Getting started in C++

CS106B/X are taught using the C++ programming language, whereas our CS106A course uses Java. Becoming exposed to another language is an excellent way to broaden your experience while simultaneously preparing for the real world where much coding is done in C/C++.

Our first goal is thus to orient you in C++. Don't let this intimidate you— C++ is syntactically similar to Java/C/C# and the core programming skills you've learned (decomposition, testing, debugging, algorithm design, etc.) are quite transferable, no matter what language you've worked in. Given the many small details and a few significant issues to cover, our first few lectures are focused on developing your "C++ legs". Although C++ is a complex and full-featured language, we will only incorporate those features that fit well with our pedagogical course goals, so no need to worry about having to learn an entire second language.

This handout points out the Java/C++ differences most relevant to our course. These are just the highlights and although we will skim these topics in lecture, plan on working through the first three chapters of the reader on your own to get the full story with additional in-depth examples.

A little history

The C programming language originated around 1970 and was designed for professional programmers writing tight low-level code. The language is known for its terseness, limited support libraries, and runtime efficiency. It is known as an unsafe language because it makes tradeoffs that value efficiency over safety (for example, access to array elements is not bounds-checked) and gives the programmer unfettered access to do what they like (e.g. pointers and typecasts). C became quite popular and even today is still one of the dominant languages in use. It is a good tool for an experienced programmer, but not as appropriate for the rest of us mere mortals who could use some safety goggles to go with our blowtorch.

C++ arrives on the scene in the 80s designed as "a better C". It extends the C language with features for data abstraction, extended facilities for user-defined types, a more extensive standard library, and improved safety, while still maintaining an emphasis on efficient runtime performance. The fact that C++ is a superset of C can be both a blessing (code compatibility) and a curse (unsafe features still exist). We switched our CS106B and X courses to C++ in 2002 and by focusing on a carefully chosen subset of features, we've been able to capitalize on the useful new functionality C++ provides and move away from some of the more treacherous areas of C.

Java is a product of the 90s and had a meteoric rise to popularity; some of it due to the Internet boom given Java was promoted for web development. Java takes much of its basic language syntax and features from C and C++. However, philosophically, Java strikes out into new territory. It is fundamentally object-oriented, cross-platform compatible, and has a huge standard library. Java strongly values safety over efficiency and provides garbage-collection, array bounds-checking, checked typecasts, and more. Java compilers are much more strict than C/C++ and close the runtime safety loopholes present in C and C++. Java's combination of features makes it a good fit with an intro course and we switched our CS106A course to Java in 2004.

All three of C, C++, and Java are widely used in industry and academia. We think it is valuable that you are learning tools that you will continue using in upper-division classes and summer internships and jobs. However, we also believe the programming skills we teach are not tied to the language. By exposing you to two languages across the intro sequence (and a few others in CS107) we're hoping to give you a solid foundation that transcends any one language or paradigm.

What does C++ look like?

To start with, here is a complete C++ program that reads some numbers from the user and computes their average.

```
/*
 * average.cpp
 * ------------
 * This program adds scores and prints their average.
 */

#include "genlib.h"
#include "simpio.h"
#include <iostream>

const int NumScores = 4;

double GetScoresAndAverage(int numScores);

int main()
{
   cout << "This program averages " << NumScores << " scores." << endl;
   double average = GetScoresAndAverage(NumScores);
   cout << "The average is " << average << "." << endl;
   return 0;
}

/*
 * Function: GetScoresAndAverage
 * Usage: avg = GetScoresAndAverage(10);
 * -------------------------------------
 * This function prompts the user for a set of values and returns
 * the average.
 */
double GetScoresAndAverage(int numScores)
{
    int sum = 0;
    for (int i = 0; i < numScores; i++) {
        cout << "Next score? ";
        int nextScore = GetInteger();
        sum += nextScore;
    }
    return double(sum)/numScores;
}
```

Familiar… but different, too.

