# 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 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

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

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.
};

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

Now we can rewrite `Push` this way:

```
void
Stack::Push(int value) {
    ASSERT(!Full());
    stack[top++] = value;
}
```

We could have also written the ASSERT:

```
    ASSERT(!(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 – in fact, we'd rather that users of the Stack abstraction *not* know about its internal implementation, in case we change it. Thus we can rewrite the class as follows:

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();       // Returns TRUE if the stack is full, FALSE 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.
    bool Full();        // Returns TRUE if the stack is full, FALSE otherwise.
};
```

Which form you prefer is a matter of style, but it's usually best to be explicit, so that it is obvious what is intended. In Nachos, we make everything explicit.

What is not a matter of style: **all data members of a class should be private.** All operations on data should be via that class' member functions. Keeping data private 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.
    bool Full();        // Returns TRUE if the stack is full, FALSE 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 – the compiler allocates space for the object on the stack, and calls the constructor function on it.

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 s1(17);
    Stack* s2 = 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 automatic or global variables, you put them after the variable name.

It is crucial that you **always** define a constructor for every class you define, and that the constructor initialize **every** data member of the class. If you don't define your own constructor, the compiler will automatically define one for you, and believe me, it won't do what you want ("the unhelpful compiler"). The data members will be initialized to random, unrepeatable values, and while your program may work anyway, it might not the next time you recompile (or vice versa!).

As with normal C variables, variables declared inside a function are deallocated automatically when the function returns; for example, the `s1` object is deallocated when `test` returns. Data allocated with `new` (such as `s2`) is stored on the heap, however, and remains after the function returns; heap data must be explicitly disposed of using `delete`, described below.

The `new` operator can also be used to allocate arrays, illustrated above in allocating an array of `int`s, of dimension `size`:

```
    stack = new int[size];
```

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

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 we allocated above with `new`, one can do:

```
    delete s2;
```

This will deallocate the object, but first it will call the *destructor* for the `Stack` class, if there is one. This destructor is 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.
    bool Full();      // Returns TRUE if the stack is full, FALSE 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;  // delete an array of integers
}
```

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

The destructor for an object is called when the object is deallocated. If the object was created with `new`, then you must call `delete` on the object, or else the object will continue to occupy space until the program is over – this is called "a memory leak." Memory leaks are bad things – although virtual memory is supposed to be unlimited, you can in fact run out of it – and so you should be careful to **always** delete what you allocate. Of course, it is even worse to call `delete` too early – `delete` calls the destructor and puts the space back on the heap for later re-use. If you are still using the object, you will get random and non-repeatable results that will be very difficult to debug. In my experience, using data that has already been deleted is major source of hard-to-locate bugs in student (and professional) programs, so hey, be careful out there!

If the object is an automatic, allocated on the execution stack of a function, the destructor will be called and the space deallocated when the function returns; in the `test()` example above, `s1` will be deallocated when `test()` returns, without you having to do anything.

In Nachos, we always explicitly allocate and deallocate objects with `new` and `delete`, to make it clear when the constructor and destructor is being called. For example, if an object contains another object as a member variable, we use `new` to explicitly allocated and initialize the member variable, instead of implicitly allocating it as part of the containing object. C++ has strange, non-intuitive rules for the order in which the constructors and destructors are called when you implicitly allocate and deallocate objects. In practice, although simpler, explicit allocation is slightly slower and it makes it more likely that you will forget to deallocate an object (a bad thing!), and so some would disagree with this approach.

When you deallocate an array, you have to tell the compiler that you are deallocating an array, as opposed to a single element in the array. Hence to delete the array of integers in `Stack::~Stack`:

```
    delete [] stack;
```

## 3.2 Other Basic C++ Features

Here are a few other 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 `enum`s.

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. The rule of thumb to follow is to only consider inlining one-line functions, and even then do so rarely.

   As an example, we could make the `Full` routine an inline.

   ```
   class Stack {
      ...
      bool Full() { return (top == size); };
      ...
   };
   ```

   There are two motivations for inlines: convenience and performance. If overused, inlines can make your code more confusing, because the implementation for an object is no longer in one place, but spread between the `.h` and `.c` files. Inlines can sometimes speed up your code (by avoiding the overhead of a procedure call), but that shouldn't be your principal concern as a student (rather, at least to begin with, you should be most concerned with writing code that is simple and bug free). Not to mention that inlining sometimes slows down a program, since the object code for the function is duplicated wherever the function is called, potentially hurting cache performance.

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++) ;
   ```

   Depending on your compiler, however, the variable `i` may 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.

