Biology Background

Introduction

The application of computer science to other disciplines seems to be increasing at an exponential rate. Each day that passes sees new ways to apply the ideas of computer science to problems in all walks of life, and especially the "hard" sciences. The fields of computational biology and bioinformatics are of particular interest here at Hunter because we have a bioinformatics/quantitative biology concentration in computer science. There are many examples of simple problems in these areas in which to demonstrate programming concepts, but to do so, the student needs to know a bit about the underlying biology. The purpose of these notes is to provide that "bit of knowledge." They are written for a prospective computer scientist rather than a biologist.

Background

A ***DNA string***, also called a ***DNA strand***, is a finite sequence consisting of the four letters `A`, `C`, `G`, and `T` in any order[1]. The four letters stand for the four ***nucleotides***: ***adenine***, ***cytosine***, ***guanine***, and ***thymine***. Nucleotides, which are the molecular units from which DNA and RNA are composed, are also called *bases*. Each nucleotide has a ***complement*** among the four: `A` and `T` are complements, and `C` and `G` are complements. Complements are chemically-related in that when they are close to each other, they form ***hydrogen bonds*** between them.

A special enzyme called ***RNA polymerase*** uses the information in DNA to create RNA. The process of creating RNA from DNA is called ***transcription***. A ***RNA string*** or ***RNA strand*** is a finite sequence consisting of the four lowercase letters `A`, `C`, `G`, and `U`. The `A`, `C`, and `G` have the same names as they do in DNA, but the `U` represents ***uracil***. When DNA is transcribed to RNA by RNA polymerase, each thymine base is converted to uracil. Hence RNA strings have `U`'s wherever DNA has `T`'s.

RNA in turn serves as a template for the construction of ***proteins***, which are sequences of ***amino acids***. Proteins are synthesized within the ***ribosomes*** of living cells by a process called ***translation***. In translation, the RNA string is viewed as a sequence of three-letter groups called ***codons***. Each codon codes for a particular amino acid. For example, `GUU` codes for *valine*, and `UCA` codes for *cysteine*. Let us count how many possible three-letter sequences are there in which each letter can be `A`, `C`, `G`, or `T`. There are four choices for the first letter, four independent choices for the second letter, and four for the third, so there are $4^3 = 64$ different codons. On the other hand, there are only 20 different amino acids. Some amino acids are coded for by multiple codons. For example, `UCA`, `UCC`, `UCG`, and `UCU` all code for *cysteine*. Some codons do not code for any amino acids; they are *stop codes*. There are three stop codons: `UAA`, `UAG`, and `UGA`.

Stop codes are used during protein synthesis to terminate reading of the RNA string. Not all of a RNA string is translated into protein; there are large regions that act like gaps. As the RNA is read, when a gap is reached, it is skipped over until a special start codon is found that tells the ribosome to begin creating amino acids again. When it sees a stop codon it stops and keeps reading until it finds another start codon, and so on, until the entire strand is read. The process is just like the way comments in shell scripts are treated by the shell. The `#` tells the shell to stop parsing the command, and the next newline character tells it to start again. The `#` is a stop code and the newline is a start code. As an example, the RNA strand

```
AUGGUUUAUGGUCUCUGA
```

[1] Some sources use lowercase while others use uppercase. Here they will be used interchangeably

is read as the following sequence of codons

```
AUG GUU UAU GGU CUC UGA
```

Consulting a table of these mappings, we see that `AUG` is a start codon that codes for *methionine* (Met), `GUU`, for *valine* (Val), `UAU`, for *tyrosine* (Tyr), `GGU`, for *glycine* (Gly), `CUC`, for *leucine* (Leu), and `UGA` is a stop codon. Therefore, the sequence *Met-Val-Tyr-Gly-Leu* is created from this RNA fragment.

Amino acids have long names like cysteine but they also have three-letter names such as Cys, for cysteine, and one-letter uppercase names, such as C for cysteine. It is not always true that the one-letter name is the first letter of the amino acid's long name. The above sequence would be written *MVYGL* using the one-letter names.

Direction and Shape

In its most common form, DNA is actually a double helix consisting of two strands that wrap around each other. Each DNA strand has ***direction***. Direction is usually indicated by putting a 5' at one end and a 3' at the other. The 5' and 3' refer to the names of the carbon atoms to which these ends attach. The carbons are part of a ring structure with multiple carbon atoms, so they get names for the purpose of distinguishing them from each other. For example,

```
5'-GTATCC-3'
```

is a fragment of DNA that runs from the 5' to the 3' position. The 5' end is called the ***upstream*** end, and the 3' end is the ***downstream*** end. The two strands of nucleotides are in reverse directions of each other. In other words, if you could unwind the helix so that the two strands were lying on a flat surface parallel to each other, in one strand the 5' end would be to the left, and in the other, it would be to the right. The two strands are chemically-related because the bases that would be across from each other on the table are complements of each other. For example, the two strands below

```
5'-G T A T C C A A T G C C-3'
   | | | | | | | | | | | |
3'-C A T A G G T T A C G G-5'
```

could be a fragment of the unwound double helix. The vertical lines connect complements in the forward and reverse strands to each other. Each `C` in one is matched by a `G` in the other, and each `A` is matched by a `T` in the other.

Characteristic Properties of DNA

Scientists use various heuristic rules in their study of DNA. Among the many metrics that they use are the following:

- A ***poly-T sequence*** of length N is a sequence of N or more consecutive `T` nucleotides.
- The ***GC content*** of a DNA strand is the ratio of the total number of `C` and `G` nucleotides to the length of the strand. For example, the sequence '`atcgtttgga`' is of length 10 and has a total of 4 `C`'s and `G`'s, so its GC content is 0.4.
- A ***CpG island*** is a `C` followed by a `G` in a DNA strand. (The *p* in between the `C` and `G` represents the fact that a *phosphodiester* bond connects `them`.)

Restriction Enzymes

Bacteria produce special enzymes called ***restriction enzymes*** that can cut DNA at specified sites, called ***cleavage sites***. Some theorize that these enzymes evolved to provide a defense mechanism against invading viruses. Inside a bacterial host, the restriction enzymes selectively cut up foreign DNA in a process called ***restriction***; the host DNA is protected from the restriction enzyme's activity.

The cleavage site is a position between two nucleotides in the DNA. The enzyme finds its site by a type of biological pattern-matching. These enzymes usually cut both strands of the DNA, but for simplicity we describe how the cut works on a single strand. The pattern specifies where in the DNA the enzyme will match. For example, the enzyme *EcoRI* has a recognition site defined by

```
5'-G'AATTC-3'
```

This means that it will search for a substring of the DNA consisting of the bases `GAATTC` in the 5' to 3' direction, and cut the DNA between the `G` and the first `A`. The apostrophe ' indicates the cleavage site. So, if the DNA string is

```
ATGAAAGGGTTTCCCTTTGAATTCCCCATGGTATTGTTGCCGGAATTCTTTCCGGCCCCC
```

it will be cut into the three pieces

```
ATGAAAGGGTTTCCCTTTG     AATTCCCCATGGTATTGTTGCCGG     AATTCTTTCCGGCCCCC
```

by *EcoRI*. The restriction enzyme *NotI* is defined by

```
5'-GC'GGCCGC-3'
```

which indicates that it will find all occurrences of the string `GCGGCCGC` in the 5' to 3' direction and cut the DNA between the first `C` and the second `G`.

You may have noticed that if you form the complement of `GAATTC`, you get `CTTAAG`, which is the string spelled backwards. Similarly, the complement of `GCGGCCGC` is `CGCCGGCG`, which is also the string spelled backwards. Certain types of restriction enzymes have this *complement palindromic* property.

There are four different types of restriction enzymes, differing in chemical ways and in the nature of their target sequence and the position of their DNA cleavage site relative to the target sequence. Some restriction enzymes have a cleavage site outside of the recognition site. *AceIII* is defined by

```
CAGCTCNNNNNNNN'
```

The `N` matches any of `A`, `C`, `G`, or `T`. Therefore, this enzyme cuts the DNA between the 7th and 8th nucleotides after its recognition site. For example

```
CAGCTCAAATGCCAGGGGGGG
```

will be cut between the `A` and the `G`:

```
CAGCTCAAATGCCA GGGGGGG
```

Some types of enzymes cut the DNA very far away from the recognition site, on the order of hundreds of nucleotides away. Some of these restriction enzymes will have more than one recognition sequence, and inversely, there are different enzymes that have the same sequence and cut in the same place.

A restriction enzyme can be described in a format known as the *Staden* format. The form is

```
enzyme_acronym/recognition_sequence/recognition_sequence/.../recognition_sequence//
```

Most enzymes have a single recognition sequence. Some have two. The cut point will be denoted by an apostrophe in the recognition sequence. Some enzymes from an actual Staden enzyme file are:

```
AatI/AGG'CCT//
AatII/GACGT'C//
AbsI/CC'TCGAGG//
AccI/GT'MKAC//
AccII/CG'CG//
AccIII/T'CCGGA//
Acc16I/TGC'GCA//
BbvI/GCAGCNNNNNNNN'/'NNNNNNNNNNNNGCTGC//
BglI/GCCNNNN'NGGC//
BglII/A'GATCT//
BinI/GGATCNNNN'/'NNNNNGATCC//
```

The first line is the enzyme named *AatI* and its cut point is between the second `G` and the first `C`. You will notice that in the fourth line, there are letters other than `A`, `C`, `G`, and `T` in the recognition sequences. These letters are part of a standard set of abbreviations defined as follows:

```
R = G or A
Y = C or T
M = A or C
K = G or T
S = G or C
W = A or T
B = not A (C or G or T)
D = not C (A or G or T)
H = not G (A or C or T)
V = not T (A or C or G)
N = A or C or G or T
```

The letters act like simple patterns for matching DNA. For example, `GTMKAC` matches `GTAGAC`, `GTCGAC`, `GTATAC`, and `GTCTAC` since the `M` matches an `A` or a `C` and the `K` matches a `G` or a `T`. There are therefore four possible combinations that the two together can match. The enzyme *BbvI* has two recognition sites, which are reverse palindromic.

Creating Functions in C/C++

Motivation

There are many reasons to create functions for your programs.

- A fragment of code that appears in your program in multiple places can be placed into a function definition and replaced by a function call instead, resulting in a smaller, more maintainable program. It is smaller because there are fewer lines of code. It is more maintainable because if you decide to change the code, you only have to do it in one place, instead of searching through the entire program for all occurrences of that code fragment. This makes it less error-prone, since it is possible to miss one of the fragments if they are dispersed throughout the program.
- By creating a function and putting a code fragment into it, the program becomes easier to read and more modular. For example, if you have code that displays a menu on the screen and gets the user's entered choice, you could create a function named `GetUserSelection()` that returns the user's choice, making it obvious to the reader what the code does. The program becomes more modular because it now contains another separately maintainable function.
- Functions can be reused more easily in other programs. You may find that you need the same code fragment in many programs. By creating a function that encapsulates it, you can put that code into other programs without having to rename variables and reorganize the program code.

There is one downside to creating a function: the ***function call overhead***. I will discuss this later.

How do you create a function?

To define a function in your C or C++ program, you write a ***function definition***, which has the form

```
result_type function_name ( parameter_list)
{ function_body }
```

where

- *result_type* is any type such as `int`, `double`, `char`, or `string`, but may also be the word `void`.
  If the result_type is `void`, it means the function does not "return" anything.
- *function_name* is any valid C++ identifier
- *parameter_list* is a list of the form *typespec parameter, typespec parameter,... typespec parameter*
  where *typespec* is a specification of a type (such as `int` or `char` but might be a bit more complex than this) and *parameter* is in the simplest case just a C/C++ identifier. (In C++ there are things that can appear here that cannot appear in a C function.)
- *function_body* is just like the body of a main program. In fact `main()` is just a special function. It consists of declarations and statements.

The first line, which contains the result type, the function name, and the parameter list is called the ***function header***.

Examples

```
double eval ( double a, double b, double c, double x)   // The header
{   // The body
    // returns value of polynomial a*x^2 + b*x + c
    return a*x*x + b*x + c;
}

double volumeOfSphere ( double radius)                  // The header
{
    // returns the volume of a sphere with given radius
    return 4*3.141592*radius/3;
}

void insertNewLines ( int N )                           // The header
{
    // insert N newlines into cout
    for ( int i = 0; i < N; i++ )
    std::cout << std::endl;
}
```

The first two examples are of functions that return something. This means that when they finish running, the value of the expression in the return statement is the value that they "return". The third example has a void return type. This means it does not return a value. It runs and does something, and it can even have return statements, but they cannot return a value.

Where do you put function definitions?

In the beginning, you should put all function definitions *after* all `#include` and `using` directives but *before* the `main()` program. Once your understanding is solidified, you will put them *after* `main()` but will put a ***function prototype*** before `main()`.

How do you use functions that you define?

You use them the same way that you use library functions, by putting *calls* to them in the code. The program that contains the call is the ***caller***, and we say that it ***calls*** the function. For example the following partly incomplete program calls the first function.

```
int main()
{
    double A, B, C, x;
    // get values A, B, C, and x here
    cout << "The value of the polynomial is " << eval(A, B, C, x)  << endl;
}
```

In the function call, you must make sure to put the arguments in the correct order. Parameters are positional, which means that it is their position in the list that matters, not their names. In the above call, the value of `A` is copied into the parameter `a`, `B` into `b`, `C` into `c`, and `x` into `x`, and then the function executes. When it returns, the value that it returns is used in the caller wherever the call was written. For example, if `A` = 1, `B` = 8, `C` = 16, and `x` = -4, then `eval(A,B,C,x)` returns 0, and it would be the same effect as if the program had the line

```
cout << "The value of the polynomial is " << 0 << endl;
```

If you change the order of the arguments, the result will be different. If you call `eval(C,B,A,x)` you will get a different answer.

Some Guidelines About Writing Functions

- Functions that return values should not have side effects. A side effect is a change to the values of variables in the program other than those that are declared within the function or in its parameter list.
- Functions should do one thing only.
- Functions should do one thing completely.
- Functions should have meaningful names.
- Functions should be grouped together in the program file.
- Functions should always have comments that summarize its purpose, pre-conditions and post-conditions, and what it returns.
- If a function is more than 50 to 100 lines, it is probably too long and should be broken into smaller functions.
- Never write a function that is essentially a main program just to call it from `main()`.

Examples of Functions Not to Write

```
double workerPay( double salary)
{
    cout << "The pay is $";
    return salary;
}
```

This just prints something that could be printed by the caller and it returns what it was passed. It has a side effect (it changes `cout`) and it returns a value too.

```
void mysqrt( double num )
{
    cout << "The square root of " << num << " is " << sqrt(num) << endl;
}
```

This just calls a library function and prints some stuff. No need to do this.

Can functions call other functions?

They certainly can, and often do. In fact `main()` is a function and `main()` calls the functions you write as well as the ones in the libraries. But to give you a better idea, the following is perfectly legitimate, albeit silly, code.

```
void func1 ( int n )
{
    cout << "In func1: " << n << "\n";
}

void func2 ( int m )
{
    cout << "In func2: " << m << "\n";
    func1 (m);
}
```

```
void func3 ( int k )
{
    cout << "In func3: " << k << "\n";
    func2(k);
 }

void func4 ( int m, int n )
{
    func3(m);
    func2(n);
}
```

Can a function call itself?

Yes, but for now we will avoid this topic. It is perilous ground and we will visit it after you can walk, run, and jump on solid ground.

Scope

The variables (and other identifiers) in a program have various properties. You know about some of them already. For example, variables have ***associated type***, and at any given instant of time, they have a ***value***. They also have ***storage requirement***s, i.e., do they need two bytes, four bytes, or something larger? Another property associated with variables and program identifiers in general is their ***scope***.

The ***scope*** of a name is the part of the program in which the name can be used. You already know one ***scope rule***: if you use a for-loop such as this:

```
for ( int i = 0; i < 10; i++) {
    // do something here
}
```

you should know that the name `i` extends only to the end of the for-loop statement itself, i.e., to everything within the body of the loop, and no further. You should also know that this is C++ and not C. The C standard does not let you declare the index within the loop, although some compilers allow it.)

For now, you should know about two types of scope.

Block Scope: A block is the code between a pair of matching curly braces. An identifier declared inside a block has block scope. It is visible from the point at which it is declared until the innermost right curly brace containing the declaration. Function parameters have block scope – they are visible only within the body of the function.

File Scope: An identifier declared outside of any function including main is visible from the point of the declaration to the end of the file in which it was declared.

Local Variables

Functions can have variable declarations. The variables declared within functions have block scope – they are visible until the end of the innermost right curly brace, which is in the simplest case, the function body. For example, in each of the following functions

```
long arithmeticsum( int num)
{
    long sum = 0;
    int  j;

    for ( j = 1; j <= num; j++)
        sum = sum + j;
    return sum;
}

long sumsquares( int num)
{
    long sum = 0;
    int  j;

    for ( j = 1; j <= num; j++)
        sum = sum + j*j;
    return sum;
}
```

both `sum` and `j` have scope that extends to the end of the function block. They are called ***local variables*** and are said to have local scope. Each function has a variable named `sum` (and a variable named `j`). They are different variables that just happen to have the same name. The variables declared in your main program have local scope too. They are visible only from the point at which they are defined until the end of the curly brace that ends the main program block.

Global Variables

Variables declared outside of any function (which therefore have file scope) are called ***global variable***s. They are visible to all functions in the file from the point of their declarations forward. There are many reasons not to use global variables in programs, because it makes program harder to read, understand, debug, and maintain. The one exception to this is constant global variables. It is acceptable to define global constants in a program, if these definitions are placed at the very top of the file after the `#include` and `using` directives. This is because it makes them easy to see and makes changing them easier.

Example

```
// various constants used in the program
const double PI       = 3.14159236;
const int    MAXSIZE = 1000;
const string MESSAGE = "The file could not be opened.\n";

int main()
{
    // stuff here
    return 0;
}
```

Function Prototypes (Declarations): Moving Towards Data Abstraction

If a program has several functions in it and their definitions precede the main program, the main program ends up at the bottom of the file. This is inconvenient, because when you read a program you usually want to first read the main program to get a general sense of how it works. The functions in it are often solving

small problems while the main program provides structure. The C and C++ languages provide the means to declare the functions in the beginning and put the definitions of them elsewhere.

What do we mean by declaring and defining functions, and how are they different?

A function definition is what you just learned how to write. It is the function's header together with its block. The function header by itself does not say what the function does, but it provides enough information to the compiler so that it can check whether the calls to the function are valid. The function header, when it is terminated by a semicolon, is called a function declaration or function prototype.

How does having the function prototype help?

This is the nice part. We put all of the function prototypes at the beginning of the file, before main(), and the definitions after main(). The comments that describe what the function does stay with the prototype at the beginning of the file. They do not need to be repeated with the function definitions. The comments and the prototype ar all that a programmer needs to figure out how to call the function and what it does. The actual function body should be thought of as a black box, a hidden piece of code that does what the comments say it does, but which the programmer does not need to see to use the function. This is the essence of procedural abstraction, an important principle of good software engineering practice, and the underpinning of object-oriented programming.

Example

```
#include <string>
#include <iostream>
using namespace std;

/*************************************************************************/
/*                          Function Prototypes                          */
/*************************************************************************/

/* eval(a,b,c,x) returns value of polynomial a*x^2 + b*x + c      */
double  eval           ( double a, double b, double c, double x);

/* volumeOfSphere(r) returns the volume of a sphere with radius r */
double  volumeOfSphere  ( double radius);

/* insertNewLines(N)  inserts N newline characters into cout      */
void    insertNewLines   ( int N );


/*************************************************************************/
/*                             Main Program                              */
/*************************************************************************/

int main()
{
    // tbe main program's body is here
}

/*************************************************************************/
/*                          Function Definition                          */
```

```
/*************************************************************************/

double eval ( double a, double b, double c, double x)
{
    return a*x*x + b*x + c;
}

double volumeOfSphere ( double radius)
{
    return 4*3.141592*radius/3;
}

void insertNewLines ( int N )
{
    for ( int i = 0; i < N; i++ )
    std::cout << std::endl;
}
```

The next step after this is to put the function declarations into one file, called a ***header file***, and the function definitions into a second file, called the ***implementation file***. This will be discussed in a future lesson.

Call-by-Reference Parameters

What if we want to write a function that actually changes the value of its parameters? For example, suppose we want a function that can be called like this:

```
swap(x,y)
```

that will swap the values of arguments x and y. In other words if `x = 10` and `y = 20` before the call, then after the call `x = 10` and `y = 10`. If we write the function like this:

```
void swap ( int x, int y )
{
    int temp = x;
    x = y;
    y = temp;
}
```

will this do the trick?

Try it and you will see that it does not. Remember that when a function is called, ***the values of the arguments are copied into the storage cells of the parameters***. The function runs and when it terminates, the parameter storage cell contents are not copied back to the arguments. When you think about it, it would make no sense. Suppose we define a function `double()` like this:

```
int double ( int x )
{

    x = 2*x;
    return x;
}
```

and we call it with the call

```
cout << double(10);
```

If the value of parameter x were copied back to its argument, it would mean we could replace a constant literal 10 by the value 20, which is impossible.

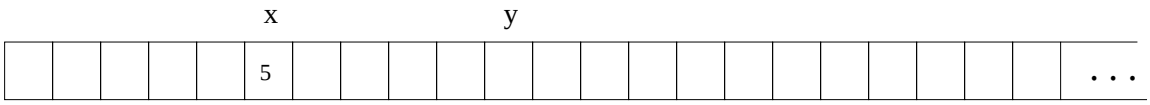
The kind of parameters we have been using so far are known as ***call-by-value parameters***. This is because the value of the argument is passed to them. So what is the alternative?

The Concept Of A Reference

We mentioned before that variables have several different properties, such as their type, their storage requirements, and their scope. Every variable also has ***contents*** and ***location***. In

```
int x = 5;
int y;
```

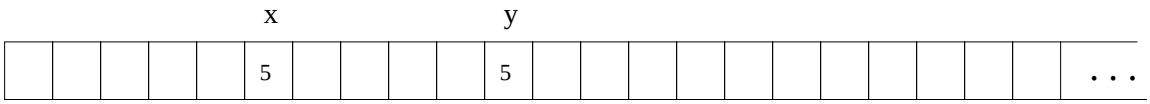
`x` is the name of a variable of type `int` with contents of `5`. `x` also has a location. We don't know its actual location exactly but we know that it has some storage location in memory and that location has a specific, numbered address. In the picture below, think of each box as a storage location capable of storing an integer:



The boxes, which are actually ***memory words*** (4-byte units), have addresses but we don't care what they are. The assignment statement

```
y = x;
```

causes the ***contents*** of `x` to be copied into the ***location*** of `y`[1]:



A reference variable is like another name for a location that already exists. A reference variable really stores an address. If we define two variables x and y as follows:

```
int  x = 5;
int& y = x;
```

then `y` is called a reference to `x`. The variable `y` stores the location, or address, of `x`, but it can be used in place of `x`. The two statements

```
cout << x;
cout << y;
```

[1] Notice the asymmetry of the = operator: the value of its right hand side operand is the ***contents of the operand***, but the value of the left hand side operand is the ***location of the operand***. In other words, putting a variable on the right hand side of "=" causes its value to be extracted whereas putting it on the left hand side causes its location to be found.

