Copied directly From: http://cs.brown.edu/courses/cs123/docs/java_to_cpp.shtml with minor edits….

Java to C++ Transition Tutorial

1 :: Introduction

C++ is a fast, powerful, and flexible programming language. It was originally developed by Bjarne Stroustrup at the AT&T Bell Labs during 1983-1985. The C++ programming language was derived from the C programming language. It attempts to retain as much of C's syntax as possible while adding most of the OOP features that you know and (have been brainwashed to) love. In that vein, C++ is a very large and complex programming language, designed to support many different programming paradigms. As such, C++ contains tons of features that you probably should never use, and it has many potholes that you must learn to avoid.

About this tutorial

This Java to C++ transition tutorial gives an overview of the C++ programming language, focusing on the most commonly used features of the language. No guide of this length could begin to discuss the intricacies of this robust language, and this guide does not purport to do so. Instead, it gives students with a background in Java and object-oriented principles a brief yet somewhat thorough introduction to the language. Code examples are used in abundance in order to increase exposure to C++'s syntax and style. Aside from covering the basics of C++, related topics such as debugging tips and makefiles are discussed in brief. This is a short tutorial, not a reference; you will most likely need to gain access to one of the recommended books if you intend to program in C++ for any substantial period of time.

This tutorial is intended for students who have taken CS15 and CS16, CS17 and CS18 or CS19. As such, a decent knowledge of and experience with the Java programming language, as well as familiarity with object-oriented programming and design are assumed. Because of these expected pre-requisites, aspects of C++ that are similar to Java are covered in minimal detail. Note that having taken CS31, though helpful, is not needed; computer hardware and memory concerns are discussed, albeit in a cursory manner.

This tutorial was created in the fall of 1997 for use in CS123. Substantial revisions, additions, and enhancements have been added in subsequent years, especially in the spring and summer of 2002. It is maintained by the CS123 TA staff.

CS123 uses C++ for graphics programming which focuses on speed, so this tutorial focuses more on the C aspects of the language (i/o and strings for example). However C++'s standard library has some great stuff in it so check out the Standard Template Library Programmer's Guide. It covers the STL, which you may need/want to use in this course.

Books and references

Below is a list of several good books that you should either read or refer to if you have any questions.

- The C++ Programming Language,
- by Bjarne Stroustrup (3rd Edition, Addison-Wesley, 1997) Bjarne Stroustrup is the creator of the C++ programming language, and this book is the reference that he has written. It is a good reference book, but not one you would sit down and read to learn the language.

(There is a copy of this book in the Sunlab.)

- C++ for Java Programmers,
- by Timothy Budd (Addison-Wesley, 1999) Recommended by the CS15 TA staff as an excellent transition.
- C++ Primer,
- by Stanley B. Lippman (Addison-Wesley Professional; 3rd edition, April 2, 1998) This book is simpler to read than the Stroustrup book, but is not as good a reference manual.
- Effective C++: 55 Specific Ways to Improve Your Programs and Designs,
- by Scott Myers (Addison-Wesley Professional; 3rd edition, May 22, 2005) A book that is not geared for beginners but that is highly recommended once you have a grounding in the language. It covers some of the fine points of good C++ coding and design and avoiding pitfalls particular to the language.

2 :: Hello World

For an introduction, let's take a quick look at the canonical first program, "Hello World!", in both Java and C++. If we examine a Java application instead of an applet, the two programs are very similar.

```
[Hello.java]
package hello;      // says that we are part
                // of a package named hello

public class Hello  // declare a class called Hello
{
    public static void main(String args[])  // declare the
                                            // function main
                                            // that takes an
                                            // array of Strings
    {
        System.out.println("Hello world!"); // call the static
                                            // method println on
                                            // the class System.out
                                            // with the parameter
                                            // "Hello world!"
    }
}

[Hello.C]
#include <stdio.h>  // include declarations for the standard I/O library
                    // where the printf function is specified

int main(int argc, char *argv[])  // declare the function main that
                                  // takes an int and an array of strings
                                  // and returns an int as the exit code
{
    printf("Hello world!\n"); // call the function printf with the
                              // parameter "Hello world!\n", where \n is
                              // a newline character
}
```

Pretty similar, eh?

You will already notice a few key changes. The first is that there can be global functions, functions which are not methods of a class, such as printf and main. The next thing you may see is that we have a #include statement. This tells the compiler to read in a file that usually contains class or function declarations. Third, notice that the Java program includes a package declaration, whereas C++ has no analagous concept of packages.

Another minor difference you may have noticed is that in Java, main takes one parameter: an array of Strings. (In C++, we use a char* instead of a String, but this will be explained later.) If we want to find out how many elements are in a Java array, we can query it by accessing the array's length property. In C++, arrays have no properties or methods; they aren't pseudo-objects as in Java. For this reason, main takes a second parameter, argc, the number of arguments contained in the argument array argv.

Finally, in Java the main method does not return a value, whereas in C++ it returns an integer. In C++, the integer returned is known as the *exit code*, which signifies whether or not the program terminated successfully. A value of 0 indicates success, and any other value means the program failed. If no value is explicitly returned, it will automatically return a value indicating success.

3 :: Classes

There are quite a few differences in syntax between how classes and functions are declared in Java and C++. The biggest difference you will immediately notice is that while all function definitions are included in the class declaration in Java, they are usually put in separate files in C++.

Note that a function *definition* is different from a function *declaration*. A definition contains the body of the function while the declaration is simply a function's name, argument types and return type.
First, in Java:

```
[Foo.java]
public class Foo          // declare a class Foo
{
    protected int _num; // declare an instance variable of type int

    public Foo()          // declare and define a constructor for Foo
    {
        _num = 5;         // the constructor initializes the _num
                          // instance variable
    }
}
```

Then, in C++:

```
[Foo.H]
class Foo               // declare a class Foo
{
    public:                 // begin the public section
      Foo();               // declare a constructor for Foo
    protected:              // begin the protected section
      int m_num;           // declare an instance variable of type int
};

[Foo.C]
#include "Foo.H"

Foo::Foo()              // definition for Foo's constructor
{
    m_num = 5;           // the constructor initializes the _num
                         // instance variable
}
```

We split the program into two files, a *header file* (which we gave the extension .H) and a *program file* (which we gave the extension .C). The header file contains the class declarations for one or more classes, and the program file contains method definitions. The program file includes the header so that it knows about the declarations.

Separating the program declaration and definition into two files has several distinct advantages. First, you can easily look at a header file and see the interface for a particular class, without being having to see its implementation. Second, separating the header and program files can speed program compilation when the implementation of a class changes.
Also take note of how the naming convention for member variables has changes. In C++, "_" and "__" are reserved for the compiler and libraries so using variables that start with those characters could lead to a crazy compiler error or worse. We prefer starting member variables with "m_" though another common way is to *end* the name with an underscore. Either is acceptible really so just be consistent. Also note that it is good practice to start global variables with "g_".
The *scope operator* :: is used when declaring methods. If I have a class called Foo and it has a method called myMethod, when defining the function in the .C file, I would call it Foo::myMethod. The scope operator is needed because a .C file could contain method definitions for multiple classes or even global functions, so we need to know for which class each method is being defined. In the example class below, we can see the scope operator in use:

```
[Foo.H]
class Foo {
    public:
      Foo();
      ~Foo();
      int myMethod(int a, int b);
};  // note the semicolon after the class declaration!

[Foo.C]
#include "Foo.H"
#include <stdio.h>

Foo::Foo()  // scope operator :: helps define constructor for class Foo
{
    printf("I am a happy constructor that calls myMethod\n");
    int a = myMethod(5,2);
    printf("a = %d\n",a);
}

Foo::~Foo()
{
    printf("I am a happy destructor that would do cleanup here.\n");
}

int Foo::myMethod(int a, int b)
{
    return a+b;
}
```

It's crucial that you remember the semicolon at the end of a C++ class declaration. Failure to include the semicolon will cause a compile-time error, but *not* at the end of the class declaration. Often the error will be reported in a perfectly viable file, such as in a header file that you included.

Constructors and initializer lists
When an instance of a class is created, you frequently want to initialize various instance variables, some of which are objects. In Java this is easy: you can initialize those variables and

perform other startup tasks in the constructor. In C++ you can do the same. C++ constructors can take a variety of parameters as in Java, plus there are some special constructors that we will discuss later. In addition, you can initialize instance variables in an *initializer list* before the rest of the constructor is called. Whether you use initializer lists for this purpose is partially a matter of personal preference. However, you will need to know its syntax: it's needed sometimes, such as when calling superclass constructors.
For the header file Foo.H:

```
[Foo.H]
class Foo
{
    public:
      Foo();
    protected:
      int m_a, m_b;
    private:
      double m_x, m_y;
};
```

The following two definitions for Foo's constructor are functionally equivalent:

```
[Foo.C] // with initializer list
#include "Foo.H"

Foo::Foo() : m_a(1), m_b(4), m_x(3.14), m_y(2.718)
{
    printf("The value of a is: %d", m_a);
}
```

OR

```
[Foo.C] // without initializer list
#include "Foo.H"

Foo::Foo()
{
    m_a = 1; m_b = 4; m_x = 3.14; m_y = 2.718;
    printf("The value of a is: %d", m_a);
}
```

Useless trivia: The order in which the instance variables are initialized is not the order in which they appear in the initializer list, but instead the order in which they are listed in the class declaration.
Useful trivia: Don't use the *this* keyword inside an initializer list!
In case you're wondering: Learning how to initialize objects requires some concepts and syntax you haven't learned yet. See the variables and memory management sections for more information.

Destructors
If you were paying attention to the first example in this section, you may have noticed that we declared a method Foo::~Foo in addition to Foo::myMethod and the constructor Foo::Foo. The special method is called a *destructor* and is executed when an instance of the class is destroyed.

We will discuss it in more detail when we reach the memory management section.
Both constructors and destructors do not return anything. In addition, destructors take no parameters.

