A Brief Overview of C++

Michael Camara

Honor Code Pledge: This work is mine unless otherwise cited

CMPSC 220

1 Introduction

C++ is an imperative, object-oriented language that Bjarne Stroustrup started developing in 1979 (Stroustrup, 2014, p. 839). Stroustrup sought to create a language that would improve upon C, particularly by facilitating abstraction and object-oriented techniques, which had previously been considered unfeasible for real-world use (p. 840). The language was initially named "C with Classes," indicative of the object-oriented principles he was trying to implement. It wasn't until the language's commercial release in 1985 that Stroustrup officially changed the name to C++, as it is known today. He continued to refine C++ with colleagues at the Bell Telephone Laboratories' Computer Science Research Center in Murray Hill, New Jersey, until standardization of the language began in 1990. Stroustrup has continued to lead the development of new C++ features, with the next major ISO standard having an expected release in 2017 (C++17).

Even though C++ is not one of the newest programming languages, it is certainly one of the most ubiquitous. The IEEE Spectrum magazine reports C++ as the third most popular programming language of 2015, only superseded by C (second place) and Java (first place) (Cass, 2015). Trends from GitHub, one of the largest version control repositories, similarly show C++ remaining a popular choice among programmers, ranking seventh out of the top languages used for projects on the site (La, 2015). These trends show that C++ offers some unique features that even more modern languages like Python and Java can't quite replicate. The following sections outline some of these features and how they relate and differ from other programming languages.

2 Basic Format

To illustrate the basic format of a C++ program, the *HelloWorld.cpp* file includes some of the fundamental components. Figure 1 shows this simple program in its entirety:

```
#include <iostream>
using namespace std;

int main() {
        cout << "Hello, World!";
        return 0;
}
```

Figure 1: Standard "Hello World" Application in C++

Beginning in lexical order, the `#include <iostream>` is a preprocessor directive that specifies that certain statements in the program use classes or functions from the standard `iostream` library. C++ is a compiled language, and therefore the compiler needs to have explicit references via `#include` directives to ensure that any required files are properly linked prior to execution. It is similar in purpose to the `import` statement in Java, which likewise allows the program to utilize classes and functions from different sources. Unlike Java, however, the `#include` directive does not simplify the syntax required to use such external sources. For instance, in Java, if you used the statement `import java.util.Random;` then you could immediately declare an object of type `Random` without needing to specify its fully qualified name (unless name clashes would require such action). This is why the `using namespace std` statement is included: it is a construct used to make names from some included source directly accessible. If this line were omitted from the code in Figure 1, then `cout` would need to be replaced with `std::cout` to prevent a compilation error. Ultimately, the `using` declaration is more of an optional convenience feature to simplify coding, but care must still be taken to avoid potential name clashes.

Function declarations in C++ are very similar to C and more modern languages like Java. They each begin with a return type, followed by a user defined name for the function, and finally a list of formal parameters enclosed in parentheses. Interestingly, as shown in Figure 1, the `main()` function that begins a program's execution has a return type of `int` in C++ instead of `void`, as seen in Java. Stroustrup (2014) mentions that "a zero returned by `main()` indicates that the program terminated successfully," which may aid in testing and debugging (p. 47). Finally, inside of the `main()` function in Figure 1 is a simple output statement. Although a programmer could customize their own input and output streams, there exist standardized versions of these streams

accessible in the `iostream` standard library: `cin` for input and `cout` for output. These streams further utilize the insertion `(<<)` operator for output and the extraction `(>>)` operator for input. Further information about this process is described in the *Input and Output* section.

3 Classes and Headers

As an object oriented language, classes are a particularly important part of C++. Although many other modern languages use classes as well, the format used by C++ appears unique with the inclusion of header files. The *Header.h* file is a simple header file, which is meant to declare the member functions and variables for a particular `ExampleClass` class while omitting any actual definitions. Figure 2 shows how the header for `ExampleClass` is included in the *ClassesAndStructs.cpp* file and used to help define the constructor of the class:

```
// Declare ExampleClass in Header.h
#include <string>
class ExampleClass {
public:
        ExampleClass(int x, std::string name);
... // Other members omitted here
};

// Define ExampleClass in ClassesAndStructs.cpp
#include "Header.h"
#include <string>
using namespace std;
ExampleClass::ExampleClass(int x, string name) {
        this -> x = x;
        this -> name = name;
}
```

Figure 2: Defining Class Members Declared In Header File

Note that the *ClassesAndStructs.cpp* file first needs to use the `#include Header.h` directive in order to link to that particular file. Next, the scope resolution operator `(::)` is used to specify the constructor as defined by the header file, using the format `TargetClass::TargetMember`. In Figure 2 the constructor is being defined, and thus we have `ExampleClass::ExampleClass`, followed by the same function signature specified in the header. Note that any attempt to define a member variable or function not explicitly declared in the linked header file will result in a compiler error. This separation of concerns may help C++ programmers by preventing deviation from set design decisions. Figure 2 also shows the use of the `this` and `->` ("arrow") operators, which are used

similarly in other languages like Java to implicitly point to the instance variables of the current object.