The ***lvalue*** of a variable is its location. The ***rvalue*** is its contents. lvalue and rvalue are just abstractions; they are not really stored anywhere, but they help to make things clearer.

print the same value because `y` is just a pseudonym, another name, for `x`. If we increment `x` and output the value of `y`, we will see it has changed as well:

```
x++;
cout << y;
```

and we can increment `y` and output the value of `x` and see that it has also changed.

```
y++;
cout << x;
```

The &-operator used in this way creates a ***reference variable***:

```
type & identifier = variable;
```

makes the identifier a reference to the variable. The reference variable type must be the same as type of the variable whose address is being assigned to it.

```
char c;
char & refTochar = c;
int x;
int &  refToint  = x;
```

are two valid reference declarations, but not this:

```
char c;
int & cref = c;
```

because `cref` is of type `int&` and it should be of type `char&`. It does not matter whether there is space to the left or right of the &-operator. The following three statements are equivalent:

```
int&  y = x;
int & y = x;
int  &y = x;
```

We can use reference variables as parameters of functions. They are then known as ***call-by-reference parameters***.

Examples

```
void swap( int & x, int & y)
// replaces x by y and y by x
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
```

```
    int a = 10;
    int b = 20;
    swap(a,b);
    cout << a << " " << b << endl;
    return 0;
}
```

This program will print 20 and then 10 because `swap(a,b)` caused the values of `a` and `b` to be swapped. This is because `x` is just another name for `a` in `swap()` and `y` is another name for `b`. Each assignment statement in `swap()` is actually altering the values of `a` and/or `b`.

```
void castActors( string & Romeo,
                 string & Juliet)
{
    Romeo = "Leonardo DiCaprio";
    Juliet = "Claire Danes";
}

int main()
{
    string lead_male_role;
    string lead_female_role;

    castActors( lead_male_role, lead_female_role );
    cout << "Romeo will be played by  " << lead_male_role << "\n";
    cout << "Juliet will be played by " << lead_female_role << "\n";
    return 0;
}
```

In this second example, the strings in the main program have no initial value but after the call to `castActors()`, they are given values. Call-by-reference parameters are the key to writing functions that must give values to multiple variables, such as functions that initialize many variables.

Overloading Functions

Functions can be overloaded in C++, but not in C. Simply put, it means that the same name can be used for two different functions, provided that the compiler can distinguish which function is being called when it tries to compile the code. This can be convenient sometimes, but most of the time there is little need for this feature of C++. Nonetheless, because you may be called upon to read a program that contains overloaded functions, I discuss them very briefly here.

The rule sounds simple at first: The same name can be used for two different functions provided that they have a different number of formal parameters, or they have one or more formal parameters of different types. So these are valid overloads:

```
int max( int a, int b );
int max( int x, int y, int z );
```

and so are these:

```
void sort( int & x, int & y );
void sort( double & x, double & y );
```

because the first pair have a different number of parameters and the second pair have different types for their parameters.

These are not valid overloads:

```
int max( int a, int b );
long max( int x, int y );
```

because they only differ in their return type. (The names of the parameters are irrelevant.) These are not valid either:

```
void sort( int & x, int & y );
void sort( int x, int y );
```

because the types are the same and kind of parameter passing is not used to distinguish them.

Overloading can get complicated because of type casting and type conversion. If you have two functions such as

```
void foo( int x, double y);
void foo( double x, int y);
```

and your program makes the call

```
foo(1,2);
```

which one should the compiler use? Guess what? It can't really decide either, so it generates an error. Unless you have a good reason to overload function names, it is best to avoid it.

Default Arguments

C++, and not C, lets you assign default values to the ***call-by-value parameters*** of a function, in right to left order. In other words, you can declare a function so that the rightmost parameters have default values, as in the following example:

```
string repeatedstr( string str, int numcopies = 1);
```

The calling program can omit the second argument, in which case `numcopies` will be assigned 1:

```
cout << repeatedstr("*");
```

will print a single '*' whereas

```
cout << repeatedstr("*", 10);
```

will print ten of them.

Note: You can not assign default values to call-by-reference parameters.

You can have any number of default arguments but the rule is that a parameter can only have a default argument if the parameter to its right has one. So these are valid:

```
void foo( int a, int b, int c = 1, int d = 2, int e = 3);
void bar(int a, int b, int c = 1);
```

but not these

```
void foo( int a = 1, int b);
void bar(int a = 1, int b = 2, int c);
```

Moreover, you can run into problems with overloading and default arguments, as in the following:

```
void f( int x, int y, int z = 1);
void f( int x, int y);
```

If the program calls `f(1,2)`, is it the first or the second function that will be invoked? Guess what? The compiler can't know either, so it generates an error.

The C++ `iostream` library has two prototypes for `getline()`:

```
istream& getline ( istream& is, string& str, char delim );
istream& getline ( istream& is, string& str );
```

Do you think it is overloaded, or do you think that `getline()` is really defined with the header

```
istream& getline ( istream& is, string& str, char delim = '\n' );
```

What do you think is the better solution?

The real benefit of being able to assign default values to parameters comes with class constructors, a topic that will be covered when we get to classes. My advice is that you avoid overloading unless you have a really good reason to use it.

Arrays

Motivation

Suppose that we want a program that can read in a list of numbers and sort that list, or find the largest value in that list. To be concrete about it, suppose we have 15 numbers to read in from a file and sort into ascending order. We could declare 15 variables to store the numbers, but then how could we use a loop to compare the variables to each other? The answer is that we cannot.

The problem is that we would like to have variables with subscripts or something like them, so that we could write something like

```
max = 0;
for ( i = 0; i < 15 ; i++ ) {
    if ( number_i > max )
        max = number_i ;
}
```

where somehow the variable referred to by `number_i` would change as `i` changed.

You have seen something like this already with C++ strings: if we declare

```
string str;
```

then we can write a loop like

```
for ( int i = 0; i < str.size(); i++ )
    cout << str[i] << "-" ;
```

in which each individual character in `str` is accessed using the subscript operator `[]`. The characters in a string form what we call an ***array***. An array is conceptually a linear collection of elements, indexed by subscripts, all of the same type. If we could create an array named `number` with 15 elements, it would look like this:

Each element could be accessed using the subscript operator, as in `number[1]` or `number[7]`, and we could write a loop like

```
max = 0;
for ( i = 0; i < 15 ; i++ ) {
    if ( number_i > max )
        max = number[i] ;
}
```

This would make it possible to manipulate large collections of ***homogeneous*** data, meaning data of the same type, with a single subscripted variable. Such is possible in C and C++ and all modern programming languages. ***Arrays are of fundamental importance to algorithms and computer science.***

Declaring a (one-Dimensional) Array

Syntax:

```
elementtype arrayname [ size_expression ]
```

where

- *elementtype* is any type that already exists
- *arrayname* is the name of the array
- *size_expression* is the number of elements in the array

This declares an array named `arrayname` whose elements will be of type *elementtype*, and that has *size_expression* many elements.

Examples

```
char    fname[24];            // an array named fname with 24 chars
int     grade[35];            // an array named grade with 35 ints
int     pixel[1024*768];      // an array named pixel with 1024*768 ints

const int MAX_STUDENTS = 100;
double average[MAX_STUDENTS]; // an array named average with 100 doubles

string fruit[5];              // an array of 5 C++ strings
```

The element type of an array is often called its ***base type***. The first example is an array with base type `char`, for example. One can say that `fname` is "an array of `char`."

Things to remember about arrays:

- The starting index of an array is 0, not 1.
- The last index is one less than the size of the array.
- If the array has `size` elements, the range is 0..`size`-1.
- Arrays contain data of a single type.
- An array is a sequence of consecutive elements in memory and the start of the array is the address of its first element.
- Because the starting address of the array in memory is the address of its first element, and all elements are the same size and type, the compiler can calculate the locations of the remaining elements. If B is the starting address of the array `array`, and each element is 4 bytes long, the elements are at addresses B, $B+4$, $B+8$, $B+12$, and so on, and in general, element `array[k]` is at address $B+12k$.
- Although C and C++ allow the size expression to be variable, you should not use a variable, for reasons having to do with concepts of dynamic allocation and the lifetime of variables.
- Use constants such as macro definitions or `const int`s as the sizes of arrays.

Initializing Declarations

Arrays can be initialized at declaration time in two different ways.

```
elementtype arrayname[size expr] = { list with <= sizeexpr vals };
elementtype arrayname[ ]         = { list with any number of values };
```

Examples

```
#define MAXSCORES 200
#define NUMFRUIT    5
const int SIZE = 100;

double  numbers[SIZE];    // not initialized
string  fruit[NUMFRUIT]   = {"apple","pear","peach","lemon","mango"};
int     power[ ]          = {0,1,2,4,9,16,25,36,49,64,81,100};
int     counts[SIZE]      = {0};
int     score[MAXSCORES]  = {1,1,1};
```

The first declaration declares but does not initialize the array named `numbers`. The next declares and initializes an array named `fruit` with five strings:

apple	pear	peach	lemon	mango
0	1	2	3	4

The third initializes an array named `power`, whose size is determined by the number of values in the brace-delimited list. When the array size is given in square brackets but the number of values in the list is less than the size, the remainder of the array is initialized to 0. In the fourth example, all elements will be set to 0, and in the last, the first three are set to 1 and the rest, to 0.

Rules

- If the array size is given in brackets, then the initialized list must have at most that many values in it. If it has fewer, the rest of the array is initialized to 0's.
- If the size is not given in brackets, then the size is equal to the number of elements in the initializer list.
- If there is no initializer list, none of the array elements are initialized. They are not set to 0.

Advice

- Always named constants for the sizes of arrays. It makes it easier to write, maintain, and read the code.

Accessing Array Elements

An element of an array is accessed by using the ***subscript operator***. The syntax is:

```
arrayname [ integer-valued-expression-within-range ]
```

Examples

```
cout << fruit[0] ;                      // prints apple
cout << powers[1] << " " << powers[2] ; // prints 1 2
fruit[3] = "apricot";                   // replaces "peach" by "apricot in fruit[3]
counts[SIZE-10] = counts[SIZE-11] + 2;  // sets counts[90] to counts[89] + 2 = 2
cout << score[power[4]];                // power[4] = 9, so this outputs scores[9]
```

Notice in the last two examples that the index expression can be arbitrary integer-valued expressions, even the value of an array of integers.

Loops and Arrays

Without loops it is very hard to use arrays. Conversely, with them they are easy to use. In general, a loop can be used to initialize an array, to modify all elements, to access all elements, or to search for elements within the array. An example follows.

Example 1. Finding a minimum element.

This example shows how an array can be initialized with values from an input stream using a for-loop. The for-loop guarantees that the array is not "overfilled" because it runs only as many times as the array has elements. The example uses the preceding declarations.

```
// read values into array from standard input
for ( i = 0; i < MAXSCORES; i++ )
    cin >> score[i];

// Let minscore be the first element as a starting guess
int minscore = score[0];

// Compare remaining array alements to current maxscore
for ( i = 1; i < MAXSCORES; i++ ) {
    if ( minscore > score[i] )  // if current element < minscore
        minscore = score[i];    // make it new minscore
    // At this point, maxscore >= score[j], for all j = 0,1,2,...i
}

cout << The minimum score is  << minscore << endl;
```

In this example, the score array is filled from values entered on the standard input stream, `cin`. After the entire array has been filled, a variable named `minscore` is set to the value of `score[0]`. Then a second loop examines each element of score from `score[1]`, to `score[2]`, and so on up to `score[MAXSCORE-1]`. If an element is smaller than `minscore`, its value is copied into `minscore`. This implies that in each iteration of the second loop, `minscore` is the smallest of the elements seen so far. This is how one searches for the minimum value in an unsorted array.

Danger

If your program tries to access an element outside of the range of the array, bad things can happen. Typically the program will be aborted by the opeating system with an error message. As an example:

```
int score[10];
for ( int i = 0; i < 20; i++ )
    cin >> score[i];
```

When `i` eventually has the value 10, `score[i]` is a memory address outside of the `score` array (since it is 0-based and the last index is 9) and the program will be aborted.

Partially Filled Arrays

Although an array might be declared to have so many elements, that does not always mean that all of the cells in the array actually have meangingful data. Sometimes a program does not know how much data it will read in from a file or external source, so it declares an array much larger than it needs. Then after it reads the data, only part of the array has been filled. If a loop that is processing the array tries to process all array elements, errors will result.

This implies that when an array is filled and the number of data items is not known in advance, then the code needs to count how many values are written into the array, and stop if it exceeds the array size, or if it runs out of data first. The while loop has two conditions to check in this case:

```
int list[100];
int temp;
length = 0;

cin >> temp;  // try to read to force eof() to become true if the file is empty
while ( length < 100 && !cin.eof() ) {
    list[length] = temp;  // temp has a value, so copy it into the list
    length++;             // increment because we copied a value into list
    cin >> temp;          // read a new value; might make eof() true
}
```

The variable `length` keeps a count of what has been put into the array and when the loop ends, it is guaranteed to be at most 100. Either the loop ends because `length` became 100 or because the data stream ended. In either case, the array has been filled with `length` many values. Therefore, all future processing of this array must iterate only from 0 to `length` -1. The following code would find the minimum element of `list`:

```
for ( i = 1; i < length; i++ ) {
    if ( min > list[i] )
        min = list[i];
 }
```

Arrays in Functions

Suppose that you want to write a function that can find the minimum element of any array it is given. Can we write such a function? Let us rephrase a simpler question for now. Can we write a function that can find the minimum integer in an array of integers? We can safely answer yes to this question, whether it is C or C++. The heart of the matter is how we can pass an entire array into a function.

How can you pass an entire array into a function?

An array parameter in a function is written as a name followed by empty square brackets:

```
result_type function_name ( ..., elementtype arrayname [ ] ,... ) ;
```

For example,

```
int max( int array[] );
void foo( int a[], int b[], double c[]);
```

The size of the array is not put between the square brackets. When you declare a function with an array parameter, you just put the type and a name for the parameter followed by a pair of empty square brackets. This tells the compiler that the corresponding argument will be an array of that type.

How does the function know how many elements are in the array?

It doesn't. Arrays do not "know" their size. It is not stored anywhere. The function acts on any array of any size whose type matches the base type.

So how can you write a function that works with an array parameter?

You must always pass the size of the array as an extra parameter. For example, if we want to write a function that finds the minimum in an integer array, the function prototype would be

```
int min( int array[], int size );
```

We would call the function by passing just the array name to it. not the name with square brackets after it. For example

```
int score[MAXSCORES];
for ( i = 0; i < MAXSCORES; i++ )
    cin >> score[i];
cout << "The minimum score is " << min(score, MAXSCORES);
```

Notice that the `score` array is passed to the `min()` function just by writing its name. The `min()` function definition would be

```
int min( int array[], int size )
{
    int minvalue = array[0];

    // Compare remaining array alements to current minvalue
    for ( int i = 1; i < size; i++ ) {
        if ( minvalue > array[i] )
            minvalue = array[i];
        //  minvalue <= array[j], for all j <= i
    }
    return minvalue;
}
```

When the function is called as in `cout << max(score, MAXSCORES)`, the `score` array is passed to the array parameter `array`, and within the function `array[i]` is in fact a ***reference*** to `score[i]`, ***not a copy of it***, but a reference to it, i.e., another name for it. This means that changes made to `array[i]` within the function are actually made to `score[i]` outside of the function. The following example will make this clearer.

```
void initSquares( int array[], int size )
{
    for ( int i = 0; i < size; i++ ) {
        array[i] = i*i;
    }
}
```

The function `initSquares()` puts into each element `array[i]` the value `i*i`. The remarkable thing about array parameters is that each element of the array parameter acts like a reference to the corresponding element of its argument, so the output of this code snippet

```
int squares[20];

initSquares( squares, 20);
for ( int i = 1; i < 20; i++ )
    cout << squares[i] << endl;
```

will be the squares 0, 1, 4, 9, ... 400, because the changes made to the array parameter `array[]` are actually changes made to the calling program's `squares` array. To emphasize this

There are three types of parameter passing:

- call by value parameters
- call by reference parameters
- array parameters

Array parameters are like call by reference parameters – changes made to the array are changes made to the corresponding array argument.

Searching and Sorting

Two of the most common types of processing of array data are searching for specific values and sorting the elements.

Searching

The problem of searching for a particular value in an array or in a list of anything is called the ***search problem***. The value being sought is called the ***search key***, or just the ***key*** when the meaning is clear.

Suppose, for example, that want to know whether anyone had a score of 75, assuming that we already read a list of scores into our array `scores`. We can search through our array of scores for the number 75 as follows:

```
cout << "Enter the number to look for: ";
cin >> searchkey;

bool found = false;  // flag to control loop
int  i     = 0;      // index to go through array

while ( !found &&  (i < length) ) {
    if ( searchkey ==  score[i] )  // found it!
        found = true;
    else
        i++;   // didn't find it yet, so advance to next element
}

// If found == false then we did not find the item
if ( found )
    cout << searchkey <<  " is in index " << i << endl;
```

There are some very important concepts here.

- We do not want to use a for-loop because if we find what we are looking for, we want to stop searching. So we use a while-loop that is re-entered if we have not found what we are looking for *and* our index variable is still within the range of the array. Therefore we set `found` to `false` initially and set it to `true` if we find a matching value in the array.
- If we do find an element in the array whose value equals `searchkey`, then we set `found` to `true`. This causes the loop to be exited. Because `i` is declared outside of the loop body, it still has a valid value after the loop exits. It was not incremented since `found` became `true`, so it is the index at which `score[i] == searchkey`.
- If we search through the entire array and do not find `searchkey`, then `found` is `false` when the loop exits, because the condition `(i < length)` became false. By checking the value of `found` after the loop, we can tell whether it was found or not.

This is the first example of a non-trivial algorithm. We can state its requirement more generally as,

> "Given an array A of N values, and a searchkey X, find the position in A in which X occurs or report that it is not there."

The preceding solution searched through the array one element at a time, starting with the first and ending with the last, until either it found it or it examined every element.

This is the best we can do when searching through a list of values that is not in sorted order. We have to look at every value in the array until we find the key. If there are N items in the array, we need to check compare our search key to N values in the absolute worst case that either it is not there or it is in the last position.

If however, the array is sorted, then it is possible to search in a much faster way. In general it is better to sort the values in the array for faster searching later. Presumably we will search the data very often, so if sorting it costs us a bit of time, it is worth the savings we get from faster searching later.

Sorting

There are many different ways to sort an array, some much faster than others. Some are easy to understand and some are harder to understand. Some require additional memory to sort, others can be done "in place", meaning by rearranging the elements of the array without using extra memory. The sorting problem can be stated as follows,

> "Given an array A of size N, rearrange the elements of A so that A[0] <= A[1] <= A[2] <= ... <= A[N-1]."

Notice that the elements are rearranged. In other words, sorting is a permutation, a rearranegment of an array.

We will start with a very intuitive sorting algorithm that is called selection sort. You will see why it is called selection sort very soon. The idea is simple:

> Find the minimum element of the array from 0 to N-1. Suppose it is in index K.
>
> Swap A[0] and A[K]. Now A[0] has the smallest element.
>
> Find the minimum element of the array from 1 to N-1. Suppose it is in index K.
>
> Swap A[1] and A[K]. Now A[1] has the second smallest element.
>
> Find the minimum element of the array from 2 to N-1. Suppose it is in index K.
>
> Swap A[2] and A[K]. Now A[2] has the third smallest element.
>
> Repeat this procedure until there are two elements remaining: A[N-2] and A[N-1}. Find the smaller of the two and swap it into A[N-2].

When the above procedure has finished, the array is in sorted order, smallest to largest. The procedure can be stated more succinctly and abstractly as follows:

```
let bottom = 0, 1, 2, ... N-2 {
    let k = index_of_minimum(A, bottom, N-1)
    swap(A[k], A[bottom)
}
```

This pseudocode translates directly into the following C++ function. Because you can sort any types of values that can be compared using ordinary comparison operators, rather than writing it with a fixed type, we use the type `T`, and a `typedef` that can be changed easily that here makes `T` another name for the `string` type.

```
typedef string T;

// Returns the index of the smallest element in A between
// A[bottom] and A[maxIndex]
int indexofMin(const T A[], int bottom, int maxIndex);

// Swaps the values stored in x and y
void swap( T & x, T & y )


// Precondition:  A[0..N-1] contains data of type T, to be sorted
// Postcondition: A[0..N-1] is sorted in ascending order.
void selectionSort ( T A[ ], int N)
{
    int smallest; // index of smallest item in unsorted part of A
    int first;    // index of first item in unsorted part of A

    first = 0;  // start with last at highest index in A
    while (  first < N-1) {
        smallest = indexofMin(A, first , N-1);
        swap( A[smallest], A[first]);
        first++;
    }
}

int indexofMin(const T A[], int bottom, int maxIndex)
{
    int m = bottom;
    for (int k = bottom+1; k <= maxIndex; k++)
        if ( A[k] < A[m])
            m = k;
    return m;
}

void swap( T & x, T & y );
{
    T temp = x;
    x = y;
    y = temp;
}
```

Notes

- This algorithm can be written symmetrically using maximums instead of minimums. That is a good exercise to try.
- If the call to `swap()` were replaced by the three instructions that do the swap, this would compile with a C compiler.
- This always takes the exact same amount of time, even if the array is already sorted. The call to `indexofMin()` always iterates its loop (`N-first`) times and the swap is always performed, even if the two indices are the same. It will swap a value with itself in this case. It is an example of an ***unnatural sort***. A sorting algorithm is ***natural*** if arrays that are already in order require less swapping than arrays that are very out of order.

Once an array has been sorted, searching it is much faster. Later we will see a method of searching called binary search, in which the portion of the array being searched is cut roughly in half in each iteration.

Multidimensional Arrays

The arrays we have studied so far have a single dimension – they represent data items that can be arranged in a sequence, and therefore they are also called ***linear arrays***. Arrays can have more than one dimension in C and C++.

Declaring a Multidimensional Array

You declare an array with multiple dimensions using a syntax that is similar to one-dimensional arrays.

Syntax:

```
elementtype arrayname [ size1 ] [ size2 ] ... [ sizeD ];
```

where

- *elementtype* is any type that already exists
- *arrayname* is the name of the array
- *size1* is the number of elements in the first dimension, size2, the number in the second dimension, and so on, and D is the number of dimensions in the array.

Basically you tack on a pair of square brackets with a number in between for every dimension that you want the array to have.

Examples

```
char    tictactoe [3][3];  // a 3 by 3 grid of chars
string  chessboard[8][8];  // an 8 by 8 grid of strings
int     pixel[1024][768];  // a 1024 by 768 grid of ints
double  cube  [5][5][5];   // a 5 by 5 by 5 cubic grid of doubles
```

In two dimensions, we think of the first dimension as the row, the next as the column. There is no rule for this and there is nothing in the language that associates the dimensions with rows or columns. It depends how you program it. In a three dimensional array, the first subscript would be like the row, the second like the column, and the third, like the plane in which this row and column are located, i.e., the height. People usually refer to the first dimension as the row, the second as the column, and the third, if it exists, as the plane.