Protection
Like in Java, there are 3 levels of protection for class members in C++: public, private, and protected. They act pretty much the same way as they do in Java. Unlike Java, C++ has no notion of package friendliness, as there are no packages. Because of this, protected members are only accessible to subclasses, while in Java the whole package can use protected members. As you've probably noticed, you put members in sections by their protection level. You can have as many sections of each protection level in a class declarations as you would like. If no modifier is specified, the protection level defaults to private.
C++ also has an additional form of control over protection levels called *friendship* that allows for a finer grain of protection. You will probably not need to use this in the majority of your coding career. To find out more, consult one of the recommended books.

Inlining
In Java, methods are declared and defined in the same place. It is possible to do this in C++ as well, by defining methods in the header file. This is known in C++ as *inlining*. Inlining is typically used only for very short methods such as accessors and mutators. For short routines, C++ can frequently optimize things which are declared inline by inserting the code for the routine directly in the calling function, thus avoiding the large overhead of calling a function. So, it is often to your advantage to inline definitions of accessors, mutators, and other short, frequently called functions that don't require use of an external library.

```
[Color.H]
class Color
{
    public:
      Color();
      int getRed() { return m_red; }       // inline declaration
      void setRed(int red) { m_red=red; }  // inline declaration
      /* same for green and blue */
    protected:
      int m_red, m_green, m_blue;
};
```

Note that medium and large-sized functions should not be inlined, *especially* if they are frequently used. Since the the compiler will insert the code for inlined functions whereever the function is called, inlining large functions will increase the size of your program, possibily slowing it down. Inlining a big function that only gets called once is fine.
<hr width = 50%>

Overloading
In Java and C++ you can have more than one function with the same name. C++ uses the types of the parameters to determine which version of the function to call. There are all kinds of rules about when C++ will do implicit casts and other fancy things for you, but if you don't feel like spending a few weeks with Stroustrup learning about them right now, simply avoid ambiguity when you do overloading. How do you do that? If possible only overload on the number of

parameters as opposed to the types of the parameters until you have learned all the rules. Or, if possible, call the functions by different names to avoid overloading entirely (OpenGL uses this method).

```
class Foo
{
    public:
      Foo();

      print(int a)    { printf("int a = %d\n",a); }
      print(double a) { printf("double a = %lf\n",a); }
};
```

On an instance "foo" of type "Foo", calling

`foo.print(5);`
will output
`int a = 5`
whereas
`foo.print(5.5)`
will output
`double a = 5.5`

Hint for later: When you learn about pointers and start overloading things so they take either a pointer type or an int, the symbol NULL is actually an int! This has brought down many a great C++ programmer. The workaround is to explicitly cast NULL to the pointer type you want.

Default parameters
You can give default values for parameters of functions in the .H file. If fewer parameters are passed than the function takes, it will use the default values. Using default values can sometimes help you avoid overloading functions or constructors. Note that parameters without default values must precede all the parameters with defaults; you can't skip arbitrary parameters in the middle of a function call. For example:

```
class Foo
{
    public:
      Foo();
      void setValues(int a, int b=5) { m_a = a; m_b = b; }
    protected:
      int m_a, m_b;
};
```

If we have an instance "foo" of class "Foo" and we did the following:
`foo.setValues(4);`
it would be the same as if we had coded: `foo.setValues(4,5);`

Inheritance

Inheritance in C++ and Java is pretty similar. Suppose we have a class B that inherits from a class A:

```
class A
{
    public:
      A();
};

class B : public A
{
    public:
      B();
};
```

This says that B has a public superclass A; there are types of inheritance other than public, but they are never used in real programs.
If you want to pass a parameter to the superclass constructor, you can do it in the initializer list:

```
[Foo.H]
class A
{
    public:
      A(int something);
};

class B : public A
{
    public:
      B(int something);
};

[Foo.C]
#include "Foo.H"

A::A(int something)
{
    printf("Something = %d\n", something);
}

B::B(int something) : A(something)
{
}
```

Not bad at all, eh? Umm, that is, as long as you don't use *multiple inheritance*.
Yes, classes in C++ can inherit from more than one superclass. You should be aware that this is possible and that it is occasionally used. However, multiple inheritance leads to many troubling issues that you don't want to deal with. *Avoid doing it* and we will all be happier people. As you gain experience with C++ you will hopefully learn where it is appropriate and where it isn't.
Note: You could make a class with only pure virtual functions and use it with multiple inheritance to get the same effect as interfaces have in Java. But wait until you know a lot more about multiple inheritance first!
Note two: In Java, you could create an interface when you wanted to group a bunch of constants together. Classes then implemented that interface to use the constants. In C++, instead use a

separate header file to define constants. Include that header file in the program files where you need the constants.

Virtual functions

To better explain virtual functions, examine the following example in Java:

```
public void someMethod() {
    Object obj = new String("Hello");
    String output = obj.toString(); // calls String.toString(),
                                    // not Object.toString()
}
```

The method toString() is defined in class Object and overridden in class String. In the above example, Java knows that obj is really of type String, so at it calls the String.toString() method. (This is polymorphism at work.) It can resolve which method to call at run-time since in Java, all methods are *virtual*. In a virtual method, the compiler and loader (or VM) make sure that the correct version of the method is called for each particular object.

As you can imagine, making everything virtual by default adds a little overhead to your program, which is against C++'s philosophy. Therefore, *in C++ functions are not virtual by default*. If you don't declare a function virtual and override it in a subclass, it will still compile even though the "correct" version of the method may not get called! The compiler may give you a warning, but you should simply remember to do this for any function that you may override later.

The virtual keyword, the opposite of the keyword final, allows you to say that a function is virtual:

```
class A
{
     public:
        A();
        virtual ~A();
        virtual void foo();
};

class B : public A
{
    public:
        B();
        virtual ~B();
        virtual void foo();
};
```

If you're uncomfortable with all of this, to be safe you can just declare all of your methods virtual. This isn't good design in reality so you should try to understand virtual and use it correctly.

The destructor should always be declared virtual. The brief explanation as to why is say you later go back and derive from a class with a non-virtual destructor. When that subclass gets destroyed the results are "undefined" according to Stroustrup. Not all of the memory may end up being released because all of the destructors won't get called. If you don't understand that, it's okay.

Just make sure you always make destructors virtual to save headaches later on.

Pure virtual functions

Java provides the keyword abstract to declare that a method is abstract or *pure virtual*. C++ also provides for making methods pure virtual. To do this, add the code = 0 after the parameter list in the function declaration.

For example, here are the Java and C++ equivalents of making a method pure virtual. First, in Java:

```
public class Foo
{
    public abstract int abstractMethod();
}
```

And then in C++:

```
class Foo
{
    public:
        virtual int abstractMethod() = 0;  // The "virtual" and "= 0" are the
                                           // key parts here.
}
```

Just like in Java, a class derived from Foo cannot be instantiated unless all pure virtual functions have been defined. Also like Java, intermediate abstract subclasses that don't define their parent's pure virtual methods need not list them in their header file.

Overriding and scope

Say we have a class A and its subclass B. Say that they both have a virtual function foo and B wants to call A's foo. In Java, you would use the super command to use A's foo from B's. However, C++ has multiple inheritance, so we need another way to specify which foo to call. The scope operator :: allows us to do this:

```
[Foo.H]
class A
{
    public:
      A();
      virtual void foo();
};

class B : public A
{
    public:
      B();
      virtual void foo();
};
```

```
[Foo.C]
#include "Foo.H"

A::foo()
{
   printf("A::foo() called\n");
}

B::foo()
{
   printf("B::foo() called\n");
   A::foo();
}
```

So, if we have an instance "b" of class "B", calling

```
b.foo();
```

will output

```
B::foo() called A::foo() called
```

4 :: C++ Variables and Memory Management

In C++, variables are declared in exactly the same way as in Java. Declaration of an integer variable would look like this under both languages:

```
int myNumber;
```

You can also assign values to local variables at the time of declaration, just as in Java:

```
int myNumber = 0;
```

(Instance variables can not be assigned a value when declared in the header file; they are initialized in the constructor instead.)

As you see above, C++ and Java declare base type variables in basically the same way. When it comes to declaring variables that can hold a class, however, things get a little more interesting. Before we go on, we must talk some about memory.

Memory

Java has shielded you from dealing with computer memory directly, and your teachers did not go too much into it.

A computer is made up of many distinct parts. Among these, the most important ones are the CPU (central processing unit) and memory. If you have a CPU and memory, and throw in some sort of I/O (input/output) device, you have a simple, yet functional computer.

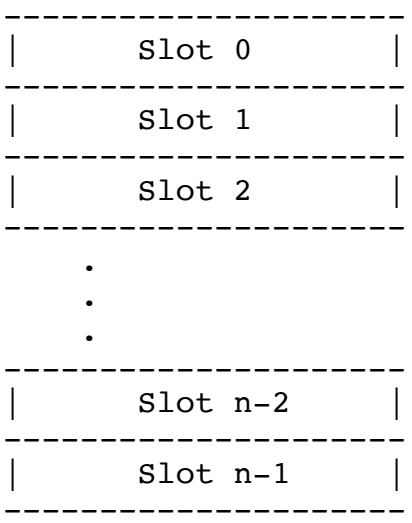
As you might guess, the memory device allows a computer to "remember" things, such as programs and data. The computer remembers everything as numbers in binary form, ones and zeros, on and off switches. A single binary digit is called one *bit* of information. Computers store everything as bits, including larger data such as strings and classes. Such data types are represented as binary numbers (groups of bits) and are stored that way.

You may wonder how a program you write can be compiled into a meaningful series of ones and

zeros that your computer understands. Well, it's the job of the compiler to take your program, parse it, and reduce it to special binary numbers called *machine language instructions* that the CPU on your computer understands. When you run your program, the computer loads these instructions into memory and executes them.

How does memory work?