Although header files can be useful for enforcing class structure, they are not absolutely required. Figure 3 from *ClassesAndStructs.cpp* shows a simple example of an inline class definition:

```
class AnotherClass {
public:
        AnotherClass(int y) { this -> y = y;}
private:
        int y;
};
```

Figure 3: Class Definition Without Header File

This format appears more familiar when compared with other object-oriented languages, although languages like Java require access modifiers before every member, while they can be grouped together in C++ under the access modifier tags as seen here. Additionally, C++ retains the ability from C to create records like structs. Although structs are very similar to classes in C++, classes have the added benefit of encapsulation, as shown in Figure 4:

```
// Instantiate ExampleStruct in the stack frame
struct ExampleStruct s;
s.x = 10;           // ALLOWED: able to directly access any variable
s.name = "Fred";    // ALLOWED: able to directly access any variable
s.someFunction();   // ALLOWED: able to directly access any function
cout << s.x << endl << s.name << endl;

// Instantiate ExampleClass object in the stack frame
ExampleClass c = ExampleClass(5, "Bill");  // Create object in stack
//c.x = 10;         // ERROR: cannot access private variable
c.setX(10);
//c.name = "Gary";  // ERROR: cannot access private variable
c.setName("Fred");
//c.privateFunction();// ERROR: cannot access private function
```

Figure 4: Comparing Structs and Classes

This shows how all members of a struct are considered public, allowing direct access from any source. Classes, however, are able to restrict access through private or protected modifiers. Thus, when the `ExampleClass` object, c, attempts to directly access its private instance variable x, the compiler throws an error since that variable was declared private previously. This encapsulation allows classes to have more meaningfully structure than records, and is one of the hallmarks of object-oriented programming.

Another interesting note about classes in C++ is how they can be instantiated. Figure 5 shows that `ExampleClass` c was instantiated on the stack frame by omitting the `new` operator following the assignment operator. In other languages, like Java, all non-static objects must be allocated on the heap by using the `new` operator, while simply keeping a pointer to that location in the stack frame. The latter operation can still be achieved, however, by changing the object type to a pointer using the unary dereferencing `(*)` operator, again shown in Figure 5:

```
ExampleClass c = ExampleClass(5, "Bill");  // Create object in stack
c.setX(10);
ExampleClass *p = new ExampleClass(5, "Bill"); // Create object in heap
p->setX(10);
```

Figure 5: Comparing Stack and Heap Instantiation

This gives C++ programmers greater flexibility for choosing how they want objects to appear in memory. As further examples will show, C++ provides a blend of high-level and low-level features that contribute to making the language so versatile.

4 Pointers

As the previous example showed, C++ retains the ability from C to declare pointers via the unary dereferencing `(*)` operator. Pointers "are variables that store addresses of values rather than the values themselves," giving C++ programmers more low-level capabilities (Prata, 2012, p. 154). The *Pointers.cpp* file includes a number of ways in which pointers can be used, as illustrated in Figure 6:

```
// Pointers are declared using the unary asterisk ("*")
cout << "Begin simplePointers():" << endl << endl;
int valueX, valueY;
int *pointerA, *pointerB;

// Pointers can be assigned to the memory address of a variable using the
// unary address-of operator ("&") to prefix the target location
valueX = 10;
pointerA = &valueX;

cout << "valueX = " << valueX << endl;
cout << "*pointerA = " << *pointerA << endl;
cout << "pointerA = " << pointerA << endl << endl;
```

Figure 6: Basic Use Of Pointers

As shown in this example, a pointer declaration includes both the dereferencing operator and a type declaration: in this case, `int`. Further, when defining a pointer, the address-of `(&)` operator can be used to indicate the specific memory address that the pointer should point to. In this case, the `int` pointer `pointerA` is assigned the address of the `int`, `valueX`. In order to access the value of the memory address that `pointerA` points to, the dereferencing operator must again be used; otherwise the physical memory address is considered instead of its value. When `valueX` changes, then so too does `*pointerA`, while `pointerA` (indicating the memory address of valueX) remains the same.