Accessing Elements of Multidimensional Arrays

Higher dimensional arrays are accessed just like one dimensional arrays. The subscripts match their dimension:

```
tictactoe[0][2] = 'x';          // puts an 'x' in first row, last column.
chessboard[7][7] = "redqueen"; // puts the string "redqueen" into the lower
                                // right hand corner of the chessboard.
```

The following example shows how we could "color" the elements of a checkerboard. It demonstrates accessing two-dimensional array accesses.

```
const char RED = 'r';
const char BLACK = 'b';
char   checkerboard[ROWS][COLS];
int i, j;

for ( i = 0; i < 8; i++ )
    for ( j = 0; j < 8; j++ )
        if ( ( i + j ) % 2 == 0 )
            checkerboard[i][j] =  RED;
        else
            checkerboard[i][j] =  BLACK;

for ( i = 0; i < 8; i++ ) {
    for ( j = 0; j < 8; j++ )
        cout << checkerboard[i][j] << ' ';
    cout << "\n";
}
```

Processing Higher Dimensional Arrays

The most commonly used higher dimensional arrays are two-dimensional. They can be used to represent a wide assortment of objects, such as graphical images, mathematical matrices, screen representations, two-dimensional surfaces in general, mazes and other 2D cellular arrangements (chess boards, game boards). Just as one dimensional arrays go hand in hand with for-loops, two-dimensional arrays are naturally paired with nested for-loops.

The following example shows a code snippet that creates a random grid of asterisks. Notice that the middle loop is not a nested loop. Sometimes two-dimensional data can be processed within a single loop.

```
const int   COLS = 100;
const int   ROWS = 80;
const int   MAXSTARS = COLS*ROWS/10;
char   grid[ROWS][COLS];
int i, j, count = 0;

srand(time(0));
```

```
// Initialize the grid to all blanks
for ( i = 0; i < ROWS; i++ )
    for ( j = 0; j < COLS; j++ )
        grid[i][j] = ' ';

while ( count < MAXSTARS ) {
    i = random(ROWS);   // generate a random row index
    j = random(COLS);   // generate a random column index
    grid[i][j] = '*';   // put an asterisk there
    count++;
}

// Print out the grid
for ( i = 0; i < ROWS; i++ ) {
    for ( j = 0; j < COLS; j++ )
        cout << grid[i][j];
    cout << "\n";
}
```

Multidimensional Array Parameters[1]

To understand how to use multidimensional array parameters, you need to remember how one-dimensional array parameters are used. When you write a function declaration such as

```
void process( double data[], int size);
```

you are telling the compiler that the function named `process` will be given an array of basetype `double` as the first argument and an integer as the second argument. The array parameter `data[]` is just the starting address of the array that is passed to it. The compiler knows that it is of type `double`, so it knows where the remaining array elements are loacted in memory, because it knows how many bytes are in the double type. In other words, the array parameter tells the compiler where the start of the array is and how many bytes are in each entry, and this is all that the compiler needs.

This same logic applies to multidimensional arrays. The compiler needs to know the size of each element and the starting address. This means that the array parameter must be written with the size of every dimension except the leftmost one, as in

```
void processdata( int data[][10][20], int size);
```

which declares that processdata is a function with a 3D array parameter whose second dimension has 10 elements, and third has 20 elements. The compiler thus knows that `data[1]` starts 200 integers after `data[0]` and `data[2]` starts 200 integers after `data[1]`, and so on.

[1] This material will not be covered in class.

About Compiling

What most people mean by the phrase "*compiling a program*" is actually two separate steps in the creation of that program. The first step is proper ***compilation***. Compilation is translating high level programming instructions into machine language instructions. The input to compilation is a source code file in a high level language such as C or C++. Source code files have extensions such as "`.c`", "`.cpp`", or "`.cc`". The output of compilation is an ***object file***, which is not quite an executable file. Object files usually have a "`.o`" or "`.obj`" extension.

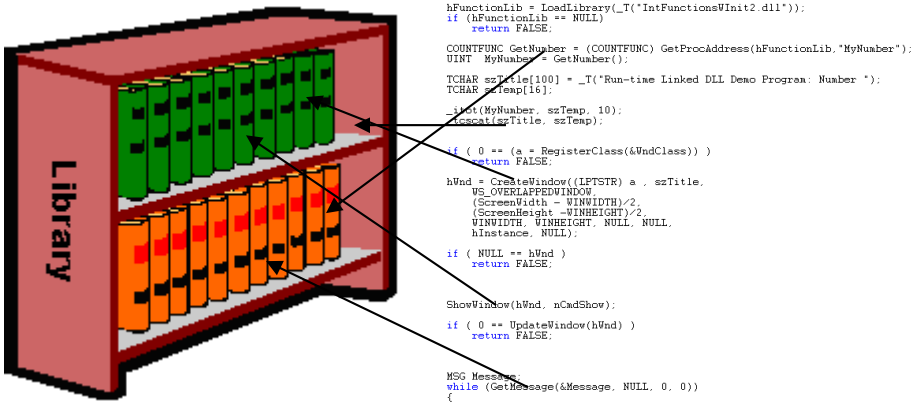
Consider the C++ code fragment

```
#include <iostream>
#include <math.h>
using namespace std;
double  number;
cout << "Enter a positive number here:";
cin >> number;
cout << "The square root is " << sqrt(number) << endl;
```

The first two lines tell the compiler to copy the contents of the header files `iostream` and `<math.h>` into the program at those points in the file. The third line tells the compiler to use the `std` namespace for resolving symbols, since the `iostream` header file just copied into the program is declared within this `std` namespace. In other words, every declaration and definition in this header file is contained within the namespace known as `std`. A namespace is essentially just a scope, so all of `iostream` has scope `std`.

In particular, `cout` and `cin` are really known to the world outside of the `std` scope by their proper names, `std::cout` and `std::cin`. If you wrote `std::cout` instead of `cout`, `std::cin` instead of `cin`, and `std::endl` instead of `endl`. you could eliminate the third line. The "`using namespace std`" directive tells the compiler that whenever it finds a symbol in the program that is not defined in the program, it should search the `std` namespace in case it is defined there. Names like `cin` and `cout` are called ***external symbols*** in a program because their definitions are not contained in the program itself.

The inclusion of the header files `<iostream>` and `<math.h>` in the program allows the compiler to determine whether the names `cin`, `cout`, and `sqrt` are being used properly, thereby allowing it to compile the code, but it cannot create an executable module, because the objects associated with the names `cin` and `cout` are not defined in your program. Because the name `cin` is defined in a separate file, the compiler cannot create a jump instruction to jump to the code that does stream extraction, i.e., "`cin >>`", because there is no memory address associated with this code. Names like `cin` and `cout` that are defined outside of the program module are said to be ***unresolved*** at compile time.



The most that the compiler can do is to create an entry in a table in the code that allows the second stage to solve the problem. This table contains the location of every instruction in the program that refers to a name whose location is unresolved, or external, to the program. The second stage is ***linking***, and it is performed by, not surprisingly, the ***linker***. The linker's job is to find the unresolved names listed in the table in the executable module and to ***link*** them to the actual objects to which they refer. To link a name means to replace it with the address to which it refers. Of course a name cannot be associated with an address unless the object that it names actually has an address, which implies that before the name can be resolved, the associated object must be incorporated into the address space of the executable file. There is a special type of linking called ***dynamic linking*** that is an exception to this rule, but how that works is a subject for a different chapter. Static linking is the type of linking in which all code needed at runtime is actually copied into the program

What is Separate Compilation?

Projects should be organized into collections of small files that can be compiled individually. Typically, large classes are given their own files and smaller classes may be grouped together into a single file. Sometimes collections of functions that are not members of any class are placed into a separate file. As long as each of the files is included in the project file, the compiler will usually compile each of them when it is given the instruction to compile the project.

The usu

A set of functions that all provide various forms of randomization, for example, would be placed into two files named `myrandstuff.h` and `myrandstuff.cpp`. The `myrandstuff.h` file contains the function prototypes.. Files that end in a "`.h`" are called ***header files***. They are usually not compiled by the compiler. Instead they are included into the implementation files at compile time so that the compiler will have access to the symbols defined in the header file. The `myrandstuff.cpp` file contains the definitions of the functions defined in the header file. The files will usually have the following form.

myrandstuff.h:

```
#ifndef MYRANDSTUFF_H
#define MYRANDSTUFF_H

// whatever header files need to be included go here
// any typedefs or other type or class definitions go here

// description of func1
void func1( ... );

// description of func2
void func2( ... );

// and so on

#endif  // MYRANDSTUFF_H
```

myrandstuff.cpp:

```
#include "myrandstuff.h"

void func1( ... )
{
   // implementation for func1
}

void func2( ... )
{
   // implementation for func2
}

// and all implementation code for remaining functions here
```

The main program only includes the header files, not the `.cpp` files, so that it can make reference to functions declared there and used in .. Therefore, the main program will contain a line of the form

```
#include "myrandstuff.h"
```

among the other header files included by it. Notice that the header file name is enclosed in double quotes, not angle braces. When the file to be included is in the same directory as the program, its name should be in double quotes.

What, you may be wondering, is the purpose of those lines at the beginning and end of the file that look like preprocessor directives?

When the macro preprocessor sees the `#include` directive, it finds the file that is to be included and copies it into the program at the point at which the `#include` directive was found. Every included file is copied into the main program.

Suppose that you have a second implementation file, say `mylist.cpp`, that uses the functions declared or the types defined in `myrandstuff.h`. It is possible that the header file `mylist.h` needs to include `myrandstuff.h`. Supoose also that the main program uses the functions declared in `mylist.h`. Then `mylist.h` has the line

```
#include "myrandstuff.h"
```

and the main program has the two lines

```
#include "myrandstuff.h"
#include "mylist.h"
```

When the compiler runs, it copies the `myrandstuff.h` file twice, one after the other, first because of the first `#include` directive, then again because when it includes `mylist.h`, it will copy `myrandstuff.h` again because of the `#include` directive in that file. Now we can explain the reason for the lines

```
#ifndef MYRANDSTUFF_H
#define MYRANDSTUFF_H

#endif // MYRANDSTUFF_H
```

The first line translates to "if the macro symbol `MYRANDSTUFF_H` is not defined then continue reading and copying lines into the file until the matching occurrence of `endif`. On the other hand, if it is defined, then skip reading code until immediately after that matching `endif`". The second line is therefore only read if the symbol was not defined. In this case that line defines it, and the code is read and copied. When the preprocessor opens the next file, `mylist.h`, the symbol `MYRANDSTUFF_H` is defined already and the code from `myrandstuff.h` is not copied twice into the `main()` function. Thus, these three lines prevent the macro preprocessor from putting multiple copies of code into other compilation units. These lines are often called a ***header guard***.

You can use whatever symbol you want in this directive, but it must be unique in your project. It is best to follow a convention that ensures this uniqueness. The most common is to use the symbol consisting of the file name. in caps, with an underscore between the root and the extension. Some people use a leading underscore also.

Compiling and Linking the Program

When a program consists of a collection of files, some of which are header files and their corresponding implementation files, and of course a single file containing the main() function, it is compiled and linked in a specific way.

Assume the project contains the files `f1.h`, `f2.h`, `f3.h`, `f1.cpp`, `f2.cpp`, `f3.cpp`, and `main.cpp` and that `main.cpp` includes all header files, but that the remaining `.cpp` files include only their own header files.

The set of steps that must be taken if this is to be done manually, using `g++`, for example, is

```
g++ -c f1.cpp
```

```
g++ -c f2.cpp
g++ -c f3.cpp
g++ -c main.cpp
```

This creates the object files `f1.o`, `f2.o`, `f3.o`, and `main.o`. Then the main program would be created by linking these together, using

```
g++ -o progname main.o f1.o f2.o f3.o
```

The compiler, `g++`, will not compile in this last step; it will just link the files and create the program named `progname`.

If you edit the file `f2.cpp`, you only need to do two steps to rebuild the program:

```
g++ -c f2.cpp
g++ -o progname main.o f1.o f2.o f3.o
```

because only the object file `f2.o` must be changed, and the main program needs to be relinked to it. If you edit a header file, such as `f3.h`, then you would also do two steps:

```
g++ -c f3.cpp
g++ -o progname main.o f1.o f2.o f3.o
```

because presumably `f3.cpp` depends on `f3.h` in such a way that the object file `f3.o` needs to be rebuilt, and then the program relinked.

This can be simplified by using a ***makefile***. A makefile is a file that is read by the make program. It contains a set of instructions for carrying out various tasks, usually for building programs. This tutorial on separate compilation does not cover how to create makefiles, but I include one here that could be used to keep the program `progname` up to date whenever any of the files change, with minimal recompilation and linking:

```
CXX        := /usr/bin/g++
CXXFLAGS   += -Wall -g
OBJECTS    := main.o f1.o f2.o f3.o

all: progname

progname: $(OBJECTS)
<TAB>$(CXX) $(CXX_FLAGS) -o progname $(OBJECTS)

main.o: f1.h f2.h f3.h

clean:
<TAB>$(RM) $(OBJECTS)
```

The `<TAB>` means that there should be a tab character in this position. Anything else and make will not work. To build the program or update it, one just types "make" on the command line in the directory containing this makefile and the program files. The makefile should be named either `makefile` or Makefile.

C Strings and Pointers

Motivation

The C++ string class makes it easy to create and manipulate string data, and is a good thing to learn when first starting to program in C++ because it allows you to work with string data without understanding much about why it works or what goes on behind the scenes. You can declare and initialize strings, read data into them, append to them, get their size, and do other kinds of useful things with them. However, it is at least as important to know how to work with another type of string, the C string.

The C string has its detractors, some of whom have well-founded criticism of it. But much of the negative image of the maligned C string comes from its abuse by lazy programmers and "hackers". Because C strings are found in so much legacy code, you cannot call yourself a C++ programmer unless you understand them. Even more important, C++'s input/output library is harder to use when it comes to input validation, whereas the C input/output library, which works entirely with C strings, is easy to use, robust, and powerful.

In addition, the C++ `main()` function has, in addition to the prototype

```
int main()
```

the more important prototype

```
int main ( int argc, char* argv[] )
```

and this latter form is in fact, a prototype whose second argument is an array of C strings. If you ever want to write a program that obtains the command line arguments entered by its user, you need to know how to use C strings.

All of this is why we turn our attention to them here.

Declaring and Initializing C Strings

A C string is an array of characters terminated by a special character called the `NULL` character, also called a `NULL` byte. The `NULL` character is the character whose binary value is 0. One can write the `NULL` character as `'\0'` in a program. Because the `NULL` character takes up one character position in the array, a C string that can store N characters must be declared to have length $N+1$.

Examples

```
char lastname[21];          // can store up to 20 characters, no more
char socsecuritynumber[10]; // can store a 9 digit social security number
char filename[500];         // can store up to 499 characters
```

Unfortunately this little fact seems to be forgotten by many programmers, and when that happens, trouble starts, as you will soon see.

You can initialize a C string in its declaration the same way that you initialize a C++ string, in one of two ways:

```
char lastname[21] = "Ritchie";
char socsecuritynumber[] = "123456789";
```

The first method specifies the length of the array. ***In this case the number of characters in the string literal must be at most one less than this length***. In the second method the compiler determines how large to make the array of characters.

Notice that there is no `NULL` character in either of these initializing declarations. The compiler ensures that the resulting string does have a trailing `NULL` byte.

You will also find declarations of C strings that look like this:

```
char* errmessage = "You did not enter a valid menu choice.";
```

or equivalently

```
char *errmessage = "You did not enter a valid menu choice.";
```

The '`*`' in these declarations means that the variable `errmessage` is a ***pointer***. Later we will talk about pointers. For now, just be aware that both of these declarations declare `errmessage` to be a C string, initialized with the given string, and terminated with a `NULL` byte.

Unlike C++ strings, C strings are of fixed size. They cannot grow automatically by appending more characters to them. They cannot be resized. They are just arrays of characters, nothing more. Later we will see how to change the amount of memory allocated to strings as they need to grow.

Using C Strings

You can treat a C string as an array of characters, using the subscript operator to access each individual character, as in

```
char name[30] = "Thompson";
/* more code here that might change contents of name */
int i = 0;
while ( i < 30 && name[i] != '\0' ) {
    name[i] = toupper( name[i] );
    i++;
}
```

which converts the characters in `name` to uppercase. Notice how the loop entry condition has two conjuncts: the first makes sure that we do not exceed the array bounds and the second looks for the end of the string in case it is shorter than 29 characters (not 30, remember!) In this simple case using a C string is almost the same as using an array. The big difference is that when you have an array, you have to keep track of its length and keep it in a separate variable. With a C string that has been properly initialized, you do not need to know its length because you can search for the `NULL` byte at the end of it.

Searching for the end of a C string is unnecessary because there is a function named `strlen()` in the C `string` library that does this for you. Its prototype is

```
size_t strlen( const char * str);
```

Do not be intimidated by the return type. It is just a `typedef` for `unsigned int`. Sizes are never negative numbers. The argument is a string, expressed not as an array but as a `char*`.

The C string library's header file is `<cstring>` in C++, not `<string>`. The C string library header file is `<string.h>` in C. Therefore, to use the `strlen()` function you need to include `<cstring>`:

```
#include <cstring>
using namespace std;

char name[30] = "Thompson";
/* more code here that might change name */
int i = 0;
while ( i < 30 && i < strlen(name) ) {
    name[i] = toupper( name[i] );
    i++;
}
```

C++ Input and Output

The C++ insertion and extraction operators both work with C strings, when used properly. The following program illustrates:

```
#include <iostream>
#include <cctype>
#include <cstring>
using namespace std;

int main()
{
    char name[10];
    int  i = 0;

    cout << "Enter at most 9 characters: ";
    cin >> name;

    while ( i < 10 && name[i] != '\0' ) {
        name[i] = toupper(name[i]);
        i++;
    }
    cout << name << endl;
    return 0;
}
```

This program will extract the characters entered by the user, until it finds a valid whitespace character or end-of-file character, storing them into the character array name. It will then append a `NULL` character to the end of `name` automatically. The insertion operator `<<` will insert the characters from `name` until it finds a `NULL` character.

There is a great danger with C strings and the extraction operator. It does not check that the input can fit into the string argument. The user can enter more characters than there are room for in the array, and these characters will be stored in the memory positions that logically follow the array of characters, overwriting whatever was there. This is called ***buffer overflow***. Purposefully trying to create buffer overflow is a standard technique used by hackers to try to break into computer systems.

To illustrate, suppose that we enter the string "`abcdefghijkl`" when prompted by the program. It will store these characters into the memory locations starting at `name[0]`, then `name[1]`, `name[2]`, and so on, and continue past the end of the array, storing `k` into the next byte after `name[9]` and `l` into the byte after that. These bytes might actually be part of the storage allocated to the integer `i`, in which case they will overwrite `i`. If the entered string is large enough, the program will crash with a segmentation fault.

To prevent the user from entering more characters than can fit into the array, you can use the I/O manipulator, `setw()` which will limit the number of characters entered to ***one less than its argument***. Remember to `#include <iomanip>` in this case:

```
#include <iostream>
#include <cctype>
#include <cstring>
#include <iomanip>
using namespace std;

int main()
{
    char name[10];
    int  i = 0;

    cout << "Enter at most 9 characters: ";
    cin >> setw(10) >> name;

    while ( i < 10 && i < name[i] != '\0' ) {
        name[i] = toupper(name[i]);
        i++;
    }
    cout << name << endl;

    return 0;
}
```

This program will prevent buffer overflow. If you intend to use the C++ extraction operator as your primary means of reading data into strings of any kind, you should make a habit of using the `setw()` manipulator.

Things You Cannot Do With C Strings (That You Can Do With C++ Strings)

You cannot assign one string to another, as in

```
char name1[10];
char name2[10];
name1 = name2;  // This is an error (invalid array assignment)
```

You cannot assign a string literal to a C string:

```
name = "Harry"; // This is illegal (incompatible types in assignment)
```

You cannot compare two C strings with comparison operators such as <, <=, >, and >= to find which precedes the other lexicographically. For example

```
( name1 < name2 )
```

will simply check whether the starting address of `name1` in memory is smaller than the starting address of `name2` in memory. It does not compare them character by character.

You cannot compare two strings for equality:

```
( name1 == name2 )
```

is true only if they have the same address.

The + operator does not append C strings. It is just an illegal operation.

Pointers, Very Very Briefly

This is a brief introduction to ***pointers***. We need to know a bit about pointers because C strings are closely related to them. Some people tremble at the mention of pointers, as if they are very hard to understand. If you understand reference variables, you can understand pointers.

A pointer variable contains a memory address as its value. Normally, a variable contains a specific value; a pointer variable, in contrast, contains the memory address of another object. We say that a pointer variable, or pointer for short, ***points to*** the object whose address it stores.

Pointers are declared to point to a specific type and can point to objects of that type only. The asterisk `*` is used to declare pointer types.

```
int*    intptr;    // intptr can store the address of an integer variable
double* doubleptr; // doubleptr can store the address of a double variable
char*   charptr;   // charptr can store the address of a character
```

We say that `intptr` is "a pointer to int" and that `doubleptr` is "a pointer to double." Although we could also say that `charptr` is a pointer to char, we say instead that `charptr` is a string. Pointers to `char` are special.

It does not matter where you put the asterisk as long as it is between the type and the variable name– these three are equivalent:

```
int* intptr;
int * intptr;
int *intptr;
```

In general, if `T` is some existing type, `T*` is the type of a variable that can store the address of objects of type `T`, whether it is a simple scalar variable such as `int`, `double`, or `char`, or a more structured variable such as a structure or a class:

```
struct Point { int x; int y; };
Point* Pointptr;
```

The ***address-of*** operator, `&`, can be used to get the memory address of an object. It is the same symbol as the reference operator, unfortunately, and might be a source of confusion. However, if you remember this rule, you will not be confused:

> If the symbol "`&`" appears ***in a declaration***, it is declaring that the variable to its right is a reference variable, whereas if it appears ***in an expression that is not a declaration***, it is the address-of operator and its value is the address of the variable to its right.

```
int x = 972384;
int y = 100;
int &ref = x;  // a declaration, so & is the reference operator
int *p   = &x; // appears on right of =, so it is in an
               // expression and is address-of operator; p stores address of x
p = &y;        // address-of operator; p stores the address of y
int &z = y;    // reference operator; z is an alias for y
```

Dereferencing Pointers

Accessing the object to which a pointer points is called ***dereferencing the pointer***. The "*" operator, when it appears in an expression, dereferences the pointer to its right. Again we have a symbol that appears to have two meanings, and I will shortly give you a way to remain clear about it.

Once a pointer is dereferenced, it can be used as a normal variable. Using the declarations and instructions above,

```
cout << *p;      // p contains address of y, which stores 100,
                 // so 100 is printed. p has been dereferenced.
cin >> *p;       // p is dereferenced before the input operation.
                 // *p is the variable y, so this stores the input into y.
*p = *p + *p;    // just like y = y + y
*p = *p * *p;    // just like y = y * y
```

The dereference operator and the address-of operator have higher precedence than any arithmetic operators, except the increment and decrement operators, `++` and `--`.

If you think about a declaration such as

```
int * p;
```

and treat the `*` as a dereference operator, then it says that `*p` is of type `int`, or in other words, `p` is a variable that, when dereferenced, produces an `int`. If you remember this, then you should not be confused about this operator.

Relationship Between Pointers And Arrays

An array name is a constant pointer; i.e., it is pointer that cannot be assigned a new value. It always contains the address of its first element.

```
int list[5] = {2,4,6,8,10};
cout << *list << endl;       // prints 2 because *list is the contents of list[0]

int * const aPtr = list;     // aPtr has the same address as list
                             // &list[0] is stored in aPtr.
aPtr[2] += 10;               // adds 10 to  list[2]
```

The fact that array names are essentially constant pointers will play an important role in being able to create arrays of arbitrary size while the program is running, rather than by declaring them statically (at compile time).