Memory can be thought of as a very large number of "slots." Each slot holds 8 bits, or one *byte*. The computers you will be working on have many, many bytes (2 gigabytes, about 2147483648 bytes, of memory is a standard configuration for computers today). To organize all these slots, you can think of the computer as arranging them in a list. Slot 0 is at the beginning, slot 1 follows it, and so on, until there are no more slots.



A key thing to realize is that all slots have a unique number associated with them. Referring to "slot 5" is always talking about the same slot.

As mentioned before, each of these slots can hold a single byte. So if we stick a byte into each of those slots, we can say things like "I want to add the byte in slot 7 to the byte in slot 20," or "I want to copy the byte in slot 100 to slot 200." (Such commands are represented by one or more machine language instructions.)

Now that you know how bytes are stored in memory, the next question is how bigger things are stored. Integers, for example, take 32 bits to store (on a 32-bit processor, and 64 bits on a 64-bit processor... see the pattern?) Well, they are just stored as 4 consecutive bytes. Larger types, such as classes, are similarly stored in consecutive memory slots. The computer stores a class in memory by turning it into several numbers. These numbers contain the values of the instance variables in your class and other such information. This class is then stored in memory in a series of consecutive slots.

We can do things with classes that we did with the numbers above. Just as we could say "add the number in slot 7 to the number in slot 20," we can say about classes, "take the class *starting* at slot 5 and do something to it." Since a class takes several slots, we deal with them in terms of the first slot they occupy. The compiler keeps track of how large each class is so that it knows how many slots after the initial one are used.

Pointers

What is a memory address?

A *memory address* is the first of the slots mentioned above.

What is a pointer?

A *pointer* is a memory address.

So, a pointer to an integer myInt is "the number of the slot that stores myInt," or more commonly, "the memory address of myInt."

How do you declare a pointer?

To declare a pointer to an integer, we place the *star operator* * between the data type and the variable name. For example:

```
int* myIntegerPointer;
```

One of the uses of the * is to tell the compiler that we want something to be a pointer when we are declaring it. So the line above means "I want a pointer to an integer" and not just an integer. So, now we have a pointer to an integer. However, we didn't assign it a value, so right now it points nowhere. When you declare a pointer, it is pointing to nothing, or worse yet, it often points to a random slot. Therefore, you can not use a pointer without first giving it somewhere to point. Well, you can try using it, but your program will crash with a segmentation fault or a bus error.

How do you make a pointer point somewhere?

As we saw earlier, pointers point to data stored in memory. We need to get the memory address of some data in order to be able to make the pointer point to it. To get the memory address of something, we use the & symbol. One of its meanings is "address of."

Now, let's make an integer and have our pointer point to it.

```
int* myIntegerPointer;
int myInteger = 1000;
myIntegerPointer = &myInteger;
```

Let's do this in a little program and see what happens.

(Note: The example programs in this section make heavy use of the printf statement and special formatting characters, such as %p or %d. To learn more about these, see the section on the standard I/O library.)

```
[main.C]
#include <stdio.h>

int main(int argc, char **argv) {
    int myInteger = 1000;             // declare an integer with value 1000
    int* myIntegerPointer = &myInteger;  // declare a pointer to an integer
                                         // and make it point to myInteger

    printf("%d\n", myInteger);           // print the value of the integer
    printf("%p\n", myIntegerPointer);    // print the value of the pointer
}
```

This program gives the following output:

```
1000
4026529420
```

1000 is the value of the integer. 4026529420 is the value of the pointer, that is, the memory address of the integer (we made that address up).

How do you change the value to which a pointer points?
Now that we have a pointer to an integer, how can we put it to use? Well, suppose all we had was a pointer to the integer, and we wanted to change the value of the integer to which it points. Before you can actually say something like "set the value of the integer at memory address x to 50," you need to tell the compiler you are talking about the integer at address x, not the address x itself.
For instance, the code myIntegerPointer = 50 does *not* mean "set the number that myIntegerPointer points to to 50," but rather "set the value of myIntegerPointer to 50." This will change the memory address that myIntegerPointer actually points to; myIntegerPointer will now improperly point to "slot 50."
In order to modify the integer, we need to *dereference* the pointer before we use it. This is where the second use of the "*" comes in.
myIntegerPointer means "the memory address of <myInteger>."
*myIntegerPointer means "the integer at memory address <myIntegerPointer>."
Let's modify the example program to show this:

```
[main.C]
#include <stdio.h>

int main(int argc, char **argv) {
    int myInteger = 1000;
    int *myIntegerPointer = &myInteger;

    // print the value of the integer before changing it
    printf("%d\n", myInteger);

    // dereference the pointer and add 5 to the integer it points to
    *myIntegerPointer += 5;

    // print the value of the integer after changing it through the pointer
    printf("%d\n", myInteger);
}
```

The output is:

```
1000
1005
```

This is the expected output. Initially, the number myInteger has a value of 1000. We then say *myIntegerPointer += 5, which means "add 5 to the number at memory address <myIntegerPointer>."

What happens if you dereference a pointer and store it in another variable?
Examine this code:

```
int myInteger = 1000;                    // set up an integer with value 1000
int* myIntegerPointer = &myInteger;      // get a pointer to it
int mySecondInteger = *myIntegerPointer; // now, create a second integer
                                         // whose value is that of the integer
                                         // pointed to by the above pointer
```

What will happen if we change the value of myInteger? Will the value of mySecondInteger change too?
Let's see:

```
[main.C]
#include <stdio.h>

int main(int argc, char **argv) {

    int myInteger = 1000;
    int *myIntegerPointer = &myInteger;

    // declare another integer whose value is the same as the integer
    // at memory address <myIntegerPointer>
    int mySecondInteger = *myIntegerPointer;

    // print the value of the first integer before changing it
    printf("%d\n", myInteger);

    // dereference the pointer and add 5 to the integer it points to
    *myIntegerPointer += 5;

    // print the value of the integer after changing it through the pointer
    printf("%d\n", myInteger);

    // print the value of the second integer
    printf("%d\n", mySecondInteger);
}
```

The output is:

```
1000
1005
1000
```

So, the answer is no: mySecondInteger is a wholly new integer at a different memory address. Changing the myInteger variable has no effect on the mySecondIntegerVariable. By assigning the value that the pointer points to to another variable, we have created a copy of that variable's value. Such a result is rarely intended, and we'll see where this can run you into trouble when we examine pointers to objects.
Let's print out the addresses of the two integers to be sure that we have a copy. To do this, we need to add the following two lines to the above program:

```
printf("%p\n", &myInteger);
```

```
printf("%p\n", &mySecondInteger);
```
The output is:

```
1000
1005
1000
4026529420
4026529412
```
As you can see, the addresses of the two numbers do actually differ.

Can more than one pointer point to the same address?

It is possible to have multiple pointers point to the same address. When this happens, changing the value of the number at that address changes the values the other pointers are pointing to, since it is the same address. Let's see an example:

```
[main.C]
#include <stdio.h>

int main(int argc, char **argv) {

    int myInteger = 1000;
    int *myIntegerPointer1 = &myInteger;

    // declare another pointer to the integer above
    int *myIntegerPointer2 = &myInteger;

    // declare a 3rd pointer. This time, however, make it equal to one of
    // the above pointers instead of getting the address again.
    int *myIntegerPointer3 = myIntegerPointer2;

    // print the values (addresses pointed to) of the pointers
    printf("%p\n", myIntegerPointer1);
    printf("%p\n", myIntegerPointer2);
    printf("%p\n", myIntegerPointer3);

    // print the value of the number the pointers point to
    printf("%d\n", *myIntegerPointer1);
    printf("%d\n", *myIntegerPointer2);
    printf("%d\n", *myIntegerPointer3);

    // let's change the number...
    myInteger = 5000;

    // ...and print the values of the pointers again
    printf("%d\n", *myIntegerPointer1);
    printf("%d\n", *myIntegerPointer2);
    printf("%d\n", *myIntegerPointer3);
}
```

The output:

```
4026529420
4026529420
4026529420
1000
```

```
1000
1000
5000
5000
5000
```
This shows that all the pointers do indeed point to the same address, and changing the number at that address affects the number all the other pointers point to as well.

Pointers to pointers
Since pointers are just numbers in memory - on our x86 machines, they're 32-bit integers - it's possible to have pointers to these pointers. To see this, let's first declare an integer and a pointer to it as we've done before:

```
int myInteger = 1000;
int* myIntegerPointer = &myInteger;
```
Now, let's declare a pointer to the above pointer myIntegerPointer. This will be a pointer to a pointer to an integer. A pointer to an integer is of type int *, so a pointer to that will be of type int **. Making it point to the pointer is a matter of assigning the pointer's address to our double-pointer:
```
int** myIntegerPointerPointer;

myIntegerPointerPointer = &myIntegerPointer;
```
If we now dereference myIntegerPointerPointer once, we have a pointer to an integer:
(*myIntegerPointerPointer) == myIntegerPointer == memory address of myInteger
If we dereference it twice, we get the integer again:
(**myIntegerPointerPointer) == the thing at memory address myIntegerPointer == myInteger
Creating an example program that demonstrates these equalities is an exercise for the reader.

Pointers to objects
Now that you know about memory and pointers, let's take a look at how we would declare variables to types other than integers. To start, assume that we have this simple class for the sake of later examples:

```
[Foo.H]
class Foo {
    public:
      Foo();              // default constructor
      Foo(int a, int b); // another constructor
      ~Foo();             // destructor

      void bar();         // random method

      int m_blah;         // random public instance variable
};
```

To declare a variable to this class and create the class in Java, you could say:

```
Foo myFooInstance = new Foo(0, 0);
```

The above is not valid C++ syntax. The new operator returns a pointer to whatever follows it. The correct C++ syntax follows:

```
Foo* myFooInstance = new Foo(0, 0);
```

We have just made a pointer to an instance of type Foo and assigned it a value, the address of the instance.
Now, let's call the method bar on the instance. In Java, you would code:

```
myFooInstance.bar();
```

In C++ you can't do this, since myFooInstance is a pointer, and pointers need to be dereferenced before being used. To call a method through a pointer in C++, you would code:

```
myFooInstance->bar(); // dereference the pointer and call the method
```

Likewise, we can access public instance variables of instances of Foo. Of course, you would never do that. :)

```
myFooInstance->m_blah = 5;
```

The *arrow operator* -> does two things for you: it dereferences the pointer, and then it calls a method on the instance or accesses a member variable. This is shorthand for saying:
(*myFooInstance).bar();
which is basically carrying out the dereference and access steps explicity. Since the arrow is shorthand for this, carrying out the two steps manually is almost never done.