5. C++ provides some new opportunities to use the `const` keyword from ANSI C. The basic idea of `const` is to provide extra information to the compiler about how a variable or function is used, to allow it to flag an error if it is being used improperly. You should always look for ways to get the compiler to catch bugs for you. After all, which takes less time? Fixing a compiler-flagged error, or chasing down the same bug using gdb?

   For example, you can declare that a member function only reads the member data, and never modifies the object:

   ```
   class Stack {
      ...
      bool Full() const;  // Full() never modifies member data
      ...
   };
   ```

As in C, you can use `const` to declare that a variable is never modified:

```
const int InitialHashTableSize = 8;
```

This is *much* better than using `#define` for constants, since the above is type-checked.

6. Input/output in C++ can be done with the >> and << operators and the objects `cin` and `cout`. For example, to write to `stdout`:

```
cout << "Hello world!  This is section " << 3 << "!";
```

This is equivalent to the normal C code

```
fprintf(stdout, "Hello world!  This is section %d!\n", 3);
```

except that the C++ version is type-safe; with `printf`, the compiler won't complain if you try to print a floating point number as an integer. In fact, you can use traditional `printf` in a C++ program, but you will get bizarre behavior if you try to use both `printf` and << on the same stream. Reading from `stdin` works the same way as writing to `stdout`, except using the shift right operator instead of shift left. In order to read two integers from `stdin`:

```
int field1, field2;
cin >> field1 >> field2;
    // equivalent to fscanf(stdin, "%d %d", &field1, &field2);
    // note that field1 and field2 are implicitly modified
```

In fact, `cin` and `cout` are implemented as normal C++ objects, using operator overloading and reference parameters, but (fortunately!) you don't need to understand either of those to be able to do I/O in C++.

# 4 Advanced Concepts in C++: Dangerous but Occasionally Useful

There are a few C++ features, namely (single) inheritance and templates, which are easily abused, but can dramatically simplify an implementation if used properly. I describe the basic idea behind these "dangerous but useful" features here, in case you run across them. Feel free to skip this section – it's long, complex, and you can understand 99% of the code in Nachos without reading this section.

Up to this point, there really hasn't been any fundamental difference between programming in C and in C++. In fact, most experienced C programmers organize their functions into modules that relate to a single data structure (a "class"), and often even use a naming convention which mimics C++, for example, naming routines `StackFull()` and `StackPush()`. However, the features I'm about to describe *do* require a paradigm shift – there is no simple translation from them into a normal C program. The benefit will be that, in some circumstances, you will be able to write generic code that works with multiple kinds of objects.

Nevertheless, I would advise a beginning C++ programmer against trying to use these features, because you will almost certainly misuse them. It's possible (even easy!) to write completely inscrutable code using inheritance and/or templates. Although you might find it amusing to write code that is impossible for your graders to understand, I assure you they won't find it amusing at all, and will return the favor when they assign grades. In industry, a high premium is placed on keeping code simple and readable. It's easy to write new code, but the real cost comes when you try to keep it working, even as you add new features to it.

Nachos contains a few examples of the correct use of inheritance and templates, but realize that Nachos does *not* use them everywhere. In fact, if you get confused by this section, don't worry, you don't need to use any of these features in order to do the Nachos assignments. I omit a whole bunch of details; if you find yourself making widespread use of inheritance or templates, you should consult a C++ reference manual for the real scoop. This is meant to be just enough to get you started, and to help you identify when it would be appropriate to use these features and thus learn more about them!

## 4.1 Inheritance

Inheritance captures the idea that certain classes of objects are related to each other in useful ways. For example, lists and sorted lists have quite similar behavior – they both allow the user to insert, delete, and find elements that are on the list. There are two benefits to using inheritance:

1. You can write generic code that doesn't care exactly which kind of object it is manipulating. For example, inheritance is widely used in windowing systems. Everything on the screen (windows, scroll bars, titles, icons) is its own object, but they all share a set of member functions in common, such as a routine `Repaint` to redraw the object onto the screen. This way, the code to repaint the entire screen can simply call the `Repaint` function on every object on the screen. The code that calls `Repaint` doesn't need to know which kinds of objects are on the screen, as long as each implements `Repaint`.