This is all we need to know for now in order to be able to talk more about C strings and the C string library.

Using The C String Library

The C string library (header file `<cstring>` in C++) contains useful functions for manipulating C strings. On a UNIX system you can read about it by typing `man string.h`. You can also read about it online in many places, such as `http://www.cplusplus.com/reference/clibrary/cstring/`. There are a couple dozen functions defined there, but we will start by looking at the most basic of these. All require the `<cstring>` header file, which is placed into the `std` namespace.

strcmp()

The `strcmp()` function is how you can compare two C strings. Its prototype is

```
int strcmp ( const char * str1, const char * str2 );
```

This compares two C strings, `str1` and `str2` and returns

0	if the two strings are identical, character for character
< 0	if `str1` precedes `str2` lexicographically
> 0	if `str1` follows `str2` lexicographically

"Lexicographically" means that the ascii (or binary) values of the corresponding characters are compared until one is different than the other. If one string is a prefix of the other, it is defined to be smaller.

```
str1      str2      strcmp(str1, str2)     comment
abc       abe       < 0                    (str1 < str2)
Abc       abc       > 0                    (str1 > str2)
abc       abcd      < 0                    (str1 < str2)
```

This little program compares two inputted strings and outputs their relationship:

```
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cctype>
using namespace std;

int main()
{
    char name1[10];
    char name2[10];

    cout << "Enter at most 9 characters: ";
    cin >> setw(10) >> name1;
    cout << "Enter at most 9 characters: ";
    cin >> setw(10) >> name2;

    if ( strcmp(name1, name2) > 0 )
        cout << name1 << " is larger than " << name2 << endl;
    else if ( strcmp(name1, name2) < 0 )
        cout << name1 << " is smaller than " << name2 << endl;
    else
        cout << name1 << " is equal to " << name2 << endl;
    return 0;
}
```

strcat()

The `strcat()` function allows you to append one string to another. Its prototype is:

```
char * strcat ( char * destination, const char * source );
```

This appends a copy of the `source` string to the `destination` string. The terminating `NULL` character in `destination` is overwritten by the first character of `source`, and a `NULL`-character is included at the end of the new string formed by the concatenation of both in `destination`. The `destination` string must be large enough to store the resulting string plus the `NULL` byte, and the two strings are not allowed to overlap. The return value is a pointer to the resulting string.

```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;

int main()
{
    char name1[100];
    char name2[100];

    cout << "Enter your first name: ";
    cin >> setw(49) >> name1;          // limit to 49 chars
    cout << "Enter your last name: ";
    cin >> setw(49) >> name2;          // limit to 49 chars

    strcat(name1, " ");                // name1 might be 50 bytes now

    cout << "Then you must be " << strcat(name1, name2) << ".\n";
    // name1 might be 50+49+1 = 100 bytes including the NULL byte
    cout << "Nice to meet you, " << name1 << endl;
    return 0;
}
```

Notice that this program uses the result of `strcat()` in two different ways. The first call to it modifies `name1` by appending a space character to it. It ignores the return value. The second call, as the argument to the insertion operator, uses the return value. `name1` is also modified, as the final output statement demonstrates when you run this program.

strcpy()

The `strcpy()` function copies one string into another. Its prototype is

```
char *strcpy(char *destination, const char *source);
```

This copies the source string, including its terminating `NULL` byte, to the destination string. The strings may not overlap, and the destination string must be large enough to receive the copy, including the `NULL` byte. The `strcpy()` function takes the place of the C++ string assignment operator. The following example shows how it can be used.

```
#include <iostream>
#include <iomanip>
#include <cstring>
```

```
using namespace std;


int main()
{
    char lastname[50];
    char firstname[30][10];
    char temp[30];
    int  i = 0, j = 0;

    cout << "Enter your family name: ";
    cin >> setw(50) >> lastname;
    cout << "Enter the names of your children (Ctrl-D to stop): ";
    cin >> setw(30) >> temp;
    while ( !cin.eof() && i < 10 ) {
        strcpy(firstname[i], temp);
        i++;
        cin >> setw(30) >> temp;
    }
    if ( i > 0 ) {
        cout << "Your children are \n";
        while ( j < i )
            cout << firstname[j++] << " " << lastname << endl;
    }
    return 0;
}
```

The `strcpy()` function is used to copy the string stored in `temp` into the array of children's first names. Even though the declaration of `firstname` makes it look like a two dimensional array, and technically it is, it is really an array of C strings, or at least the program treats it as an array of C strings.

Summary

The following table summarizes the differences between the most common string operations using C++ strings and C strings.

C++ string operation	**C string equivalent**
`n = name.size();`	`n = strlen(name);`
`str1 = str2;`	`strcpy(str1, str2);`
`str1 = str1 + str2;`	`strcat(str1, str2);`
`if ( str1 > str2 )`	`if ( strcmp(str1, str2) > 0 )`
`if ( str1 < str2 )`	`if ( strcmp(str1, str2) < 0 )`
`if ( str1 == str2 )`	`if ( strcmp(str1, str2) == 0 )`

Safer Functions

The three functions described above all work in the same way internally. They all need to find the end of the string by looking for the `NULL` byte. This is not a problem when the strings are always `NULL`-terminated, but either for malicious reasons or because of honest mistakes, they can be called with strings that are not `NULL`-terminated, and in this case, many problems can result. Therefore, some time ago, safer versions of these functions were written that take another argument, which is a length argument. They all have the letter 'n' in their names:

```
int   strncmp(const char *str1,  const char *str2,   size_t n);
char *strncat(char *destination, const char *source, size_t n);
char *strncpy(char *destination, const char *source, size_t n);
```

Each differs from its unsafe counterpart in that the third parameter limits the number of characters to be processed. For example, `strncmp()` only compares the first `n` characters of the two strings when given argument `n`, and `strncat()` only appends the first `n` characters of `source` to `destination`, and `strncpy()` only copies the first `n` characters of `source` into `destination`.

More C String Library Functions

strchr()

Sometimes you would like to find a particular character in a C string. The strchr() function lets you do that easily. Its prototype is

```
char *strchr(const char *str, int ch);
```

This returns a pointer to the first occurrence in the string `str` of the character `ch`. Note that it does not return an index, but a pointer! If the character does not occur in the string, it returns a `NULL` pointer. For example,

```
char psswdline[] = "nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin";
char * colonptr = strchr( psswdline, ':' ) ;
```

would set `colonptr` to point to the first colon in the string `psswdline`. One could repeatedly call this function to get all of the positions, but there is a better way to do something like this using `strtok()`.

strspn() and strcspn()

Suppose that you wanted to find the length of the prefix of a string that contains only the characters in a given set of characters. For example, suppose that you wanted to find the longest prefix of a string that contained only the characters `a`, `c`, `g`, and `t`. Of course you could write your own loop, and it would not be very hard (I hope), but there is already a function that does this. Its prototype is

```
size_t strspn(const char *s, const char *accept);
```

`strspn( str, "acgt")` would return the length of the longest prefix of `str` that contained only those characters. If hat length was equal to `strlen(str)`, then that would imply that `str` had no other characters in it.

On the other hand, what if you wanted the length of the longest prefix of `str` that *did not contain* any of the characters in a given set of characters? That is what `strcspn()` is for.

```
size_t strcspn(const char *s, const char *reject);
```

The "c" in the name is for *complement*. Thus, `strcspn(str, ":")` would return the length of the longest prefix of `str` that did not contain a colon. This is something like `strchr()` except that you get a length instead of a pointer to the colon.

strstr()

This function finds substrings of a string. Finding a substring of a string is a harder problem to solve efficiently than the preceding ones. It takes a bit of ingenuity to do it well. For a beginning programmer, this function is handy. If you want to find the first occurrence of the string "`acgaa`" in a string consisting of thousands of occurrences of the characters `a`, `c`, `g`, and `t`, this will do it nicely. Its prototype is

```
char *strstr(const char *haystack, const char *needle);
```

The choice of parameter names is from its man page on my Linux system. `strstr(haystack, needle)` returns a pointer to the first occurrence of the string pointed to by `needle` in the string pointed to by `haystack`. If it has no such substring, `NULL` is returned.

strtok()

The `strtok()` function is different from all of the others in that it can remember the fact that it was called before by your program. The technical way of stating this is that ***it retains state***. It allows you to ***parse*** a string into a sequence of ***tokens***. Parsing a string means decomposing it into its grammatical elements. The individual elements are called tokens in this case. For example, a comma-separated-values file consists of lines each of which is a sequence of tokens separated by commas, as in

```
Harry,Jones,32,photographer,333-33-1234
```

The `strtok()` function will let us break this line into the sequence of tokens

```
Harry
Jones
32
photographer
333-33-1234
```

with very little programming effort. Its prototype is

```
char *strtok(char *str, const char *delim);
```

The first argument is a pointer to the string to be parsed and the second is a pointer to a string consisting of the characters that can act as a delimiter. A delimiter is always a single character. If we want to break up a string on commas, we would use the delimiter string ",". If we want to allow both commas and semicolons, we would use the delimiter string ",;".

`strtok()` remembers its state in the following way. The first time you call it, you give it a pointer to the string you want to parse. Then you repeatedly call it passing a `NULL` first argument instead, and it knows that you still want it to parse the string that you first passed to it. It returns a pointer to a null-terminated string containing the next token on each call until there are no more tokens, when it returns a `NULL` pointer instead. An example will illustrate.

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

int  main(int argc, char *argv[])
{
    char *str, *token, *delim;
    int j = 1;

    if ( argc < 3 ) {
        cerr << "usage: argv[0] string-to-parse  delim-chars\n";
        exit(1);
    }
    str   = strdup(argv[1]);
    delim = strdup(argv[2]);

    token  = strtok(str, delim);
    cout << j << " " << token << endl;

    while ( 1 ) {
        token = strtok(NULL, delim);
        if ( token == NULL )
            break;
        cout << j++ << " " << token << endl;
    }
   return 0;
 }
```

The program expects two command-line arguments: the string to be parsed and a string of delimiters. For example, if we name this program `parseline`, we would use it like this:

```
parseline "Harry,Jones,32,photographer,333-33-1234" ","
```

It would decompose the first string on commas, printing

```
1 Harry
2 Jones
3 32
4 photographer
5 333-33-1234
```

There are several important points to make about `strtok()`.

- It treats adjacent delimiters as a single delimiter. Also, it ignores delimiters at the beginning and end of the string. Thus

  ```
  parseline ,"Harry,Jones,32,,,photographer,,,333-33-1234," ","
  ```

  would produce the same output as the first invocation above. It will never return an empty string. If you need to detect when a token might be empty, you cannot use `strtok()` for this purpose[1].

- It cannot be used on constant strings, meaning strings declared as arrays of chars or with the `const char*` type because it modifies its first argument as it works. Therefore you should copy the string to be parsed into a temporary copy. The `strdup()` function makes a non-constant copy of a string and returns a pointer to it. The program above copies the command-line arguments using `strdup()` and then calls strtok() on the copy.

- The returned token must also be a non-constant string, so it too should be declared as a `char*`.

- If you call `strtok()` with a non-`NULL` first argument, it stops parsing the previous string and starts parsing the new string.

Character Handling: The ctype Library

The C language, and C++ as well, has a rich library of character classification and mapping functions, all of which are declared in the C header `<ctype.h>`, or `<cctype>` in C++. By a mapping function, we mean one that associates one character to another. For example, the function `toupper()` returns the corresponding uppercase letter for a given lower case letter.

All of the functions in the `<ctype>` library take a single character as their argument, but this character is declared as an `int`, not a `char`, and they all return an `int` as well. For example

```
int toupper(int ch);
```

is the prototype for the `toupper()` function. You can pass a character to them, as if they had a `char` parameter, e.g.

```
char ch;
ch = toupper('a'); // store 'A' into ch
```

Although the return type is an `int`, you can still assign the return value to a variable of type `char`. The two useful mapping functions in the library are

[1] You can use `strsep()`, but not all systems have this function.

```
int toupper( int ch);
int tolower(int ch);
```

which return the corresponding character in the opposite case when given a letter, and return the same character as the argument otherwise. Because these functions return an `int`, you have to be a bit careful about how you use them. For example, the instruction

```
cout << toupper('a');
```

prints the code for `'A'` (most likely 65 on your computer) instead of the character `'A'` because it can have an argument of type `int`. If you want to print the `'A'` then either you have to force that return value to be interpreted as a char. You can do this by assigning the return value to a variable of type `char`, as I showed above, or by casting, as in

```
for (char c = 'a'; c <= 'z'; c++ )
    cout << (char) toupper(c);
```

which will print the uppercase letters.

Classification functions take a single character and report on the type of character that it is. There is much overlap in the classifications. The disjoint classes are the lower and upper case letters, digits, space characters, and punctuation. Several functions report on various combinations of those. For example, `isalnum()` reports whether a character is a digit or an alphabetic character. The set of classification functions includes the following:

```
int islower(int ch); // tests whether ch is a lowercase letter
int isupper(int ch); // tests whether ch is an uppercase letter
int isalpha(int ch); // true iff ( islower(ch) || isupper(ch) )
int isdigit(int ch); // tests whether ch is a digit
int isalnum(int ch); // true iff ( isalpha(ch) || isdigit(ch) )
int isblank(int ch); // tests whether ch is a blank (tab or ' ')
int isspace(int ch); // tests for ' ','\t', '\n', '\f', '\r', '\v'
int iscntrl(int ch); // tests for control characters
int isgraph(int ch); // tests whether is printable but not blank
int isprint(int ch); // tests whether ch is printable
int ispunct(int ch); // tests whether ch is punctuation
```

All of these return a non-zero for true and a zero for false. The precise definitions of these functions is different than the comments above; it actually depends upon the ***locale***, which is a collection of environment settings that defines such things as the language, character set, how monetary units are displayed, and so on. The descriptions above are stated for the standard "C" locale, which is what most users have by default.

Processing Command-Line Arguments

In a UNIX environment, when you type a command such as

```
g++ main.cpp utils.cpp fileio.cpp
```

or

```
rm file1 file2 file3 file4
```

and press the `Enter` key, the shell program parses this command-line into words separated by whitespace characters. A ***word*** is usually any sequence of non-whitespace characters[1]. The first word on the command line, except in certain unusual commands, is the name of a program or a shell built-in command to be run. In the above examples, it is `g++` and `rm` respectively. The words that follow the program name are called ***command-line arguments***. In the first example above, the command-line arguments are `main.cpp`, `utils.cpp`, and `fileio.cpp`. In the second example, they are `file1`, `file2`, `file3`, and `file4`.

In UNIX and in other POSIX-compliant operating systems, the operating system arranges for the program name and the command-line arguments to be made available to the program itself via parameters to the `main()` function. Programs can ignore this information by writing the main function as

```
int main () { /* program here ... */ }
```

However, the C and C++ standards require compliant implementations of C and C++ to accept a `main()` with two parameters as follows:

```
int main ( int argc,  char * argv[]  ) { /* program here ... */ }
```

The first parameter is an integer specifying the number of words on the command-line, including the name of the program, so `argc` is always at least one. The second parameter is an array of C strings that stores all of the words from the command-line, including the name of the program, which is always in `argv[0]`. The command-line arguments, if they exist, are stored in `argv[1]`, ..., up to `argv[argc-1]`. If you have not seen the `char*` type, refer to the notes on C strings and pointers on this website.

Also note that there is nothing special about the names of two parameters `argc` and `argv`; they can be whatever names you want them to be. It is a convention to use the names `argc` and `argv`, although you will often find programs that use `ac` and `av` instead. You can name them `foo` and `bar` but that would be pretty bad programming style.

A simple C++ example that illustrates how a program can access the command-line arguments is below. This simple program does nothing more than display the name that the user typed to execute the program, followed by the command-line arguments that it received from the shell.

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
```

[1] Certain symbols such as the shell redirection operators, semicolons, quotes, and so on, are not considered words in this sense.

```
    cout << argv[0] << ": ";
    for ( int i = 1; i < argc ; i++ ){
         cout << argv[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Whenever you write a program that expects command-line arguments you must check whether the expected number of arguments was provided by the user. Otherwise, the program will attempt to access locations in the array of arguments that do not exist.

For example, suppose that you write a program that expects the names of two files on the command-line, the first being the name of a file to open for reading and the second being the name of a file to open for writing. Suppose that the name you give to the program executable is `myprog`. Then proper use of `myprog` would be something like

```
myprog inputfile outputfile
```

There have to be at least three words on the command-line for your program to run properly. There might be more, but it can ignore those words. Therefore, the program should only run if the first parameter to `main()` is at least 3. The program must therefore begin with

```
int main ( int argc,  char * argv[]  )
{
    /* declarations here */

    if ( argc < 3 ) {
        /* handle the incorrect usage here */
        cerr << usage:  << argv[0] << " inputFileName   outputFileName \n";
        exit(1);
    }
    /* rest of program */
}
```

If the user did supply the correct number of arguments, then it is safe for the program to access the strings from the second parameter. The program might look something like

```
int main ( int argc,  char * argv[]  )
{
    ifstream fin;
    ofstream fout;

    if ( argc < 3 ) {
        /* handle the incorrect usage here */
        cerr << usage:  << argv[0] << " inputFileName   outputFileName \n";
        exit(1);
    }

    fin.open(argv[1]);
    if ( fin.fail() ) /* handle the error here */

    fout.open(argv[2]);
    if ( fout.fail() ) /* handle the error here */
    /* rest of program */
}
```

A Refinement

If the user types a command such as

```
../../proj1/testcode/myprog infile
```

and forgot to include the output file, the above code would produce output such as

```
usage:  ../../proj1/testcode/myprog inputFileName   outputFileName
```

If you do not want to display the entire path name of the program, but prefer that it only displays

```
usage:  myprog inputFileName   outputFileName
```

then you have to strip off the leading part of the `argv[0]` string so that the only thing left is what comes after the final '/' character. The C string library has a function that will make this easy, provided you are familiar with pointers.

You can use the `strrchr()` function, whose prototype is

```
char *strrchr(const char *source, int ch);
```

This function finds the last occurrence in the string `source` of the character `ch`. If `ch` is not in `source`, then it returns a `NULL` pointer. Therefore a strategy for displaying the characters of the program name after the final '/' is to check whether it has a slash, and if it does, display the part after it. The simplest way to do this is to take advantage of the fact that the program is allowed to modify the `argv[]` array. In particular, it can change what `argv[0]` points to. The following C++ program does just this.

```
#include <iostream>
#include <cstring>
using namespace std;

int main(int argc, char * argv[])
{
    char *forwardslashptr;

    forwardslashptr = strrchr( argv[0], '/');
    if ( forwardslashptr != NULL )
        /* argv[0] does contain the '/', so reset it so it points to
           the character just past the '/' character. Nothing needs
           to be done if it has no slash.
        */
        argv[0] = forwardslashptr+1;

    cout << argv[0] << ": ";
    for ( int i = 1; i < argc ; i++ ){
         cout << argv[i] << " ";
    }
    cout << endl;
    return 0;

}
```

This program uses "pointer arithmetic". The line

```
argv[0] = forwardslashptr+1;
```

sets the `argv[0]` string, which is, remember, a pointer to an array of characters, to the value obtained by adding one to the address stored in `forwardslashptr`. The compiler translates pointer addition to add however many bytes are needed by the type of thing that the pointer points to. In other words, if a pointer points to a `char` and a `char` takes up one byte, it adds 1. Therefore the above instruction causes `argv[0]` to point to the first character after the slash.

Namespaces, Structures and Classes

Namespaces

A ***namespace*** is a collection of name declarations and/or definitions. Your program's interaction with namespaces so far has been limited to using them via the ***using directive***, as in

```
#include <iostream>
using namespace std;
```

The reason that a program with a line like

```
cout << "Hello world.";
```

fails to compile unless the `using namespace std` directive precedes this statement is that the name `cout`, defined in the header file `<iostream>` has been placed into the namespace `std` within that header file and is unknown outside of that namespace. A namespace defines its own scope and the only way to refer to the members of that namespace outside of it are either to

- put a using namespace directive into the program before the point at which you want to refer to those members, or
- precede each member's name with ***explicit scope resolution***.

The former is what you've done so far, but you have also seen that instead of using the namespace in the program you can write something like

```
std::cout << "Hello world";
```

which tells the compiler to use the name `cout` defined in the namespace `std`. The "`::`" is called the ***scope resolution operator***. The name on the left of it is the name of a scope region such as a namespace, and the name on the right is a member of that scope region. The construct `std::cout` is how you refer to the `cout` member of the namespace `std`.

All of the declarations that you write in a program have scope of one form or another. Recall that block scope is the scope that extends from a point of a declaration within a block to the end of that block, and that file scope is the region of program text from the point of a declaration that is not in any block until the end of the file containing that declaration. All of the names that have file scope in your program actually belong to an unnamed namespace called the global namespace. This does not mean that the namespace is named global! It has no name, but we call it the global namespace.

You can create namespaces of your own. To create a namespace with the name `spacetime` for example, you would write

```
namespace spacetime
{
     /* declarations and definitions here */
}
```

The keyword `namespace` introduces the namespace definition. The name of the namespace follows, in this case `spacetime`, and then what looks like a block: a pair of curly braces with whatever declarations and/or definitions you want to place in them. Notice though that there is no semicolon after the namespace definition. It is not required.

Example

```
namespace spacetime
{

    void printdate( ostream & out, int y, int m, int d )
    {
        out << m <, "/" << d <, "/" << y;
    }

    void printpoint( ostream & out, double x, double y, double z);
}
```

If you then put a directive

```
using namespace spacetime;
```

in your program, you will be able to call the function `printdate()` by writing just the function name itself, but if you do not put this using directive into the program, then you must ***qualify the name*** `printdata()` with the name of its namespace, i.e., you must call `spacetime::printdata()` to use it:

```
spacetime::printdata(2012, 11, 6);
```

The `printpoint()` function is declared in the namespace but its definition is not contained in it. Suppose for some reason that we wanted to write the definition of `printpoint()` outside of the namespace. (Imagine that the namespace has prototypes of functions and only prototypes, and different programmers need to write different implementations of these functions in their own code.) Then you would have to write the function definition as follows:

```
void spacetime::printpoint( ostream & out, double x, double y, double z)
{
    /* code here */
}
```

To summarize, a namespace is a collection of names. To refer to any of those names outside of the namespace itself, you must either insert a `using namespace` directive before the first use of any of those names, or you must qualify those names with the namespace name followed by the scope resolution operator.

Structures

Suppose that we would like to store a student's academic record for processing. It will have string data such as names and id numbers, numeric data such as GPA and the number of credits earned, the courses the student has taken so far, the grades in each, and so on. You could create a separate variable for each data item, and that might be fine for a program that manages one student record, but if you can imagine a program that reads in a list of these records from a text file, perhaps stored one record per line, and has to store each record separately, then you should realize that putting the separate parts of the student records in separate variables is a messy solution. Clearly the data types you have learned about so far, such as strings, scalar types, and arrays, cannot be used very easily to store this complex set of data.

One solution in C and C++ is a type known as a ***structure***. A structure is a ***heterogeneous collection*** of data. It can store data of different types, each with its own name. A structure is introduced with the `struct` keyword, as in the example below.

