrings = new char\* [3];

cppStrings[0] = new char[9];

cppStrings[1] = new char[13];

cppStrings[2] = new char[4];

strcpy(cppStrings[0], "XoaX.net");

strcpy(cppStrings[1], "Michael Hall");

strcpy(cppStrings[2], "C++");

for (int i = 0; i < 3; i++){

cout << cppStrings[i] << endl;

}

int iLength = strlen(caName) + strlen(caLesson); // assume name and lesson are init.

char\* cpFullName = new char[iLength + 2]; // +2 for one 'space' and /0 (null char)

strcpy(cpFullName, caName);

delete [] cpFullName; // syntax to de-allocate a char\*

string to string ‘deep’ copy: **// copies individual characters**

strcpy performs deep copy. It copies data contained in memory at address of new pointer.

* *char\** ***strcpy****(char\* dest, const char\* src);*

**Tokenize a char array**

char caString[] = “XoaX.net”;

char caDelim[] = “,”;

char\* cpSub = strtok(caString, caDelim);

while (cpSub)

{

cout << cpSub << endl; // iterating over a char\* until not NULL

cpSub = strtok(NULL, caDelim);}

# **7 : STL Strings**

CHAR OR STRING TO INT ??

INT TO CHAR OR STRING ??

BUILD A STRING OR CHAR[] WITH INTS AND STRINGS??

std::string name;

* Strings are objects that don’t need to be initialized.
* Strings can be initialized directly to string literals.
* String variables can be assigned a string literal value after initialization.

***Extracting a char from a string at a given index i.***

std::string nestedStr = “Ryan”;

char ch = nestedStr.at(i);

***Converting a string to char array***

std::string name = "Ryan";

name.c\_str(); // straight conversion, no storage

***Copying a stl string to char array***

std::string name = "Ryan";

char myName[5]; // char array size needs to account for null character.

strcpy(myName, name.c\_str());

***Finding STL String’s Size or Length***

std::string name = "Ryan";

std::cout << name.length() << std::endl;b

// name.length() does NOT include the null “character”.

***Returning substrings***

std::string name = "Ryan";

std::string s1 = name.substr(0, 2);

std::cout << s1 << std::endl;

***Checking if stl string contains using find():***

std::string s1 = "Ryan";

std::string s2 = "an";

if (s1.find(s2) != string::npos) // find needs to be evaluated as binary exp.

{

std::cout << "Found" << std::endl;

}

***Copying an STL String using ‘soft copy’***

string name = "Ryan";

string cpyName = name; // copies memory address

cout << name << endl;

***Appending or concatenating STL Strings:***

string name1 = "Ryan";

string name2 = "Smith";

name1.append(name2); // Output: RyanSmith

cout << name1 << endl;

std::string newString = "";

newString += "aa";

cout << newString << endl; // Output: aa

***Compare STL Strings:***

string name1 = "Ryan";

string name2 = "Ryan";

if (name1 == name2) // Are equal when == 0

cout << "Same" << endl;

else

cout << "Different" << endl;

***Tokenize STL Strings (using stringstream)***

// #include <sstream>

string line = "GeeksForGeeks is a must try";

vector <string> tokens;

// stringstream class check1

stringstream check1(line);

string intermediate;

// Tokenizing w.r.t. space ' '

while (getline(check1, intermediate, ' '))

{

tokens.push\_back(intermediate);

}

// Printing the token vector

for (int i = 0; i < tokens.size(); i++)

cout << tokens[i] << '\n';

***Defining a string array***

const int size = 10;

std::string strArray[size];

Difference between (cstring v string.h) v string ?

* **<cstring>** c string functions, **C++ header**, places identifiers in the standard namespace. std::printf()
* **<string.h>** c string functions, **C header**, deprecated in C++, places in global namespace. print()
* **<string>** string object class and related functions, **C++ header**, works on **string objects** not cstrings.

Can you switch on strings in C++?

* No in C/C++. Yes in other OOP languages like Java, C#, ect..

**Char Array (C-String) v STL Strings:**

* stl strings are objects v char arrays are not, they are an array of primitives.
* stl strings can hold characters of any size dynamically, char arrays need to be sized on declaration.
* char arrays memory address cannot change, strings can point to something different.
* char arrays can only change value by indexing individual elements.
* char arrays can not be larger than initialized size, strings don’t need to worry about size.