Instances
In Java, the only way to create an object is to new one and store a reference to it in a variable. In C++, it is possible to declare objects without newing them explicitly. For example, here we declare a local variable of type Foo without using new and a pointer:

```
Foo myFooInstance(0, 0);
```

This line of code creates a variable of type Foo and passes the specified parameters to its constructor. If we wanted to create a Foo instance using the default constructor instead, we could say:

```
Foo myFooInstance; // same as Foo myFooInstance();
```

In Java, myFooInstance would be a null reference. In C++, it's an actual instance. Let that sink in for a minute because you might not be used to it.
If we don't want to refer to the instance later, say, because it is being passed as a parameter, we can leave out the variable name:

```
// ... suppose the class Bar defines the method setAFoo(Foo foo)
...
Bar bar;
bar.setAFoo( Foo(5,3) ); // pass an instance of Foo
```

Calling methods and accessing public instance variables of an instance has the same syntax that you're used to in Java:

```
myFooInstance.bar();
```

```
myFooInstance.m_blah = 5;
```
Like pointers, instances may be local variables or member variables. If an instance is a member variable of a class, its constructor can be called in the class's constructor's initializer list, as in the following example:

```
[Bar.H]
#include "Foo.H" // must include Foo.H since we declare an instance of it

class Bar {
    public:
      Bar(int a, int b);
    protected:
      Foo m_foo; // declare an instance of Foo
};

[Bar.C]
Bar::Bar(int a, int b) : m_foo(a,b) // call Foo::Foo(int,int) and initialize
m_foo
{
    Foo fooLocal; // create another instance of Foo, this time as a local var
    // do something with the two Foos, m_foo and fooLocal
}
```

References

Suppose you allocate a chunk of memory for an object. Sometimes, it may be useful to refer to this block of memory with more than one name. We can sort of already do this with pointers, since multiple pointers can point to the same object. There is also a way to do it without using pointers; we can use something called *references* instead. Look at the program below to see how references can be used:

```
[main.C]
#include <stdio.h>

int main(int argc, char **argv) {

    int foo = 10;
    int& bar = foo;

    bar += 10;
    printf("foo is: %d\n", foo);
    printf("bar is: %d\n", bar);

    foo = 5;
    printf("foo is: %d\n", foo);
    printf("bar is: %d\n", bar);
}
```

Here, we have allocated memory to hold an integer and named it foo in the first line. The & sign you see in the second line declares a reference to an integer variable. By assigning foo to bar, bar does not become a copy of foo, but instead refers to the same memory location as foo. When you change the value of bar, it also changes the value of foo and vice versa. References are essentially the same as pointers, except that they are dereferenced like instances, can never be NULL, and can only be assigned to once, at creation. This makes references "safer" than

pointers. The output of the above program should look like:

```
foo is: 20
bar is: 20
foo is: 5
bar is: 5
```

Since references can be assigned to only at creation, references that are members of a class must be assigned to in the constructor's initializer list:

```
[Bar.H]
class Foo;

class Bar {
    protected:
      Foo & m_foo; // declare an reference to a bar
    public:
      Bar(Foo & fooToStore) : m_foo(fooToStore) {}
};
```

References are used most commonly when dealing with parameters; see the parameters section for more information.

Converting between pointers and instances
You can convert between pointers and instances using the "*" and "&" operators that were mentioned above.
Example 1 (making a pointer from a local variable):

```
Foo myFooInstance(0, 0);     // create local variable instance of foo
Foo* fooPointer;             // declare a pointer to Foo classes.
fooPointer = &myFooInstance; // set the value of the pointer to be the
address
                            // of foo.
```

Now, the following two statements have the same effect:

```
myFooInstance.bar(); // call bar through the instance
fooPointer->bar();   // call bar through the pointer
```

Example 2 (making a local variable from a pointer):

```
Foo* fooPointer = new Foo(0, 0); // create a pointer to Foo and give it a
                                 // value
Foo myFooInstance = *fooPointer; // dereference the pointer and assign it
                                 // to myFooInstance; copy is made (!)
```

The above code may not have the result that you expect. If you remember from earlier, we had an example of a pointer to an integer that we dereferenced and stored in a second integer variable. We discovered that the second integer was actually a copy of the first. A similar thing is

happening here: the instance that fooPointer points to and myFooInstance are actually two separate instances.
The first line news an instance and assigns the address of that instance to the pointer. The second statement dereferences the pointer and assigns *the instance* to myFooInstance. Here, the compiler performs a bitwise copy of the instance pointed to by fooPointer and assigns it to myFooInstance, or, if you have defined an assignment operator, a member-wise copy is done. So, saying fooPointer->m_blah = 5; would *not* change the value of blah in myFooInstance. Doing things like this yields really confusing code and is a potential source of really big, juicy bugs. For this reason, it is usually a bad idea to do this.
By the way, while we're at it, let's talk about the copy constructor!
A *copy constructor* is a constructor that is invoked when one instance of a class is initialized with another instance of the same type.
The syntax for a copy constructor is:

```
class Foo {
    Foo(const Foo &classToCopy); // copy constructor
};
```

A copy constructor usually assigns all the values of instance variables in the class that is passed in to the instance variables that this constructor is called on.

5 :: Memory Management
Clearly, declaring and using variables is a major aspect of programming. The memory needed to store these variables varies with the type of the variable and where it is declared. There are two major categories of storage:

- *local storage*: valid only within a certain scope. Local storage is also known as *automatic memory* or storage *on the stack*.
- *global storage*: valid throughout the execution of the program. Global storage is also called the *free store*, *dynamic memory*, or storage *on the heap*.

Local storage
The following block of code shows an integer and a instance of the class Bar being allocated in local storage:

```
{
    int myInteger; // memory for an integer allocated
                   // ... myInteger is used here ...

    Bar bar; // memory for instance of class Bar allocated
             // ... bar is used here ...
}
```

The { and } symbols mark the beginning and the end of a *block*. When program flow enters the block, memory needed to store an integer is allocated for myInteger, and memory needed to store the class instance is allocated for the variable bar. When the end of the block is reached, this memory used to store myInteger and bar is freed up and those variables cease to exist. Trying to use the variables after the block is closed will yield compile errors, just as in Java.

Allocating memory with new

In the example above, we're out of luck if we want to use bar outside of its block. If we want to do this, we need to put bar in global storage instead. In C++, we can request a block of memory in global storage for certain data types by using new, and we return the memory by using delete. As you've seen in the pointers section, the syntax for the new operator is as follows:

`new ClassName(`*`params`*`);`

On success, a chunk of memory that is the size of the object is allocated and a pointer to that memory is returned. If the memory can not be allocated to store the instance, (which will most certainly never happen on our machines) an exception is thrown (std::bad_alloc) . Note that if the class has a constructor that takes no parameters, the parentheses are optiona

The following C++ code shows how you can allocate memory and use it later:

```
[Bar.H]
class Bar {
    public:
      Bar() { m_a = 0; }
      Bar(int a) { m_a = a; }
      void myFunction(); // this method would be defined elsewhere (e.g. in
Bar.C)
    protected:
      int m_a;
};

[main.C]
#include "Bar.H"

int main(int argc, char *argv[])
{
    // declare a pointer to Bar; no memory for a Bar instance is allocated
now
    // p currently points to garbage
    Bar * p;

    {
        // create a new instance of the class Bar (*p)
        // store pointer to this instance in p
        p = new Bar();
    }

    // since Bar is in global storage, we can still call methods on it
    // this method call will be successful
    p->myFunction();
}
```

Notice that you can still use the object generated by the new statement even if you are outside the block. You can see that except for pointer statements, this segment of code is the same as in Java. (Indeed, Java is doing exactly the same thing behind the scenes.)

Deallocating memory with delete

In Java, you allocate memory for an object using new, and a garbage collector frees the memory automatically when no existing object references it. In C++, you have to be much more responsible than that. Whatever memory you allocate in global storage, you must explicitly free, or your program will swell in size and contain what are called *memory leaks*. To avoid leaks, you

need to keep track of all the memory you have newed and free it when you no longer need it. Actually, it is very easy to free memory that you have newed. Use delete to deallocate the memory when you are done with it. For example, to free the memory allocated above, add this line at the end of the function:

```
delete p; // memory pointed to by p is deallocated
```

Remember that *only objects created using new should be deleted with delete!* Instances created in local storage are automatically recycled and should not be deleted explicitly. For example, the following code will make your program crash:

```
Bar bar; // bar not created with new
// ... use the instance of Bar ...

delete &bar; // EEK! bar is in local storage... chaos ensues!
```

Arrays and single instances are deleted differently. See the arrays section for more information on deleting arrays.

Managing memory: Classes

Technically speaking, deleting objects is pretty easy. So, it seems, it should be just as easy to avoid leaking memory in your programs. Unfortunately, this is not the case; people often write code that has leaks everywhere. Later, in the debugging section, we will introduce some methods for eliminating leaks. However, there is an easy and effective way of avoiding them: good programming style.

We mentioned class destructors earlier, but we didn't mention their use. In Java, you don't have to deallocate memory. You seldom need to fill in the finalize() method for an object. In C++, memory that is newed is not deallocated automatically, so you have to explicitly free it. Since you can free memory at any time your program is running, the question is when to do it. The following is a good rule of thumb: memory allocated in a constructor should be deallocated in a destructor, and memory allocated in a function should be deallocated before it exits.