```
struct User
{
    string firstname;
    string lastname;
    int    score;
};
```

This defines a new data type named `User` that has three members. Each member is a variable that can be assigned data independently of the other members. The first two members of the structure are `string` variables named `firstname` and `lastname`. The third member is an `int` named `score`.

The syntax of the structure definition is

```
struct structurename { list_of_member_declarations };
```

where *`structurename`* is a valid identifier and *`list_of_member_declarations`* is a list of variable declarations. ***The structure definition must be terminated by a semicolon.***

The structure does not have to contain different types of data. Here is a useful structure whose members are all integers:

```
struct Date
{
    int year;
    int month;
    int day;
};
```

Because a structure definition defines a type and not a variable, it can be placed into the global scope so that all functions have access to the definition.

Structures can be initialized when they are declared using the same form of initialization as arrays:

```
User default_user = {"Jody", "Jones", 0 };
```

The values are assigned to the members in the same order as they appear in the declaration. I.e.., it is ***positional assignment***. Variables whose type is a structure are declared the same way as ordinary variables:

```
User current_user;
User default_user;
```

Having declared a variable of the given structure type, the individual members of that variable are accessed using the "dot operator". The syntax is

```
variable_name.member_name
```

as in

```
cin  >> current_user.firstname;
cin  >> current_user.lastname;
current_user.score = 0;
```

Two variables of the same structure type can be assigned one to the other, and the entire structure is copied from one to the other:

```
current_user = default_user;
```

is the same as

```
current_user.firstname = default_user.firstname;
current_user.lastname  = default_user.lastname;
current_user.score     = default_user.score = 0;
```

Suppose a text file has lines consisting of a first name, last name, and a score, such as

```
Elmer Fudd 82
```

A program fragment like the following could be used to read the data from the file so that it can be sorted or processed in other ways.

```
int main()
{
    User    scoredata[MAXUSERS];  //  an array of 100 User structures
    User    temp;                 // temp variable for input
    fstream fin;
    int     length = 0;

    /* open a file and associate with the ifstream fin */
    /* skipping this part ... */

    fin >> temp.firstname >> temp.lastname >> temp.score;
    while ( length < MAXUSERS && !fin.eof() ) {
        scoredata[length] = temp;  // struct-to-struct assignment!!
        length++;
        fin >> temp.firstname >> temp.lastname >> temp.score;
    }

    /* now process the data */
}
```

The following function could be used to sort the data by score in ascending order. It uses a simple selection sort.

```
void sortByScore ( User list[], int len)
{
    int smallest; // index of smallest item in sorted part
    int first;    // index of first item in unsorted part
    User temp;

    for (first = 0; first < len-1; first++) {
        // find index of smallest element in range first to N-1
        smallest = first;
        for (int k = first+1; k <= len-1; k++)
            if ( list[k].score < list[smallest].score)
                smallest = k;

        temp = list[smallest];
        list[smallest] = list[first];
        list[first] = temp;
```

```
    }

}
```

Structures can contain other structures. There is no restriction on the type of the members of a structure. They can contain arrays, structures, arrays of structures, and so on.

Example:

```
struct Date
{
    unsigned int year;
    unsigned int month;
    unsigned int day;
};

struct User
{
    string firstname;
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

Note: ***If an array is a member of a structure, it should be declared with constant size.***

To access the members of a structure that is a member of another structure, you need the dot operator twice. The following example illustrates how the array elements and substructure members are accessed. This function reads from an input stream into a call by reference parameter of type `User`, assuming the data is stored in the stream in the order

```
firstname lastname birthmonth birthday birthyear score1 score 2 score3 score4 score5
```

```
void get_one_record( istream & instream,
                     User & record )
{
    instream >> record.firstname >> record.lastname;
    instream >> record.birthday.month
             >> record.birthday.day
             >> record.birthday.year;
    for ( int i = 0; i < 5; i++ )
        instream >> record.scores[i];
}
```

A program fragment to read a file of such records into an array of `User` structures would look like the following:

```
int main()
{
    User    scoredata[MAXUSERS];
    User    temp;
    fstream fin;
    int     length = 0;
```

```
    /* open a file and associate with the ifstream fin */
    /* skipping this part ... */

    // This code looks  the same as reading a single value. The details
    // are hidden in the function calls.
    get_one_record(fin, temp);
    while ( length < MAXUSERS && !fin.eof() ) {
        scoredata[length] = temp;  // struct-to-struct assignment!!
        length++;
        get_one_record(fin, temp);
    }

    /* now process the data */

}
```

This program fragment demonstrates a subtle point. Although you cannot assign one array to another, as in this code fragment:

```
int a[10]={0}, b[10];
b = a;
```

because it is a syntax error, you ***can*** assign structures containing arrays to each other, as in this fragment:

```
struct S {
    int list[10]
};
S a, b;
for ( int i = 0; i < 10; i++)
    a.list[i] = i;
b = a;
```

and it will do a copy of the entire `a` structure to the entire `b` structure including all elements of their arrays. In the listing above, the variable `temp` is assigned to `scoredata[length]` and this is a structure to structure assignment of a structure type containing an array.

Classes

In the preceding example, the `get_one_record()` function is intimately connected to the definition of the `User` structure above. If that structure is changed, this function must also be changed. In order to access the members of the `User` structure, that structure is an explicit call-by-reference parameter of the function.

The connection between the function and the data definition is a strong one, but the programmer is free to screw everything up because, as the program is currently designed, there is no language mechanism to prevent the programmer from breaking the program by failing to make modifications correctly.

In general, the connection between structure definition and the functions that act on them is weak and prone to error. If the definition changes, so must every function that acts on instances of it. To reduce this chance, one should put all functions and the definition into a single file, with clear instructions. The programmer is still able to foul everything up because there is no language mechanism to prevent the programmer from breaking the program. This is one reason for the invention of an alternative language model known as ***classes***.

A ***class*** is a collection of data and functions. It is like a structure with functions in it, except that it is much more than that. The data and the functions are part of a single entity. When you declare a variable whose type is a class, it has a special name. It is called an ***object***. *Do not get confused between objects and members.* The members of a class are the things defined between the curly braces, its elements. They are like members of a structure. An object of a class is an instance of that class, i.e., a variable whose type is that class.

Access to the members of a class may be restricted by the class's designer. Every member of a class, whether a function or data, has either

- public,
- protected, or
- private

access. We will ignore protected access for now and focus on public and private access.

- A ***public*** member is one that can be accessed by any function in the program. It is just like a member of a struct.
- A ***private*** member cannot be accessed by any functions in the program except those that are members of that class (and some others, to be explained later).

In object-oriented programming, we ***hide*** the data of our class so that no code can see it except the functions of the class to which it belongs. This is called ***data-hiding***. We put all related data together into the class's private data part. This is ***data encapsulation***. The functions that act on this data are the only things that are made public. These functions are the only means by which other parts of the program can access or change the data of the class. Only the function prototypes are put in the public part of the class. The actual definitions will be written someplace else. This is because it is no one's business ***how they work***, but just ***what they do***. This concept of separating how functions work from what they do is called ***procedural abstraction***. The idea of encapsulating the data and functions that act on it into a single entity is called ***data abstraction***.

We will introduce class definitions by example rather than by formal syntax rules.

In the simplest case, the syntax of a class definition is similar to that of a structure definition, except that it includes function prototypes and has access qualifiers:

```
class User
{
// The public interface:
public:
    void get_one_record  ( istream & instream ); // note no User parameter
    void print_one_record( ostream & outstream); // note no User parameter

// The private stuff:
private:
    string firstname;
    string lastname;
    int    scores[5];   // an array of 5 scores
    Date   birthday;    // This member is also a struct. Members can be ANY
                        // previously DECLARED type.
};
```

Points to Remember:

- The keyword `class` introduces the definition.
- The definition ends with a semicolon (just like a `struct`).
- `public:` (don't forget the colon) introduces the list of members that have ***public access***. Until another qualifier is reached, all following members, whether functions or data, have public access.
- The phrase `private:` (again there is a colon) introduces the list of members that have ***private access***s. Until another qualifier is reached, all following members, whether functions or data, have private access.
- If the class has **no access qualifier**, then all members default to ***private access*** until another access qualifier is reached. So the above could also have been written as

```
class User
{
    string firstname;
    string lastname;
    int    scores[5];   // an array of 5 scores
    Date   birthday;    // This member is also a struct. Members can be ANY
                        // previously DECLARED type.
// The public interface:
public:
    void get_one_record  ( istream & instream ); // note no User parameter
    void print_one_record( ostream & outstream); // note no User parameter
};
```

  because the members `firstname`, `lastname`, `scores`, and `birthday` precede the `public:` section.

- Function members are not defined in the class, but only declared. This means only their prototypes are written in the class definition.
- As with structures, array members should have constant size.

The public members of the class are accessed the same way that the members of a structure are accessed, using the dot operator. For example, a main program could contain the lines

```
User user1, user2;
user1.get_one_record( cin );
```

Two instances of the `User` class named `user1` and `user2` are declared, and the member function `get_one_record()` is called on the `user1` instance. We use the language "called on an object" or "called on an instance" to emphasize that the member function is a member of the class but that when it runs, it is acting on the data contained in a specific instance of that class. In this case `get_one_record()` will run on `user1`.

Now we turn to the function definitions. Since the functions are declared within the class but not defined there, their definitions are written outside of the class definition. Because their definitions are outside of the class definition, their form is a bit different than that of a function which is not a member function of a class. This is because a class acts like a namespace; the names declared within the class definition have scope that is limited to that definition. In order to use those names outside of the class definition, they must be qualified in the same way that names in a namespace must be qualified. Thus, the definitions of the `get_one_record()` and `print_one_record()` functions would be as shown in the listing below.

```
// Read a user record from the input stream into a User structure
void User::get_one_record( istream & instream)
{
```

```
    instream >> firstname >> lastname;
    instream >> birthday.month    // birthday is the class member, but it is a
             >> birthday.day      // struct and the dot is needed to get to its
             >> birthday.year;    // month, day, and year members
    for ( int i = 0; i < 5; i++ )
        instream >> scores[i];    // scores is the class member too.
}

// Write a User structure to the output stream, with some formatting
void User::print_one_record( ostream & outstream)
{
    outstream << lastname
              << ", " << firstname
              << "\t" << birthday.month
              << "/" << birthday.day
              << "/" << birthday.year
              << "\t" ;
    for ( int i = 0; i < 5; i++ )
        outstream << scores[i] << " ";
    outstream << endl;
}
```

It is only the name of the function that gets qualified, not the entire prototype. It is incorrect to write the definition like this:

```
User::void get_one_record( istream & instream)
{
    ... // INCORRECT
}
```

because this would mean that `void` is a name defined in the namespace `User` and `get_one_record` is a name defined in the global scope.

Observe that the functions do not have a parameter that is an instance of the class, but they access the private members such as `firstname` and `lastname`. This may seem confusing because `firstname` and `lastname` are not written using the dot operator. Think about it though. If you were to try to use the dot operator, what would be on its left side within this function?

When the function is called, it is called like this:

```
user1.get_one_record( cin );
```

In other words, it is called on a particular instance of the `User` class named `user1`, and so when it runs, the member variables `firstname`, `lastname`, `score` and so on are the members of the object `user1`. This is why they are written without the dot operator.

Separating Class Interface and Implementation

A class interface, or definition, should always be written separately from its implementation and they should be placed into separate files. The interface should be placed in a `.h` file (a header file) and its implementation in a `.cpp` file. This is in the spirit of procedural abstraction. The client code, and the people who write it, do not need to see how the functions are written; they only need to see the prototypes. Because the implementation file has to have access to the class definition, it needs to include the interface file usually, so you should put an `#include` directive in the `.cpp` file.

Imagine that you are writing some class that you would like to distribute to many users. You want to protect your code because you think it may have value. The way that you do this is by compiling the implementation file into an object file and then distributing the header file and the the object file (either in a `.o` file or compiled into a library file.) The header file should be thoroughly documented.

Header Guards

As with any header file, you need to enclose the code in a ***header guard***. The `#ifndef` directive is used to prevent multiple includes of the same file, which would cause compiler errors. #ifndef X is evaluated to true by the preprocessor if the symbol `X` is not defined at that point. X can be defined by either a `#define` directive, or by a `-DX` in the compiler's command-line options. The convention is

```
#ifndef __HEADERNAME_H #define __HEADERNAME_H

//  interface definitions appear here

#endif // __HEADERNAME_H
```

For those wondering why we need this, remember that the `#include` incorporates the named file into the current file at the point of the directive. If we do not enclose the header file in this pair of directives, and two or more included files contain an include directive for the header file, then multiple definitions of the same class (or anything else declared in the header file) will occur and this is a syntax error.

Constructors

When you declare an object of a class, such as in this declaration of `user1`:

```
User user1;
```

you are telling the compiler to create an instance of the `User` class named `user1`. For this particular example, it may not seem too hard for the compiler to figure out how to create a variable of this type. But there are questions. Should the compiler initialize the elements of the `scores` array to some particular value? Should the last and first names be given a particular initial value?

C++ provides a way to initialize the members of a class object when the object is instantiated. When you define a class, you have the option to define a special member function called a ***constructor***. A constructor is a member function that is executed when the object is initialized. Its definition is different from that of an ordinary function in that:

- Constructors have the same name as the class.
- They have no return values and cannot contain a return statement.

As an example prototype for a constructor for our `User` class:

```
User( string fname, string lname, Date bday );
```

defines a constructor with three arguments. Notice that there is no return type. Like ordinary functions, constructors can have arguments. Like ordinary functions, they can be overloaded. Unlike ordinary functions,

- Constructors can have a special feature called an ***initializer list***.

- Whether or not you create a constructor for a class, it does have one. *If you do not provide a constructor for a class, the compiler will generate one for it.* This automatically generated constructor will create an uninitialized object of that class. *If you do provide any constructor for your class, the compiler will not generate one for it.* A constructor that you define is called a ***user-defined constructor***.
- If a class must initialize its data members, then the class needs a user-defined constructor because the compiler-generated constructor will not be able to do this.
- Your program cannot call a constructor! It is called by the run-time library when the object must be initialized.

We illustrate with an example. We enhance the `User` class to contain two overloaded constructors.

Example

```
class User
{
public:
    User();
    User( string fname, string lname, Date bday );
    void get_one_record  ( istream & instream ); // note no User parameter
    void print_one_record( ostream & outstream); // note no User parameter
private:
    string firstname;
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

The first constructor has no parameters. It is an example of a ***default constructor***. *A default constructor is a constructor that can be called with no arguments.*

The second has three arguments. It can be called with a first and last name and a date to be assigned to the birthday member, and it could initialize the scores array to all zeros.

Your program cannot call the constructor. It is called automatically during program execution. When it is called depends upon the type of variable that it is. For now you can think of variables with block scope as being instantiated when the declaration within the block is reached.

The above constructors could be used to declare `User` objects as follows:

```
User user1;                                // default constructor invoked
Date bday = {1950,12,30};                  // Create a date object
User user2("Bjarne", "Stroustrup", bday); // second constructor invoked
User user3, user4;                         // default constructor invoked twice
```

- The default constructor is called when you declare an object of the class in the ordinary way, just by writing the variable name after the class name, as in the first line.
- The non-default constructor is called by writing the class name, then the variable name, and after that the parenthesized argument list, as in the third line. This is very different from ana ordinary function call!

The following are illegal ways to declare objects that have user-defined constructors:

```
User user5();  // ILLEGAL - no parentheses allowed
User user6;
const Date bday = {1900,1,1};
user6.User("Sam", "Spade", bday); // ILLEGAL - user6 already exists and
                                  // cannot call by name on an existing object
User user7 = {"Sam", "Spade", bday }; // Cannot initialize using ={}
```

- You cannot put empty parentheses after the variable name. The compiler sees this as a prototype for a function. The first line for example looks like a function named `user5` that returns a `User` object and has no parameters.
- You cannot use a constructor to give values to an object that already exists. The object `user6` was created in line 2, and then a constructor was called on it in line 4, but it was already constructed. That is an error.
- Although you can initialize structures using the curly brace initializer, in general you cannot do this with classes. The only time you can do this with classes is if they have no private section, roughly. Just avoid it.

Defining Constructors

The two constructors above could be implemented as follows.

```
User::User()
{
    firstname = "";
    lastname  = "";
    for ( int i = 0; i < 5;  i++) scores[i] = 0;
    birthday.year = birthday.month = birthday.day = 0;
}

User:User (string fname, string lname, Date bday )
{
    firstname = fname;
    lastname  = lname;
    for ( int i = 0; i < 5;  i++) scores[i] = 0;
    birthday = bday;
}
```

Notice that the constructor name is qualified since its definition is outside of the class definition. The default constructor assigns empty strings to string variables and zeros to integer variables. The non-default constructor assigns the string arguments to the string member variables, and the `Date bday` to the birthday member of the object, using a structure assignment.

We can also define a default constructor in another way. Instead of writing it with no arguments, we can give default arguments to all of its parameters, so that it can be called with no arguments. For example:

```
class User
{
public:
    User( string fname = "", string lname = "" );
    User( string fname, string lname, Date bday );
    void get_one_record  ( istream & instream );
    void print_one_record( ostream & outstream);
private:
    string firstname;
```

```
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

The first constructor can be called with no arguments, in which case `lname` and `fname` will be assigned empty strings.

The following class interface is illegal because there are two constructors that can be called with no arguments, i.e., two default constructors.

```
class User
{  // ILLEGAL CLASS DEFINITION
public:
    User();                                          // default constructor
    User( string fname = "", string lname = "" );    // default constructor
    User( string fname, string lname, Date bday );
    void get_one_record  ( istream & instream );
    void print_one_record( ostream & outstream);
private:
    string firstname;
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

An important rule to remember is that a class cannot have more than one default constructor.

Initializer Lists

Constructors are different from ordinary functions in another way: they can have a special section called an ***initializer section***, or ***initializer list***, in their definitions. A simple one looks like this:

```
User:User (string fname, string lname, Date bday ):
          birthday(bday), firstname(fname), lastname(lname)
{
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
}
```

An initializer list is a comma-separated list of ***initializations*** introduced by a colon, immediately after the function header and before the function block. *An initialization is not an assignment operation.* It looks a bit like a function call. You write the name of a member variable from the class, followed by a pair of parentheses, between which is the expression that initializes that variable. Its form is

```
member-variable-name(initial-value)
```

In the above example

```
firstname(fname)
```

is an initialization. Its effect is to copy the value of `fname` into `firstname`, as if it were an assignment operation, but in fact it is more generally a call to a special initializer for the variable. You can think of it like an assignment operation, but it is not. In general, the members of a class can be objects of other classes, as we will see later, and the initializer will invoke their constructors when possible. Note that the `birthday` member, which is a struct, has no "constructor" but we can initialize it just the same using

```
birthday(bday)
```

because the compiler creates what is effectively a constructor for it.

Two important thing to remember about initializer lists are that

- they are executed before the body of the constructor,
- the order in which the individual initializations take place is not the order in which they are listed but the order in which they appear in the class's declaration, which is called their ***declaration order***.

The member variables will be assigned initial values in the following order:

- `firstname` gets a copy of `fname`
- `lastname` gets a copy of `lname`
- `birthday` gets a copy of `bday`

We will say more about initializer lists later.

Explicit Constructor Calls

There is another way for constructors to be invoked, using an explicit constructor call. This is something you may occasionally have to do, but it is an advanced idea. For now we just note it in passing. The following is an alternative way to declare and initialize an object of the `User` class:

```
User newuser;
newuser = User( "Sam", "Spade", bday);
```

Let us break down what this means. The object `newuser` is declared in the first line. The default constructor is used to create `newuser`. It has whatever values are given to it by the default constructor. In the second line, the non-default constructor for the `User` class is invoked explicitly, with what looks like a function call (even though I said that you can never do this earlier). *But this explicit call to the constructor is not on any object.* It creates an object with no name, called an ***anonymous object***. That anonymous object is then copied into the `newuser` object.

To confuse things even further, you can invoke a default constructor explicitly, as in

```
User newuser;
newuser = User();
```

This is different from the illegal

```
User newuser(); // remember that this is illegal!!
```

because the empty parentheses follow the class name, not the object name.

Classes with Class Member Variables

Classes can have members of any type. The members can be members of another class. We can modify our example by turning the `Date` structure into a class to demonstrate.

```
class Date
{
public:
    Date ( int year = 1972, int month = 1, int day = 1 );
    void set     ( unsigned int y, unsigned int m, unsigned int d);
    int  getday  ( );
    int  getmonth( );
    int  getyear ( );

private:
    unsigned int year;
    unsigned int month;
    unsigned int day;
};
```

Now `Date` is a class with a single constructor that can be called with no arguments. If it is called with no arguments, the date is set to January 1, 1972. Assume that our `User` class remains the same. It now has a member that is a member of the `Date` class. We are forced to rewrite our member functions because they access the members of the `Date` class, which are now private. The `User` class is not allowed to access the private members of the `Date` class. Just because it contains a `Date` member does not give it this privilege. Its functions are not member functions of the `Date` class.

It has to use the public member functions of the `Date` class to do its work. They now become

```
User::User()
{
    firstname = "";
    lastname  = "";
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
    birthday.set(0,0,0);
}

User:User (string fname, string lname, Date bday ):
           birthday(bday), firstname(fname), lastname(lname)
{
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
}
```

The first constructor calls the `set()` member function of the `Date` class to set the date of its `birthday` member. The second constructor did not change. A very subtle thing is going on in the second constructor. The initializer list

```
birthday(bday), firstname(fname), lastname(lname)
```

has an initialization of the `birthday` member. This has to invoke an initializer for the `Date` class. There is only one user-defined constructor for this class and its prototype looks like

```
Date ( int year = 1972, int month = 1, int day = 1 );
```

If we were calling this constructor, we would have to ahve three integer arguments, but we do not. We just have a Date argument, so this is not the constructor that is used to perform the initialization. What is? The compiler generates an initializer that copies the members of `bday` into `birthday`.

We could also create a third `User` class constructor with the prototype

```
User( string firstname, string lastname, int y, int m, int d );
```

and implement it like this:

```
User::User( string firstname, string lastname, int y, int m, int d ):
      birthday(y,m,d), fname(firstname), lname(lastname)
{
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
}
```

explicitly calling the constructor of the `Date` class on the `birthday` member of the `User` class. Although this makes the `User` class a bit more flexible, it breaks the data abstraction because the `User` class is given values for the private member variables of the `Date` class in its own constructor.

const Member Functions

Every member function of a class falls into one of two categories: either it modifies some data member of the class or it does not. If it does not modify any member of the class, it is called an ***accessor*** function – it retrieves data from the class object but does not modify it. Usually a class will have several accessor functions. If it modifies any part of the class object, it is called a ***mutator*** function.

You can tell the compiler and the reader of the code that a function is an accessor by putting the `const` keyword after the parameter list in the function's prototype as well as in its definition. The `const` keyword used in this way says that the function is guaranteed not to modify the object on which it is called. To illustrate, we will supplement our `Date` class to use the `const` modifier wherever we can:

```
class Date
{
public:
    Date ( int y = 1972, int m = 1, int d = 1 );
    void set     ( unsigned int y, unsigned int m, unsigned int d);
    int  getday  ( ) const;
    int  getmonth( ) const;
    int  getyear ( ) const;

private:
    unsigned int year;
    unsigned int month;
    unsigned int day;
};
```

The three functions that do not change the object are the *getX()* type functions: `getday()`, `getmonth()`, and `getyear()`. They are marked as `const`. We can now modify the `User` class interface as well:

```
class User
{
// The public interface:
public:
```

```
    User();
    User( string firstname, string lastname, Date bday );
    User( string firstname, string lastname, int y, int m, int d );
    string firstname() const;
    string lastname()  const;
    int    highscore() const;
    int    lowscore()  const;
    void   getbirthday( unsigned int &m, unsigned int &d) const;
    void   read_record  ( istream & instream );
    void   write_record ( ostream & outstream) const;

// The private stuff:
private:
    string fname;
    string lname;
    int    scores[5];
    Date   birthday;
};
```

Remember to put the `const` keyword after the header in the function definitions as well.

If you tell the compiler that the `getbirthday()` member function of the `User` class is `const`, but you do not mark the functions that it calls on that `birthday` member as `const`, then the compiler will issue an error message. This is because the compiler assumes that any function not marked `const` may modify the object on which it is called. The `getbirthday()` function looks like this

```
void   User::getbirthday( unsigned int &m, unsigned int &d) const
{
    m = birthday.getmonth();
    d   = birthday.getday();
}
```

If `getmonth()` and `getday()` are not `const` member functions of the `Date` class, the compiler will assume that they modify the `Date` object, `birthday`, on which they are called. Since `getbirthday()` is supposed to be a `const` function, this contradicts the requirement, so the compiler issues an error message.

You should get into the habit of marking every accessor as a const member function.

Static Member Variables

A member of a class, whether a data member or a member function, can be designated as ***static*** by preceding its declaration with the keyword `static`. Static data members are a bit easier to understand so we will start with them. A ***static data member*** of a class is one that is not a part of any of the objects of the class, but is instead a single member that they all share. It is like a global variable in a program except that it is restricted to be "global" among the objects of the class.

To declare that a data member is static, you have to do two things:

1. You must precede its declaration within the class with the `static` keyword (before the type and any other qualifiers such as `const`), and
2. You must put its definition outside of the class definition, in a surrounding namespace scope, without the `static` keyword but with its name qualified by the :: operator.

As this is surely confusing, here is an example.

Example

```
class Myclass
{
public:
    static int counter;             // declares counter to be static
    static const int maxvalue = 10; // a static constant member
    int sum;
};

int Myclass::counter;        // defines Myclass::counter
const int Myclass::maxvalue; // defines Myclass::maxvalue
```

In this example, the variables `counter` and `maxvalue` are static. The declarations within the class are just declarations, not a definitions. (In this example, you cannot see the difference; trust me for now that there is a difference.) Each must be defined outside of the class but within a surrounding namespace scope. The simplest thing to do is to put its definition in the same file, so that `Myclass` and `Myclass::counter` and `Myclass::maxvalue` are defined in file scope (known as the global namespace.)

If we declare a couple of objects of class `Myclass`,

```
Myclass obj1, obj2;
```

and each will have its own copy of `sum`, but they will share `counter`.