**Char Array v Char Pointer**

* char array gets a block of memory reserved upon declaration; char pointers don’t need to.
* char arrays can be initialized to string literal, char pointers cannot (w/o const keyword).
* char arrays can only point to the memory address it was initially assigned, char pointers memory address can be re-assigned to point to another string or char array somewhere else in memory.

**STL Strings v char arrays[] v char\* v const char\***

* **STL strings** are best for readability, quick prototyping, large data values, frequently changing values.
* **char arrays** are faster than strings, exist on the stack, and allows for individual character manipulation. Char arrays are preferred fwor constant values and single characters.
* **char\*** is memory efficient as only one pointer is required to refer to the string.
* **char\*** don’t require size initialization upon declaration of pointer but do require a valid chunk of memory to point to be initialized to a value.
* **char\*** works best where speed is at a premium and there exists an established char array or string it needs to reference outside of scope and heap access is required.
* **const char\*** work best when performance is needed but modification of the string value is not. Const char ptrs are very similar to char arrays, but there string value cannot change.

**strlen() v sizeof() v size() length()**

*What is sizeof() used for? Does it work on data structures?*

* C++ sizeof operator is an operator that finds the size of data types and expressions in total bytes.
* **CANNOT** use sizeof to determine that total amount of memory taken up by a **vector**.
* **CAN** be used on **primitive arrays** to calculate total memory or declared size.

// ### STRLEN() ###

**Works**: string literals, const char\*, char\*, char array[] (initialized to string literal)

Does NOT work: STL strings, char arrays with comma separated lists, int arrays

DOES NOT count NULL Terminator.

// ### SIZEOF() ###

**Works**: string literals, char arrays with comma seperated lists, int arrays

Does NOT work: STL strings, pointers.

DOES count NULL terminator '/0' for char arrays set to string literal

Adds NULL terminator '/0' to string literals, but not comma separated lists

// ### SIZE() ###

**Works**: everything but pointers

Does NOT work: Pointers.

DOES count NULL terminator '/0'

Adds NULL terminator '/0' to string literals, but not comma separated lists

Size(“string literal”) syntax needed for string literals.

// ### .LENGTH() ###

**ONLY Works**: STL String objects

DOES NOT COUNT Null Terminator ‘/0’.

**Classes**

The **public** portion of a class definition is known as its **interface**.

The **private** portion is known as **Data Hiding**.

*What is a* ***friend class*** *in C++, when would you use one?*

* It’s a **keyword** given to a function or a class that gives *access to private members* of the function or class that it’s a friend of. Similar to protected variables of a parent class.
* For example, say you create an dynamic array class, and you have a separate iterator class.  You would want to make this iterator class a friend so it can have direct access to private members of dynamic array class.

# **8 : Constructors**

**Copy Constructors**

CPoint(const CPoint& myCPoint)

{

mdX = myCPoint.mdX;

}

* **Copy Constructor** passes in a object parameter of the same type of the class being constructed.
* If copy constructor isn’t defined, the C++ compiler implictly creates a default copy constructor for each class which does a member-wise copy between objects.
* A copy constructor needs to be defined explicitly only if an object has pointers or any runtime allocation of the resource like*a file handle*, a network connection, etc.

In C++, a Copy Constructor may be called for the following cases:

* When an object of the class is returned by value.
* When an object of the class is passed (to a function) by value as an argument.
* When an object is constructed based on another object of the same class.

*What is a copy ellison?*

* Copy elision refers to a compiler optimization technique that eliminates **unnecessary copying of objects**.
* Makes ‘*returning by value’*or ‘*pass-by-value’* feasible in practice.
* When executing a copy constructor, the compiler bypasses creating duplicate objects.

*What is RVO?*

* **Return Value Optimization** is a technique that gives the compiler some additional power to terminate the temporary object created which results in changing the observable behavior/characteristics of the final program.
* It is guaranteed to be called in a modern compiler program.
* The returned object is constructed in place of the function call. It does not allow the creation of a local object that is used as a return value.
* Using the “**-fno-elide-constructors”**flag mandatorily calls copy constructor (bypassing RVO).

Copy Elision v RVO?

* Copy Elision is an abstract concept that can be flagged in the compiler config file. Whereas, RVO is a specific form of copy elision, the actual technique the compiler performs to optimize duplicates.