Although some programs may not necessarily need them to perform correctly, pointers can be used in a number of interesting ways that other languages can't easily duplicate. For instance, they can be used in conjunction with arrays as shown in the `arrayPointers()` function of the *Pointers.cpp* file. When a pointer points to an array, then meaningful operations can be performed on the pointer that otherwise would have trivial outcomes. For instance, Figure 7 shows a number of these operations as performed on an `int` array with 10 elements:

```
int intArray[10] = {};
int *pointerC = &intArray[0];
pointerC++;
*pointerC = 20;
...
pointerC = &intArray[5];
*pointerC = 30;
...
pointerC[-2] = 50;
```

Figure 7: Pointers On Arrays

The pointer begins at the memory address of the first element in the `intArray`. Since an array is a "homogeneous sequence of objected allocated in continuous memory; that is, all elements of an array have the same type and there are no gaps between objects of the sequence," then we can easily traverse this array using a pointer (Stroustrup, 2014, p. 649). The increment (`++`) and decrement (`--`) operators can be used to move one space forward or backward in the array, such that after the `pointerC++` statement, then `pointerC == &intArray[1]`. Similarly, the pointer can be directly moved to any location in the array by using the address-of operator on any specific element. The pointer can even temporarily point to memory locations relative to its current position by using subscript notation on the pointer directly. As Figure 7 shows, `pointerC[-2] = 50` will access the value in the memory address two places "behind" it and then change the value to 50; however, the

pointer's memory address will remain unchanged after this operation. Care must be taken with these operations, however, since it may inadvertently allow manipulation of elements outside of the array bounds.

5 Input and Output

At first glance, the input and output format of C++ may look strange; but it follows many of the same conventions as other languages. C++ handles these tasks using streams that are primarily found in the `iostream` library, although additional libraries like `ostream, istream, iomanip, fstream`, and others exist for more advanced purposes. The `cout` class is used with the insertion (`<<`) operator to send data to the right of the operator to the output stream. Conversely, `cin` is used with the extraction (`>>`) operator to send data obtained from some input source (typically the keyboard) to be stored as date indicated to the right of the operator. In both cases, these operators can be "chained" together indefinitely, as shown in Figure 8:

```
// Insertion << chaining
int num = 10;
string name = "Michael";
char c = 'J';
cout << num << endl << name << endl << c << endl;

// The >> operator can be chained like the << operator to simplify input
cout << "\nEnter your first name, your favorite number, and your favorite food
         in the format: " << "name number food" << endl;
cin >> name >> num >> food; // store first token in name,
                            // the next in num, the last in food
```

Figure 8: Simple Input and Output Chaining

When using `cout`, the insertion operator can almost be viewed as a type of concatenation operator for strings, simply appending the individual pieces together. When using `cin`, each instance of the extraction operator pulls one token from the given input stream. So, following the example in Figure 8, if the user enters "Michael 8 Pizza," then `cin` will delimit the string by spaces and thus take each word separately, which it stores into separate variables. This can allow for great efficiency when parsing a document with standardized input, which might be more tedious in other languages. C++ also has some built-in error handling for `cin`, as shown in the Figure 9 (note: this example was taken and modified from Stroustrup (2014) p. 355):

```
int num;
// cin will return 1 (true) if the input stream exits successfully,
// or will return 0 (false) if an error is encountered
cout << "Enter a number: ";
cin >> num;
if(!cin) {
        // cin.eof() == true if the stream reached the end of the file
        if(cin.eof()) {
                cout << "End of file reached";
        }
        // cin.bad() == true if it encountered some critical error that it likely cannot recove
        else if(cin.bad()) {
                cerr << "Critical failure, aborting input operation";
        }
        // cin.fail() == true if there was some formatting error that occurred,
        // e.g. trying to store a char into an int variable
        else if(cin.fail()) {
                cout << "Incorrect input format. Try again using a valid number.";
        }
}
```

Figure 9: Input Error Handling

As this example shows, `cin` will actually return a true or false value upon completion, indicating whether the stream exited successfully or not. If it did not exit successfully, then the error can be determined by three other `cin` functions. `cin.eof()` indicates that the end of a file was reached, which would be useful when parsing a collection of tokens to know when input should cease. `cin.bad()` indicates when a critical error has occurred that likely cannot be repaired; or, as Stroustrup (2014) describes, "something really nasty, such as a bad risk read, happens" (p. 355). Finally, `cin.fail()` usually indicates some type of formatting error that can be caught and corrected, such as trying to insert a token of one type into some incompatible type.