```
obj1.sum = 1;
obj2.sum = 2;
obj1.counter = 0;
obj2.counter = 20;
cout << obj1.counter << endl;  // outputs 20
```

Notice that we accessed `counter` using the member access syntax (i.e., using the dot operator) on specific classes. Because counter does not belong to any one object but is really like a part of the class itself, we can also access it using the syntax `Myclass::counter`. So we can also write

```
Myclass::counter = 20;
obj2.counter    += 10;
cout << Myclass::counter << endl;  // outputs 30
```

Does this look a bit familiar? Recall how, when formatting output on an output stream, you used the instructions

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
```

to display numbers as fixed decimals. The `ios::fixed` argument is a static member of the class known as `ios`.

The `maxvalue` member is a constant member. We are allowed to initialize it within the class declaration. We could have also initialized it in the definition ouside of the class definition.

A ***static member function*** is a member function that cannot access the non-static parts of the class. This implies that it can only access the static data members of the class. If multiple objects share a static data member, they each have the ability to update that member. Although this is possible, it is better if they all call a function that is not called on any particular object. This is one use of a static member function. In the above example, we could define a static member function that resets the counter:

```
class Myclass
{
public:
    static int counter;             // declares counter to be static
    static const int maxvalue = 10; // a static constant member
    static void reset();            // a static function that resets counter
    int sum;
};

int Myclass::counter;        // defines Myclass::counter
const int Myclass::maxvalue; // defines Myclass::maxvalue

void Myclass::reset() : counter(0) {}
```

The keyword `static` is not repeated when the function is defined.

A Bit More About I/O in C++

Member Functions of the `istream` Class

The extraction operator is of limited use because it always uses whitespace to delimit its reads of the input stream. It cannot be used to read those whitespace characters, for example. The `istream` class, has an assortment of other methods of input that are useful for different types of tasks. Remember that all input file streams (`ifstream` objects), all `iostream` objects (`cin`, `cout`, and `cerr`), and all input string streams (`istringstream` objects) are types of `istream` objects, meaning that they "inherit" all public member functions (and other public and protected members) from `istream`. This is a very short tutorial on some of the member functions of the `istream` class that may prove to be useful.

The get() Member Functions

The `get()` member function has six overloaded versions, but only four will be discussed here. The first two versions are for reading single characters at a time.

One Character at a Time

The first of these is

```
int get();
```

This extracts a single character from the stream and returns its value cast to an integer. It does not return a `char` because the `char` type is unsigned, and when get fails, it might return -1, depending upon the implementation. `get()` reads every single character: blanks, tabs, newlines, and so on, one at a time. When there are no characters, the `eof` bit is set. This means that you can use this form of `get()` as follows.

```
int main()
{
        char c;
        while ( (c = cin.get()) && !cin.eof()  )
                cout << c;
        return 0;
}
```

This little program will copy the standard input stream to the standard output stream. If you compile this into a command named `copy`, for example, and type

```
copy < oldfile > newfile
```

it will make a copy of `oldfile` in `newfile`.

A second overload has the signature

```
istream& get ( char& ch );
```

This also extracts a single character from the stream, but does not return it. Instead it stores it in its call-by-reference parameter `ch`. Like the first, if the end of file is reached, it sets the `eof` bit and does not alter the value of `ch`. The preceding program could be written using this version of `get()` as follows.

```
int main()
{
        char c;
        while ( cin.get(c) && !cin.eof() )
                cout << c;
        return 0;
}
```

Reading Strings

The two remaining overloads can be used to read data into ***C strings***, ***not C++ strings***! Unlike the versions that read one character at a time, *these reads are delimited*. This means that they read until either they read a certain number of characters or they see a delimiter character, which is by default a newline, and they do not remove that delimiter from the input stream. Because the delimiter is not removed from the input stream, the next time they are called on that stream, they will read nothing because the delimiter is the next character they would read. Therefore, for the next call to read beyond the delimiter, it must be removed by some other means. These functions are really more useful when it is expected that the delimiter *will not be found*, and that finding the delimiter is the exceptional case. The examples will clarify this point.

There are two prototypes:

```
istream& get (char* str, streamsize n );
istream& get (char* s, streamsize n, char delim );
```

The only difference between these is that the first uses the newline character as the delimiter and the second lets the caller choose the delimiter. They both extract characters from the input stream and store them into the C string beginning at `str`. Characters are extracted until either `(n - 1)` characters have been extracted or the delimiting character is found. In other words, if the delimiter is reached before `(n-1)` characters have been read, reading stops, or if `(n-1)` characters have been read but the delimiter has not been reached, reading also stops. In either case the string is padded with a `NULL` byte after the last character. The extraction also stops if the end of file is reached in the input sequence or if an error occurs during the input operation. If `get()` extracts no characters, *it sets the failbit on the stream.*

If the delimiter is found, it is not extracted from the input sequence. The next call to this version of the `get()` function will return nothing, will not remove it, and will set the failbit to true preventing any further input operations on the stream. If you want to force this character to be extracted, you must use the `getline()` member function.

The point of providing a second argument is to prevent the C string from being overflowed. You should always set it to no more than the size of the array pointed to by the first argument.

The number of characters read by any of the previous input operations can be obtained by calling the member function `gcount()`. The `gcount()` function is useful when using the string-reading versions of `get()` because you can test when they have stopped reading anything. The next program will read a single line of text, breaking it up into chunks of size 10 with a `':'` between the chunks.

```
int main()
{
    char str[10];
    cin.get(str, 10);
        while ( cin.gcount() > 0 ) {
        cout << str << ":";
```

```
        cin.get(str, 10);
    }
    cout << endl;
        return 0;
}
```

The following program shows how to use `get()` properly when either stopping criterion may occur: a delimiter was found or (n-1) characters were read. It uses `getline()` to read the delimiter, and it clears the failbit when it reads nothing.

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

#define CHUNKSIZE  14

int main()
{
    char str[CHUNKSIZE];
    char junk[2];
    istringstream iss;
    int i = 0;

    string strdata =
    "12345678901234\n"
    "123456789012345678\n"
    "123456\n"
    "1234567890123\n"
    "123456789012\n"
    "12345678901234567890123456789012345678\n"
    "12345678901\n"
    "123456789012345678901\n"
    "12345678901234567\n"
    "1234567890123\n";

    iss.str(strdata);  // Make iss a stream like a text file with this string

    while ( !iss.eof() ) {
        iss.get(str, CHUNKSIZE);
        // if get() read no characters, it set the failbit, so we clear it
        // Otherwise we print what we got
        if ( iss.gcount() == 0 )
            iss.clear();
        else
            cout << i << "  " << str << endl;

        // If gcount < CHUNKSIZE-1 we found the delimiter, so remove it
        if ( iss.gcount() < CHUNKSIZE-1 )
            iss.getline(junk, 2);
        i++;
    }
    cout << endl;
    return 0;
}
```

The output is the sequence of chunks that are read by successive calls to `get()`. A line of output can never be larger than the chunk size minus one, and will be smaller when the newline is found first. After each call to `get()` we check whether `get()` read nothing. If it did, it set the failbit, and no input will be possible on the stream after that, so we must clear it.

After that we check whether (`CHUNKSIZE-1`) chars were read. If less, it found the delimiter, so `getline()` is used to extract it. Because the last character in the stream is a newline, `getline()` will read it. The next call to `get()` will read nothing, and `iss.gcount()` will be less than (`CHUNKSIZE-1`), causing `getline()` to run again. This time `getline()` will encounter the end-of-file condition and set the eofbit on the stream and the loop will exit.

The getline() Member Functions

The `getline()` functions that are members of the `istream` class are not the same as the global `getline()` function. The ones that are part of `istream` are called on an `istream` object, just like the `get()` functions above, e.g.

```
char str[10];
cin.getline(str, 10);
```

The global `getline()` function is called with a stream object as its argument, and has a C++ string argument:

```
string s;
getline(cin, s, 10);
```

It is easy to get these confused. The distinction, once again:

- ***member function needs C string***
- ***global function needs C++ string***

The `getline()` member functions of the `istream` class are similar to the `get()` member functions that take a C string argument, except that they extract the delimiter and they set the failbit under different conditions. Their prototypes are

```
istream& getline (char* str, streamsize n );
istream& getline (char* str, streamsize n, char delim );
```

Unlike the `get()` functions, these work nicely for reading text files line by line, and they are suitable for any type of reading tasks where the text is delimited and the delimiter must be read and discarded. The only caveat is that the size of the read should be large enough that the delimiter is found within the first (n-1) characters, otherwise you have to deal with the failbit being set. Thus, if you know the line size, for example, is never larger than 200 characters, call it with

```
cin.getline(str, 200);
```

Now the details. Calling

```
input.getline(str, n, delim) ;
```

where `input` is an input stream, `str` is a C string, and `delim` is some character used as a delimiter, will extract characters from the stream until one of three conditions occurs:

1. end-of-file occurs on `input`, in which case it sets the eofbit on the `input` stream;
2. the delimiter `delim` is the next available input character, in which case the delimiter is extracted but not stored into `str`;
3. `n` is less than one or `(n-1)` characters are stored into str, in which case the failbit is set on the `input` stream.

These checks are made in the stated order. This way, if there are `(n-1)` characters followed by a delimiter, it will not set the failbit, because it first finds the delimiter before checking whether `(n-1)` characters have been stored.

The implication of the above is that when you use `getline()`, you have to check whether it stopped reading because it found the delimiter, or it stopped because it read `(n-1)` characters, or it reached end-of-file. The following examples show how to do this.

The first program reads from an input file specified on the command line and outputs a copy of that file on the standard output, i.e., the terminal.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

/* copy from file argument to stdout*/
int main(int argc, char* argv[] )
{
    char buffer[10];
    ifstream fin;

    if ( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " file\n";
        exit(1);
    }

    fin.open( argv[1]);
    if ( !fin ) {
        cerr << "Could not open" << argv[1] << " for reading.\n";
        exit(1);
    }

    fin.getline(buffer, 10);
    while (  !fin.eof()  ) {
        cout << buffer;
        if ( fin.fail() )
            fin.clear();
        else
            cout << endl;
      fin.getline(buffer, 10);
    }
    cout << endl;
    return 0;
}
```

It uses a string of size 10, which we will call `buffer`, to store the characters read by `getline()`. Therefore, we repeatedly call `getline()` with 10 as its `streamsize` argument.

After the appropriate error-checking, it enters a loop in which it checks for end-of-file upon entry, and repeatedly calls `getline()`. It checks whether the failbit was set by the immediately preceding call. If it was not set, this means that `getline()` found the delimiter, so the program prints a newline (since `getline()` does not store it into `buffer`). If the failbit was set, it implies that 9 characters were read but no delimiter was found, and all we have to do is unset the failbit using the `clear()` member function of the `istream` class. It cannot be the case that we reached end-of-file, otherwise we would not be in the loop, so clearing the flags is safe here.

An alternative to the loop in the above listing is the following:

```
    while ( !fin.eof() ) {
        fin.getline(buffer, 10);
        cout << buffer;
        if ( !fin.eof() && fin.fail() )
            fin.clear();
        else
            cout << endl;
    }
    return 0;
```

Instead of calling `getline()` before entering the loop and again at the bottom, we can call it upon entering the loop. But in this case we cannot simply clear the flags if the failbit is set, because the eofbit might be set by the last call to `getline()`. Therefore the condition for clearing is slightly different. Also, we can remove the last output of a newline when the loop ends, since that is now superfluous.

Vectors

Vectors are one of the container class templates defined in the Standard Template Library. To understand vectors, you need to understand template classes.

There are three ways to declare a vector.

Syntax

```
#include <vector>   // must include this file
using namespace std;   // might need this as well

vector<base type> arrayname;            // contains 0 cells
vector<base type> arrayname(size expression);
vector<base type> arrayname(size expression,  value);
        // contains size-expression cells, each initialized to value
```

The expression can be any expression that evaluates to a number. If the number is not an integer, it is truncated.

Examples

```
vector<int>     grades(5,0);  // vector of 5 ints, all 0
vector<string>  trees(50);    // vector of 50 strings
vector<Point>   hexagon(6);   // vector of 6 Points
               // because Point had a default constructor)
```

or, more interestingly:

```
cout << "Enter the number of sides:";
cin  >> n;
vector<Point>  polygon(n);
vector<double> chordlengths(n*(n+1)/2);
```

But,

```
vector<MyClass>   Object(2);
```

will be illegal if `MyClass` does not have a default constructor.

To access an individual element**, use the vector name and an index:**

```
grades[0]  = 100;
cin >> grade[1];
cout << "The grade is " << grade[1];
for (int i = 0; i < 5; i++)
    cin >> grades[i];
```

To initialize a vector **to 0:**

```
for (int k = 0; k < 5; k++)
    grades[k] = 0;
```

To compute the average of the values:

```
sum = 0.0;
for (int k = 0; k < 5; k++)
    sum += grades[k];
average = sum/5;
```

Example

This simulates rolling a pair of dice with `NSIDES` many sides 20,000 times and counts how many times each possible sum (2,3,4,5,..., 2*`NSIDES`) occurs.

```
#include <vector>

// use vector to simulate rolling of two dice
const int NSIDES = 4;
int main()
{
    int sum, k;
    Dice d(NSIDES);     // Dice defined elsewhere
    vector<int> diceStats(2*NSIDES+1);  // room for largest sum
    int rollCount = 20000;

    for (k = 2; k <= 2*NSIDES; k++) // initialize to zero
        diceStats[k] = 0;

    // could have done this at declaration time
    for(k=0; k < rollCount; k++)      // simulate all the rolls
    {   sum = d.Roll() + d.Roll();
        diceStats[sum]++;
    }

    cout << "roll\t\t# of occurrences" << endl;
    for(k=2; k <= 2*NSIDES; k++)
       cout << k << "\t\t" << diceStats[k] << endl;
    return 0;
}
```

vector parameters

Vectors can be passed as parameters to functions.

```
int Sum(const vector<int> & numbers, int length)
{
     sum = 0;
     for (int k = 0; k < length; k++)
        sum += numbers[k];

     return sum;
}

void Shuffle(vector<string> & words, int count)
```

```
{
    RandGen gen;     // for random # generator
    int randWord;
    string temp;
    int k;
    // choose a random word from [k..count-1] for song # k

    for (k=0; k < count - 1; k++)
    {   randWord = gen.RandInt(k,count-1);  // random track
        temp = words[randWord];             // swap entries
        words[randWord] = words[k];
        words[k] = temp;
    }
}
```

Collections and Lists Using `vectors`

A vector's size is not the same as its capacity. Suppose we have

```
vector<string>   trees(8);
```

and we have filled it with 6 tree names as follows.

birch	oak	ebony	cherry	maple	ash		
0	1	2	3	4	5	6	7

The ***capacity*** is 8 but the ***size*** is 6. We don't have to keep track of this in our program if we use the methods of the `vector` class.

The `vector` class has methods of growing itself and keeping track of how big it is.

```
vector::size()               // returns current size
vector::push_back(value)     // adds another value to a tvector
                             // and if it does not have enough
                             // cells it doubles capacity

vector<double>   prices(1000);   // prices.size() == 1000
vector<int>      scores(20);     // scores.size() == 20
vector<string>   words;          // words.size() == 0;

words.push_back("camel");   // size() == 1, capacity() = 2
words.push_back("horse");   // size() == 2, capacity() = 2
words.push_back("llama");   // size() == 3, capacity() = 4
words.push_back("okapi");   // size() == 4, capacity() = 4
words.push_back("bongo");   // size() == 5, capacity() = 8
```

`size()` always returns current size, not the number of elements added by `push_back`. If a vector is initially size 0, and `push_back` is used exclusively to grow it, `size()` will return the number of elements pushed onto it.

`vector::reserve(`*`size expression`*`)`
`//`**allocates an initial capacity but keeps size at 0**:

```
vector<int>   votes;
votes.reserve(32000);       size() == 0    but capacity == 32000
vector<int> ballots(32000) size() = 32000 and capacity == 32000
for (int i = 0; i < 100; i++){
    cin >> x;
    votes.push_back(x);
} // what is capacity now?
```

Vector Idioms: Insertion, Deletion, Searching

Typical operations in data processing are:

- insert into a vector (or array)
- delete data from a vector
- search a vector for data

Building an unsorted vector

```
for (int i = 0; i < 100; i++){
    cin >> x;
    v.push_back(x);
}
```

or, reading from a file:

```
vector<double>  v;
ifstream fin;
fin.open("inputdata.txt");
double x;
while ( fin >> x ) {
   v.push_back(x);
}
```

The data are in the order read from the file now.

Deleting from a vector using pop_back()

The `pop_back()` member function of the vector class deletes the last element of a vector and reduces the size by 1. It does not affect capacity. E.g., assume

```
vector <double>  v(5)  contains 8,4,2,10,3

v.pop_back();   =>    8  4  2  10
```

```
v.pop_back();    =>    8  4  2
v.pop_back();    =>    8  4
```

If the vector is unsorted, deletion from position `pos` is easy. We overwrite the item in position `pos` by copying the last element into `v[pos]`, then we delete the last element with `pop_back()`:

```
int lastIndex = v.size() - 1;
v[pos] = v[lastIndex];
v.pop_back();
```

Searching an unsorted vector (linear search)

To search an unsorted vector it is necessary to look through the entire vector. To look for the cell with the value key:

```
int k;
for (k = 0; k < v.size(); k++){
    if ( v[k] == key )
        break;
}
if (k < v.size())
    // not found
```

Or, the function:

```
void LinSearch(const vector<double> & v, double key, int & loc)
{
    int k;
    for (k = 0; k < v.size(); k++){
        if ( v[k] == key )
            loc = k;
            return;
    }
    loc = -1;
}
```

Sorted vectors

Vectors can be built in sorted order by inserting data in the right position during creation. This makes later searching faster but makes creation a little slower.
Idea:

```
while there is more data available
    read the next data item
    let k be the index of the largest element of
       the vector that is smaller than the item
    put this data item into position k+1, shifting
       all larger elements of the vector up one cell
```

This is one way to do it. The author does it slightly differently. To be more precise declare

```
vector<double>  sortedNums;
double s;
while there is more data available
    let count =  sortedNums.size(); //current size of vector
    cin >> s;  //read the next data item s
    sortedNums.push_back(s)  //push onto the end of vector
                             // now it has count+1 items
    let k = count;
    while ( 0 < k && s <= sortedNums[k-1]) {
       sortedNums[k] = sortedNums[k-1];
       k--;
    }
```

Use the example data

4.5 10 6.3 3.0 1.0

Suppose we have a sorted vector with some large number of items. To delete the item at index n, 0 <= n < size(), we can shift items n+1 to size()-1 down 1 and delete the last:

```
for ( k = n; k < a.size()-1; k++)
     a[k] = a[k+1];
a.pop_back();
```

More generally, a function to delete an item from an `int` vector

```
void  delete( vector<int> & a, int p)
{
    int k;
    if ( (p < 0) || (a.size() <= p))
         return;

    for ( k = p; k < a.size()-1; k++)
         a[k] = a[k+1];
    a.pop_back();
}
```

Searching a sorted vector

If a vector is sorted we can use more efficient method called binary search.

Binary Search

```
int bsearch(const vector<string>& list, const string& key)
// precondition: list.size() == # elements in list
// postcondition: returns index of key in list, -1 if key not found
{
    int low = 0;                    // leftmost possible entry
    int high = list.size()-1;       // rightmost possible entry
    int mid;                        // middle of current range
    while (low <= high)
```

```
    {
        mid = (low + high)/2;
        if (list[mid] == key)        // found key, exit search
        {   return mid;
        }
        else if (list[mid] < key)    // key in upper half
        {   low = mid + 1;
        }
        else                         // key in lower half
        {   high = mid - 1;
        }
    }
    return -1;                       // not in list
}
```

Example

Search for each of: `"ash"` `"kapok"` `"elm"` in

ash	birch	cherry	dogwood	ebony	imbuya	kapok	maple

Pointers and Dynamic Variables

Motivation

Sometimes when you write a program you do not know at compile-time how large an array needs to be, or how many instances of a class you might need because it depends what happens when the program is running. For example, imagine a program that asks a user interactively to enter some data that must be stored in an array, but the program does not know how many items will be entered. If the array is declared of fixed size at compile time, either it can be too small or too large, depending on how much data is entered. C and C++ have (different) instructions for solving this problem. We look at C++'s solutions here.

Pointers

A pointer variable contains a memory address as its value. Normally, a variable contains a specific value; a pointer variable, in contrast, contains the memory address of another object. We say that a pointer variable, or pointer for short, ***points to*** the object whose address it stores.

Pointers are declared to point to a specific type of object and can point to objects of that type only. The asterisk `*` is used to declare pointer types.

```
int*    intptr;    // intptr can store the address of an integer variable
double* doubleptr; // doubleptr can store the address of a double variable
char*   charptr;   // charptr can store the address of a character
```

We say that `intptr` is "a pointer to int" and that `doubleptr` is "a pointer to double." Although we could also say that `charptr` is a pointer to char, we say instead that `charptr` is a string. Pointers to `char` are special.