*What is the* ***explicit*** *keyword for?*

* Prefixing the **explicit** keyword to the constructor prevents the compiler from using that constructor for implicit conversions.
* You have pass in values in parentheses as part of construction.
* The reason you might want to do this is to avoid accidental construction that can hide bugs.
* Converting Constructor allows for implicit construction whereas explicit keyword does not.

*When would a constructor use an* ***implicit conversion****?*

* When you don’t pass in arguments when constructing an object, but instead have the new object = <some argument value>. <https://en.cppreference.com/w/cpp/language/converting_constructor>

# **9 : Initializer Lists**

class fraction

{

private:

int numerator;

int denominator;

public:

// member-initializer lists are constructors included in class’s definition

fraction(int n, int d) : numerator(n), denominator(d) {}

};

**Initialization order**

The order of member initializers in the list is irrelevant: the actual order of initialization is as follows:

* If the constructor is for the most-derived class, virtual bases are initialized in the order in which they appear in depth-first left-to-right traversal of the base class declarations.
* 2nd, direct bases are initialized in left-to-right order as they appear in this class's base-specifier list.
* 3rd, non-static data member are initialized in order of declaration in the class definition.
* Finally, the body of the constructor is executed.

*Difference between standard assignment constructors and initializer list constructors?*

* An initializer list is a compact notation equivalent to a sequence of assignment statements. But they have the advantage of running before the constructor's body, so the member variables are ready to use as soon as the body runs.
* Initializer Lists prevents object duplication.

# **10 : Preprocessor Directives**

constexpr auto PI = 3.14159;

#define MAX(a, b) ((a >b) ? a : b)

#define WINDOWS

#ifdef WINDOWS

int dx = 1.0;

#endif

int main()

{

cout << PI \* 2 << endl; // Output: 6.28…

}

* Preprocessor directive starts with **'#'** symbol syntax
* Preprocessor directives occur before code is compiled.
* Use preprocessor directives to define macros.
* Use preprocessor directives to include header files.
* **Preprocessor** **if** statements determine which code will be compiled vs. which code will be executed.
* If statement is executed and plugged in before compiling takes place; good for multiple platforms

*What is the difference between Preprocessor Directives and Header Files?*

* **Preprocessor directives** are used to instruct the preprocessor to perform certain tasks before the code is compiled, while **header files** contain declarations and definitions that can be included in a C/C++ source file. #including header files is a type of preprocessor directive instruction.

*What is the difference between a header file with a .h extension and one without?*

* The difference between a header file with a .h extension and one without is the naming convention.
* Files with a .h extension are typically used for C++ header files, while files without an extension are used for C++ template header files.
* Both types of files serve the same purpose and **can** contain the same type of declarations and definitions.

# **11 : Templates**

// template <typename T> syntax needs to be directly above function using template

template <typename T> // T is the template datatype (T can be any variable name)

T Max(T xVal1, T xVal2) // Max is a func that pass and returns the **T template datatype**

{

if (xVal1 > xVal2)

{

return xVal1;

}

else

{

return xVal2;

}

}

template <typename T>

void Swap(T& xrItem1, T& xrItem2)

{

T xTemp = xrItem1;

xrItem1 = xrItem2;

xrItem2 = xTemp;

}

*What’s the difference between Template functions and Template classes?*

* A function template in C++ is a single function template that works with multiple data types simultaneously, the class template defines a family of classes in C++.
* The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated

*Do you need to specify a data type when calling a template function or member template function?*

* Yes. Need to be specified in angle brackets <>.

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

int main()

{

int result1 = add<int>(2, 3);

double result2 = add<double>(2.2, 3.3);

}

*How do you use Template functions?*

* Define it first (shown below) and to call the template, specify datatype in angle brackets and params.

template <typename T>

T functionName(T parameter1, T parameter2, ...)

{

// code

}

**Calling a template function**

functionName<dataType>(parameter1, parameter2, ...);

*When do you use Template functions?*

* A function template is used to perform the same function on multiple data types. An alternative approach to this is using function overloading. However, using a function template is a better approach to writing less and more maintainable code

*Can template functions exist within a class?*

* Yes, member templates refers to both member function templates and nested class templates.