2. You can share pieces of an implementation between two objects. For example, if you were to implement both lists and sorted lists in C, you'd probably find yourself repeating code in both places – in fact, you might be really tempted to only implement sorted lists, so that you only had to debug one version. Inheritance provides a way to re-use code between nearly similar classes. For example, given an implementation of a list class, in C++ you can implement sorted lists by replacing the insert member function – the other functions, delete, isFull, print, all remain the same.

### 4.1.1 Shared Behavior

Let me use our Stack example to illustrate the first of these. Our Stack implementation above could have been implemented with linked lists, instead of an array. Any code using a Stack shouldn't care which implementation is being used, except that the linked list implementation can't overflow. (In fact, we could also change the array implementation to handle overflow by automatically resizing the array as items are pushed on the stack.)

To allow the two implementations to coexist, we first define an *abstract* Stack, containing just the public member functions, but no data.

```
class Stack {
  public:
    Stack();
    virtual ˜Stack();          // deallocate the stack
    virtual void Push(int value) = 0;
                               // Push an integer, checking for overflow.
    virtual bool Full() = 0;   // Is the stack is full?
};

// For g++, need these even though no data to initialize.
Stack::Stack {}
Stack::˜Stack() {}
```

The Stack definition is called a *base class* or sometimes a *superclass*. We can then define two different *derived classes*, sometimes called *subclasses* which inherit behavior from the base class. (Of course, inheritance is recursive – a derived class can in turn be a base class for yet another derived class, and so on.) Note that I have prepended the functions in the base class is prepended with the keyword virtual, to signify that they can be redefined by each of the two derived classes. The virtual functions are initialized to zero, to tell the compiler that those functions must be defined by the derived classes.

Here's how we could declare the array-based and list-based implementations of Stack. The syntax : public Stack signifies that both ArrayStack and ListStack are kinds of Stacks, and share the same behavior as the base class.

```
class ArrayStack : public Stack {  // the same as in Section 2
  public:
    ArrayStack(int sz);   // Constructor:  initialize variables, allocate space.
    ~ArrayStack();        // Destructor:   deallocate space allocated above.
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();     // Returns TRUE if the stack is full, FALSE 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.
};

class ListStack : public Stack {
  public:
    ListStack();
    ~ListStack();
    void Push(int value);
    bool Full();
  private:
    List *list; // list of items pushed on the stack
};

ListStack::ListStack() {
    list = new List;
}

ListStack::~ListStack() {
    delete list;
}

void ListStack::Push(int value) {
    list->Prepend(value);
}

bool ListStack::Full() {
    return FALSE;  // this stack never overflows!
}
```

The neat concept here is that I can assign pointers to instances of ListStack or ArrayStack to a variable of type Stack, and then use them as if they were of the base type.

```
Stack *s1 = new ListStack;
Stack *s2 = new ArrayStack(17);

if (!stack->Full())
    s1->Push(5);
if (!s2->Full())
    s2->Push(6);

delete s1;
delete s2;
```

The compiler automatically invokes `ListStack` operations for `s1`, and `ArrayStack` operations for `s2`; this is done by creating a procedure table for each object, where derived objects override the default entries in the table defined by the base class. To the code above, it invokes the operations `Full`, `Push`, and `delete` by indirection through the procedure table, so that the code doesn't need to know which kind of object it is.

In this example, since I never create an instance of the abstract class `Stack`, I do not need to *implement* its functions. This might seem a bit strange, but remember that the derived classes are the various implementations of Stack, and Stack serves only to reflect the shared behavior between the different implementations.

Also note that the destructor for `Stack` is a virtual function but the constructor is not. Clearly, when I create an object, I have to know which kind of object it is, whether `ArrayStack` or `ListStack`. The compiler makes sure that no one creates an instance of the abstract `Stack` by mistake – you cannot instantiate any class whose virtual functions are not completely defined (in other words, if any of its functions are set to zero in the class definition).

But when I deallocate an object, I may no longer know its exact type. In the above code, I want to call the destructor for the derived object, even though the code only knows that I am deleting an object of class `Stack`. If the destructor were not virtual, then the compiler would invoke `Stack`'s destructor, which is not at all what I want. This is an easy mistake to make (I made it in the first draft of this article!) – if you don't define a destructor for the abstract class, the compiler will define one for you implicitly (and by the way, it won't be virtual, since you have a *really* unhelpful compiler). The result for the above code would be a memory leak, and who knows how you would figure that out!