The * is a *pointer declarator operator*. In a declaration, it turns the name to its right into a pointer to the type to the left. It is a *unary, right associative* operator. It does not matter whether you leave space to the right or left of it, as demonstrated below. These three are equivalent:

```
int* intptr;
int * intptr;
int *intptr;
```

In general, if `T` is some existing type, `T*` is the type of a variable that can store the address of objects of type `T`, whether it is a simple scalar variable such as `int`, `double`, or `char`, or a more structured variable such as a structure or a class:

```
struct Point { int x, int y };
Point* Pointptr;
```

You can even write a declaration like this:

```
struct Spline
{
    int x;
    int y;
    Spline *nextpoint;
};
```

which we will not even attempt to explain here (since it is a topic for the sequel to this class). What do you think it defines?

Pointers are even more general than this. The following are all valid declarations that declare pointers to many different kinds of things.

```
int  *intp;        // pointer to an integer memory cell
char *pc;          // pointer to char, also known as a C string
int * N[10];       // N is array of pointers to int (NOT POINTER TO ARRAY OF INTS)
int (*B)[10];      // B is a pointer to an array of 10 ints
int* f(int x);     // f is a function returning a pointer to an int
int  (*pf)(int x); // pf is a pointer to a function returning an int
int* (*pg)(int x); // pg is a pointer to a function returning a pointer to an int
```

Notice the difference in how an array of `int*` is declared versus a pointer to an array of `int`s, and how a pointer to a function is declared versus a function that returns pointers to other things.

The `typedef` statement is useful for declaring types that are pointers to things:

```
typedef int* intptr;
intptr  intp, intq;       // p and q are both pointers to int
intptr  func( intptr np); // func returns a pointer to an int
```

This is even more interesting:

```
typedef bool (*MessageFunc) (char* mssge);
// This defines a MessageFunc to be a pointer to a function with a
// null-terminated string as an argument, returning a bool.
```

A `MessageFunc` is a pointer to a function. This makes it possible to pass one function to another, as in the following `handle_error()` function, which calls `f()` with a message and either exits the program or not depending on the boolean value returned by `f()`.

```
void handle_error( MessageFunc f, char* err_mssge)
{
    if ( f(err_mssge) )
        exit(1);
}
```

The address-of operator, `&`, can be used to get the memory address of an object. It is the same symbol as the reference operator, unfortunately, and might be a source of confusion. However, if you remember this rule, you will not be confused:

If the symbol "`&`" appears ***in a declaration***, it is declaring that the variable to its right is a reference variable, whereas if it appears ***in an expression that is not a declaration***, it is the address-of operator and its value is the address of the variable to its right.

```
int x = 200;
int y = 100;
int &ref = x;  // a declaration, so & is the reference operator
int *p   = &x; // appears on right of =, so it is in an
               // expression and is address-of operator; p stores address of x
p = &y;        // address-of operator; now p stores the address of y
int &z = y;    // reference operator; z is an alias for y
```

Dereferencing Pointers

Accessing the object to which a pointer points is called ***dereferencing the pointer***. The "*" operator, when it appears in an expression, dereferences the pointer to its right. Again we have a symbol that appears to have two meanings, and I will shortly give you a way to remain clear about it.

Once a pointer is dereferenced, it can be used as a normal variable. Using the declarations and instructions above,

```
p = &y;
cout << *p;         // p contains address of y, which stores 100,
                    // so 100 is printed. p has been dereferenced.
cin >> *p;          // p is dereferenced before the input operation.
                    // *p is the variable y, so this stores the input into y.
*p = *p + *p;       // just like y = y + y
*p = *p * *p;       // just like y = y * y
(*p)++;          // * has lower  precedence than the post-increment ++
                    // so this increments *p which is y so y =  101
int  a[100] = {0}; // a is an array of 100 ints, initially all 0
int *q = &a[0]; // q is the address of a[0]
q++;                // q is address of a[1]
*q  = 2;         // now a[1] contains  2
cout << *q++;       // increment q (so q is address of a[2]) and print a[2]
```

The dereference operator and the address-of operator have higher precedence than any arithmetic operators, except the increment and decrement operators, `++` and `--`.

If you think about a declaration such as

```
int * p;
```

and treat the `*` as a dereference operator, then it says that `*p` is of type `int`, or in other words, `p` is a variable that, when dereferenced, produces an `int`. If you remember this, then you should not be confused about this operator.

Relationship Between Pointers And Arrays

An array name is a constant pointer; i.e., it is pointer that cannot be assigned a new value. It always contains the address of its first element.

```
int list[5] = {2,4,6,8,10};  // list is the address of list[0]
cout << *list << endl;       // prints 2 because *list is the contents of list[0]

int * const aPtr = list;     // aPtr has the same address as list
                             // &list[0] is stored in aPtr.
aPtr[2] += 10;               // adds 10 to  list[2]
```

The fact that array names are essentially constant pointers will play an important role in being able to create arrays of arbitrary size while the program is running, rather than by declaring them statically (at compile time).

This is all we need to know for now in order to be able to talk more about C strings and the C string library.

Classes, Structures, and Pointers

There is a special notation that is used when a dereference operator is combined with a member access operator. Suppose we have the following declarations:

```
struct Point
{
    double x;
    double y;
};

Point * pptr;
```

and we dynamically allocate a `Point` variable using `pptr` :

```
pptr = new Point;
```

Then to access the `x` member of the `Point` pointed to by `pptr`, we could write either of these:

```
(*pptr).x
pptr->x
```

They are equivalent. The ***arrow operator*** `->` dereferences the pointer to its left and accesses the member of the structure or class to which it points to its right, so `pptr->x` is a shortcut for `(*pptr).x`.

The same thing is true for classes. Given

```
class CPoint
{
public:
    void print(ostream & out) const;
private:
    double x;
    double y;
};

CPoint * cpptr;
```

we would write

```
cpptr->print(fout);
```

to call the `print()` member function of the class on the dynamic object pointed to by `cpptr`.

The `this` Pointer

C++ defines a special pointer variable that is part of every class, called `this`. The `this` pointer points to the object on which a function is being called. In essence it points to the object itself. A few examples will make this clearer.

```
class Simple
{
    public:
        Simple()
        Simple * addr()
        void set(double x )
        double val()
    private:
        double x;
};

Simple*  Simple::addr()
{
    return this;  // return a pointer to the object itself
}

void Simple::set( double x )
{
    this->x = x;
}
```

In this first example, the `addr()` member function returns a pointer to the object on which it is called, so we can use it as follows:

```
Simple s1;
Simple s2;

cout << "The address of s1 is " << s1.addr() << endl;
cout << "The address of s2 is " << s2.addr() << endl;
```

and the output might look like

```
The address of s1 is 0xbfc03718
The address of s2 is 0xbfc03710
```

The `set()` member function is poorly designed because the parameter has the same name as the private variable. Without a `this` pointer we could not write its definition. But with it, we can write the assignment

```
this->x = x;
```

so that we can tell that the left-hand side is the private member of the object and the right-hand side is the parameter.

Dynamic Memory Allocation

Variables declared with block scope are created on what we call the ***runtime stack***. They get created when the program's execution enters the block and they get destroyed when it leaves that block. Global variables, i.e., variables with file scope, get created when the program first starts up and they get destroyed when the program terminates. You have no control over the lifetimes of either of these types of variables.

C++ provides an operator named `new` that lets you create a new, nameless variable while the program is running. The simplest use of it looks like this:

```
int * p;
p = new int;
```

The `new` operator has a single argument which is the name of an already defined type. When it is executed, it creates an object of this type. The value of the expression

```
new int
```

is the address of the new object of type `int`. This can be assigned to a pointer of type `int`. We can say informally that `new` "returns" the address of an `int`, even though this is not an accurate statement, because `new` is not a function and it has no return value. A variable that si created by the new operator is called a ***dynamically allocated variable***, or just a ***dynamic variable***, because it is created at runtime by the program.

More generally

```
T * p = new T;
```

creates a new dynamic variable of type `T` and assigns its address to the `T` pointer `p`.

If the type is the name of a class, the constructor of that class is invoked. For example,

```
Class MyClass
{
public:
    MyClass();
    MyClass( int x, int y);
    // other stuff here
};
MyClass *cptr = new MyClass;
```

creates an instance of `MyClass`, causing the default constructor to be invoked, and assigns its address to `cptr` and

```
MyClass *c2ptr = newMyClass(4,5);
```

creates a second instance of `MyClass`, invoking the non-default constructor.

You can also use the initializer notation with elementary types, like this:

```
int * p = new int (100);
```

which creates a new variable of type `int` with initial value 100.

More About Memory Management

Block scope variables "live on the stack" and file scope variables are allocated in a special part of your program's memory when the program starts up. Variables created with the `new` operator come from a part of the memory called the ***heap*** or ***freestore***. The heap is of finite size. If you allocate too much of it the program will terminate. It is pretty big and most programs that are written properly do not run out of heap space. Programs that have memory-related errors may run out of heap space. This program will use up its heap store on a 32-bit machine:

```
int main()
{
    for ( int i = 0; i < 100000000000; i++ )
        int *p = new int;
    return 0;
}
```

because it asks for 400 billion bytes of heap, and a program's heap on a 32-bit machine is not that large. When `new` cannot allocate memory, it causes the program to exit[1]. With older C++ compilers, `new` returns a `NULL` pointer instead of making the program terminate.

It is very important that you assign the address of the new variable to a pointer. If you do not, your program will have created a chunk of memory that is completely inaccessible. When it does this, it is just wasting the machine's memory. That memory cannot be reused until the program terminates. If you reassign the value of a pointer to a new dynamic variable you also create unusable memory.

```
int *p;
p = new int (10);
p = new int (20);
```

The first dynamic variable with value 10 cannot be accessed after `p` is assigned the address of the second one. It is an example of a ***memory leak***.

To enable you to return memory that you no longer need, C++ has a `delete` operator. The `delete` operator has a single argument – the pointer whose memory is to be deleted. It deletes the dynamic variable pointed to by that pointer:

```
delete p;
```

deletes the dynamic variable pointed to by `p`.

Since `p` contains just an address, how does the runtime system know how many bytes of memory should be deleted? Remember that pointers are declared to point to specific types of objects. If `p` points to a 4 byte `int`, 4 bytes starting at `p` will be deleted. If it points to an 8 byte `double`, 8 bytes will be deleted.

If you do something like this:

```
int x;
int *p = &x;
delete p;
```

your program will crash, because `x` is a block scope variable, not a dynamic variable, and the machine will be very unhappy about your trying to delete what `p` points to.

This type of code is also a major problem:

```
int *p1, *p2;
p1 = new int(10);
p2 = p1;
// do stuff ...
delete p1;
cout << *p2 ; // get junk!
```

When the memory pointed to by `p1` is deleted, since `p2` also points to it, it now points to junk. The address it contains is meaningless, and trying to dereference it is equally meaningless. We say that p2 is a ***dangling pointer***. Dangling pointers are a common programming error.

[1] It throws an exception that is unhandled unless you specifically handled it. Exception handling is covered in CSci 235.

Dynamic Arrays

A ***dynamic array*** is an array that is created on the heap using the `new` operator. The advantage of a dynamic array over a static array is that its size can be determined at runtime. The syntax is slightly different when creating dynamic arrays:

```
int * q;
q = new int[100]; // 100 ints are allocated and q points to the start of the array
```

The general form of a statement to dynamically allocate an `N` element array of type `T` is

```
T * p = new T[N];
```

Notice that what comes after the new token is the type name followed by the square brackets with the expression in square brackets. If we allocated q dynamically as above we can use it as if it were an ordinary array, as follows:

```
for (int i = 0; i < 100; i++)
    q[i] = i*i;
```

Now `q` can be treated like the name of an array, but it is different, because unlike an array name, `q` is not a `const` pointer. It can be assigned to a different memory location. Note that `*q` is the first element of the array, i.e., `q[0]`, and `*(q+1)` is the same as `q[1]`. More generally, `*(q+k)` is the same as `q[k]`.

To delete the memory associated with a dynamic array, you have to use a slightly different form of the `delete` operator:

```
delete[] q;  // or delete [] q;
```

The square brackets go between the `delete` and the pointer name. You do not put anything in between the square brackets.

```
delete   q[];   // WRONG
delete [100] q; // WRONG
delete    q;    // WRONG
```

A few examples will illustrate. The first asks the user to enter a string and stores it in an array of fixed size. It then allocates an array of twice the size and puts two copies of the string into it.

```
#include <iostream>
#include <cstring>
using namespace std;

int main( )
{
    char input[100];

    cout << "Enter a string:";
    cin >> input;
    int len = strlen(input);
    char * ditto = new char[2*len];

    strcpy(ditto, input);
```

```
    strcat(ditto, input);
    cout << ditto << endl;
    delete [] ditto ;

    return 0;
}
```

The next program shows how a function can create a dynamic array and return a pointer to it.

```
#include <iostream>
#include <cstring>
using namespace std;

int * doublesize ( int a[], int size )
{
    int * p = new int[size*2];
    for ( int i = 0; i < size; i++ )
        p[i] = a[i];

    return p;
}


int main( )
{
    int n;
    int *squares, *moresquares;

    cout << "How many squares to create?";
    cin >> n;
    if ( n <= 0 || n > 10000 )
        return 1;

    squares = new int[n];
    for ( int i = 0; i < n; i++ )
        squares[i] = i*i;

    p2 = doublesize(squares, n);

    for ( int i = n+1; i < 2*n; i++ )
        p2[i] = i*i;

    for ( int i = 0; i < 2*n; i++ )
        cout << p2[i] << " ";
    cout << endl;

    delete [] squares;
    delete [] p2;

    return 0;
}
```

Binary Search

Binary search is a method of searching a sorted array. Assume that the array is named `list` and that it has N elements. The elements are `list[0]`, `list[1]`, `list[2]`, ..., `list[N-1]`. The array is being searched to see if it contains a particular value. That value is called the ***search key***. For example, if the array contains names of animals, and we are searching for the word "gorilla", then "gorilla" is the search key.

The idea is to compute the middle position, middle = N/2, and compare `list[middle]` to the search key. If it is bigger, then the key must be in the lower half, so we repeat this procedure in the lower half. For example, if `list[middle]` contains "leopard", then if "gorilla" is in the list at all, it must be somewhere between `list[0]` and `list[middle-1]`. So we repeat the same algorithm on the array that starts at 0 and ends at middle-1.

Summarizing, if `list` is the array being searched, N is the size of the array, and `key` is the item being sought, binary search is of the form

```
start of search region = 0;
end of search region   = N-1;
while key is not found in list and the search region is not empty
    middle = (start of search region + end of search region)/2
    if key is smaller than list[ middle ]
        search the lower half
    else if key is larger than list[ middle]
        search the upper half
    else
        key is found at position middle
```

This can be refined as follows:

```
int first = 0;
int last  = N-1;
bool found = false;

while ( !found and (last - first >= 0 ) ) {
    int middle = (first + last)/2;
    if (key < list[ middle ] )
        last = middle - 1;
    else if key > list[ middle]
        first = middle + 1;
    else
        found = true;
}
if ( found )
    return middle;
else
    return -1;
```

This can be packaged into a function and tidied up a bit to make it a little more efficient, but the idea is still the same. We can eliminate the Boolean variable `found` since it is redundant. We can return from within the loop if we find our key.

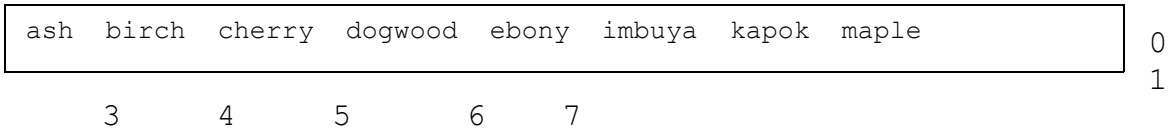
```
int bsearch(const element list[], int size, const element & key)
// precondition: size == # elements in list AND list is sorted
// postcondition: returns index of key in list, -1 if key not found
{
    int low = 0;            // leftmost possible entry
    int high = size-1;      // rightmost possible entry
    int mid;                // middle of current range
    while (low <= high) {
        mid = (low + high)/2;
        if ( key < list[mid] )        // key in lower half
            high = mid - 1;
        else if ( key > list[mid])    // key in upper half
            low = mid + 1;
        else                          // key == list[mid]
            return mid;

    }
    return -1;                      // not in list
}
```

We can analyze the performance of this algorithm. Suppose `list` has N elements and N is a power of 2, i.e., $N = 2^k$. Each time that the while loop is executed, the size of the list being searched is cut in half. Either `high` is reduced to `mid-1` or `low` is increased to `mid+1`. Therefore the size of the sub-array being searched goes roughly from 2^k to 2^{k-1} to 2^{k-2}, and so on, until it becomes size 4, then 2, then 1, and then 0 in the worst possible situation. (Actually, it is slightly less than this each time, because we are removing the middle element each time.) So in the worst case, it loops $\log_2 N$ times. In other words, the total number of comparisons in which the key is compared to an element of the list is at most $2 * \log_2 N$ (twice in each loop iteration). This is MUCH smaller than the time used by linear search.

Example

If we search for "kapok" in



We first compare "kapok" to "dogwood" because 3 = (0 + 7)/2. Since "kapok" is larger, we set first = 4 and compare "kapok" to "imbuya" because (4+7)/2 = 11/2 = 5. Since it is still larger, we set first = 6 and we compare "kapok" to "kapok" because (6+7)/2 = 13/2 = 6, and that is where we find the key.

Lab 1 - Working in the Terminal and More

Getting Used to the Linux computers in the Lab 1000G.

In CSci 127, you should have learned the basics of getting around on the Department's Lab 1000G Linux systems. This tutorial assumes that you know at the very least how to login and logout when in Lab 1000G, and how to use the Linux desktop environment, including its menus and file browsers. (Our lab uses Fedora Linux, which defaults to the Gnome desktop, but you might have learned another such as KDE.) When you are not in the lab but connect remotely to the Linux hosts, you will have no choice but to work within a ***terminal window***. Even when you are in the lab, you will have to do much of the work within a terminal window. Therefore, the main purpose of this tutorial is to teach you the basics of working within terminal windows.

Why, you may be wondering, do you have to work in a terminal window? There are two reasons.

- UNIX systems, which includes Linux systems, support the ability for someone who is in a remote location to login to their UNIX account. For example, you can connect from your home computer to a computer in the lab and work on your files on that lab computer from home. However, for reasons of efficiency, the only way that you can do this is to work within a terminal window. If you are at home and try to run an application on a CS lab machine that uses the full graphical user interface (GUI), everything will "crawl"[1]. Therefore, to be able to work from home or work or any other remote location, you need to know how to use the command-line interface in a terminal.
- When you are in the lab and you want to work on C++ programs, you will need to use the command-line tools for compiling and running your program. All of your programs will be run by typing their names on the command line because you will not learn in this course or any of its sequels how to design programs that create their own graphical user interfaces. So at the very least you will need to work in a terminal to run your programs. Furthermore, many of the tools for working on software are command line tools. The most basic tool is, of course, the compiler/linker. We will be using the GNU compiler collection for this purpose.

How can you open a terminal window when you are in the lab?

- After you have logged in and are looking at your desktop, you can open a terminal in one of two ways. There may be a Terminal icon on the toolbar. In this case you can click it. If not, then open the *Applications* menu, select *System Tools*, and select the *Terminal* tool that is displayed in its sub-menu. If you do not have a *Terminal* icon in your menu-bar, you can drag it to your toolbar for easy access in the future.

Once you have an open *Terminal* window, you can start entering Linux commands in that window.

You should be aware that, although you are working on a computer that is physically in the lab, all of the files that you will work with are actually stored on the CS Department file server, whose name is *biocs*[2]. Each CS computer, including *eniac*, remotely mounts the *biocs* file system containing your home directory. This makes it possible for you to access your files from any computer in the CS network. Although you may not understand how this works, just trust it for now. Later we will explore commands related to file systems and mounting, and some of it will be demystified.

[1] The reason for this is simply that the amount of data that must be passed back and forth with each mouse movement, mouse click, and so on is generally in the range of tens of thousands of bytes or more, whereas in a terminal it is dozens of characters that get transferred back and forth.

[2] Its full name is *biocs.geo.hunter.cuny.edu*.

What if you cannot login to the computer in the lab?

- To login to any computer in the lab, you will need to have a Computer Science Department account. You should have been given that account at the start of the semester. Use your account's username and password in the dialog boxes on the Linux startup screen. If you have problems logging in, talk to your instructor or email the lab administrator, Tom Walter at twalter@hunter.cuny.edu with the description of your problem.

Some Rules When Working in the Lab

- It is very important that you ***do not turn off the computer*** on which you are working! Other students or instructors may be logged in to these machines remotely. If you turn off your computer they lose their work! If something goes wrong and the machine crashes or becomes unresponsive, do not turn it off, put a sign on it, saying that it is out of order, login to another machine and email Tom Walter describing what happened (twalter@hunter.cuny.edu).
- Never unplug any machines! If you want to use your laptop in the lab, that is fine, but you will have to run it on its battery.
- Never unplug any cables in general, and in particular, do not unplug a machine's Ethernet cable! If you want to connect to the network from your laptop while in the lab, use its wireless card.

Accessing Your Account From Other Locations

From anywhere outside of the CS network, the only way to access any CS machine is by logging in to the CS *gateway* machine, which is named *eniac*. Its full name is *eniac.geo.hunter.cuny.edu*. To login to *eniac* remotely you will need an `ssh` client (for working on the lab machines and/or *eniac* remotely) and / or `sftp` client (for transferring files to and from your CS lab account). There are many free clients available on the Internet; here are a few:

- SSH Client 3.2.9, Windows, SSH and SFTP client (link via my website)
- PUTTY, Windows, SSH client
- FileZilla, cross-platform, SFTP client
- Apple systems should come with SSH client built in, you may need to enable it
- Linux systems come with SSH client built in, and many SFTP clients.

iIf you never used `ssh`, read my tutorial *"SSH, SFTP, and Remote Logins"*.

Assuming that you have a Computer Science Department account and an ssh client, you can login to *eniac*. Once you login to *eniac*, you *must* login to one of the lab machines (e.g., `ssh cslab10`). The names of the lab machines are *cslab1, cslab2, ..., cslab28*. It does not matter which one you use; all of your files are accessible identically from every host, and they are all the same type of machine (same computing power, memory, etc.) Just pick a number you like.

Getting around in Unix/Linux

Passwords

If you just got a new account on the system, the first thing you must do is change your password. *Always*. *No exceptions*. If someone intercepted the email that contained your password, they can access your account and even change your password so that you cannot get into it!

To change the password, type the command `passwd` at the prompt in the terminal window. Note that it is 'password' without the 'or'. You will be asked to enter the current password and then asked for the new password, twice. If you make any mistakes, the password will not be changed. The prompt will always be denoted by the dollar sign `$` in these notes.

```
$ passwd
Changing password for user sweiss.
(current) UNIX password: you type here
New UNIX password: you type here
Retype new UNIX password: you type here
passwd: password has been changed.
$
```

If you give a poor password, it will not accept it. You must enter a strong password, which is one that contains:

- at least one uppercase letter
- at least one lowercase letter
- at least one digit
- preferably some punctuation mark such as a hyphen, period, underscore, and so on.