**BubbleSort with Template Functions - O(n^2)**

template <typename PData> // n-1 comparisons + n-2 comparison = n(n-1)-n/2 = O(n^2)

void Bubblesort(PData xaArray[], int iLength){ // moves largest num to end of array

for (int iEnd = iLength - 1; iEnd > 0; --iEnd) {// stops at end = end - 1

for (int iIndex = 0; iIndex < iEnd; ++iIndex) { // compares from begin

if (xaArray[iIndex] > xaArray[iIndex + 1]) {

Swap(xaArray[iIndex], xaArray[iIndex + 1]);

}

} // moves left-right during inner pass foreach outer pass,

} // then moves the end index back 1 when the inner pass finishes.

}

**Template Function Specialization**

// Write a template function that works for most data types but not all(top)

// Write a 2nd overloaded template function that works for data types not covered

template <typename T> // default function template

void PrintType(T xItem)

{

std::cout << "Unknown Type" << std::endl;

}

template <> // Function names need to be exactly the same for this to work.

void PrintType<>(int iItem) // version used for specific arguments.

**Template Classes**

* Class templates are useful when a class defines something that is independent of the data type.
* Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

*How do you use Template classes?*

* First declare the template by proceeding class header with template <class T>. ***Note****:* Not all variable types (parameters, private members, and return types) have to use the T template type.

**// Simple Template Class Example**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <string>

#include <stdexcept>

using namespace std;

template <class T>

class Stack {

private:

std::vector <T> elems;

public:

void push(T const&); // push element

void pop(); // pop element

T top() const; // return top element

bool empty() const { // return true if empty.

return elems.empty();

}

};

template <class T>

void Stack<T>::push(T const& elem) {

// append copy of passed element

elems.push\_back(elem);

}

template <class T>

void Stack<T>::pop() {

if (elems.empty()) {

throw out\_of\_range("Stack<>::pop(): empty stack");

}

elems.pop\_back(); // remove last element

}

template <class T>

T Stack<T>::top() const {

if (elems.empty()) {

throw out\_of\_range("Stack<>::top(): empty stack");

}

return elems.back(); // return copy of last element

}

int main()

{

Stack<int> intStack; // stack of ints

Stack<string> stringStack; // stack of strings

// manipulate int stack

intStack.push(7);

cout << intStack.top() << endl;

// manipulate string stack

stringStack.push("hello");

cout << stringStack.top() << std::endl;

stringStack.pop();

stringStack.pop();

}

*When do you use Template classes?*

* Class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

*Can swapping different data types cause logic errors when using a template function?*

* Function template will prevent you from swapping objects of different types, because the compiler knows the types of the *a* and *b* parameters at compile time.

*Difference between a Macro and a Template?*

Like Macros, **Templates** are expanded at compiler time. The difference is, that the compiler does type-checking before template expansion. **Templates** (like inline functions) are type safe, Macros are not.

# **12 : Function Pointers**

void Function1()

{

std::cout << "Function1" << std::endl;

}

int main()

{

void (\*fnPrintName)(); // declare and initialize FP.

fnPrintName = Function1; // Assumes Function1 was defined somewhere in code

fnPrintName(); // fnPrintName being a function pointer, not a function

}

* Pointers can point to functions instead of variables.
* Function Pointers are typed based on the argument type they take in and return type
* Function pointers declaration names are defined within parentheses proceeded by a asterisk.
* Function pointer declaration need the return type before the name and a parenthesis after the name.
* Those arguments and return type need to exactly match the function that the function pointer points to.
* Can NOT have arrays of functions but can have arrays of function pointers.
* Like passing in arrays, a function definition might have a function\* parameter but when calling, you can pass a function name (not func\*) directly without the ‘()’ parentheses after the function name.

**FUNCTION Pointer Array Example**

double Add(double a, double b){ return a\*b; }

double Multiply(double a, double b){ return a+b;}

int main()

{

// Function Pointer array example | Add and Multiply assigned to fn ptr array

double (\*fnOperation[2]) (double, double) = {Add, Multiply};

cout << fnOperation[0](2.0, 3.0) << endl; // Output: 5

cout << fnOperation[1](2.0, 3.0) << endl; // Output: 6

}

**Member Function Pointers**

int (MyClass:: \*myfnptr)(); // member function ptr with no args and returns an int.

* **Declaration of member function pointers** are the same as local pointer except the member function pointer is proceeded by the **class name and scope operator ::**
* Typed by class, can only point to member functions of the same class.
* Member Function Pointers CAN NOT point to non-member functions, or local functions.
* Local (normal) function pointers CAN NOT point to member functions.
* Const member variables disables assignment and move semantics for a class.