6 Scope

C++ exhibits static, block-level scoping that is very similar to the scoping seen in Java. The *Scoping.cpp* file provides a number of examples showing how scoping is affected by different constructs. Importantly, this file contains an example of a namespace, which allows for additional organization by serving as a container for "variables, functions, structures, enumerations, classes, and class and structure members" (Prata, 2012, p. 482). Specifically, these namespaces were designed to help with name clashes, as seen when multiple libraries are included that may possess

constructs bearing the same name. As seen previously, the scope resolution operator (`::`) can be used to help target the specific class, or namespace, the might be needed for a particular purpose. Figure 10 provides a brief snippet of how this operator can be used to isolate `someFunction()` from a particular namespace:

```
int x = SomeLibrary::SomeClass().someFunction();//Target fnc in class in SomeLibrary namespace
std::cout << "x: " << x;                        //x = 5
int x2 = SomeLibrary::someFunction();          //Target fnc in SomeLibrary namespace
std::cout << "\nx2: " << x2;                    //x = 10
int x3 = WithinScopeClass().someFunction();    //Target fnc in class in global namespace
std::cout << "\nx3: " << x3;                    //x = 15
int x4 = someFunction();                       //Target fnc in global namespace
std::cout << "\nx4: " << x4;                    //x = 20
```

Figure 10: Using the Scope Resolution Operator (::)

Prata (2012) describes two terms relating to scope that help explain how it works in C++. First, "a declarative region is a region in which declarations can be made;" and second, "the potential scope for a variable begins at its point of declaration and extends to the end of its declarative region" (p. 483). Creating a new namespace, class, or function introduces a new declarative region in which member names won't conflict within the same namespace. Further, the potential scope is more limiting than it is in Java: for example, if a function is located at the end of a file, it is considered outside of the potential scope for all constructs located lexically above it. For this reason the `main()` function is often located at the end of a file to ensure that no necessary constructs "hide" below it. Finally, there exists a file-level declarative region known as the global namespace, such that any non-nested construct (class, function, variable, etc.) is considered a member of this global namespace and can be accessed without explicit use of the scope resolution operator, as demonstrated also in Figure 10 (p. 484).

7 Parameters and Resource Management

While some other languages use a particular means of parameter passing, like pass-by-value for C and Java, C++ allows the programmer to select between three different techniques: pass-by-value, pass-by-reference, and pass-by-const-reference. Figure 11 shows several function signatures used in the *ParametersEtc.cpp* for illustrating these types of parameter passing:

```
void passByValue(ReallyBigObject reallyBigObject) {...}
void passByReference(ReallyBigObject &reallyBigObject) {...}
void passByConstReference(const ReallyBigObject &reallyBigObject) {...}
```

Figure 11: Function Signatures For Parameter Passing Techniques

The first function, `passByValue()`, simply accepts an object as a parameter, similar to Java or C. In this case, C++ makes a copy of the object for use within the function body. Unlike Java, which makes a copy of a reference to the indicated object, C++ copies the entire object itself by default. This might be efficient for small objects, but can lead to performance issues for larger ones. The second function, `passByReference()`, uses the address-of (`&`) operator to target the memory address of the indicated object. In this case, the function is able to directly access the object in memory to access or manipulate it, without the need for copying it into the function itself. Finally, the `passByConstReference()` is similar to `passByReference()`, except it additionally includes a `const` declaration that prefixes the parameter. This `const` declaration is frequently used in C++, indicating a constant value that cannot be altered once defined. Thus, this function signature prevents the indicated memory address from being altered, potentially preventing code in the function from inadvertently changing it.

The *ParametersEtc.cpp* file further uses a number of timing experiments to show the significant difference in performance between these functions. First, a `ReallyBigObject` is created, which is simply an object with a very large list data structure. The time is then recorded between the object being passed to one of the functions to when control returns back to the `main()` function. Unsurpsingly, the `passByValue()` function takes a significant amount of time to operate, since it must recreate the entire large object for the function's use. The `passByReference()` and `passByConstReference()` functions, however, perform nearly instantaneously, since only the address of the `ReallyBigObject's` memory address is passed to these functions, foregoing any need for copying.