The following C++ class definition is an example of poor memory management in a class:

```
[Foo.H]
#include "Bar.H"

class Foo {
    private:
      Bar* m_barPtr;
    public:
      Foo() {}
     ~Foo() {}
      void funcA() {
          m_barPtr = new Bar;
      }

      void funcB() {
         // use object *m_barPtr
      }

      void funcC() {
        // ...
        delete m_barPtr;
```

```
    }
};
```

Notice that in the above class, some memory is allocated when funcA is called. This memory is freed up when the function funcC is called.
Here is some code that uses the above class:

```
{
    Foo myFoo; // create local instance of Foo
    myFoo.funcA(); // memory for *m_barPtr is allocated

    // ...
    myFoo.funcB();
    // ...
    myFoo.funcB();
    // ...

    myFoo.funcC(); // memory for *m_barPtr is deallocated
}
```

The above code does not leak any memory. When funcA is called, we allocate some memory that is used internally by myFoo. Calling funcB then uses the memory. Finally, calling funcC frees up the memory. Since we have deleted all newed memory, this code contains no memory leaks.
Now, let's take a look at some code that uses the Foo class improperly:

```
{
    Foo myFoo;
    //...
    myFoo.funcB(); // oops, bus error in funcB()

    myFoo.funcA(); // memory for *m_barPtr is allocated

    myFoo.funcA(); // memory leak, you lose track of the memory previously
                   // pointed to by m_barPtr when new instance stored
    //...
    myFoo.funcB();

}   // memory leak! memory pointed to by m_barPtr in myFoo is never
deallocated
```

The above snippet has a couple of errors. First of all, we call funcB before calling funcA. This means that the memory funcB operates on has not been allocated yet. This will cause a bus error, and your program will crash since m_barPtr is pointing to some random memory. Now assuming calling funcB first did not cause a crash, we proceed to call funcA two times in a row. The first time we call it, we allocate the memory and store it in a variable. The second time we call it, we allocate a new chunk of memory and assign it to the same variable again. This means that we have now lost the pointer to the previously allocated block of memory and have no way of finding it again. This causes a leak, since this memory can never be deallocated.
Now take a look at the class below, which uses a constructor and destructor correctly:

```
[Foo.H]
#include "Bar.H"

class Foo {
    private:
      Bar* m_barPtr;
    public:
      Foo() { m_barPtr = new Bar; }
     ~Foo() { delete m_barPtr; }
      void funcA() {}

      void funcB() {
        // use object *m_barPtr
      }

      void funcC() {
        // ...
      }
};
```

Memory is always allocated in the constructor at the time a Foo object is allocated. The memory is automatically deleted when myFoo is deleted or goes out of scope. Using the constructor above, it is impossible not to allocate the memory before we call funcB, nor is it possible to forget to delete the memory, since the destructor is automatically called.

Managing memory: Pointers, references, and instances

After learning about pointers, references, and instances, you may have been wondering when each type should be used in your programs. Unfortunately, there is no hard and fast rule. What is important is that you know how memory is managed for each.

When dealing with pointers, you explicitly newed a class or something, tying up some memory. Unless you explicitly call delete after you are done using that instance, it will tie up memory until your program exits.

The memory where a local variable instance is stored, on the other hand, is automatically managed by the computer. When a local variable ever goes out of scope, the memory that it ties up will be freed up automatically. It doesn't matter if you have a pointer or reference to it somewhere; if it goes out of scope, it will be destroyed, and the pointers and references to it will point to nothing. Note that this is different from Java's garbage collection.

A reference can be thought of as just another name for the value to which it refers. Consequently, references are automatically managed by the computer, and you don't need to worry about deleting them or anything.

Managing memory: Parameters

As you know, in Java parameters are passed *by reference*. When you pass a reference to an object in Java, you can change the actual object by calling methods on it or accessing its public instance variables. In C++, parameters can be passed either by reference or *by value*. Think of passing by value as passing a copy instead of the real thing.

Here's an example of passing by reference. We define the function IncrementByTwo to take a reference to an integer. Since the function has a reference, it can alter the integer that is passed in to it:

```
void IncrementByTwo(int & foo) { foo += 2; }
```
You can increment an integer variable by calling:

```
int bar = 0;
IncrementByTwo(bar);
```
The variable bar will now have been increased by two.
Now, let's define the same function, only this time we will pass by value:

```
void IncrementByTwo(int fooVal) { fooVal += 2; }
```
If we use the same code above, bar will still be 0 after IncrementByTwo has been called. This is because the formal parameter fooVal contains a *copy* of bar. So, passing by value here will not give the result that we want.
A third way to pass fooVal is to instead pass a pointer to it. In this example, we define IncrementByTwo to take a pointer to the integer:

```
void IncrementByTwo(int* fooPtr) { *fooPtr += 2; }
```
We call the function by passing a pointer:

```
int bar = 0;
IncrementByTwo(&bar); // note the & sign; remember, we pass a
pointer to bar
```
Since we passed a pointer to bar, it will be incremented by two just as we wanted.
So the question remains, how do we pass around objects in C++? Well, just as our integer bar above, objects can be passed by reference, by value, or by passing a pointer to the object. Since objects are often newed, meaning that you have a pointer to them, they are most commonly passed by a pointer. Of course, they can be passed by reference as well. However, objects are generally not passed by value, since that implies that a copy of the object is being made. For small types like integers, making a copy is not a big deal; but for objects, this can take up a lot of time. If you're passing by value to make sure that the object you passed in won't be changed, instead make the input parameter const. (See below for a description of const parameters.)
Return values
Return values can be passed in all the ways discussed above. Discussing return values, however, allows us to note a common C++ pitfall: passing a local variable outside of its scope. Above, we mentioned that variables in local storage are automatically destroyed when the block they are in closes. So, if you return a pointer or reference to a variable declared in this manner, and the variable leaves scope at some time, the pointer or reference will point to trash.
To see this, look at the following example:

```
[FooFactory.C]
#include "FooFactory.H"
#include "Foo.H"

Foo* FooFactory::createBadFoo(int a, int b) {

    Foo aLocalFooInstance(a,b); // creates an local instance of the class Foo
    return &aLocalFooInstance;  // returns a pointer to this instance

}   // EEK! aLocalFooInstance leaves scope and is destroyed!
```

Here, we've created an instance of the Foo class, passing its constructor the input parameters of the createBadFoo method. We then get the memory address of this instance and return it. At this point, everything is fine: we have a pointer to the instance of Foo that we just created. At the next step, however, the function ends since we returned, causing aLocalFooInstance to be destroyed. Now, that pointer we returned is pointing to, well, garbage.
Note that this next example is flawed as well, since we return a reference to a local variable:

```
Foo& FooFactory::createBadFoo(int a, int b) {

    Foo aLocalFooInstance(a,b); // creates an local instance of the class Foo
    return aLocalFooInstance;  // returns a reference to this instance

}   // EEK! aLocalFooInstance leaves scope and is destroyed!
```

The solution to this problem is to either return a pointer to an instance in global storage, or to return an actual instance:

```
Foo* FooFactory::createFoo(int a, int b) {
    return new Foo(a,b);  // returns a pointer to an instance of Foo
}

Foo FooFactory::createFoo(int a, int b) {
    return Foo(a,b);  // returns an instance of Foo
}
```

The moral of the story: *never* return pointers to variables you did not new, unless you can be completely sure that they will never leave scope.

6 :: Arrays
It would be an understatement to say that Java arrays are nice and C++ arrays are ugly. In C++, arrays behave like pointers but they are not exactly the same thing. Pointers are variables which contain a changeable address. Arrays are "array-typed" addresses that can't be changed.

Declaring arrays
As with most types in C++, arrays can be allocated either in local storage or in free storage using new.
Here's the syntax for declaring an array using new:
<type> *arrayName = new <type>[array size];
For example, declaring an array of 100 integers looks like this:
int* integerArray = new int[100];
We can also declare an array in local storage as follows:
<type> arrayName[array size];
Let's declare that 100 element integer array again:
int integerArray[100];
What is important to note here is that the integerArray variable is of type "int[100]" (which behaves like int*) regardless of which way we used to declare it. The value of integerArray is the memory address of the first element contained within it. The remainder of the elements are stored in *consecutive memory locations* following the first element.

Indexing arrays

To access a particular element in the array, the syntax is:

arrayName[element]

Arrays are indexed from 0, just as in Java. So, to access the fourth element in the array, we say:

arrayName[3]

When you index into an array, it is dereferenced automatically. In the last couple of examples, we declared an array called integerArray. It was an array of 100 integers, so it was of type "int[100]". Even though integerArray is a pointer to integers, if we access one of them, integerArray[10] for example, that returns an integer (int) not a pointer to one (int *). For example, we can do something like this:

```
integerArray[10] = integerArray[42] + integerArray[0] - 5;
```

How does indexing an array work?

Remember that arrays store all their elements in memory consecutively? Indexing into an array just specifies how far to go along in memory from the first element.

integerArray[0] means "*the integer at memory address integerArray*"

integerArray[1] means "*the integer at memory address integerArray + 1", and so on.*

So, alternately, you could index an array yourself, without using the [] operators by adding an offset to the beginning of the array. The element at index index of the array is accessed by saying:

*(arrayName + index)

What the above line does is add some number (index) to the address of the first element. This yields the address of the index'th element of the array. And as you saw before, if you want to access something at a memory address, you have to dereference that memory address, hence the "*". Doing all this is called *pointer arithmetic*: we're performing an actual arithmetic operation on the pointer, the memory address itself, to index into the array.

Using either array syntax or pointer arithmetic to access array elements should produce the same compiled code but sometimes smart pointer arithmetic can lead to faster, tighter code. Therefore, if you have a choice, and generally you do, use the syntax that makes your code easier to read. In other words, you should usually avoid pointer arithmetic.

