C++ Primer

Elizabeth Dinella

Introduction

C++ was originally created in the 1980s as an extension of the C language to add support for classes. Since its inception, C++ has grown enormously to include a host of features such as exception handling, templates, an expansion of library functions, stronger type checking, inheritance, and other object-oriented concepts. This document will serve as a primer for C++ in this course as well as a source of resources for a more complete documentation. The content is organized into three sections: *Classes & Objects, General Use C++,* and *C++ for LLVM*. *Classes* & Objects will cover classes in C++, access modifiers, and object-oriented concepts such as single and multiple inheritance, and polymorphism. *C++ for LLVM* will serve as an introduction to the LLVM API which will be heavily used in the assignments for this course. Conversely, *General Use C++* will detail features that are not specific to the LLVM API, but will likely be frequently used in the assignments for this course. Lastly, we include a *References* section that includes links to external sources.

Classes & Objects

"Classes are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable." [1]

Readers that are familiar with other languages such as Java are presumably familiar with many of the object-oriented concepts discussed in this section. Nevertheless, there are some notable features that may be new to readers such as multiple inheritance, compilation, destructors, and manual object management.

C++ Class Syntax

Class implementation is usually split between two files: a *header* and an *implementation* file. The header file typically contains the class declaration including list of member variables and member function header. A function header includes the name of the function, parameters and their types, and return type. The function body with the actual instructions to execute when called are written in the implementation file. The header and implementation files typically share

the same name and end in "`.h`" and "`.cpp`" respectively. An example of a Time class split between a header and implementation file is shown in the resources at the end of this document [2].

C++ Scope Resolution Operator

In C++, in order to access a member function, or member variable, or member class, you can use the double colon (::) operator. Examples include ClassName::functionName, ClassName::variable, namespace::ClassName, or EnumName::Member. Within class scope, the member functions and variables are accessible without explicit scope resolution.

Class Compilation

The compilation of a C++ program consists of two stages: *compilation* and *linking*. Compilation includes preprocessing (macro and include expansion), compilation to LLVM IR (and LLVM optimization), object code generation, and finally *linking*. Linking puts together one or many object code files into an executable program. In this stage, the compiler matches function calls with their definitions and ensures that each function that is called has exactly one definition. Linking is a common source of errors for users that are new to C++ classes.

Access Specifiers

Access specifiers identify access rights for the members they are applied to. Access can be either `private`, `public`, or `protected`. By default, a class has `private` access to all of its members. This means that members of the class are only accessible from within members of the same class. Conversely, a `public` member variable is accessible from everywhere that the object is visible. `Protected` member variables are accessible from members of the same class and members of their child classes.

These rules change slightly when the `friend` keyword is introduced. This feature will be detailed in the **Friendship** section.

Constructors and Destructors

Each class has a specific *constructor* function that is called each time a new object of the class is created. The *constructor* has the same name as the class and is typically used to initialize member variables and/or perform some setup. The *default constructor* is a special constructor that is called when an object of the class is declared but is not initialized with any arguments. For example, consider the following declarations:

```
1 Rectangle rectb;   // ok, default constructor called
2 Rectangle rectc(); // oops, default constructor NOT called
```

[1]

The default constructor is called for rectb. Note that rectb is not even constructed with an empty set of parentheses. This is because the empty set of parentheses make of rectc a function declaration instead of an object declaration: it is a function that takes no arguments and returns a value of type Rectangle.

Another special constructor type is the *copy constructor.* This is executed when an object of the same type is passed to the constructor. By default, it performs a shallow copy of all member variables. If a class has a pointer type member variable, a shallow copy may not be enough. Multiple classes may share the same object the pointer refers to which can cause issues if this is not what the developer intended. If a different semantics is preferred, the developer can write their own copy constructor with the header: `MyClass::MyClass(const MyClass& x)`

Likewise, the *destructor* function is called each time an object of the class is destroyed. It has the same name as the class (and the constructor), but is preceded with the tilde sign (~). It is typically used to perform some cleanup of heap allocated memory.