Things that are mostly the same between C++ and Java

A quick glance at the program above might mistake it for Java, so that gives you some idea how similar the basics are. In fact, enumerating all the features that are the same would be quite a long list indeed! To give you an idea of some things you don't need to re-learn, consider these features shared by Java and C++:

Both case-sensitive
Same use of punctuation (semicolons, curly braces, commas, parentheses, square brackets, …)
Same comment sequence (`/* comment */`, `// comment`)

Same primitive variable types (e.g. **`int`**, **`double`**, **`char`** …)
(although Java's **`boolean`** type is called **`bool`** in C++)
Same operators for arithmetic, comparison, logical (e.g. **`+`**, **`%`**, **`*=`**, **`++`**, ==, **`<`**, **`&&`**, **`||`**,…)
(same syntax, precedence, associativity, short-circuiting, conversions for mixed-types, etc.)
Same control structures (e.g. **`if/else`**, **`for`**, **`while`**, **`switch`**, **`return`**, **`break`** …)

All of the basic C++ language syntax is thoroughly covered in Chapter 1 of the reader. We highly recommend that you carefully read this chapter and work through some of the review questions and exercises to refresh your fundamental skills.

Compiler/language strictness

Both Java and C++ compilers are pretty assertive about making sure your code meets certain standards (all variables declared, right number of parameters in calls, types used correctly, and so on) but there are some areas where a C++ compiler is noticeably lax compared to what you're used to in Java. Some of this is historical accident (improved compilers make possible better error-checking) and some is by design (C++ being targeted at experienced programmers who are assumed to not make mistakes and don't want to pay the costs for extra checks). Here are a few of the pitfalls that you'll need to be on the lookout for yourself, since the compiler isn't much help on these:

Forgetting to initialize a variable before you use it. In C++, trying to use the value of a variable that hasn't been assigned will just get whatever junk contents it has. Yuck! Most C++ compilers will produce a warning when you do this, but unlike Java, it is not a hard error. Depending on how much you've been relying on the compiler to remind you, you'll probably need to bump up your own attention to making sure variables are properly initialized.

Forgetting to return a value from a non-void method. C++ will allow you to "fall off the end" with no return statement and will use a junk value for the function result. Double yuck! Again, some C++ compilers will provide a warning, but not a hard error as you may have been reliant on from Java.

Using a non-boolean expression where a boolean is expected. The original C language has no explicit boolean type and it interprets any non-zero value as true, zero as false. C++ inherits this from C. For example, in the test of an **`if`** or **`while`** statement, you can use any expression (boolean or not). Combine this with the fact that an assignment statement returns a value and you have set the stage for an insidious and all-too-common error in C/C++ programming:

```
if (x = 3)       // oops! meant == but used =
      cout << "Ack!" << endl;
```

What does the above code do? It always prints Ack because it assigns x the value 3 and that value is non-zero, so the test expression is true.

Programming paradigms

Different programming languages support different paradigms. Java is an example of a language in the *object-oriented* paradigm. All Java code is written within the context of a class and classes form the patterns from which you create objects that operate at runtime. When you execute Java code, it's all about sending messages to objects. C++ supports the object-oriented paradigm, so it also allows you to define classes, create objects, and send messages to them, but the language also supports the procedural paradigm. In the *procedural* or *imperative* paradigm, you can define routines outside of any class and execute code by a sequence of calls to such routines, without use of any objects or classes. A C++ program can operate strictly in one or the other paradigm, but often make use of

both object-oriented and procedural features. C++ is termed a *hybrid* language, since it combines more than one paradigm.

In this course, our approach to C++ is one that I call "procedural programming using objects". Much of our code will be written in the procedural style, in the form of global functions that operate sequentially but that code will use many objects (e.g. the standard string and stream classes, as well as our own library classes). For the first half of the quarter, we will not be defining any new classes at all. In the second half we will work on class implementation.