What happens if I run off the end of an array?

If we have the array of 100 integers integerArray, what happens when we access element -10, or element 200? In Java, a nice java.lang.ArrayIndexOutOfBoundsException would be thrown. In C++, there is no such luxury.

integerArray[-10] for example, using the indexing scheme mentioned above, translates to the memory address:

(integerArray - 10)

What does that point to? Who knows. It could point to some data of a class, a program instruction, there is really no way of knowing.

What happens then if we say:

integerArray[-10] = 5;

Well, if the line does not crash, we just have set the value of some block in memory to the value 5. What happens now? Your program may behave in a funny way, or you could get a "segmentation fault", a "bus error", and your program would crash. When using arrays, **you must be very, very careful not to index beyond their bounds, or your program will crash in general.**

Multidimensional arrays

Multidimensional arrays with dynamic sizes do not exist in C++, but you can fake them with arrays of arrays.

So where in Java you could declare a 10x10 array like this:

int twoDArray[][] = new int[10][10];

In C++, you have to do it another way. We will declare a 10 element array, whose elements are 10 element arrays of integers.

As you recall, an array of ints is like a "int *". We want an array of arrays of integers, so prefixing another "*" will give us something of type "int **".

So, declaring the array would look like this:

int** twoDArray;

To make twoDArray be an array of arrays, we just have to new 10 arrays of pointers to integers:

twoDArray = new int*[10];

Ok, now we have an array of 10 pointers to ints, each of which we want to point to an actual array. This is done like so:

```
for(int i=0; i < 10; i++)
twoDArray[i] = new int[10];
```

The loop does steps through each array in our array of arrays and make sure each of the arrays contains 10 integers.

So in short, declaring and setting up an x by x array of elements, integers in this case, looks like this:

```
int** twoDArray = new int*[10];
for(int i=0; i < 10; i++)
twoDArray[i] = new int[10];
```

A little more complex than in Java, no? The more dimensions you have, the more arrays of arrays you will have, and the more loops you will need to set up the arrays.

In reality, our 2d (10x10) array of ints does not really need to be 2d. We have 100 integers, and we want to index the integer at location (row, col) at any given time. To do this, we could declare a 100 element array of integers and index it ourselves:

int* myArray = new int[100];

To index myArray, let's say row 0 of the 2d matrix is the first 10 elements in this array. Row 1 is the next 10, and so on. Given this scheme, we can find the correct element in the array using the formula:

myArray[row * 10 + col]

Here, we multiply row by 10, since this is how much we have to move in the array to find the first element of row row. We add col to it to specify how far in the row of 10 elements we move to find the col'th element in that row.

In general, for a 2d array of size width * height, you can declare a 1d array of width * height elements. Then index it yourself by saying:

array[row * width + column]

Indexing multidimensional arrays

Indexing multidimensional C++ arrays is the same as indexing arrays in Java. To index an

element of our two dimensional array, we would say:
twoDArray[row][col]
This line does a couple of things that gives us the result we need. We choose a particular array from the array of arrays and then choose the desired element from that array. The call 2dArray[row] returns a one-dimensional array, which we then index again using the [] operator to choose an element.
Note: If you have a large 2d array (*cough* image *cough*) it's often faster to represent it as a 1d array instead because of better cache coherency. Every time you call new you're getting a block of memory from who knows where. In other words, you aren't guaranteed contiguous areas of memory.

Deleting arrays

In Java, you did not need to delete arrays; they went away when you stopped referencing them. In C++, this is not the case. When you're done using an array that you created using new, you have to remember to delete it.
If you recall, deleting a single element looks like this:
delete <pointer>;
To delete an array, we need to tell the compiler we are deleting an array, and not just an element. The syntax is:
delete [] arrayName;
Deleting 2d arrays takes a little more work. Since a 2d array is an array of arrays, we must first delete all the elements in each of the arrays, and then we can delete the array that contains them. Assuming we had created a 2d array called 2dArray, we could delete it as follows:

```
for(int i=0; i < (number of arrays); i++)
delete [] 2dArray[i];
delete [] 2dArray;
```

C++ strings: Character arrays

There is no string base type in C++, so arrays of characters, or char*s, are used instead. A "string" contains all of the characters that make it up, followed by '\0' to denote the end of the string. ('\0' is the ASCII NUL character which has a decimal value of 0, as opposed to the ASCII '0' character which has a decimal value of 48.)
Declaring a string:
char* myString = "Hello, this is a string.";
This creates an array of 25 chars: 24 for the characters above, and one at the end with the value of '\0' to denote the end of the string.
Since strings are arrays, you cannot concatenate them using + or compare them using ==; instead, there are special functions to perform these tasks in the standard C library.
The C++ standard library has a string class (std::string) that can make your life a lot easier.

7 :: Types, Values, and Expressions

Basic Java types such as int, double, char have C++ counterparts of the same name, but there are a few differences. As you now know, there is no built in string class. Also, there is no built in null keyword. Instead NULL is defined to be 0 in many C libraries so including one of those lets you use the constant NULL which is easier to see in code than a 0.

Enumerated types

In C++, you can define enumerated types using the enum keyword. Enumerated types are sometimes useful for expressing a value that has a limited range. For example, we might create

an enum for the life cycle of a caterpillar:

```
enum CatLifeCycleType
{
   LARVA,
   CATERPILLAR,
   PUPA,
   BUTTERFLY
};
```

You can now create a variable of type CatLifeCycleType and assign to it values such as LARVA or PUPA.
Useless fact: By default, the values declared in an enum statement are assigned integer values starting at 0 and increasing. So, in the above example, LARVA represents the number 0, CATERPILLAR is number 1, and so on.
Useful fact: You may bypass this default numbering and assign each enumerated value an actual integral value. For example, redefining the above example:

```
enum CatLifeCycleType
{ LARVA = 1, CATERPILLAR = 2, PUPA = 3, BUTTERFLY = 4 };
```

Useful fact: An enumerated type can be cased off of in a switch statement.

The const keyword
In C++, the keyword const means different things according to its context. When you add const in front of a variable, it means that variable is treated like a constant. You will not be able to change the value of a const variable once you assign it. An example of its usage would be:

```
const float PI = 3.14156;
```

If an object is declared as const, then only the const functions may be called. If const is used with a member function, that means only const objects can call that function. For example, suppose you have the following class:

```
[Foo.H]

class Foo
{
    public:
      void ChangeValue(int newVal) { m_val = newVal; }
      int GetVal() const { return m_val; }

      const float PI = 3.14156;

      protected:
      int m_val;
};
```

If an instance of foo is declared as const, you cannot call ChangeValue on it. Correspondingly,

since GetVal is declared const it cannot modify m_val.
The const keyword can be used in another way that you might not expect. Parameters in a function may be declared const, which means that those parameters will not be changed during the function call. For example, consider the following function:
int multiply(const int a, const int b) { return a*b; }
Now, does this mean that only constants can be passed into multiply? *No*. Rather, it means that during this function, the parameters a and b will be treated as constants.
There are several reasons why it is good practice to use const whereever you can. One reason is efficiency. If you use const liberally and correctly there is a chance the compiler might be able to perform a few optimizations. Secondly, declaring parameters and/or functions as const improves program readability. If you declare a function const, for example, anyone reading your code will know right away that the function doesn't change the object on which it is called. In addition, the compiler will return an error if a const function modifies its object, or if a const parameter is modified in its function.

Math expressions
People often make errors when they write numeric expressions. These are often the most notorious and most difficult bugs that can ever be present in your programs. Here are a few tips you should know:

- Don't compare floating point numbers using ==. Their values are not exact, so sometimes what you expect to be equal isn't.
- Use doubles as parameters when calling math functions.
- For efficiency reasons, > is preferred to >=, < is preferred to <= , == 0 is preferred to == non-zero.
- If you try to implicitly cast a number to a lower precision, say a double to an int, Java will issue a compile-time error. You must make such a cast explicit for the code to compile. In C++, the implicit cast compiles without error.
- Integer precision truncates, but floating point precision does not! For example:

```
int x = 5; int y = 2; double z = 5.0;
double a = x / y;                       // a equals 2
double b = z / y;                       // b equals 2.5
double c = (double)x / (double)y;       // c equals 2.5
```

- This can be a source of very annoying bugs. To avoid these "errors" from occurring, explicitly cast an int to a double before performing a computation, as is done for the third example above. You can later cast back to an int if necessary.
- Since integer division truncates, for int expressions, multiply first before dividing (e.g., a * b * c / d / e;).
- Use an small number EPSILON to simulate zero in calculations with floats and doubles:

```
double a ;
int b ;
double EPSILON = 1e-6;
// You want to equally preide a and execute loop b times
// instead use:
// for ( double  i= 0 ; i < b; i+= a / b);
//
for (double i = 0 ; i < b - EPSILON; i += a / b);
```

8 :: Libraries and Utility Functions

Unlike Java, most C++ utilities are not yet within separate namespaces (packages or classes). Instead, the C++ libraries contain many global functions for your convenience. For example, in Java the cosine function is a static member in the Math class, whereas in C++ it is a global function declared in a header file. Three often-used collections of global functions are the math library, math.h, the standard I/O library, stdio.h, and the C string functions in string.h.

For information on any library routines, simply read the man pages. For instance, type man cos in a shell for information on the cosine function.

Math library

To use the standard math library, all you have to do is put the statement

#include <math.h>

at the beginning of your file.

The math library contains functions for trigonometry and for computing exponentials and powers. It also contains several commonly used constants.

Trigonometric functions

The functions sin(x), cos(x), and tan(x) each take x in radians. They both take and return doubles.

asin(x) returns the arc sine of x in the range -pi/2 to pi/2.

acos(x) returns the arc cosine of x in the range 0 to pi.

atan(x) returns the arc tangent of x in the range -pi/2 to pi/2.