### 4.1.2 Shared Implementation

What about sharing code, the other reason for inheritance? In C++, it is possible to use member functions of a base class in its derived class. (You can also share data between a base class and derived classes, but this is a bad idea for reasons I'll discuss later.)

Suppose that I wanted to add a new member function, `NumberPushed()`, to both implementations of `Stack`. The `ArrayStack` class already keeps count of the number of items on the stack, so I could duplicate that code in `ListStack`. Ideally, I'd like to be able to use the same code in both places. With inheritance, we can move the counter into the `Stack` class, and then invoke the base class operations from the derived class to update the counter.

```
class Stack {
  public:
    virtual ˜Stack(); // deallocate data
    virtual void Push(int value); // Push an integer, checking for overflow.
    virtual bool Full() = 0; // return TRUE if full
    int NumPushed();         // how many are currently on the stack?
```

```
  protected:
    Stack(); // initialize data
  private:
    int numPushed;
};

Stack::Stack() {
    numPushed = 0;
}

void Stack::Push(int value) {
    numPushed++;
}

int Stack::NumPushed() {
    return numPushed;
}
```

We can then modify both ArrayStack and ListStack to make use the new behavior of Stack. I'll only list one of them here:

```
class ArrayStack : public Stack {
  public:
    ArrayStack(int sz);
    ~ArrayStack();
    void Push(int value);
    bool Full();
  private:
    int size;        // The maximum capacity of the stack.
    int *stack;      // A pointer to an array that holds the contents.
};

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

void
ArrayStack::Push(int value) {
    ASSERT(!Full());
    stack[NumPushed()] = value;
    Stack::Push();      // invoke base class to increment numPushed
}
```

There are a few things to note:

1. The constructor for ArrayStack needs to invoke the constructor for Stack, in order to initialize numPushed. It does that by adding `: Stack()` to the first line in the constructor:

   ```
   ArrayStack::ArrayStack(int sz) : Stack()
   ```

The same thing applies to destructors. There are special rules for which get called first – the constructor/destructor for the base class or the constructor/destructor for the derived class. All I should say is, it's a bad idea to rely on whatever the rule is – more generally, it is a bad idea to write code which requires the reader to consult a manual to tell whether or not the code works!

2. I introduced a new keyword, `protected`, in the new definition of `Stack`. For a base class, `protected` signifies that those member data and functions are accessible to classes derived (recursively) from this class, but inaccessible to other classes. In other words, protected data is `public` to derived classes, and `private` to everyone else. For example, we need `Stack`'s constructor to be callable by `ArrayStack` and `ListStack`, but we don't want anyone else to create instances of `Stack`. Hence, we make `Stack`'s constructor a protected function. In this case, this is not strictly necessary since the compiler will complain if anyone tries to create an instance of `Stack` because `Stack` still has an undefined virtual functions, `Push`. By defining `Stack::Stack` as `protected`, you are safe even if someone comes along later and defines `Stack::Push`.

   Note however that I made `Stack`'s data member `private`, not `protected`. Although there is some debate on this point, as a rule of thumb you should never allow one class to see directly access the data in another, even among classes related by inheritance. Otherwise, if you ever change the implementation of the base class, you will have to examine and change all the implementations of the derived classes, violating modularity.

3. The interface for a derived class automatically includes all functions defined for its base class, without having to explicitly list them in the derived class. Although we didn't define `NumPushed()` in `ArrayStack`, we can still call it for those objects:

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

ASSERT(s->NumPushed() == 0); // should be initialized to 0
```

4. Conversely, even though we have defined a routine `Stack::Push()`, because it is declared as `virtual`, if we invoke `Push()` on an `ArrayStack` object, we will get `ArrayStack`'s version of `Push`:

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

if (!s->Full()) // ArrayStack::Full
    s->Push(5); // ArrayStack::Push
```

5. `Stack::NumPushed()` is not `virtual`. That means that it cannot be re-defined by `Stack`'s derived classes. Some people believe that you should mark *all* functions in a base class as `virtual`; that way, if you later want to implement a derived class that redefines a function, you don't have to modify the base class to do so.