Filename extensions

Whereas Java source files are named with a **`.java`** extension (e.g. **`Binky.java`**), our C++ source files are named with a **`.cpp`** extension (e.g. **`boggle.cpp`**). Other conventions for C++ source file extensions include .C, .cc, and .cxx.

#include (Reader page 1-5)

In a C++ program, you must inform the compiler about features you are using from outside this file. This is done with **`#include`** statements at the top of the file, such as

```
#include <iostream>
#include "genlib.h"
```

A **`#include`** is kind of like a Java **`import`** statement. A **`#include`** statement names a particular interface file and directs the compiler to bring in that named interface so that its features are accessible. For example, the first **`#include`** above is needed so you can access the I/O stream library, the second is for the CS106 genlib interface. Angle brackets are used for standard library interfaces, and quotes for our own interfaces. All our CS106 programs rely on in the genlib interface and must always include it.

Alternate cast syntax (Reader page 1-18)

In Java, you saw the use of the typecast to convert between types, such as when converting an integer to a floating-point value. C++ supports the same cast where the converted type is enclosed in parentheses, but you will also see an alternate syntax that puts the parentheses around the value being converted.

```
val = (double)num;  // Java-style cast
val = double(num);  // C++-style cast
```

Either form is acceptable in your programs but the textbook and lecture examples will tend to use C++ style, so you should have at least reading familiarity with this form.

Functions and main (Reader pages 1-6 to 1-8 and pages 1-32 to 1-34)

In Java, all code is organized in methods, which must be defined within a class. In C++, it is possible to define routines that exist outside any class context. These *functions* are like Java methods in terms of return value, parameters, and so on, but a function does not operate on a particular object and there is no object receiver when you call a function. Functions defined outside a class are called *global functions* or *free functions*.

There is a global function **`main`** that every C++ program must have, analogous to the **`public static void main`** method in the main class of a Java program. A C++ program starts executing at the main function, steps through its code, and exits when main is done. The main function takes no arguments and returns an integer, which reports whether the program was successful. By convention, 0 means all is well and any non-zero return is an error code.

Here is a program showing main and two other functions.

```
void Binky(int num)
{
   for (int i = 0; i < num; i++)
      cout << "Hello!" << endl;
}

int Winky()
{
   cout << "Enter your favorite number: ";
   return GetInteger();
}

int main()
{                                  // execution starts here
   int val = Winky();
   Binky(val);
   return 0;                       // and stops here
}
```

A few things to note

- When you call a global function there is no receiver (ie no **`xxx.`** before the function name). A global function is not defined within a class and there is no object being operated on. There is no "**`this`**" in context when the function is executing.
- The **`main`** function takes no parameters and returns an integer (usually 0).
- Functions must be declared before use. In the above code, both **`Binky`** and **`Winky`** are defined before **`main`** which calls them. If you did want to define **`main`** first, you would need a *forward declaration* of **`Binky`** and **`Winky`** to avoid a compiler error. A forward declaration is just the function header (return type with parameters) ended by a semicolon, e.g.

  ```
  void Binky(int num);
  ```

  A forward declaration gives the compiler advance notice of a function to be defined later in the file. This allows calls to that function to be properly checked for correctness even before the full function has been seen. A forward declaration is also called a *prototype*.
- It's largely personal preference whether you list functions top-down (**`main`** followed by the functions it calls), or bottom-up (starting with lower-level functions and working up to **`main`**). However, a top-down listing requires separate prototypes since functions are called before they are defined. The program flow can be easier to follow if organized top-down, but bottom-up has the convenience of not maintaining separate prototypes. Either strategy is fine with us.
- Our naming convention is to capitalize function names, e.g. **`ReadFile`** or **`DrawHistogram`**.