`this` keyword

"The keyword `this` represents a pointer to the object whose member function is being executed. It is used within a class's member function to refer to the object itself." [1]

It is often used to resolve ambiguity between a member variable of the object executing the call and an object of the same class passed as a parameter. For example, consider the member function isitme.

More details on the specifics of the `this` keyword are included in the resources [3].

Static members

A static member of a class is a variable that is shared between all objects of the class. Likewise, a static member function is not associated with any specific object. Therefore, they do not have a `this` pointer.

Friendship

The `friend` keyword defines access relationships between functions and classes. If a class A is a "friend" of class B, member functions in class A can access `private` and `protected` members of class B. Friendship is defined in the class giving access (Class B in this example). An example of friend classes is shown here.

Similarly, if a function is a "friend" of class B, it can access `private` and `protected` members of class B within its function body. An example of a friend function is shown here.

Inheritance

Inheritance creates an "*is-a*" relationship between the *derived* and *base* class. The derived class inherits member variables and functions from the base class. It can also include its own member variables and functions. An example can be found here. The Rectangle class is derived from the base class, Square.

Recall that derived classes can access `protected` members in the base class.

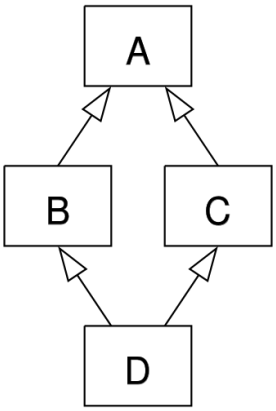
A base class can inherit with access modifiers `public`, `private`, or `protected`. A `public` inheritance (shown in the example) behaves as expected. The variables inherited from the base class keep their access status. Conversely, in `private` inheritance, all inherited members become private. In `protected` inheritance, public members become protected and all other members are unchanged.

Multiple Inheritance and Virtualism

As described in the introduction, C++ was originally created to add support for classes to C. Thus, C++ includes many object oriented concepts that did not exist in C. In this section, we assume that the reader has a basic understanding of classes and inheritance.

Unlike Java, C++ supports multiple inheritance. This means that an object can have one or more parent class. At first glance this seems like a useful feature that should exist in all programming languages that support inheritance. However, multiple inheritance is often a source of confusion and implementation troubles.

Consider the infamous "Diamond Problem":



Both B and C inherit from A and thus contain A's member variables. This could lead to ambiguities and duplicate copies of A's member variables in class D. To solve this, we can use *Virtual Inheritance* as follows:

```
Class B : public virtual A { }
Class C : public virtual A { }
```

More information about virtual inheritance and vtables is detailed in the resources.

Polymorphism

Class inheritance provides many useful features. One of these, is that a pointer to a derived class is type-compatible with a pointer of the base class. More concretely, if class B is derived from base class A, a pointer to class A can also refer to an object of type class B. This is intuitive as class B has an "*is-a*" relationship to class A.

General C++

Templates

C++, like other statically typed languages, requires the developer to specify types in nearly all declarations. However, this can lead to unnecessary repetition of code that is somewhat type independent. Consider the code for the STL vector. Regardless of the element type, the code to add an element, remove an element, calculate size, and even sort are likely identical. Templates provide a way to write programs that are *generic* or independent of any particular type. Below is an example of a templated swap function. So long as the types of n1 and n2 are equivalent, this function will be applied successfully.

```
1. template <typename T>
2. void Swap(T &n1, T &n2)
3. {
4.     T temp;
5.     temp = n1;
6.     n1 = n2;
7.     n2 = temp;
8. }
```

However, C++ templates are notoriously used for much more than just generic programming. For more information on templates, we direct the reader to resources.

C++ Type Checking

In the introduction, we briefly noted that C++ has stronger type checking than C. In this section, we will describe how the stronger type checking manifests in differences in enums and void pointers. In C, an enum is simply an alias to integer types. However, the C++ enum defines an actual type. So,

```
enum direction {
     North, South, West, East
 };
```

```
    direction myDir;

    myDir = 3;
```

will result in a type error in C++, but not in C.