6. Member functions in a derived class can explicitly invoke public or protected functions in the base class, by the full name of the function, `Base::Function()`, as in:

```
void ArrayStack::Push(int value)
{
    ...
    Stack::Push();      // invoke base class to increment numPushed
}
```

Of course, if we just called `Push()` here (without prepending `Stack::`, the compiler would think we were referring to `ArrayStack`'s `Push()`, and so that would recurse, which is not exactly what we had in mind here.

Whew! Inheritance in C++ involves lots and lots of details. But it's real downside is that it tends to spread implementation details across multiple files – if you have a deep inheritance tree, it can take some serious digging to figure out what code actually executes when a member function is invoked.

So the question to ask yourself before using inheritance is: what's your goal? Is it to write your programs with the fewest number of characters possible? If so, inheritance is really useful, but so is changing all of your function and variable names to be one letter long – "a", "b", "c" – and once you run out of lower case ones, start using upper case, then two character variable names: "XX XY XZ Ya ..." (I'm joking here.) Needless to say, it is really easy to write unreadable code using inheritance.

So when is it a good idea to use inheritance and when should it be avoided? My rule of thumb is to only use it for representing *shared behavior* between objects, and to never use it for representing *shared implementation*. With C++, you can use inheritance for both concepts, but only the first will lead to truly simpler implementations.

To illustrate the difference between shared behavior and shared implementation, suppose you had a whole bunch of different kinds of objects that you needed to put on lists. For example, almost everything in an operating system goes on a list of some sort: buffers, threads, users, terminals, etc.

A very common approach to this problem (particularly among people new to object-oriented programming) is to make every object inherit from a single base class *Object*, which contains the forward and backward pointers for the list. But what if some object needs to go on multiple lists? The whole scheme breaks down, and it's because we tried to use inheritance to share implementation (the code for the forward and backward pointers) instead of to share behavior. A much cleaner (although slightly slower) approach would be to define a list implementation that allocated forward/backward pointers for each object that gets put on a list.

In sum, if two classes share at least some of the same member function signatures – that is, the same behavior, *and* if there's code that only relies on the shared behavior, then there *may* be a benefit to using inheritance. In Nachos, locks don't inherit from semaphores, even though locks are implemented using semaphores. The operations on semaphores and locks are different. Instead, inheritance is only used for various kinds of lists (sorted, keyed, etc.), and for different implementations of the physical disk abstraction, to reflect whether the disk has a track buffer, etc. A disk is used the same way whether or not it has a track buffer; the only difference is in its performance characteristics.

## 4.2 Templates

Templates are another useful but dangerous concept in C++. With templates, you can parameterize a class definition with a *type*, to allow you to write generic type-independent code. For example, our `Stack` implementation above only worked for pushing and popping *integers*; what if we wanted a stack of characters, or floats, or pointers, or some arbitrary data structure?

In C++, this is pretty easy to do using templates:

```
template <class T>
class Stack {
  public:
    Stack(int sz);     // Constructor:  initialize variables, allocate space.
    ~Stack();          // Destructor:   deallocate space allocated above.
    void Push(T value); // Push an integer, checking for overflow.
    bool Full();       // Returns TRUE if the stack is full, FALSE otherwise.
```

```
  private:
    int size;         // The maximum capacity of the stack.
    int top;          // Index of the lowest unused position.
    T *stack;       // A pointer to an array that holds the contents.
};
```

To define a template, we prepend the keyword `template` to the class definition, and we put the parameterized type for the template in angle brackets. If we need to parameterize the implementation with two or more types, it works just like an argument list: `template <class T, class S>`. We can use the type parameters elsewhere in the definition, just like they were normal types.

When we provide the implementation for each of the member functions in the class, we also have to declare them as templates, and again, once we do that, we can use the type parameters just like normal types:

```
     // template version of Stack::Stack
template <class T>
Stack<T>::Stack(int sz) {
    size = sz;
    top = 0;
    stack = new T[size];   // Let's get an array of type T
}

     // template version of Stack::Push
template <class T>
void
Stack<T>::Push(T value) {
    ASSERT(!Full());
    stack[top++] = value;
}
```

Creating an object of a template class is similar to creating a normal object:

```
void
test() {
    Stack<int> s1(17);
    Stack<char> *s2 = new Stack<char>(23);

    s1.Push(5);
    s2->Push('z');
    delete s2;
}
```

Everything operates as if we defined two classes, one called `Stack<int>` – a stack of integers, and one called `Stack<char>` – a stack of characters. `s1` behaves just like an instance of the first; `s2` behaves just like an instance of the second. In fact, that is exactly how templates are typically implemented – you get a complete *copy* of the code for the template for each different instantiated type. In the above example, we'd get one copy of the code for `int`s and one copy for `char`s.