Default arguments

One minor convenience supported in C++ is the concept of default arguments. Consider the string member function **`find`** that searches a string for a given character. The function takes two arguments: the character to look for and the starting index to search from. In most situations, the starting index is zero (to search the entire string starting from the beginning) and it would be handy if the client didn't have to always supply that zero. In C++, you can define the function with a default argument, so the value could be assumed zero unless otherwise indicated. If the client calls the function without specifying a second argument, the default value is used.

A default argument is added to the function prototype after the parameter name with an equal sign:

```
int find(char ch, int start = 0);
```

Only the lastmost argument(s) can be given default arguments (otherwise when a call was made without all arguments, the compiler wouldn't know which ones were missing).

You call **find** with one or two arguments. If the second is not given, the default value is used.

```
str.find('c', 3);
str.find('c');     // uses 0 for second argument
```

Default arguments are merely a minor convenience. A similar effect can be achieved in Java with several functions of the same name that have different argument lists (**find** that takes one parameter, another **find** that takes two, etc.).

Reference parameters (Reader pages 1-34 to 1-38)

In Java, all parameters are passed by value. Passing by value means that a copy of the variable being passed is made and inside the called function any changes to the parameter only affect the copy.

C++ has two different parameter passing mechanisms. The default is *pass-by-value* (like Java). The other option is *pass-by-reference* which is indicated by adding **&** to the type of a parameter. When a parameter is passed by reference, changes to the parameter **do** change the original. A parameter passed by reference is not copied, instead a reference to the original variable is taken out, and within the called function, the parameter becomes a synonym for the original. Any changes to the parameter are reflected as changes to the original variable.

Consider the **Adjust** function that increments the first parameter by the amount of the second:

```
void Adjust(int & val, int delta) // val is ref param, delta is not
{
   val += delta;
}
```

Note that parameter **val** is passed by reference. Its value is updated within the body of **Adjust** and we want that change to be visible in the calling function.

Given this call:

```
int x = 10;
Adjust(x, 5);
```

And after the function executes, **x** will now contain the value 15. [1]

The difference between an integer parameter and an integer reference parameter can be a little subtle. Any call to **Adjust** must supply two integer arguments. The first argument is pass-by-reference and thus must be the name of a modifiable integer variable. In C++ speak, we call this an *lvalue*, which is a fancy way of saying it can appear on the left hand side of an assignment statement. The second argument can be any form of integer value (be it a constant, an expression that evaluates to an integer, a double that can be truncated to an integer, an integer variable, and so

[1] An aside on simulated pass-by-reference: it is possible in C++ to get a pointer to a variable and explicitly pass that pointer to a function that dereferences it to change the original variable. This is a way of simulating pass-by-reference and this technique dates back to the C language, which doesn't contain references. What a reference parameter is doing behind the scenes is basically the same thing but without dealing with the grungy ***** and **&** syntax and pointer pitfalls.

on). The restriction on what constitutes a valid integer reference makes sense if you think about it. For example, consider this call:

```
Adjust(3+5, 4*5);
```

Although it is fine to pass an expression for the second argument, there is no sensible interpretation for using an arithmetic expression as a reference for the first argument, thus this call produces is a compile error.

A classic use of pass-by-reference is shown by this **Swap** function to exchange two values:

```
void Swap(double & a, double & b) // a, b are ref params
{
   double tmp = a;
   a = b;
   b = tmp;
}
```

When you call this function, you pass two variables to exchange:

```
double x = 1, y = 3.4;
Swap(x, y);
```

And after the function executes, x will have the value 3.4 and y 1. You cannot write such a Swap function in Java because of its lack of reference parameters!

Even when the called function does not modify the parameter, a C++ programmer will often mark any struct or object parameter as pass-by-reference. These items can be often large in size and using pass-by-reference avoids the inefficiency of making copies of these large items as they are passed around. As a habit, we will typically pass objects by reference for this reason of efficiency.