These timing experiments also reveal an interesting fact about the resource management strategy used in C++. In Java, when an object is no longer referenced in memory then it is targeted for garbage collection; but the actual execution of the garbage collection is somewhat arbitrary–it might not happen right away. In C++, however, there is a system known as "Resource Acquisition Is Initialization" (RAII) whereby "when the thread of execution leaves a scope, the destructors for every fully constructed object and sub-object are invoked" (Stroustrup, 2014, p. 701). These destructors allow for the programmer to specify specific actions to take when a particular object

is no longer referenced. Typically, this involves using the `delete` operator, which manually frees a particular memory location. In addition to destructors, C++ allows for special copy constructors, which activate whenever an object of a class is copied (e.g. via pass-by-value). By adding simple print statements within the destructor and copy constructor created for the `ReallyBigObject` class, Figure 12 shows some sample output for the execution of *ParametersEtc.cpp*:

```
Creating really big object...
Elapsed time: 6
... // Other output omitted
Passing object by value and returning by value...
Copying ReallyBigObject
Returning from passAndReturnByValue()...
Copying ReallyBigObject
Destroying ReallyBigObject
Elapsed time: 20
```

Figure 12: Copy Constructor and Destructor Used For Pass-By-Value

In this example a new function named `passAndReturnByValue()` is called, which is identical to the `passByValue()` function, except it additionally returns the same `ReallyBigObject` that was passed as a parameter. The output shows that the copy constructor is activated as soon as the parameter is passed, indicative of pass-by-value, and it again activated when the same object is returned. As Prata (2012) mentions, "returning an object involves the time cost of calling a copy constructor to generate the copy and the time cost of calling the destructor to get rid of the copy" (p. 770). So, in this simple example, the copy constructor is called twice (once when entering the function and once while leaving the function), and the destructor is called once (when destroying the original copy of the `ReallyBigObject` as the scope of `passAndReturnByValue()` is exited). This ultimately results in approximately three times the performance cost of creating a single `ReallyBigObject`. This shows how RAII, for better or worse, activates immediately after the scope of an object is exited; and also how pass-by-value functions must be used with caution for large objects to prevent such extreme performance issues.

8 Multiple Inheritance

C++ further provides utility to programmers by allowing them to use multiple inheritance of classes. In other languages, like Java, only single inheritance is permitted, whereby each class is only allowed to extend one parent class. In C++, however, a single class can extend any number

of classes, inheriting each of their members. In *MultipleInheritance.cpp*, several nearly identical parent classes are created (ParentA, ParentB, and ParentC). Each parent has a function unique to them (`f(), g(), and h()`), as well as a function that shares the same name across each parent (`sharedFunction()`. A Child class is then created using the syntax: `class Child : public ParentA, public ParentB, public ParentC {...}`, which extends each Child with each of the parent classes (note the use of ":" to signify "extends" in C++). Figure 13 then shows the output from creating a Child object and calling each of its inherited functions:

```
Child c = Child();
c.f();  // "This is from ParentA"
c.g();  // "This is from ParentB"
c.h();  // "This is from ParentC"
c.i();  // "This is from Child"
//c.sharedFunction(); //ERROR: sharedFunction() is ambiguous
```

Figure 13: Multiple Inheritance

Note that the unique functions inherited from the parents (`f(), g(), and h()`) are executed as expected, but the `sharedFunction()` fails to execute, giving a compiler error that signifies the function is ambiguous. This kind of error follows naturally from allowing multiple inheritance: when multiple parent classes share identically named members, the child class cannot precisely determine which one to use. However, this can be bypassed by either renaming the parent functions, or overriding the ambiguous function in the child class to specify exactly which parent the operation should be taken from.

9 Conclusion

C++ clearly offers a variety of useful features that any programmer can utilize. Although some of the syntax might seem unusual for programmers coming from other object-oriented languages, there are enough familiar elements to allow comprehension of the core features. Further, C++ gives programmers both high-level and low-level control of their programs, making this language very versatile for those willing to explore its many intricacies.

10 Bibliography

1. Cass, S. (2015). The 2015 Top Ten Programming Languages. IEEE Spectrum. Retrieved from http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages.

2. La, A. (2015). Language Trends on GitHub. GitHub. Retrieved from https://github.com/blog/2047-language-trends-on-github.

3. Prata, S. (2012). C++ Primer Plus (6th ed). Upper Saddle River, NJ: Pearson Education, Inc.

4. Stroustrup, B. (2014). Programming: Principles and Practices Using C++ (2nd ed). Upper Saddle River, NJ: Pearson Education, Inc.