So what's wrong with templates? You've all been taught to make your code modular so that it can be re-usable, so *everything* should be a template, right? Wrong.

The principal problem with templates is that they can be *very* difficult to debug – templates are easy to use if they work, but finding a bug in them can be difficult. In part this is because current generation C++

debuggers don't really understand templates very well. Nevertheless, it is easier to debug a template than two nearly identical implementations that differ only in their types.

So the best advice is – don't make a class into a template unless there really is a near term use for the template. And if you do need to implement a template, implement and debug a non-template version first. Once that is working, it won't be hard to convert it to a template. Then all you have to worry about code explosion – e.g., your program's object code is now megabytes because of the 15 copies of the hash table/list/... routines, one for each kind of thing you want to put in a hash table/list/... (Remember, you have an unhelpful compiler!)

# 5 Features To Avoid Like the Plague

Despite the length of this note, there are numerous features in C++ that I haven't explained. I'm sure each feature has its advocates, but despite programming in C and C++ for over 15 years, I haven't found a compelling reason to use them in any code that I've written (outside of a programming language class!)

Indeed, there is a compelling reason to avoid using these features – they are easy to misuse, resulting in programs that are harder to read and understand instead of easier to understand. In most cases, the features are also redundant – there are other ways of accomplishing the same end. Why have two ways of doing the same thing? Why not stick with the simpler one?

I do not use any of the following features in Nachos. If you use them, *caveat hacker*.

1. **Multiple inheritance.** It is possible in C++ to define a class as inheriting behavior from multiple classes (for instance, a dog is both an animal and a furry thing). But if programs using single inheritance can be difficult to untangle, programs with multiple inheritance can get really confusing.

2. **References.** Reference variables are rather hard to understand in general; they play the same role as pointers, with slightly different syntax (unfortunately, I'm not joking!) Their most common use is to declare some parameters to a function as *reference parameters*, as in Pascal. A call-by-reference parameter can be modified by the calling function, without the callee having to pass a pointer. The effect is that parameters look (to the caller) like they are called by value (and therefore can't change), but in fact can be transparently modified by the called function. Obviously, this can be a source of obscure bugs, not to mention that the semantics of references in C++ are in general not obvious.

3. **Operator overloading.** C++ lets you redefine the meanings of the operators (such as + and >>) for class objects. This is dangerous at best ("exactly which implementation of '+' does this refer to?"), and when used in non-intuitive ways, a source of great confusion, made worse by the fact that C++ does implicit type conversion, which can affect which operator is invoked. Unfortunately, C++'s I/O facilities make heavy use of operator overloading and references, so you can't completely escape them, but think twice before you redefine '+' to mean "concatenate these two strings".

4. **Function overloading.** You can also define different functions in a class with the same name but different argument types. This is also dangerous (since it's easy to slip up and get the unintended version), and we never use it. We will also avoid using default arguments (for the same reason). Note that it can be a good idea to use the same name for functions in different classes, provided they use the same arguments and behave the same way – a good example of this is that most Nachos objects have a `Print()` method.

5. **Standard template library.** An ANSI standard has emerged for a library of routines implementing such things as lists, hash tables, etc., called the standard template library. Using such a library should make programming much simpler if the data structure you need is already provided in the library. Alas, the standard template library pushes the envelope of legal C++, and so virtually no compilers (including

g++) can support it today. Not to mention that it uses (big surprise!) references, operator overloading, and function overloading.

6. **Exceptions.** There are two ways to return an error from a procedure. One is simple – just define the procedure to return an error code if it isn't able to do it's job. For example, the standard library routine `malloc` returns NULL if there is no available memory. However, lots of programmers are lazy and don't check error codes. So what's the solution? You might think it would be to get programmers who aren't lazy, but no, the C++ solution is to add a programming language construct! A procedure can return an error by "raising an exception" which effectively causes a `goto` back up the execution stack to the last place the programmer put an exception handler. You would think this is too bizarre to be true, but unfortunately, I'm not making this up.