Constants (Reader pages 1-5 to 1-6)

The **const** type modifier is used when defining named constants. A constant is like declaring a variable with a name and a type, and includes an initializer to set the value.

```
const int MaxEntries = 100;
```

The **const** modifier assures that once assigned, the value will not change; any attempt to modify a constant will result in a compiler error.

Constants allow you to associate a name with a value and then use the name in place of the value for purposes of readability and code maintenance. These are like the **static final** fields you used as constants in Java. By convention, C++ constants are often named in uppercase.

```
const double PI     = 3.141519;
const double TWO_PI = (PI*2);
```

Constants are usually listed at the top of the file, after the **#include** statements. These statements are defined outside any class context. Similar to functions they exist globally— these constants are accessible anywhere within the file once they have been defined.

The C++ **const** modifier has many other uses (parameters, member functions, etc.) but managing those details is a topic we defer to CS107. In our course, we use **const** only for program-level constants.

enum (Reader section 2.1)

The **enum** keyword is used to introduce a new type with a limited set of named values. For example, the declaration below creates a new type for day of the week:

```
enum weekdayT {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

This declaration creates **weekdayT** as an enumerated type, with possible values ranging from **Sun** to **Sat**. The type **weekdayT** can be used for variables, parameters, return types, and so on.

```
weekdayT day;

day = Sun;
if (day == Mon) ...
```

Behind the scenes, **weekdayT** is just a special kind of integer and each name is mapped to a value (**Sun** is 0, **Mon** is 1, etc.) but using an enumerated type is less error-prone and more self-documenting than that using integers with defined constants.

struct (Reader section 2.6)

The **struct** keyword is used to define a new type that is a collection of fields in one package. Structures are used to group logically related data into one variable. For example, an inventory program might track an item using a struct containing the name, price, and stock status:

```
struct itemT {
   string name;
   double price;
   bool inStock;
};
```

This type declaration establishes **itemT** as a new record type containing three fields. This type can be used for variables, parameters, return types, and so on.

```
itemT it;

it.name = "Hula hoop";
it.price = 25.99;
it.inStock = true;
```

The fields within a struct are accessed using *dot notation*. This notation should be familiar to you from Java, which uses dot notation to access fields within an object. In fact, a struct is like an object, where the fields are public to everyone and there are no methods to operate on it. Structs are most appropriate when the data being managed has no behavior and there is no need for encapsulation.

Pointers (Reader section 2.3)

A *pointer* is just a special type of variable that stores a memory address. That address allows you to access some other piece of data, called the *pointee*. You can then access the data to read and write by reference through the pointer.

Pointers are one of the trickier features of C++. Even though Java doesn't let you explicitly manipulate pointers, there are there just the same, so you have more experience with pointers than you might think! In Java, all array and object variables are pointers, even though the syntax doesn't make this apparent. When a Java program declares an array or object, space is allocated only for the pointer. You must assign that pointer to hold the location of an object in the heap in order to use it.

C++ makes pointers much more explicit than Java. You indicate for each declaration whether a variable is a pointer or not. All pointer variables are declared with an explicit **`*`** and you can have pointers to any type. Consider the declarations below:

```
int x, y;
int *p, *q;
```

This declares **`x`** and **`y`** as integers and **`p`** and **`q`** as pointers to integers. As with any variable, you must first initialize a pointer by assigning it a value. You can assign the pointer an address in one of four ways:

1) using the result of a new call	`p = new int;`
2) from another pointer variable	`q = p;`
3) to NULL	`p = NULL:   // C++ NULL is uppercase`
4) to the location of an int	`p = &x;`

To dereference a pointer to access its pointee, the **`*`** operator is used.