The math library also contains the functions atan2(y,x) and hypot(x,y) for converting rectangular coordinates to polar coordinates.

Exponentials

exp(x) computes the exponential function e^x.

expm1(x) computes (e^x)-1 accurately even for tiny x.

log(x) computes the natural logarithm of x.

log1p(x) computes log(1+x) accurately even for tiny x.

log10(x) computes the base-10 logarithm of x.

Powers

pow(x,y) returns x raised to the power y.

sqrt(x) returns the square root of x

cbrt(x) returns the cube root of x

Additional functions

ceil(x) returns the smallest integer >= x.

floor(x) returns the largest integer <= x.

fabs(x) returns the absolute value of x.

Constants

The following are a subset of the constants in the math library:

```
e           M_E             2.7182818284590452354
pi          M_PI            3.14159265358979323846
pi/2        M_PI_2          1.57079632679489661923
sqrt(2)     M_SQRT2         1.41421356237309504880
1/sqrt(2)   M_SQRT1_2       0.70710678118654752440
```

Standard I/O library

In C++ there are two commonly used I/O libraries. One is the C++ iostreams library, which is a fancy object-oriented library. The other is the C standard I/O library. This tutorial has chosen to teach the latter.

The downside to standard I/O is that it does not automatically do type checking for you, so if you pass in the wrong type of parameters or the incorrect number of parameters, the program will seg fault or simply give incorrect output. This is not a problem that will be picked up by the compiler.

To use it, simply

#include <stdio.h>

Output

The printf function is used to send output to your shell, called stdout. Unlike many other functions, printf takes an arbitrary number of parameters. It first takes a string that can contain special characters describing the format of the output. Next it takes the variables which are to be printed.

Example:

```
printf("Hello world\n");
printf("Hello %d.\n",42);
```

Output:

```
Hello World
Hello 42
```

Input

The function scanf is used for getting input from the user in a shell. scanf works the same way as printf, except that you must pass the address of the variable you want it to stick the input in to. For example:

```
int x;
scanf("%d", &x ); // note the &x!
printf("Your number was %d\n",x);
```

You can also get a line of input with the gets() function. You have to allocate a buffer in advance for it to stick the input into.

Formatting symbols

These are some of the formatting symbols that you can use in the standard I/O functions above:

```
\n        Newline
\t        Tab
\\        \
\0xxx     Character w/ASCII xxx, where xxx is a 3-digital octal
number

%d        Placeholder for a decimal integer
%f        Placeholder for a float
%lf       Placeholder for a double
%c        Placeholder for a character
%u        Placeholder for a unsigned integer
```

```
%p       Placeholder for a pointer
%s       Placeholder for a null-terminated charracter array
(i.e. a char*)

%5.2f    Placeholder for a float with max. 5 sig. figs and up to
2 of them
       after the decimal point
%.2f     Same as above, except no restrictions placed on sig.
figs
%-5.2f   Same as above, except left justified
```
Example:

```
int a=5; int b=2; double x=11.4345345;
printf("%d,%d: %lf\n",a,b,x);
printf("%d,%d: %.2lf\n",a,b,x);
```
Output:

```
5,2: 11.4345345
5,2: 11.43
```
For more information on formatting, type man -s5 formats in a shell.

File I/O

Doing I/O on files is easy. The library has two functions called fprintf and fscanf that are exactly the same as printf and scanf except that they take a "file pointer" as their first parameter.

So, how do we get one of these file pointers? There is a function called fopen that will open a file and return a file pointer to it. fclose will close the file when you are done. For example:

```
FILE* outfile = fopen("/tmp/wack","r+");
fprintf(outfile,"Hello, file.\n");
fclose(outfile);
```
The second parameter to fopen is a mode. You can open a file with several modes, including the following:

```
r        For reading only
w        For writing only, truncate to 0 bytes
a        For writing only (starting at the end), don't truncate
(append)
r+       Open for reading and writing; don't truncate
w+       Open for reading and writing; truncate to 0 length
```
fopen will return NULL if the file can't be opened. Be sure to check! Here's the corrected example from above:

```
FILE* outfile;
if((outfile=fopen("/tmp/wack","r+") == NULL) {
  fprintf(stderr, "Error.  Can not open /tmp/wack for update.\n");
  exit(1);  // terminates the application; include <stdlib.h> to use
}
// ...
```
Three files are pre-opened for you when your program starts:

```
stdout   Standard out
stderr   Standard error (send error messages here)
stdin    Standard input
```

String functions

The C string functions in the standard library are available by using

```
#include <string.h>
```

It has a detailed man page (man string) that you should read before attempting to use the string functions. Here are some of the functions:

strlen

Returns the length of a string.

strcmp

Compares two strings. (Remember, a string is an array of characters, so you can't just use the == operator to test for equality, just as in Java). Returns 0 (i.e. false!) if the two strings are equal.

strncmp

Compares the first n characters of two strings.

strcat

Append one string on to the end of another.

strcpy

Copies a string.

strncpy

Copies the first n characters of a string.

strchr

Finds the next occurrence of a certain character in a string.

strtok

Breaks a string up into tokens. The syntax for this function is kind of strange, so be sure to read the man page.

You should ALWAYS use the "n" version of the function if it will involve playing with memory to avoid corrupting memory.

9 :: Flow of Control

Flow of control constructs are very similar in Java and in C++, with the difference being that Java has a boolean type and C++ does not. The counterparts of true and false in C++ are any expressions which evaluate to non-zero and zero values.

The if statement

The syntax for this statement is the same in C++ and Java:

```
if(<predicate>)
<do this if predicate is true (!=0)>
else
<do this if predicate is false (==0)>
```

As in Java, the else block is optional.

The switch statement

The switch statement is the same as in Java also.

```
switch (<variable to case on>) {
case <value 1>:
```

```
 <stuff to do if above variable == value 1>
 break;
case <value 2>:
 <stuff to do if above variable == value 2>
 break;
case <value 3>:
 <stuff to do if above variable == value 3>
 break;
default:
 <stuff to do if none of the cases matches the variable>
 break;
}
```

The variable or expression that the switch statement cases on can be of any type whose equality can be tested using ==. Integers and enumerated types are two commonly used values in a switch statement.

In switch statements, when a case section is done executing, flow of control will fall through into the case below. To avoid this, always put a break statement at the end of each case, and save yourself confusion later. If you must have a case block fall through for some reason, make sure you comment it so you (and others) know it is intentional.

Like Java, you do not have to have a default case.

Boolean expressions

A *predicate* is any boolean expression, i.e. an expression that evaluates 0 for false, or non-zero for true. As far as syntax goes, C++ and Java predicates are identical. Just be sure to remember that zero means false and non-zero means true.

You can combine boolean functions in any order to generate predicates. Listed below are several boolean functions that you should know. The functions are listed in *decreasing* order of precendence, and "false" really means zero and "true" means non-zero:

```
!x              Returns false if x is true and vice-versa.
x < y           Returns true if x is less than y, else false.
x > y           Returns true if x is greater than y, else
false.
x <= y          Returns true if x is less than or equal to y,
else false.
x >= y          Returns true if x is greater than or equal to
y, else false.
x == y          Returns true if x and y are equal, else false.
x != y          Returns true if x and y are not equal, else
false.
x && y          Returns true only if both x and y are true.
x ^^ y          Returns true if either x or y is true (not
both)
x || y          Returns true if one of x or y is true (or both)
```

All these operators can be combined in any way you want to generate complex expressions. Since it is easy to forget the precendence rules, *always use parentheses* in your expression to make it easier to read and to be sure that it does what you want.

Remember that testing for equality uses ==, not =. In Java, if you accidentally tried to test if two

things are equal using =, you would get a compile-time error. In C++, you will not get a compiler error and your program will just behave unexpectedly! This mistake is very easy to make and hard to track down, so be as diligent as you can to avoid it.

10 :: Iteration

Loops in Java and in C++ are practically identical. Here is a list of different types and syntaxes:
The for loop
Syntax:

```
for(<initialize counters>; <loop condition>; <increment
counters>)
   <statement>
```

For example:

```
for(int i = 0; i < 10; i++)
printf("I am counting to 10\n");
```

The loop body gets executed while the loop condition is true, and the loop terminates the first time it is false.
It should be noted that in some C++ compilers, including some on our system, the scope for a counter declared in the for loop definition is considered to be *outside* the loop. For this reason, the following code is valid in Java but could produce "Multiple variable declaration" compile errors in C++:

```
for(int i = 0; i < 10; i++) {
   // do something...
}
for(int i = 0; i < 10; i++) {
   // oops, counter i already declared...possible C++ compile
error...
   // to correct, change this loop to read "for (i = 0; ...", or
declare
   // i at start of function to avoid ambiguity
}
```

The while loop
Syntax:

```
while(<expression>)
   <statement>
```

The statement gets executed so long as the expression in the parentheses evaluates to true (non-zero). This is just like a for loop without a counter variable:

```
for(;<expression>;)
   <statement>
```

The do...while loop
Syntax:

```
do
  <statement>
while(<expression>);
```

In a do...while loop, the statement is executed before the expression is evaluated, so the statement is executed at least once, even if the expression is false. This differs from a while loop or a for loop in this respect.

11 :: The Command Line
As you already know, you can pass in command line arguments to a program when you execute it in a shell. Many shell commands take in command line arguments. One example is ls -l. Here, we are executing the program ls and passing it the command line argument -l.
How does a program read in command line arguments? Take a look at the following code:

```
[main.C]
#include <stdio.h>

int main (int argc, char* argv[]) {
  printf("Total number of arguments: %d\n", argc);
  printf("Your executable name: %s\n", argv[0]);
  for (int i = 1; i < argc; i++)  {
     printf("Argument # %d: %s\n", i, argv[i]);
  }
}
```

The program will print out all the command line arguments passed to it. Notice how argc is used to control the number of strings (char*s) we read from argv. *argv[0] will always be the name of the executable itself!* argv[1] is the first command line argument, argv[2] is the second one, etc.
If you compiled this program and named the executable my_exec, a sample execution could produce the following output:

```
$ my_exec testa testb
Total number of arguments: 3
Your executable name: my_exec
Argument # 1: testa
Argument # 2: testb
$
```

Often times, you want to input numbers as command line arguments. However, all arguments are read in as strings. To convert strings to numbers, try the functions atoi (ascii-to-int) and atof (ascii-to-float) defined in stdlib.h.
As we saw in the Hello World program, the main function returns an integer in C++. In this program we return 0 (implicitly) since the program flow reached the end of the main function successfully.

