A Quick Introduction to C++

Wayne A. Christopher

CS 162, Spring 1992

> "There are two ways of constructing a software design: one way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies."
>
> C. A. R. Hoare, "The Emperor's Old Clothes", CACM Feb. 1981

The CS 162 project is written in a subset of C++, and you will be expected to understand and modify this code. Since you have already used C in previous classes, we hope that it will be easy for you to make the change to C++.

The best way to learn a language is to read clear programs in that language. We have tried to make the code for the first assignment as readable as possible, so you should look over it as you read this introduction. Of course, your TA's will answer any questions you may have.

You should not need to buy a book on C++ for this class, but if you are curious or expect to use the language more in the future, there is a large selection at Cody's and other bookstores. Be sure to get one that describes Version 2.0 of the language – our compiler (GNU C++) is not up to this level yet, but it's close. The one most likely to be technically complete is by the author of C++, Bjarne Stroustrup. The **Annotated Reference Manual** by Stroustrup and Ellis is a bit out of date but is very good if you want to know the motivation behind the design of the language. Other books may be more readable — if you have any opinions, please tell us.

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.

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

   ```
   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`.

1 Basic Concepts

Before giving examples of C++ features, we will first go over some of the basic concepts of object-oriented languages. If this discussion seems a bit obscure, it will become clearer once we have some concrete examples to talk about.

1. **Classes and objects**. A class defines a set of objects, or *instances* of that class. One declares a class in a way similar to a C *structure*, and then creates objects of that class. A class 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).
   (b) What to do when you delete an object (the **destructor** for that object).
3. **Private and 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.

2 Classes

C++ classes 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)
{
    if (top == 10) {
        fprintf(stderr, "Error: Stack overflow\n");
        exit(1);
    }

    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.

   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 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.
    int Full();       // Returns non-0 if the stack is full, 0 otherwise.
    int top;          // Index of the lowest unused position.
    int stack[10];    // A pointer to an array that holds the contents.
};

int
Stack::Full()
{
    return (top == 10);
}
```

Now we can rewrite `Push` this way:

```
void
Stack::Push(int value)
{
    if (Full()) {
        fprintf(stderr, "Error: Stack overflow\n");
        exit(1);
    }

    stack[top++] = value;
}
```

We could have written the line with `Full()` this way also:

```
    if (this->Full()) {
```

but in a member function, the `this->` is implicit.

The purpose of member functions is to encapsulate the functionality of a type of object along with the data that the object contains. A member function does not take up space in an object of the class.

2. **Private members.** One can declare some members of a class to be *private*, which are hidden to all but the member functions of that class, and some to be *public*, which are visible and accessible to everybody. Both data and function members can be either public or private.

   In our stack example, note that once we have the `Full()` function, we really don't need to look at the `top` or `stack` members outside of the class. Thus we can rewrite the class as follows:

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    int Full();       // Returns non-0 if the stack is full, 0 otherwise.
  private:
    int top;          // Index of the top of the stack.
    int stack[10];    // The elements of the stack.
};
```

   Before, given a pointer to a `Stack` object, say `s`, any part of the program could access `s->top`, in potentially bad ways. Now, since the `top` member is private, only a member function, such as `Full()`, can access it. If any other part of the program attempts to use `s->top` the compiler will report an error.

   You can have alternating `public:` and `private:` sections in a class. Before you specify either of these, class members are private, thus the above example could have been written:

```
class Stack {
    int top;          // Index of the top of the stack.
    int stack[10];    // The elements of the stack.
  public:
    void Push(int value); // Push an integer, checking for overflow.
    int Full();       // Returns non-0 if the stack is full, 0 otherwise.
};
```

Which form you prefer is a matter of style.

In many cases, it is best to make all data members of a class private and define *accessor functions* to read and write them. This adds to the modularity of the system, since you can redefine how the data members are stored without changing how you access them.

3. **Constructors and the operator new.** In C, in order to create a new object of type `Stack`, one might write:

```
struct Stack* s = (struct Stack *) malloc(sizeof (struct Stack));
InitStack(s, 17);
```

The `InitStack()` function might take the second argument as the size of the stack to create, and use `malloc()` again to get an array of 17 integers.

The way this is done in C++ is as follows:

```
Stack* s = new Stack(17);
```

The `new` function takes the place of `malloc()`. To specify how the object should be initialized, one declares a *constructor* function as a member of the class, with the name of the function being the same as the class name:

```
class Stack {
  public:
    Stack(int sz);    // Constructor:  initialize variables, allocate space.
    void Push(int value); // Push an integer, checking for overflow.
    int Full();       // Returns non-0 if the stack is full, 0 otherwise.
  private:
    int size;         // The maximum capacity of the stack.
    int top;          // Index of the lowest unused position.
    int* stack;       // A pointer to an array that holds the contents.
};

Stack::Stack(int sz)
{
    size = sz;
    top = 0;
    stack = new int[size];   // Let's get an array of integers.
}
```

There are a few things going on here, so we will describe them one at a time.

The `new` operator automatically creates (i.e. allocates) the object and then calls the constructor function for the new object. This same sequence happens even if, for instance, you declare an object as an automatic variable inside a function or block. (Although we explain this construct here, in Nachos we avoid automatic allocation of objects, to make it explicitly clear when the constructor is being called.) In this example, we create two stacks of different sizes, one by declaring it as an automatic variable, and one by using `new`.

```
void
test()
{
    Stack stack1(17);
    Stack* stack2 = new Stack(23);
}
```

Note there are two ways of providing arguments to constructors: with `new`, you put the argument list after the class name, and with variables declared as above, you put them after the variable name.

The `stack1` object is deallocated when the `test` function returns. The object pointed to by `stack2` remains, however, until disposed of using `delete`, described below. In this example it is inaccessible outside of `test`, and in a language like Lisp it would be garbage collected, but we could have returned `stack2` as a value, for instance.

Note how the `new` operator is used to allocate arrays, in this case an array of 10 `int`s:

```
int* nums = new int[10];
```

Note that you can use `new` and `delete` (described below) with built-in types like `int` and `char` as well as with class objects.

4. **Destructors and the operator delete.** Just as `new` is the replacement for `malloc()`, the replacement for `free()` is `delete`. To get rid of the `Stack` object pointed to by `s`, one can do:

```
delete s;
```

This will deallocate the object, but first it will call the *destructor* for the `Stack` class, if there is one. This destructor would be a member function of `Stack` called ~Stack():

```
class Stack {
  public:
    Stack(int sz);    // Constructor:  initialize variables, allocate space.
    ~Stack();         // Destructor:   deallocate space allocated above.
    void Push(int value); // Push an integer, checking for overflow.
    int Full();       // Returns non-0 if the stack is full, 0 otherwise.
  private:
    int size;         // The maximum capacity of the stack.
    int top;          // Index of the lowest unused position.
    int* stack;       // A pointer to an array that holds the contents.
};

Stack::~Stack()
{
    delete stack;
}
```

The destructor has the job of deallocating the data the constructor allocated. Most classes won't need destructors, and some will use them to close files and otherwise clean up after themselves.

As with constructors, a destructor for an auto object will be called when that object goes out of scope (i.e., at the end of the function in which it is defined), so in the `test()` example above, the stack would be properly deallocated without your having to do anything. **However,** if you have a pointer to an object you got from `new`, and the pointer goes out of scope, then the object won't be freed — you have to use `delete` explicitly. Since C++ doesn't have garbage collection, you should generally be careful to delete what you allocate.

Many classes will not need destructors, and if you don't care about core leaks (i.e., space not getting freed when it is no longer used) when one object allocates other objects and keeps pointers to them, you will need them even less frequently.

In some versions of C++, when you deallocate an array you have to tell `delete` how many elements there are in the array, like this:

```
delete [size] stack;
```

However, GNU C++ doesn't require this.

3 Other C++ Features

Here are a few minor C++ features that are useful to know.

1. When you define a `class Stack`, the name `Stack` becomes usable as a type name as if created with `typedef`. The same is true for `enums`.
2. You can define functions inside of a `class` definition, whereupon they become *inline functions*, which are expanded in the body of the function where they are used. This is usually a matter of convenience, but it should not be overused. A few examples of this appear in the second assignment.
3. Inside a function body, you can declare some variables, execute some statements, and then declare more variables. This can make code a lot more readable. In fact, you can even write things like:

   ```
   for (int i = 0; i < 10; i++) ;
   ```

   The variable `i` is still visible after the end of the `for` loop, however, which is not what one might expect or desire.
4. Comments can begin with the characters `//` and extend to the end of the line. These are usually more handy than the `/* */` style of comments.

4 Style Guidelines

It is as easy to write unreadable and undebuggable code in C++ as it is in C, and perhaps easier, given the more powerful features the language provides. For this project, we suggest you adhere to the following guidelines (and tell us if you catch us breaking them):

1. Words in a name are separated SmallTalk-style (i.e., capital letters at the start of each new word). All class names and member function names begin with a capital letter, except for member functions of the form `getSomething()` and `setSomething()`, where `Something` is a data element of the class (i.e., accessor functions). Note that you would want to provide such functions only when the data should be visible to the outside world, but you want to force all accesses to go through one function. This is often a good idea, since you might at some later time decide to compute the data instead of storing it, for example.
2. All global functions (except for `main` and library functions) should also be capitalized.
3. Minimize the use of global variables. If you find yourself using a lot of them, try and group some together in a class in a natural way or pass them as arguments to the functions that need them if you can.
4. Minimize the use of global functions (as opposed to member functions). If you write a function that operates on some object, consider making it a member function of that object.
5. For every class or set of related classes, create a separate `.h` file and `.cc` file. The `.h` file acts as the *interface* to the class, and the `.cc` file acts as the *implementation* (a given `.cc` file should `include` it's respective `.h` file). If, for a particular `.h` file, you require another `.h` file to be included (e. g., `synch.h` needs `thread.h`) you should include it in the `.h` file, so that the user of your class doesn't have to track down all the dependencies himself. To protect against multiple inclusion, bracket each `.h` file with something like:

```
#ifndef STACK_H
#define STACK_H

class Stack { ... };

#endif
```

Sometimes this will not be enough, and you will have a circular dependency. In this case, you will have to do something ad-hoc: if you only use a pointer to class `Stack` and do not access any elements of the class, you can write, in lieu of the actual class definition:

```
class Stack;
```

This will tell the compiler all it needs to know to deal with the pointer. In a few cases this won't work, and you will have to move stuff around or alter your definitions.

5 Features Not Used

At this point, we do not expect to use the following features of the language in any code that we give you. If you use them, *caveat hacker*.

1. **Inheritance.** This would probably be useful in some places, but if one is not comfortable with classes then derived classes (not to mention multiple inheritance) can be rather confusing.

2. **Operator overloading.** C++ lets you redefine the meanings of the operators (such as `+` and `>>`) for class objects. This is dangerous at best, and when used in non-intuitive ways, a source of great confusion.

3. **Function overloading.** You can define different functions with the same name but different argument types. This is also dangerous, and we use it only for constructors. We will also avoid using default arguments.

4. **References.** Reference variables are rather hard to understand in general. Their most common use is to declare some parameters to a function as *reference parameters*, like those in FORTRAN. This can be useful but is also a source of obscure bugs, and the semantics of references are in general not obvious.

6 Compiling and Debugging

The Makefiles we will give you works only with the GNU version of make, which should be called "gmake" on `po`. You may want to put "alias make gmake" in your .cshrc file.

You should use **gdb** to debug your program rather than **dbx**. Dbx doesn't know how to decipher C++ names, so you will see function names like `Run__9SchedulerP6Thread`.

On the other hand, in GDB (but not DBX) when you do a stack backtrace when in a forked thread (in homework 1), after printing out the correct frames at the top of the stack, the debugger will go into a loop printing the lower-most frame (`ThreadRoot`), and you have to type control-C when it says "more?". If you understand MIPS assembly language and can fix this, please tell us.

7 Example: A Stack of Integers

The complete, working code for the stack can be found in the directory `~c162/stack-example` on the machine `po`. You should read through it and play around with it to make sure you understand the features of C++ described in this paper.

To compile the stack test, do `g++ stack.cc test.cc`, which gives you an `a.out` file.