```
*p = 5;      // assigns p's pointee value 5
*q = *p;     // copies p's pointee value to q's pointee
```

You need to be diligent about your use of pointers in C++ because the language does not have the same safety features as Java. For example, a Java compiler requires that you initialize a variable before you use it. C++ will allow you to use the random contents. For an ordinary integer variable, using a junk-valued integer might throw off your calculations, but trying to interpret the junk contents as an address for a pointer variable usually has catastrophic, program-crashing results.

You can declare pointers to any type (including pointers to pointers!). Here is an example of a pointer to the **`itemT`** struct from above:

```
itemT *ip;

ip = new itemT;
(*ip).name = "Yo-Yo";   // parens needed, . higher than *
ip->price = 9.95;       // cleaner alternate for (*x).
```

The new and delete operators (Reader section 2.7)

C++'s **`new`** operator is basically the same as Java's. It is used for allocating variables in the heap and returning a pointer to the allocated space. Here are some examples of calls to **`new`**:

```
double *p = new double;
int arr[] = new int[10];
itemT *ip = new itemT;
Binky *b = new Binky();
```

In C++, just as in Java, an unsuccessful **`new`** request throws an exception that halts execution.

On significant way in which dynamic memory differs in the two languages has to do with releasing heap memory when no longer needed. Java provides *automatic garbage collection*, which means that the system tracks use of heap memory and when it determines a variable is no longer in use, it will automatically recycle it. C++ has no such facility, and it is responsibility of the programmer to explicitly tell the system to delete a piece of memory that is no longer needed. Thus it is the programmer's job to call **`delete`** to release unneeded memory previously allocated with **`new`**.

As a minor complication, there are two variants of **`delete`**, one with brackets and one without. The special form **`delete[]`** is used to deallocate an array that was earlier allocated using **`new[]`** (i.e an entire array allocated such as **`new int[num]`**). The basic rule is that every **`new`** call should be balanced by exactly one **`delete`** call and similarly each **`new[]`** call is matched by a **`delete[]`** call.

Figuring out what and when to delete can be a difficult task, and getting it wrong can have disastrous consequences. Typically you write the program first with no attempt to clean up memory and then slowly and carefully add the necessary delete statements to clean up where appropriate.

Arrays (Reader section 2.4)

Java arrays and C++ arrays have fairly similar syntax, but operate somewhat differently underneath. Below shows some C++ code for allocating and manipulating some arrays:

```
int arrOnStack[50];
int *arrInHeap;

arrInHeap = new int[50];

for (int i = 0; i < 50; i++) {
   arrOnStack[i] = i;
   arrInHeap[i] = arrOnStack[i] * 2;
}

delete[] arrInHeap;
```

Some things to note:

- A Java array declaration has the bracket next to the type, C++ has the brackets to the right side of the variable name.
- Both Java and C++ arrays are indexed from 0 to numElements –1.
- C++ arrays do not do bounds-checking. If you attempt to access an element off either end, it will have unpredictable results (rather than halt with an exception as Java would).
- C++ arrays do not know their length. You will need an additional variable to track the length of a C++ array.
- C++ arrays can be declared on the stack or allocated with **`new`** in the heap (Java arrays can only be allocated in the heap).

Classes (Reader section 8.1)

Defining a C++ class is similar to its Java equivalent, but there are a few differences to pick up. One trivial note is that C++ programmers favor different vocabulary. C++ programmers typically call an object's internal variables *data members* (Java calls them instance variables or fields) and the routines that operate on an object are called *member functions* (Java calls these methods).

A Java class is defined in a single **`Classname.java`** file. In C++, the class definition is separated into two files, the interface (also called the header or .h file) and the implementation (the .cpp file). The interface file is defines the class "skeleton", listing the data members and the member functions including types and access modifiers. The implementation file contains the code for the functions.

Although managing two files instead of one sounds like a burden, it allows for a nice division between the two roles of client and implementer. The client of a class only needs to know the available operations and doesn't need to know and shouldn't be bothered with how those operations are implemented. The interface file doesn't contain those details and is the only file that the client needs to see. The implementation file is intended only for the eyes of the implementer.

Here's a simple Location class for an x,y coordinate pair. It has two data members, a constructor, and three member functions: two accessors and one modifier.

Here is the class declaration in **location.h** (by convention, the name of the file is lowercase):