*When should I use a function pointer?*

* When using callback functions (aka asynchronous functions).

What is a Callback Function?

* A Callback function is a function that is not called explicitly by the programmer.
* Callback functions relies on some mechanism that continually waits for events to occur.
* When an event occurs, a callback function is coupled to that event and that callback function runs.
* This mechanism is typically used when an operation(function) takes a long time for execution and the caller of the function does not want to wait till the operation is complete.

*What is a void pointer and when is it used?*

* A void pointer is a pointer that has no associated data type with it. A void pointer can hold an address of any type and can be typecasted to any type.
* It’s rarely used in C++ (built for C). Functors and Templates are used in its place.

# **13 : Polymorphism**

int main()

{

CShape\* qpaShapes[2]; // Array of base class objects, Cshapes (ptrs)

CCircle qCircle1(1.0); // Construct a specific CShape Object, Cirle

CSquare qSquare1(2.0); // Construct a specific CShape Object, Square

qpaShapes[0] = &qCircle1;

qpaShapes[1] = &qSquare1;

for (int i = 0; i < 5; i++)

{

// Area() is marked as virtual in base class.

double dArea = qpaShapes[i]->Area();

std::cout << "Area[" << i << "] = " << dArea << endl;

dAreaSum = dAreaSum + dArea;

}

}

* Polymorphism means to have ‘many forms’.
* Polymorphism is a runtime selection of overridden function based on the class of the object.
* Polymorphism works with class inheritance and requires parent class’ function be marked ‘virtual’.
* Polymorphism allows class-specific code to be invoked without having to specificy which definition to invoke in the calling code.
* Polymorphism is useful when we want to use a base class interface to make all the derived class objects in a base class array look all the same from the user's perspective.
* Computation takes place depending on what type of object the pointer points to.
* Polymorphism is good for less coupling, less code bloat, and code reuse.
* Polymorphism is bad for readability and debugging. Polymorphism is less used these days.
* Polymorphic function calls use dynamic dispatch incur a run time penalty. CPU branch predictions.

**Virtual Overriding functions (**Virtual functions are overridden)

* **Virtual functions** - A method that has multiple versions of the same method in which invocation is determined at run time by the class type of the object which invoked it
* Denoted by the **‘virtual’** keyword in **front of function prototype of the base class**.
* Regular, non-virtual function invocations are determined at compile time.
* **Virtual Functions** are used in **polymorphism** to invoke the version of the virtual function associated with the object type the pointer is pointing to.
* **Virtual Functions** take **longer** to **invoke** because the **address of the function must be looked up in a table**.
* **Virtual Functions** calls use dynamic dispatch that incur a run time penalty because Virtual Machine calls mess with the CPUs branch prediction.

class Base

{};

class Derived : public Base // Need public keyword before Base

{};

int main()

{

Base base;

Base\* baseptr = &base;

Derived derived;

Derived\* derptr = &derived;

baseptr = &derived;}

* For a function to be considered virtual, the base class’s version of the same function name must be marked with the virtual keyword in front of it.
* If the **base pointer is associated with a derived object** and a **virtual function is called**, the compiler knows to **run the derived version of that function**, even though the pointer is of base class type.
* If baseptr points to the derived object, the compiler will still run base version of the functions**,** unless that function is tagged with a ‘virtual’ keyword in the base class.
* Derived class pointers can NOT point to an object of the base class, however, a base class pointer can point to an object of a derived class object.
* Making a destructor virtual ensures that the base class and derived classes don't leak.

**Pure Virtual Functions:**

virtual double Area() = 0;

* A Pure Virtual Function is a function that must be overridden in a derived class.
* Pure Virtual Functions need not be defined.
* Pure Virtual classes disables the ability to directly instantiate objects of the class type.
* Pure Virtual classes allows for the ability to use pointers to instantiate objects of that base class type.
* Any class that contains pure virtual without re-defining it is an abstract class.
* If a class has **only pure virtual functions** and no data members (Car class) - it's an **i*nterface.***
* If a class contains at least one but **not all pure virtual functions**, it's an ***abstract class*.**

**Interface (API)** - a set of related methods, outside of any class.

* Only pure virtual functions, no data fields.
* Important for classes that only support single inheritance.
* Useful when unrelated classes need to provide a common way to invoke conceptually related functionality.  Great for plug in play architecture where components are interchanged at will.  Interfaces are prominent in strategy patterns.