While I'm at it, there are a number of features of C that you also should avoid, because they lead to bugs and make your code less easy to understand. See Maguire's "Writing Solid Code" for a more complete discussion of this issue. All of these features are legal C; what's legal isn't necessarily good.

1. Pointer arithmetic. Runaway pointers are a principal source of hard-to-find bugs in C programs, because the symptom of this happening can be mangled data structures in a completely different part of the program. Depending on exactly which objects are allocated on the heap in which order, pointer bugs can appear and disappear, seemingly at random. For example, `printf` sometimes allocates memory on the heap, which can change the addresses returned by all future calls to `new`. Thus, adding a `printf` can change things so that a pointer which used to (by happenstance) mangle a critical data structure (such as the middle of a thread's execution stack), now overwrites memory that may not even be used.

   The best way to avoid runaway pointers is (no surprise) to be *very* careful when using pointers. Instead of iterating through an array with pointer arithmetic, use a separate index variable, and assert that the index is never larger than the size of the array. Optimizing compilers have gotten very good, so that the generated machine code is likely to be the same in either case.

   Even if you don't use pointer arithmetic, it's still easy (easy is bad in this context!) to have an off-by-one error that causes your program to step beyond the end of an array. How do you fix this? Define a class to contain the array *and its length*; before allowing any access to the array, you can then check whether the access is legal or in error.

2. Casts from integers to pointers and back. Another source of runaway pointers is that C and C++ allow you to convert integers to pointers, and back again. Needless to say, using a random integer value as a pointer is likely to result in unpredictable symptoms that will be very hard to track down.

   In addition, on some 64 bit machines, such as the Alpha, it is no longer the case that the size of an integer is the same as the the size of a pointer. If you cast between pointers and integers, you are also writing highly non-portable code.

3. Using bit shift in place of a multiply or divide. This is a clarity issue. If you are doing arithmetic, use arithmetic operators; if you are doing bit manipulation, use bitwise operators. If I am trying to multiply by 8, which is easier to understand, `x << 3` or `x * 8`? In the 70's, when C was being developed, the former would yield more efficient machine code, but today's compilers generate the same code in both cases, so readability should be your primary concern.

4. Assignment inside conditional. Many programmers have the attitude that simplicity equals saving as many keystrokes as possible. The result can be to hide bugs that would otherwise be obvious. For example:

   ```
   if (x = y) {
     ...
   ```

Was the intent really `x == y`? After all, it's pretty easy to mistakenly leave off the extra equals sign. By never using assignment within a conditional, you can tell by code inspection whether you've made a mistake.

5. Using `#define` when you could use `enum`. When a variable can hold one of a small number of values, the original C practice was to use `#define` to set up symbolic names for each of the values. `enum` does this in a type-safe way – it allows the compiler to verify that the variable is only assigned one of the enumerated values, and none other. Again, the advantage is to eliminate a class of errors from your program, making it quicker to debug.

## 6 Style Guidelines

Even if you follow the approach I've outlined above, it is still 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 the Nachos project, and in general, 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 should be capitalized, except for `main` and library functions, which are kept lower-case for historical reasons.

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 using a particular `.h` file requires another `.h` file to be included (e.g., `synch.h` needs class definitions from `thread.h`) you should include the dependency 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. For example, you might have a `.h` file that uses a definition from one `.h` file, but also defines something needed by that `.h` file.