```
class Location {

  public:
     Location(int xVal, int yVal);          // ctor like Java
     int getX();
     int getY();
     void translate(int dx, int dy);

  private:
     int x, y;
};                          // this ending semicolon is important!
```

A couple of things to point out right away:

- The members are grouped into public and private sections (as opposed to Java, where each member is individually declared public or private). All data members should be private.
- The interface has no code for the member functions, code goes in the implementation file.
- There must be a semi-colon at the end of the class. Forgetting this is a really easy mistake to make and it can be surprisingly hard to track down the compiler error that results.
- By convention, we typically capitalize the name of our classes. The names of members begin with a lowercase letter and each subsequent word is capitalized.

The implementation file that goes with this interface contains the code to implement the member functions. Here are the contents of **location.cpp**:

```
#include "location.h"       //.cpp file includes its .h

Location::Location(int xVal, int yVal) {
     x = xVal;
     y = yVal;
}

int Location::getX() {
   return x;
}

int Location::getY() {
   return y;
}

void Location::translate(int dx, int dy) {
     x += dx;
     y += dy;
}
```

Note that each member function is prefixed by **Classname::**. The double colon is the C++ *scope* operator and it tells the compiler that you are providing the implementation for the function named **translate** within the class named **Location**. If you forget the **Location::**, the compiler will think you are defining a global function named **translate**, which is not what you wanted at all. This mistake will produce error messages about undeclared variables **x** and **y** since global functions have no "**this**" and there are no instance variables in scope.

Here is some client code that creates and uses two Location objects. The first Location object is allocated on the stack, the second is dynamically allocated in the heap:

```
Location loc(10, 5);

loc.translate(9, -3);
cout << loc.getX() << ", " << loc.getY() << endl;

Location *lp = new Location(0, 0);
lp->translate(-10, 30);
```

A few things to note:

- In Java, objects are always dynamically allocated using **`new`**. In C++, you have the choice of either allocating objects on the stack or dynamically in the heap using pointers. Typically, we use the stack to allocate our objects, but occasionally will have reason to use the heap.
- You access the members of an object by selecting the field, just as you do with a struct, e.g. **`loc.getX()`**. For an object accessed through a pointer, you will use **`->`**.

The CS106 libraries

In CS106B/X, we have a few small libraries that are specifically designed for our course. In most cases, these libraries offer tidier or more convenient ways of accomplishing tasks you could do using the standard C++ libraries. The routines in the libraries list below are packaged as global functions. You can call them from any context (as long as you have included the necessary interface file) without an object receiver.

`genlib.h` This library must be included in all CS106 programs. It sets up base facilities needed in our programs and provides the **`Error`** function will report an unrecoverable error condition and halt.

`simpio.h` This library provides four simple functions to read data entered by the user (**`GetInteger`**, **`GetLong`**, **`GetReal`**, and **`GetLine`**). You could directly read from the standard C++ input stream, but these convenient routines handle the error-checking and retry when the user enters an incorrect type. You can read more about this library on reader page 1-14.

`random.h` This library has functions for generating different types of random events (**`RandomInteger`**, **`RandomReal`**, and **`RandomChance`**). Behind the scenes, it uses the standard C++ **`rand`** function. You can read all about our random library in reader section 3.2.

`strutils.h` This library provides a smattering of handy string utility functions (**`StringToInteger`**, **`ConvertToUppercase`**, and a few others). You can read more details on reader pages 3-19 to 3-22.

`graphics.h` Our base graphics library has functions that set up a graphics window and allow you to do rudimentary black-and-white line drawings. The graphics model is a canvas on which you make persistent marks (this is different than the paintComponent() style used in CS106A). Section 5.3 of the reader discusses the simple graphics library.

`extgraph.h` The extended graphics library provides additional functions for color, text, filled shapes, mouse handling, and more. The additional graphics features are documented in our **`extgraph.h`** header file.

Strings (Reader section 3.3)

Just as in Java, there is a standard C++ class for managing a sequence of characters. The C++ `string` class is quite similar to Java's `String` class. Consider the C++ code below:

```
string s = "apple";
string t = "banana";

for (int i = 0; i < s.length(); i++)
      cout << s[i];      // square brackets to access chars

t = s.substr(0, 3);     // creates new substring

t[0] = 'm';       // C++ strings are mutable!
s = t + "binky";  // concat works for strings & chars

if ( s == "Stanford")
   s += "!";

int index = s.find("an");
```

A few things to note:

- These string objects are allocated on the stack and are not handled through pointers. We access string fields using ordinary dot notation.
- The contents of a C++ string can be changed, such as by accessing a character location with [] and reassigning it or calling member functions insert and replace. (This is unlike Java strings which are immutable).
- You compare C++ strings using the standard relational operators (==, <, >= , etc.). These comparisons are case-sensitive.
- In C++, using + for string concatenation only works for adding strings and characters. If you want to add integers or objects into a string, you must manually convert them to strings first. (There are some handy conversion functions in our strutils library).

Input/output streams (Reader section 3.4)

Pretty much every programming language has a facility for input/output with similar functionality, but each has their own particular design and syntax, which make them annoyingly different. Since I/O tends to be one of messier, detail-oriented parts of a language, you could spend a lot of time wading through arcana trying to completely master it. My general philosophy is to take the time up front to be comfortable with the basics, but plan on looking up the more specialized details (how to print with exactly two decimal places, how to read in a string of only lowercase letters, etc.) on a need-to-know basis. This handout just skims the basics to get you started.

In Java, calls to `System.out.print` print things to the console. In C++, you use the insertion operator `<<` on the standard `cout` stream. The code below will print a string, a number, and a newline to the console:

```
int x = 19;
cout << "hello" << x << endl;  // endl prints a newline
```

In Java, you typically print multiple things by first concatenating them into one string and then printing that string. In C++ you can chain together uses of `<<` to print multiple things. There are fancy ways of using stream *manipulators* to control the formatting of what is printed. Table 1-3 on reader page 1-14 gives some of the more common manipulators.

To read from a C++ stream, you can use the extraction operation **>>**. The console input stream is named **cin**. For example, the following code reads two integers from the console:

```
int x, y;
cin >> x >> y;
```

(I know **<<** and **>>** can be easily confused. Here's a way to help keep it straight: The stream is always on the far left and the arrows point in the direction of the information flow. If printing information out to the stream, the arrows point from the variables to the stream. If reading information in from the stream, the arrows point from the stream to the variables).

We will avoid using raw stream extraction on the console because it's tedious to handle the inevitable user typing errors (such as entering a letter when a number was expected). Instead, use our convenience functions in simpio.h that handle these details. For example, the code below shows a typical way to interact with the user at the console:

```
cout << "What is your name?";
string name = GetLine();
cout << "How old are you?";
int age = GetInteger();
cout << "How much would you pay for this fabulous course?";
double price = GetReal();
```

Reading and writing to files (as opposed to the console) uses the same basic approach. You use the classes **ifstream** and **ofstream** (input and output file streams) and the same insertion and extraction operators. There are also facilities for reading char-by-char and line-by-line. Here is some sample code that copies one file to another one line at a time:

```
#include <fstream>

int main() {
   ifstream in;
   ofstream out;

   in.open("data.txt");
   out.open("copy.txt");
   if (in.fail() || out.fail()) Error("Can't open files.");

   string line;
   while (true) {
      getline(in, line);
         if (in.fail()) break; // no more lines to read
      out << line << endl;
   }
   in.close();
   out.close();
}
```

You can read all about streams and file streams in reader section 3.4.