Another source of type unsafety in C is the implicit promotion of void pointers. Although malloc returns a `void *`, the following program compiled with gcc produces no errors or warnings.

```
#include <stdlib.h>

int main() {
      int* arr;
      arr = malloc(5 * sizeof(int));
}
```

However, compiling with g++ produces the following error:

```
void_ptr.cpp:5:8: error: assigning to 'int *' from incompatible type 'void *'
        arr = malloc(5 * sizeof(int));
              ^~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

In C++, casting the `void *` to `int *` is required.

Command Line Arguments

In this course, we will often pass additional information from the user when the program is run. The syntax to implement this in C++ is exactly the same as in C. However, it is worth reviewing as it will be used quite frequently in the assignments.

```
int main(int argc, char* argv[]) {
    if (std::string(argv[1]) == std::string("blue")) {
        std::cout << "my favorite color is blue too!" << std::endl;
    }
}
```

New and Delete keywords

C++ includes dynamic memory features that did not exist in C. The traditional `malloc` and `free` keywords are still available in C++; however, it is widely accepted that `new` and `delete` should be used in their place.

`new` and `delete` have slightly different syntax for scalar types and arrays.

```
int *foo = new int;         // allocates memory for one int
...
delete foo;                 // deletes memory pointed to by foo


int *baz = new int[100];    // allocates memory for 100 int array
...
delete [] baz;              // deletes memory pointed to by baz
```

`delete` with `[]` indicates that the entire array should be deleted rather than a single element.

C++ Standard Library

The standard libraries in C++ are a superset of the standard C libraries. In addition, the C++ libraries include a diverse and powerful set of features. In this document, we will detail, arguably, the most important features: iterators, strings, I/O streams, and data structures.

1. **Iterators -** Iterators provide an interface to traverse containers (vectors, sets, maps, etc.) and view / edit specific elements stored in such containers. In many ways, iterators are a generalization of pointers. C++ separates the traversing of a container from the container itself.

```
// Accessing the elements of a vector without using iterators
for (j = 0; j < 4; ++j) {
    cout << v[j] << " ";
}

// Accessing the elements of a vector using iterators
for (i = v.begin(); i != v.end(); ++i) {
    cout << *i << " ";
}
```

2. **Strings -** This powerful library simplifies the tedious character array manipulation tasks left to the developer in C. C++ strings are a wrapper around the array of characters that would be explicitly manipulated in C. Developers can:

    a. Create and delete strings using constructor and deconstructors
    b. Assign values to strings using the = operator
    c. Compare contents of strings using the == operator
    d. Iterate over strings using C++ iterators
    e. Access particular characters using the `[]` operator
    f. Erase a substring from a string

The many more string features in the C++ standard library are left to the resources at the bottom of the document.

3. **I/O Streams -** The IOStream library is intended to replace the stdio library for an easier and more flexible input and output developer experience. A `stream` has some source or sink. The IOStream library supports standard input, standard output, standard error, a file, or an array of characters. The >> and << operators are conveniently overloaded to simplify the use of IOStreams. The following example shows the basics of writing to the console and a file:

```
#include <iostream>
#include <fstream>

int main() {
    /*
    Writing to the console:
    cout is defined to access to screen by default
    endl (rather than std::end) appends a newline (\n)
    */

    std::cout <<  "Hello World!" << std::endl;