12 :: The Preprocessor

Before the "real" compiler actually touches your program, a program called the *preprocessor* processes it. The job of the preprocessor is to do simple text substitutions and the like. Preprocessor commands all start with the # character. While many C programs have relied heavily on use of the preprocessor, one of C++'s goals has been to eliminate most preprocessor use. However, there are still a few things that you must know.

#include statements

The preprocessor allows you to include the contents of one file in another. This is performed using the #include statement. If the name of the file to be included is enclosed in angle brackets < >, the compiler will search a standard list of directories for the file you wish to include; you can add entries to this list with the compiler flag -I. On the other hand, if the name of the file is in quotes " ", it will search the current directory for the files in addition to the other directories. For example:

```
#include <math.h>
#include <stdio.h>
#include "MyHeader.H"
```

The #include directive is typically used to load header files. The following is a non-exhaustive list of times when you will need to include the header file for a class:

- when you create an instance of the class
- when you call a member function on an instance of the class
- when you access a public instance variable on an instance of the class
- when you access a static function or member of the class
- when you define methods for the class (i.e. Foo.C must include Foo.H)
- when you declare a member instance in the class declaration
- when the class you're declaring is a subclass of the class (i.e. SubClass.H must include SuperClass.H)

Including lots of header files in a header file is discouraged. See the section on forward declarations for more information.

#define statements

The #define preprocessor directive will tell the preprocessor to do text substitution. For example, the following directive will substitute every occurrence of FIVE with the number 5:

```
#define FIVE 5
```

The #define statement can also do substitution with parameters, allowing you to write simple macros. However, using #define macro substitutions can have many negative side effects. Since C++ has features that make using macros largely unnecessary, we advise that you avoid them. To declare constants, you could declare an extern const int in some header file and assign that int a value in some program file. To declare short functions, simply write inline code.
A macro can be undefined with the #undef directive.

Conditional compilation
You can use the preprocessor to conditionally compile code. The statements used to do this are #if, #ifdef, or #ifndef as well as a #endif following the conditionally compiled section. For example:

```
#define COMPILE_SECTION

// ...

#ifdef COMPILE_SECTION

// some code here
// this code would be compiled since COMPILE_SECTION is defined
above

#endif
```

Conditional compilation can be used for avoiding circular includes (see below), writing code for multiple platforms, or optionally showing debugging messages.

Circular includes
In general, things can't be defined twice in C++. This means that two files cannot include each other. For example, if Foo.H does a #include "Bar.H" then Bar.H cannot #include "Foo.H". Even if the restriction on multiple definitions didn't exist, this would clearly lead to infinite recursion. Luckily, we can use conditional compilation to avoid this. All of your header files should have something like this:

```
[Foo.H]
#ifndef Foo_Header
#define Foo_Header

// put all of the header file in here!!

// remember this endif or you will have big, big problems!
#endif
```

The first time the file is read, Foo_Header isn't defined, so it defines it and reads the code in the header file. The second time, Foo_Header will already be defined, so your code will be skipped, making your class declaration defined only once.

Forward declarations
You've now learned how to prevent circular includes, but you might be wondering how to deal with two classes that actually need to know about each other. It turns out that you don't need to know anything about a class other than its name in order to declare a pointer to it. So, if a class contains a pointer to another class, instead of including the first class' header in the second's header, simply make a forward declaration. Then, include the header in the .C file:

```
[Foo.H]
#ifndef _FOO_H_ALREADY_INCLUDED_
#define _FOO_H_ALREADY_INCLUDED_
```

```
class Bar;       // forward declaration; says "class Bar exists, but we
                 // don't know anything about it"
class Foo {
   // ...
   protected:
    Bar* m_bar;  // We can declare a pointer to a bar.  We can't call
                 // any methods or declare a non-pointer bar until we
                 // include its header file.
};

#endif

[Foo.C]
#include "Foo.H"
#include "Bar.H" // must include here if we want to instantiate a Bar

// ...
```

In fact, forward declarations should not just be used to avoid circular includes. Using forward declarations instead of #include statements in your header files can drastically decrease the time it takes to compile your program after you change it. For this reason, use forward declarations as much as possible and avoid including header files unnecessarily. Your header files should have mainly forward declarations, and your program files should have mainly #include statements. Forward declarations work on references too!

13 :: Build Process

This is just a short introduction to the entire build process of how your code goes from your .C and .H files into an executable. In Java, you simply run javac, and your .java files become .class files and you run your program using the Java Virtual Machine which handles most of what you must do manually in C++. This is why we use the utility called make, because it handles all of this build process for you. For more information, see CS31 or CS167.

- For each .C file in your project, the preprocessor copies in all the .H files that are #included and creates a very large temporary file.
- Then, each expanded temporary file (from the preprocessor) is compiled into an object (.o) file, which contains the methods and data in the .C file in a format that can be executed. There is also a table of symbols (variables and functions) that are referenced but not defined in this particular .o file. (e.g. C Library functions, stuff from the support code, et cetera)
- In order to resolve all of these symbol references, the final step is to link the .o files together into an executable. The linker needs all of the .o files and any external libraries (e.g. libcs123.so) that have any referenced but unresolved symbols. The executable is basically a concatenation of your .o files with some information about external libraries.
- When you run your program, any external libraries that were linked in dynamically will be (you guessed it) dynamically loaded by the solaris loader. Now your program can execute without issue.
- So, you need to find out which files need to be recompiled, build them, then rebuild your linked object (executable) every time you make any modifications to source. To make this easier, we use make and the associated Makefile to automate the process.

14 :: Debugging Tips

There are several kinds of errors you can encounter in your code:

- compile errors
- run time errors
- numerical errors
- algorithmic errors
- memory leaks

Compile errors

Compile errors are the easiest problems to tackle. Most of the time, they are typos or obvious mistakes such as passing the incorrect numbers of parameters to functions. More so than with Java, it's likely that each actual error in your code will cause several compile errors to be returned, possibly in multiple files. As you learn to program in C++ you'll learn what the compile errors are really telling you. In the beginning, solve your errors one at a time and try to compile again.

Run time errors

Run time errors can have many causes, the most notable (and obvious) among these are the ones which crash your program. Often times, stepping through your code manually is the quickest way to find these bugs. Failing that, a powerful debugger called dbx is at your disposal. dbx allows you, among many other things, to set breakpoints, to examine values of variables, and to check memory access. However, dbx is a cumbersome tool (especially for beginners) and may not be necessary to track down a run time error. If you think you know what part of your code caused the error, it might save you time by just checking the code directly.

Numerical errors

Numerical errors are caused by limitations of your underlying software and hardware implementation. For example, you can not have a char value greater than 255, and there is limited accuracy for floating point numbers. There is no way to eliminate a numerical error. However, you can anticipate the range of your numerical calculations and choose the proper variable types and algorithms. Numerical errors are very difficult to track down. The best approach for avoiding numerical errors is careful and incremental coding.

Algorithmic errors

Often, students spend time debugging their code over and over with out thinking, "*Is it my code which is wrong or my algorithm?*" If you have spent three hours debugging ten lines of code, most likely, the problem is the latter. On such occasions, *stop coding!* Go home, eat some food, take a shower, think about what could be going wrong with your code. Most times, you will shout out "*Eureka!*", like Archimedes, and run back to the Sun Lab to finish your program in 10 minutes. If you cannot figure out the problem, don't be afraid to ask a TA. We don't encourage people to come with a piece of code for us to debug at TA hours, but if you really put some serious thinking into the problem and still cannot figure it out come ask a TA on hours.

Memory leaks

Finally, when your program is just about complete, you should try to get rid of any memory leaks in your code. Memory leaks occur when you allocate memory with new that you fail to deallocate before the program quits. To find out if you have leaky code, you can use the bcheck utility. In a shell, type bcheck <executable_name> <program_arguments>. This will output a file named <executable_name>.errs that will contain a list of all your leaks, along with the file names and line numbers where they occured. (Memory leaks can be detected from dbx as well; in fact, bcheck is just a wrapper around dbx used for checking memory leaks.)

Additional information

The CS36 web site also has a collection of C++ resources and tools that are useful, including a list of common C++ compile errors.

15 :: Miscellaneous Tips

This is just a collection of miscellaneous pieces of advice, amalgmated from students and TAs past and present about the use of C++ and things to keep in mind in general. Some of this may be repetitive, but on the other hand it's in an easy to reference location. If you have any suggestions on other things to add to this list, please let us know.

- Whenever possible, when using references as parameters, use const references - this will help prevent you from modifying objects that you don't intend to modify, and will help you catch bugs in certain situations, like using = incorrectly and accidentally copying some object.
- Do not forget to declare a method you intend to override virtual.
- Initialize all pointers that aren't being newed immediately to NULL. It can save a lot of headaches in debugging. Also, after deleting a pointer, set the pointer to NULL for the same reason.
- It's not discussed in this document as it's beyond the scope, but if you intend to use the STL, avoid newing STL classes whenever possible.
- When instantiating a local variable of type Foo, if Foo takes no arguments, then saying: Foo myFoo(); is a syntax error. Do not use the parentheses. You will get strange compile errors if you do.
- If you get weird memory errors, one thing to check is that you are not returning local arrays. Arrays are pointers too.

If all else fails, try doing a make clean; make.