**Abstract class** - an incomplete class that declares but doesn't define all of it's methods.  Abstract classes can have **some definitions** and have data members, but can not be instantiated directly.  Useful when all derived classes share some functionality the parent abstract class implement.

**Multiple inheritance** - Useful for combining aspects of two disparate class hierarchies (for example, if two frameworks define their own base class for exceptions,  you can use mutliple inheritance to define a class that can be used in either framework).

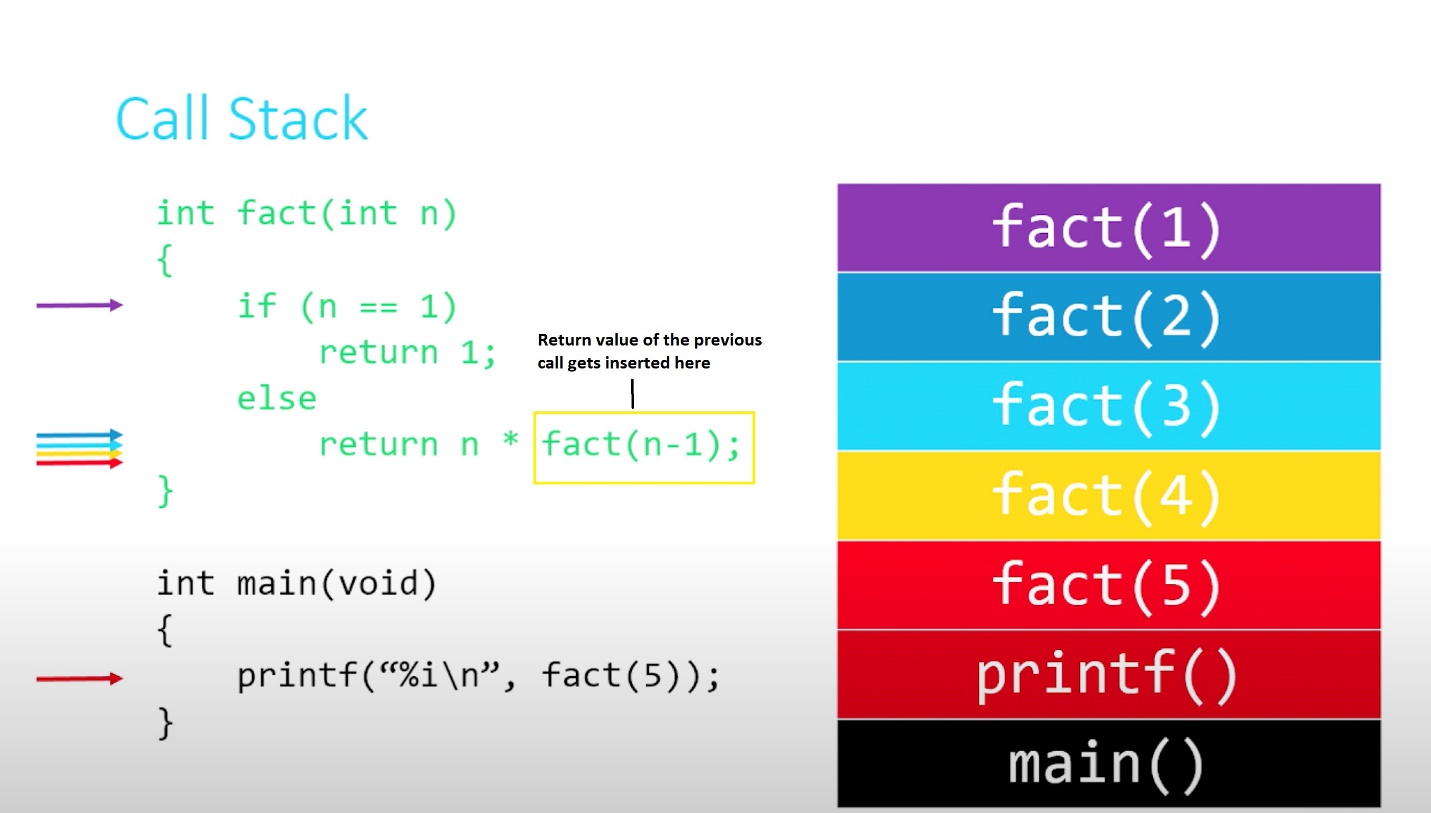
* The problem with Multiple inheritance is ambiguity (the diamond problem).
* The solution is to either disambiguate the reference with a namespace B::flag = nflag, or make the B, and C classes virtual so that only one copy of A's base member variable flag can exist.
* MI other issue is inadvertently hidding members based the order the base classes are initialized.