    //writing to a file

    std::ofstream myfile;
    myfile.open("out.txt");

    myfile << "Hello File!" << std::endl;

    myfile.close();
}
```

For more details on reading from a file / stdin, we refer the reader to the resources. However, according to the LLVM style guide, IOStreams are forbidden [6]. Instead, developers are encouraged to use LLVM's `raw_ostream`. For more details, we refer the reader to the *C++ for LLVM* section.

4. **Data Structures** - The C++ Standard Library (STL) defines many useful containers that will feel familiar to readers with experience in Java or Python. The STL includes vectors, lists, queues, stacks, sets, and maps. Each of these containers can be traversed using iterators. Additionally, many containers share common functions. This manifests in a seamless developer experience regardless of the container or element type. For example, `size()`, `empty()`, `max_size()`, `==`, `!=`, `swap`, a default constructor, copy constructor, and assignment constructor are shared by all STL containers. A detailed list of STL containers and properties is provided in the resources.

C++ for LLVM

LLVM data structures

C++ STL data structures are for general purpose use and have performance that is platform dependent.

In order to create very fast specialized, but platform independent structures, LLVM created its own data structures. To the user, these structures feel similar to STL data structures, but have some notable differences. For each of the following, the developer can choose to use the STL data structure or the LLVM specialized structure. This choice should be made with efficiency and their use cases in mind.

In general, STL structures are variable size. To conserve memory, on creation, the underlying array does not have any space allocated for elements. Once the user calls `push_back` or `insert`, memory is allocated for the element. Each time the user inserts an element to a container that is at capacity, the memory allocated for the structure is doubled. As the number of elements increases, the frequency of allocation calls decreases. So, for a large n, the STL version is often a good choice of data structure. For a small number of elements, LLVM provides alternatives.

- **LLVM SmallVector** is optimized for a small n. It is created with some number of elements in place. In this way, it avoids allocation when the actual number of elements is below that threshold. Inserts below that threshold will be must faster to a SmallVector than an STL vector.
- **LLVM DenseMap** is an unsorted alternative to the STL map. The STL map guarantees that iteration order of the container is the same as the insertion order. The LLVM DenseMap keeps keys and values next to each other in memory in order to speed up lookup.
- **LLVM StringMap** is a specialized structure with only strings as keys. This is useful as string keys are difficult to support efficiently. Long strings are inefficient to compare and copy. The LLVM StringMap supports an arbitrarily long key value. Like the LLVM DenseMap, the iteration order is not guaranteed.
- **LLVM SmallSet** is an unsorted alternative to the STL set. Because STL sets have a defined order, searches are O(log n). Conversely, in the LLVM SmallSet searches are O(n). Additionally, as the LLVM SmallSet does not have a defined order, it cannot be iterated over. For a small n that does not need to be regularly queried, LLVM SmallSet could be an efficient choice.

raw_ostream

The `raw_ostream` is an LLVM alternative to `iostream`. Instead of using `std::cout` the developer should pipe output to `outs()`. Analogously, the alternative to `std::cerr` is `errs()`.

```
int main() {
    outs() <<  "Hello World!\n"
}
```

Dynamic Casting

A dynamic cast converts pointers to a more specific type in its class hierarchy at runtime.

Consider the "*is-a*" relationship between an LLVM `AllocaInst` and an LLVM `Instruction`. [4] In the assignments, we will often write control flow similar to the following structure:

```
if (auto *AI = dyn_cast<AllocationInst>(Val)) {
  // ...
}
```

[5]

This allows us to handle specific instructions (`BinaryOperator`, `CallInst`, `AllocaInst`, etc.) differently. The **`dyn_cast`** operator checks if the parameter is an instance of the templated type. If so, it returns a pointer to the downcasted class. If not, `dyn_cast` returns a null pointer.

Resources

[1] C++ tutorial - Classes
[2] C++ Separate Header and Implementation Files Example
[3] Should you use the this pointer in the constructor?, C++ FAQ
[4] LLVM AllocaInst
[5] LLVM Programmers Manual
[6] LLVM Coding Standards
[7] Bruce Eckel's Free Electronic Books Thinking in C++ book
[8] C++ Language - C++ Tutorials
[9] string - C++ Reference
[10] <iterator> - C++ Reference
[11] C++ Templates: The Complete Guide
[12] C++ Exception Handling
[13] Differences Between C and C++
[14] Basic Input/Output - C++ Tutorials

[15] Input/output with files - C++ Tutorials
[16] The C++ Standard Template Library (STL)
[17] STL Containers
[18] Understanding Virtual Tables in C++