A Quick Introduction to C++

Tom Anderson

"If programming in Pascal is like being put in a straightjacket, then programming in C is like playing with knives, and programming in C++ is like juggling chainsaws."

Anonymous.

1 Introduction

This note introduces some simple C++ concepts and outlines a subset of C++ that is easier to learn and use than the full language. Although we originally wrote this note for explaining the C++ used in the Nachos project, I believe it is useful to anyone learning C++. I assume that you are already somewhat familiar with C concepts like procedures, for loops, and pointers; these are pretty easy to pick up from reading Kernighan and Ritchie's "The C Programming Language."

I should admit up front that I am quite opinionated about C++, if that isn't obvious already. I know several C++ purists (an oxymoron perhaps?) who violently disagree with some of the prescriptions contained here; most of the objections are of the form, "How could you have possibly left out feature X?" However, I've found from teaching C++ to nearly 1000 undergrads over the past several years that the subset of C++ described here is pretty easy to learn, taking only a day or so for most students to get started.

The basic premise of this note is that while object-oriented programming is a useful way to simplify programs, C++ is a wildly over-complicated language, with a host of features that only very, very rarely find a legitimate use. It's not too far off the mark to say that C++ includes every programming language feature ever imagined, and more. The natural tendency when faced with a new language feature is to try to use it, but in C++ this approach leads to disaster.

Thus, we need to carefully distinguish between (i) those concepts that are fundamental (e.g., classes, member functions, constructors) – ones that everyone should know and use, (ii) those that are sometimes but rarely useful (e.g., single inheritance, templates) – ones that beginner programmers should be able to recognize (in case they run across them) but avoid using in their own programs, at least for a while, and (iii) those that are just a bad idea and should be avoided like the plague (e.g., multiple inheritance, exceptions, overloading, references, etc).

Of course, all the items in this last category have their proponents, and I will admit that, like the hated goto, it is possible to construct cases when the program would be simpler

This article is based on an earlier version written by Wayne Christopher.

using a goto or multiple inheritance. However, it is my belief that most programmers will never encounter such cases, and even if you do, you will be much more likely to misuse the feature than properly apply it. For example, I seriously doubt an undergraduate would need any of the features listed under (iii) for any course project (at least at Berkeley this is true). And if you find yourself wanting to use a feature like multiple inheritance, then, my advice is to fully implement your program both with and without the feature, and choose whichever is simpler. Sure, this takes more effort, but pretty soon you'll know from experience when a feature is useful and when it isn't, and you'll be able to skip the dual implementation.

A really good way to learn a language is to read clear programs in that language. I have tried to make the Nachos code as readable as possible; it is written in the subset of C++ described in this note. It is a good idea to look over the first assignment as you read this introduction. Of course, your TA's will answer any questions you may have.

You should not need a book on C++ to do the Nachos assignments, but if you are curious, there is a large selection of C++ books at Cody's and other technical bookstores. (My wife quips that C++ was invented to make researchers at Bell Labs rich from writing "How to Program in C++" books.) Most new software development these days is being done in C++, so it is a pretty good bet you'll run across it in the future. I use Stroustrup's "The C++ Programming Language" as a reference manual, although other books may be more readable. I would also recommend Scott Meyer's "Effective C++" for people just beginning to learn the language, and Coplien's "Advanced C++" once you've been programming in C++ for a couple years and are familiar with the language basics. Also, C++ is continually evolving, so be careful to buy books that describe the latest version (currently 3.0, I think!).

2 C in C++

To a large extent, C++ is a superset of C, and most carefully written ANSI C will compile as C++. There are a few major caveats though:

1. All functions must be declared before they are used, rather than defaulting to type `int`.
2. All function declarations and definition headers must use new-style declarations, e.g.,

   ```
   extern int foo(int a, char* b);
   ```

   The form `extern int foo();` means that `foo` takes *no* arguments, rather than arguments of an unspecified type and number. In fact, some advise using a C++ compiler even on normal C code, because it will catch errors like misused functions that a normal C compiler will let slide.
3. If you need to link C object files together with C++, when you declare the C functions for the C++ files, they must be done like this:

```
extern "C" int foo(int a, char* b);
```

Otherwise the C++ compiler will alter the name in a strange manner.

4. There are a number of new keywords, which you may not use as identifiers — some common ones are `new`, `delete`, `const`, and `class`.

3 Basic Concepts

Before giving examples of C++ features, I will first go over some of the basic concepts of object-oriented languages. If this discussion at first seems a bit obscure, it will become clearer when we get to some examples.

1. **Classes and objects**. A class is similar to a C *structure*, except that the definition of the data structure, *and* all of the functions that operate on the data structure are grouped together in one place. An *object* is an instance of a class (an instance of the data structure); objects share the same functions with other objects of the same class, but each object (each instance) has its own copy of the data structure. A class thus defines two aspects of the objects: the *data* they contain, and the *behavior* they have.
2. **Member functions**. These are functions which are considered part of the object and are declared in the class definition. They are often referred to as *methods* of the class. In addition to member functions, a class's behavior is also defined by:
   (a) What to do when you create a new object (the **constructor** for that object) – in other words, initialize the object's data.
   (b) What to do when you delete an object (the **destructor** for that object).
3. **Private vs. public members**. A public member of a class is one that can be read or written by anybody, in the case of a data member, or called by anybody, in the case of a member function. A private member can only be read, written, or called by a member function of that class.

Classes are used for two main reasons: (1) it makes it much easier to organize your programs if you can group together data with the functions that manipulate that data, and (2) the use of private members makes it possible to do *information hiding*, so that you can be more confident about the way information flows in your programs.

3.1 Classes

C++ classes are similar to C structures in many ways. In fact, a C++ struct is really a class that has only public data members. In the following explanation of how classes work, we will use a stack class as an example.

1. **Member functions.** Here is a (partial) example of a class with a member function and some data members:

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    int top;          // Index of the top of the stack.
    int stack[10];    // The elements of the stack.
};

void
Stack::Push(int value) {
    ASSERT(top < 10); // stack should never overflow
    stack[top++] = value;
}
```

   This class has two data members, `top` and `stack`, and one member function, `Push`. The notation *class*::*function* denotes the *function* member of the class *class*. (In the style we use, most function names are capitalized.) The function is defined beneath it.

   As an aside, note that we use a call to `ASSERT` to check that the stack hasn't overflowed; ASSERT drops into the debugger if the condition is false. It is an extremely good idea for you to use ASSERT statements liberally throughout your code to document assumptions made by your implementation. Better to catch errors automatically via ASSERTs than to let them go by and have your program overwrite random locations.

   In actual usage, the definition of `class Stack` would typically go in the file `stack.h` and the definitions of the member functions, like `Stack::Push`, would go in the file `stack.cc`.

   If we have a pointer to a `Stack` object called `s`, we can access the `top` element as `s->top`, just as in C. However, in C++ we can also call the member function using the following syntax:

```
    s->Push(17);
```

   Of course, as in C, `s` must point to a valid `Stack` object.

   Inside a member function, one may refer to the members of the class by their names alone. In other words, the class definition creates a scope that includes the member (function and data) definitions.

   Note that if you are inside a member function, you can get a pointer to the object you were called on by using the variable `this`. If you want to call another member function on the same object, you do not need to use the `this` pointer, however. Let's extend the Stack example to illustrate this by adding a `Full()` function.

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();       // Returns TRUE if the stack is full, FALSE otherwise.
    int top;          // Index of the lowest unused position.
    int stack[10];    // A pointer to an array that holds the contents.
};
```