# **14 : Big ‘O’**

Big O runtime cost (operations needed) vs. memory footprint (memory storage required).

* O(1) – constant time
* O(log n) – logarithmic time
* O(n) – linear time
* O(n log n) – quasilinear time
* O(n^x) – Polynomial time – n raised to the x operations
* O(x^n) – Exponential time – some number performed n exponential times
* O(n!) – Factorial time

# **15 : Recursion**



# **16 : lValues & rValues**

**Dsda**

# **17 : Questions**

**C++ Class Citizens**

*What is a first-class citizen in C++?*

Being "first-class” means **entity can be programmatically manipulated at run-time** with three properties:

* It can be used, wherever "ordinary" values can, i.e., passed, assigned, and returned from functions
* It can be constructed, without restriction, wherever "ordinary" values can, i.e., locally, in an expression, etc.
* It can be typed in a way like "ordinary" values, i.e., there is a type assigned to such an entity, and it can be freely composed with other types.

*What is a second-class citizen in C++?*

A second-class construct is one which is an intrinsic element of the language with the following properties:

* It must form part of the lexical syntax of the language
* It may have operators applied to it

*What are closures in programming? Does C++ support closures?*

* A closure is a persistent scope which holds on to local variables after code block executes.
* **Closure**, also **lexical closure** or **function closure**, is a technique for implementing [name binding](https://en.wikipedia.org/wiki/Name_binding) in a language with [first-class functions](https://en.wikipedia.org/wiki/First-class_function).
* True closures are not supported in C++. Lambdas are in C++v11

*What are lambda expressions?*

* A lambda is an unnamed function that is useful (in actual programming, not theory) for short snippets of code that are impossible to reuse and are not worth naming.

**Questions (Miscellaneous)**

*Difference between* ***#include <filename>*** *and* ***#include "filename"****?*

* **#include <filename>** The preprocessor searches in directories pre-designated by the compiler/IDE for C standard library and target platform header files.
* **#include "filename"** syntax is for programmer-defined header files that are typically included in the same directory as the file containing the directive.

*Difference between a 32-bit vs 64-bit application?*

* The size of the memory addresses and general-purpose registers than an application uses.
* 64-bit has faster processing but uses more memory.
* 64-bit data structures have more cache misses.

*What are the two major networking issues?*

* **Latency**: the time it takes a given bit of information to get from one point to another on the network
* **Bandwidth**: the rate at which data moves through the network once communication is established.

*How is mobile programming difference from programming on a normal computer?*

* Mobile devices use mobile specific operating systems like Android and iOS.
* Mobile device operating systems have different paradigms for file system access, memory access, and inter-application communication.
* Mobile programming requires attention to power consumption, storage, and bandwidth.
* Mobile devices use different input devices and accessories for controls and gameplay.

*What is AJAX?*

* Web development technique or style.
* Asynchronous JavaScript and XML.
* Architectural style for building interactive web applications.
* Data exchanges with the server occur in the background over HTML.

*What is JSON?*

* JavaScript Object Notation is a lightweight data-interchange format.
* Easy for humans to read-write, easy for machines to generate-parse.
* Language independent but uses conventions familiar with the C family and Java

Built on 2 structures:

* Name/value pairs – realized as an object, record, struct, dictionary, hash table, keyed list, or associative array in programming language
* Ordered list of values – realized as an array, vector, list, or sequence.

*C++ versus Java?*

* Different design goals
* Java cares about security, portability, and simplicity.  Java is compiled to virtual machine byte code.
* C++ cares about performance, backwards compatibility with C, and programmer control.
* C++ is compiled in native machine code.  Operator overload, multiple inheritance.  Auto type casting and using pointers are another feature of C++.

**More Questions**

* std::shared\_ptr?
* smart\_ptr?
* Interface-based Programming?
* Assert
* Try-catch blocks? When and how?
* Interfaces vs. Namespaces? When to use one or the other?
* Using vs Typedef vs typename keywords