In this case, you will have to do something ad-hoc. One thing to realize is that you don't always have to completely define a class before it is used. If you only use a pointer to class `Stack` and do not access any member functions or data from the class, you can write, in lieu of including `stack.h`:

```
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.

6. Use `ASSERT` statements liberally to check that your program is behaving properly. An assertion is a condition that if FALSE signifies that there is a bug in the program; `ASSERT` tests an expression and aborts if the condition is false. We used `ASSERT` above in `Stack::Push()` to check that the stack wasn't full. The idea is to catch errors as early as possible, when they are easier to locate, instead of waiting until there is a user-visible symptom of the error (such as a segmentation fault, after memory has been trashed by a rogue pointer).

   Assertions are particularly useful at the beginnings and ends of procedures, to check that the procedure was called with the right arguments, and that the procedure did what it is supposed to. For example, at the beginning of List::Insert, you could assert that the item being inserted isn't already on the list, and at the end of the procedure, you could assert that the item is now on the list.

   If speed is a concern, ASSERTs can be defined to make the check in the debug version of your program, and to be a no-op in the production version. But many people run with ASSERTs enabled even in production.

7. Write a module test for every module in your program. Many programmers have the notion that testing code means running the entire program on some sample input; if it doesn't crash, that means it's working, right? Wrong. You have no way of knowing how much code was exercised for the test. Let me urge you to be methodical about testing. Before you put a new module into a bigger system, make sure the module works as advertised by testing it standalone. If you do this for every module, then when you put the modules together, instead of *hoping* that everything will work, you will *know* it will work.

   Perhaps more importantly, module tests provide an opportunity to find as many bugs as possible in a localized context. Which is easier: finding a bug in a 100 line program, or in a 10000 line program?

# 7 Compiling and Debugging

The Makefiles we will give you works only with the GNU version of make, called "gmake". 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 sometimes go into a loop printing the lower-most frame (`ThreadRoot`), and you have to type control-C when it says "more?". If you understand assembly language and can fix this, please let me know.

# 8 Example: A Stack of Integers

We've provided the complete, working code for the stack example. 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 simple stack test, type `make all` – this will compile the simple stack test (`stack.cc`), the inherited stack test (`inheritstack.cc`), and the template version of stacks (`templatestack.cc`).

# 9 Epilogue

I've argued in this note that you should avoid using certain C++ and C features. But you're probably thinking I must be leaving something out – if someone put the feature in the language, there must be a good reason, right? I believe that every programmer should strive to write code whose behavior would be immediately obvious to a reader; if you find yourself writing code that would require someone reading the code to thumb through a manual in order to understand it, you are almost certainly being way too subtle. There's probably a much simpler and more obvious way to accomplish the same end. Maybe the code will be a little longer that way, but in the real world, it's whether the code works and how simple it is for someone else to modify, that matters a whole lot more than how many characters you had to type.

A final thought to remember:

> "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

# 10 Further Reading

James Coplien, "Advanced C++", Addison-Wesley. This book is only for experts, but it has some good ideas in it, so keep it in mind once you've been programming in C++ for a few years.

James Gosling. "The Java Language." Online at "http://java.sun.com/" Java is a safe subset of C++. It's main application is the safe extension of Web browsers by allowing you to download Java code as part of clicking on a link to interpret and display the document. Safety is key here, since after all, you don't want to click on a Web link and have it download code that will crash your browser. Java was defined independently of this document, but interestingly, it enforces a very similar style (for example, no multiple inheritance and no operator overloading).

C.A.R. Hoare, "The Emperor's Old Clothes." *Communications of the ACM*, Vol. 24, No. 2, February 1981, pp. 75-83. Tony Hoare's Turing Award lecture. How do you build software that really works? Attitude is everything – you need a healthy respect for how hard it is to build working software. It might seem that addding this whiz-bang feature is only "a small matter of code", but that's the path to late, buggy products that don't work.

Brian Kernighan and Dennis Ritchie, "The C Programming Language", Prentice-Hall. The original C book – a very easy read. But the language has evolved since it was first designed, and this book doesn't describe all of C's newest features. But still the best place for a beginner to start, even when learning C++.

Steve Maguire, "Writing Solid Code", Microsoft Press. How to write bug-free software; I think this should be required reading for all software engineers. This really *will* change your life – if you don't follow the recommendations in this book, you'll probably never write code that completely works, and you'll spend your entire life struggling with hard to find bugs. There is a better way! Contrary to the programming language types, this doesn't involve proving the correctness of your programs, whatever that means. Instead, Maguire has a set of practical engineering solutions to writing solid code.

Steve Maguire, "Debugging the Development Process", Microsoft Press. Maguire's follow up book on how to lead an effective team, and by the way, how to be an effective engineer. Maguire's background is that he is a turnaround artist for Microsoft – he gets assigned to floundering teams, and figures out how to make them effective. After you've pulled a few all-nighters to get that last bug out of your course project, you're probably wondering why in heck you're studying computer science anyway. This book will explain how to write programs that work, *and* still have a life!

Scott Meyers, "Effective C++". This book describes how 50 easy ways to make mistakes C++; if you avoid these, you will be a lot more likely to write C++ code that works.

Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley. This should be the definite reference manual, but it isn't. You probably thought I was joking when I said the C++ language was continually evolving. I bought the second edition of this book three years ago, and it is already out of date. Fortunately, it's still OK for the subset of C++ that I use.