What does a command look like in UNIX?

First of all, when you use a terminal, you are really providing input to a special program generally called a ***command line interpreter***, which reads your entered command, interprets what it means, and then executes it. In UNIX these command line interpreters are called ***shells***, and on most Linux systems, the specific shell that people use is called `bash`. `bash` is short for "Bourne-again shell."

In all shells, a simple command is of the form

```
$ commandname command-options argument1 argument2 ... argumentn
```

The command is executed only after you type a newline character. The command has three parts: the command name, command options, which are, as the name implies, optional things you can give to the command, and arguments. Many commands require no arguments. Examples of simple commands:

`date`	display the current date and time
`who`	list who is currently using the computer
`man <command>`	display the online manual page for the given command

This last command is very important – it is how you can learn all about a command whose name you know. Remember it. In fact, try typing the commands

```
$ man who
```

and

```
$ man date
```

Then type

```
$ man man
```

to learn about the `man` command itself!

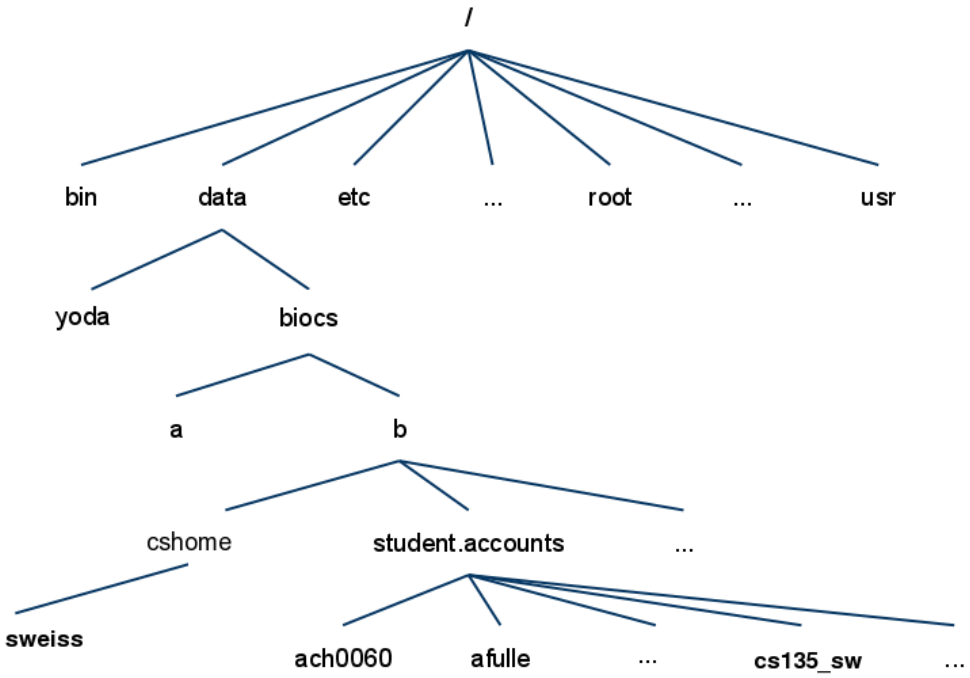


Fig. 1: Partial directory structure on *eniac*.

Navigating the File System

Whenever you are logged into a Unix/Linux system, you have a unique, special directory called your ***current*** or ***present working directory*** (***PWD*** for short). The present working directory is where you "are" in the file system at the moment, i.e., the directory that you feel like you are currently "in". This is more intuitive when you are working with the command line, but it carries over to the GUI as well.

Many commands operate on the *PWD* if you do not give them an explicit argument. We say that the *PWD* is their *default argument*. (Defaults are fall-backs – what happens when you don't do something.) For example, when you enter the "`ls`" command (list files) without an argument, it displays the contents of your *PWD*. The dot "." is the name of the *PWD*:

```
$ ls .
```

and

```
$ ls
```

both display the files in the *PWD*; the first command because the name of the directory ("`.`") is provided, the second because it is the default directory for `ls`.

When you first login, your present working directory is set to your ***home directory***. Your home directory is a directory created for you by the system administrator. It is the top level of the part of the file system that belongs to you. You can create files and directories within your home directory, and usually almost nowhere else. Usually your home directory's name is the same as your username.

In Unix/Linux, files have ***filenames*** and ***pathnames***. The filename is the name of the file, i.e., the name listed when you type the `ls` command. You might name one of your files, `hwk1.cpp` for example, and its filename would be `hwk1.cpp`. Someone else might also create a file with that exact same name. How can they be told apart? The answer

is by their pathnames. A pathname is a unique name for the file. It specifies the location of the file in the file system by its position in the hierarchy. The POSIX.1-2008 standard specifies a system-dependent limit on the number of characters in a pathname (including the terminating null byte).[3].

There are two types of pathnames: ***absolute*** and ***relative***. An absolute pathname is a pathname that starts at the root. It begins with a "/" and is followed by zero or more filenames separated by "/" characters. All filenames except the last must be directory names. For example, `/data/biocs/b/student.accounts/cs135_sw` is the absolute pathname to the directory that we will use in class, as is `/data/biocs/b///student.accounts////cs135_sw`. The extra slashes are ignored. Observe that UNIX (actually POSIX) uses slashes, not backslashes (as in Windows), in pathnames: `/usr/local`, not `\usr\local`.

The image in Figure 1 shows a partial directory structure on biocs. Your home directory is located in `student.accounts` which is in a directory called `b` which in turn is in a directory called `biocs`, which in turn is in `data` which is at the root level of the file system. The absolute pathname to the home directory of `jbgoode` would be `/data/biocs/b/student.accounts/jbgoode`.

A relative pathname is a pathname to a file relative to the PWD. It never starts with a slash! If the current working directory of `jbgoode` is his home directory, and this contains a directory named `cs136`, which contains another directory called `lab01` which contains the file called `lab01_solution.pdf`, then the relative pathname of that file from his PWD is

```
cs136/lab01/lab01_solution.pdf
```

and its absolute pathname is

```
/data/biocs/b/student.accounts/jbgoode/cs136/lab01/lab01_solution.pdf
```

Dot-dot or **..** is a shorthand name for the parent directory. The parent directory is the one in which the directory is contained. It is the one that is one level up in the file hierarchy. If my PWD is currently `/data/biocs/b/student.accounts/cs135_sw/`, then

`..` is the directory `/data/biocs/b/student.accounts`

and

`../../` is the directory `/data/biocs/b/`.

Tilde, or `~`, is a shorthand for your home directory.

```
$ ls ~
```

displays the contents of your home directory. If your username is `jbgoode`, then `~jbgoode` is also a shorthand for your home directory.

Simple Versions of Unix/Linux commands

Listed below are the most basic UNIX commands related to navigating around the file system. Only the simple forms of each are listed. Many have more complex forms. You can learn more about them using the `man` command.

`ls`	list content of the current working directory
`ls dir_name`	list content of the directory named `dir_name`
`cd dir_name`	`cd` stands for change directory, changes the current working directory to `dir_name`
`cd ..`	move one directory up in the directory tree
`cd`	change the current working directory to your home directory

[3] The variable `PATH_MAX` contains the maximum length of a string representing a pathname. On many Linux systems it is 4096 bytes.

`pwd`	print the name of the present working directory
`cp file1 file2`	copy `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names
`mv file1 file2`	move `file1` into `file2`, where `file1` and `file2` can be either relative or complete path names
`rm file`	remove a file (there is no undoing it, so be very careful!)
`mkdir path`	make a new directory at the specified `path`
`rmdir path`	remove the *empty* directory specified by the `path` (there is no undoing it, so be very careful!)

Class Directory

I will put code, tutorials and labs that you need to access in the directory

```
/data/biocs/b/student.accounts/cs135_sw
```

Do not be bothered about why it is `cs135_sw` and not `cs136_sw`. There are so many students who are in both classes that it is easier just to use a single directory for both. The 135 and 136 classes will share materials.

If you don't want to worry about remembering the path to this directory you can make a link to it in your home directory. To do that, first navigate to your home directory and then create the link:

```
$ cd
$ ln -s /data/yoda/b/student.accounts/cs135_sw cs135link
```

Then, you can always navigate to it by typing

```
$ cd ~/cs135link
```

This directory has several subdirectories, including one named labs and one named tutorials. Files placed in these directories can be copied into your own directories so that you have your own private copies of them. It is a good idea to copy assignments and tutorials when they become available. There will be code examples as well, and these should also be copied. You have only read permission on the files inside this directory, so if you ever need to edit something, you should copy the files first to your own directory (you can do so from the command line, or simply drag files from one directory to another in the gui - of course the second option is not available when you are logged in remotely).

Compiling Your C++ Programs

One of the things that you going to do over and over again in this class is compiling and running your code. We will use the GNU C++ compiler, named g++, for this purpose. I have written a more detailed tutorial on the GNU C++ compiler, which is on my website; see *A Tutorial and Brief Summary of GCC*.

The exercise portion of Lab01 goes through several examples of how to use g++ to compile and build your programs.

Documentation and Comments

You should make a habit of writing documentation and comments even for the simplest programs. I will talk about documentation in separate files later on in the semester, for now, let's start with comments. Comments should serve as a way of documenting your code. You do not need to explain the programming language in the comments, since we assume that whoever reads your code knows C++. You do have to explain the purpose of variables (unless you give them descriptive names) and the methods of doing things. You might look at your own code several weeks or months (or even years) after you have written it and not know why you chose to do something; the comments will help you to remember if you write them well.

Every distinct source code must contain a preamble with a title, author, brief purpose and description, date of creation, and a revision history. The description should be a few sentences long at the minimum. A revision history is a list of brief sentences describing revisions to the file, with the date and author (you in most of the cases) of the revision. This is an example of a suitable preamble:

```
/*************************************************************************
   Title          : simple_C++_io.cpp
   Author         : Stewart Weiss
   Created on     : January 26, 2006
   Description    : Copies standard input to output a char at a time
   Purpose        : To demonstrate typical text I/O in C++
   Usage          : simple_C++_io
   Build with     : g++ -o simple_C++_io simple_C++_io.cpp
   Modifications  :

*************************************************************************/
```

For the simple programs at the beginning of the course you may not really need all the parts of the preamble - just leave them blank if that is the case. The required parts, those you should always specify, are: Title, Author, Description, Usage, and Build with.

Sometimes there is information you may want to include at the very top of the program, to help the reader understand more about it. I usually include notes in the preamble for this reason. Following is an example of a preamble that you will find in one of the demo programs in our class directory. In fact, the preamble contains the name of the file itself (nestedfor_01.cc), so you can look for it.

```
/*************************************************************************
   Title          : nestedfor_01.cc
   Author         : Stewart Weiss
   Created on     : Feb. 28, 2007
   Description    :
   Purpose        : To introduce the concept of a nest for-loop, this
                    program shows how the inner and outer loop index variables
                    change.
   Usage          : nestedfor_01
   Build with     : g++ -o nestedfor_01 nestedfor_01.cc
   Modifications  :
   Notes          : The program introduces a convenient function found in
                    UNIX operating systems called usleep(). usleep() is given
                    integer argument that is the number of microseconds to
                    delay the program. usleep(1000000) delays 1 second for
                    example, and usleep(500000) delays 1/2 second. It adds
                    effect to the program.

*************************************************************************/
```

Lab 3: Pseudo-Random Numbers and Strings

Random number generation in C/C++

Random number generation is critical to many simulations and games. C (and therefore C++) provides two different functions that can generate pseudo-random integers in the range from `0` to `RAND_MAX` inclusive. `RAND_MAX` is a large positive integer that is implementation dependent. It really does not matter what its actual value is for most applications. There are many pseudo-random number generators available and some are better than others. What is meant by the word "better"? The quality of a pseudo-random number generator is measured by various statistical measures, but intuitively, the better it is, the more its output will be like truly random numbers. Pseudo-random number generators are called ***weak*** or ***strong*** depending on how good they are.

If you want a very strong pseudo-random generator, you may have to write it yourself. The functions in the C/C++ library are pretty strong, but not suitable for all applications. The two functions of choice in that library are `rand()` and `random()`. Because the textbook for the course describes `rand()`, these notes will be limited to `rand()`. Almost everything to be said about `rand()` is true of `random()` as well.

The `rand()` Function

The `rand()` function is part of the C Standard Library and therefore you need to include the header `<cstdlib>` in your C++ program to use it. (In C, you would include `<stdlib.h>`.) The function generates a pseudo-random integer in the range `0` to `RAND_MAX`. `RAND_MAX` is a constant defined in `<cstdlib>`. Its default value may vary between implementations but it is granted to be at least `32767`. On the `cslab` hosts in the lab this header file has the following line:

```
#define RAND_MAX 2147483647
```

which gives `RAND_MAX` the value $2^{31} - 1$. (You can see this line by typing the command "`grep RAND_MAX /usr/include/stdlib.h`".)

What if you want to generate a random number between 1 and 100? How would you use this function to do this? One simple way is to use the modulus operator `%` to narrow the range. This is not the best way, but it works. To illustrate:

```
number = rand() % 50;
```

assigns to `number` a value between 0 and 49. If we add 1 to it we get a number from 1 to 50.

```
number = ( rand() % 50 ) + 1;
```

More generally, suppose we want to generate pseudo-random numbers in the range `[lower...upper]` inclusive, where `lower` and `upper` are any pair of integers such that `lower < upper`. There are (`upper - lower +1`) numbers in this range. Let

```
rangesize = upper-lower+1;
```

The expression

```
rand() % rangesize
```

generates a number between 0 and `rangesize-1`, or equivalently, between 0 and `upper-lower`. To get numbers that range between `lower` and `upper`, we just have to add `lower` to the result. This way, the numbers lie between `0+lower` and (`upper-lower+lower`), which is from `lower` to `upper`:

```
number = ( rand() % rangeSize ) + lower;
```

Example: To generate numbers in the range from 100 to 400, let *lower = 100* and *upper=400* in the above formula. Then the rangesize is $(400-100+1) = 301$. Using this, the assignment would be

```
number = (rand() % 301) + 100;
```

Note. I mentioned above that this is not a very good method of narrowing the range of pseudo-random numbers. This is because the `%` operator is throwing away many valuable bits of information. For greater strength, it is usually better to use a different approach in which you use division instead of modulus.

The `srand()` Function

The `rand()` function cannot be used reliably without first giving its internal generator a starting value. The process of giving it a starting value is called ***seeding the generator***. There is a function whose sole purpose is to seed `rand()`. Its name is easy to remember: `srand()`. This function seeds the pseudo-random number generator. The seed is passed as the argument to the function. The prototype (or declaration) of `srand()` is

```
void srand( unsigned int)
```

Therefore you use it like this:

```
srand(1000000);
```

In this case you gave it a seed of 1000000.

When it is given the same seed, `rand()` will always produce the same sequence. Therefore, if you give `srand()` the same seed each time a program is run, the sequence of numbers generated by `rand()` will be the same. This little program can be used to illustrate this:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    srand(100);
    for ( int i = 0; i < 10 ; i++ )
        cout << rand() << endl;

    return 0;
}
```

Compile it and run it repeatedly. You will see the same ten numbers each time.

To get a new sequence each time, the value given to `srand()` must be different in each run. The most common way of doing this is to seed `srand()` with the current time. UNIX has a function that returns the current time. The function `time()` (declared in the C header file `<ctime>`) returns the current time as an integer. You give it a 0 as an argument:

```
    current_time = time(0);
```

and it returns an integer value. To be more precise, `time(0)` returns the number of seconds that have elapsed since since 00:00 hours, Jan 1, 1970 UTC. (This is due to historical reasons, since it corresponds to a UNIX timestamp, but is widely implemented in C libraries across all platforms.) The important fact is that the value returned by time(0) is guaranteed to be different almost every time that the program runs, it is a perfect choice for the seed. Try running this program:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0));
    for ( int i = 0; i < 10 ; i++ )
        cout << rand() << endl;

    return 0;
}
```

and you will see the sequence vary from one run to another.

A Bit About Strings

C++ allows you to declare variables of type `string` in your code. In order to declare and use such variables, the program must include the `<string>` header file, which is placed into the `std` namespace, so you need the lines

```
#include <string>
using namespace std;
```

in the beginning of your program. The type `string` is different from elementary types such as `int`, `float`, and `char`. It is a ***class***. C++ has libraries that define complex things called ***classes***. You should think of a class, for now, as a collection of data values together with operations and functions that act on those values.

Declaring string objects

There are several ways to declare and initialize string objects. Two equivalent ones are

```
string filename = "input_file";
```

and

```
string filename("input_file");
```

Which one you use is a matter of taste.

Concatenating strings

The word ***concatenation*** means juxtaposing two strings one after the other. For example, the concatenation of "`butter`" and "`flies`" is "`butterflies`". The operator used for adding numbers, `+`, has a different meaning when it is placed between two string objects; it forms their concatenation[1].

If `username` and `hostname` are two string variables, and `username` contains the string "`stewart.weiss`" and `hostname` contains the string "`hunter.cuny.edu`", then the statement

```
cout << username + "@" + hostname;
```

will output "`stewart.weiss@hunter.cuny.edu`", and if `email` is a string variable

[1]The reuse of the + sign here is an example of an **overloaded operator**: it works on objects of the string class differently than it does on numbers.

```
email = username + "@" + hostname;
```

will assign "`stewart.weiss@hunter.cuny.edu`" to `email`. We could also have written the above with the @ symbol as a character, not a string:

```
email = username + '@' + hostname;
```

The concatenation operator lets you concatenate characters and string variables, string literals and string variables, and two string variables, but not string literals or characters with each other. Assume `username` and `hostname` are string variables that have values. The following illustrates what you can and cannot do:

```
username + hostname              // legal expression
username + '@'                   // legal expression
username + "@hunter.cuny.edu"    // legal expression
"hunter" + ".cuny.edu"           // illegal expression
"hunter" + '.'                   // illegal expression
```

Getting the size of a string object

Variables of the string class have a `size()` function that can return their size, which means the number of characters in them. To use this function, you have to put "`.size()`" after the variable name, just the way you use the `open()` function after an `ifstream` or `ofstream` variable. For example:

```
string input_line;
cin >> input_line;
cout << "You entered " << input_line.size() << " characters.\n";
```

In the third line, the value of `input_line.size()` is the number of characters presently stored in that string variable.

Accessing individual characters in a string object

There is a way to access each individual character in a string object using what is called the ***subscript operator***. The subscript operator is a pair of square brackets `[]` placed after the name of the string, with a number in between. The numbering starts at 0 and goes up to the size of the string minus 1. The number in brackets is called the ***index value***. Thus:

```
cout << "The first character you entered was " << input_line[0] << ".\n";
```

will display the character in the first position in the string `input_line`. Notice that the first position is at index 0, *not 1*! It will be hard getting used to this idea and you will likely forget, but try, try to remember that the positions in the string are numbered 0, 1, 2, 3, and so on up to `size()`-1:

```
int last_index = input_line.size() - 1;
cout << "The last character you entered was " << input_line[last_index] << ".\n";
```

will output the character in the last position of the string. You can modify individual characters using an assignment statement and the subscript operator:

```
input_line[0] = 'L';
```

will replace the first character in the string by the letter 'L'.

Warning: C++ compilers do not check if you are trying to access or modify a location outside of bounds of your `string` object. But if you do, your program may work incorrectly or even crash! Be careful!

Pair Programming Tutorial

This material is excerpted almost entirely from the webpage http://ecee.colorado.edu/~ecen2120/Manual/pair.html.

Background

Pair programming is a structured technique, with clear guidelines and methods, for improving the process of programming by using collaboration. Pair Programming is most effective when programmers have some basic understandings:

- Programmers should think of their work as a shared, collective project. Each programmer should be able to see his or her contribution in the final product, and each programmer should take equal ownership for the programming process that created that product: Each programmer should "own" both the process and the product of their shared work.
- Programmers should avoid the temptation to break projects into smaller parts to be completed independently and the integrated together. In pair programming, programmers work together, side by side, for most of the time. Pair programming aims to be truly collaborative.
- Programmers agree to perform specific activities while programming with their partners. Sitting side by side at one computer, programmers assume either the role of the Driver or the Navigator. Both of these roles are "programming". The pair programming method depends on these set roles being carried out fully. Partners can switch roles when they wish and should work to feel comfortable with both sets of practices.
- Pair programming requires that partners communicate with each other openly and honestly. Yet, the Driver and the Navigator must work to balance their abilities to voice their own opinions while also being open to their partners' input. This means staying away from a "my way or the highway" attitude. It also means having the ability to speak up and voice their own concerns. Pair programmers need to balance too much ego with too little.

Pair programming may seem to be more effort, or to be less efficient because it requires two people to work together, when most other programming practices are done with just one person. The efficiency of pair programming is shown in the outcome: generally reduced development time, generally better programs, and improved awareness of the logic and practices of programming.

Role Guidelines and Procedures

Effective pair programming requires some planning, and an understanding of the Driver and Navigator roles.

Set Up

Physical Space. The work space needs to be set up so that both programmers are able to sit next to each other, with both orienting to the computer. Programmers should have any materials on hand that they may need.

Planning Period. Before starting on their destination, programmers need to develop a mental map of where they are going. This includes discussing what they intend to do, their thoughts on the problem or design, and any questions or suggestions they may have at the start. At the end of this period, programmers have a shared sense of where they are going.

Role Assignment. Programmers must have an initial discussion to address who will navigate and who will drive.

The Driver

The Driver does the following:

1. Controls whatever is being used to record the program as it is developed, such as a pencil, mouse, or keyboard.
2. Has responsibility for the details of developing or writing the code. (We will refer to this as *entering code*.)
3. Talks out loud about what they are thinking about or why they are entering any particular element of the code. The intent here is to make reasoning or thought processes explicit so they can be discussed, rather than leaving them implicit and perhaps difficult to understand. This is one key to pair programming.
4. Responds constructively to feedback given by the Navigator. Recognizes that comments from the Navigator are part of the process, and doesn't become annoyed, defensive or dismissive. The Driver must be open to working out/talking through problems.

The Navigator

The Navigator does the following:

1. Does not enter the code. This is important. Partners should make every attempt to talk through the ideas rather than simply hand off the keyboard or pencil to each other. Hand-offs prevent truly understanding the program as it is being developed.
2. Keeps an objective view and thinks strategically about the direction in which the program is going. Navigators keep a big picture view.
3. Watches and alerts the Driver if there are tactical or strategic problems with the work.
4. Continues to think of alternatives and looks up resources that may help the partners pursue other ways of going about solving a problem. These are things the Driver shouldn't do while focusing on developing code, yet could be critical to developing a good program.
5. Looks for the strategic implications of the developing code. Asks questions at the appropriate times, such as: where are we going, and what does it mean to be doing it this certain way over others?
6. Asks questions of the Driver, especially if the Driver is doing something that is unclear, cloudy, or unknown to the analyzer. Asks these questions in constructive and supportive ways, rather than in ways that are directing or commanding. Listens to the Driver's answers and discusses if appropriate. These include: Try this ..., I notice that ..., etc.

Procedural Checklist

Before Entering Code:

- Physical space is correctly arranged and materials needed are present.
- Discussed what you are working on and who is doing what.

Driver:

- Should be the only one entering code.
- Talks through their actions.
- Accepts feedback and direction from the navigator.

Navigator:

- Focuses on the big picture.
- Thinks strategically–is watchful and alert to what the Driver is doing.
- Shares alternatives and consulting resources when needed.
- Asks questions of the Driver.

Programmers:

- Both are listening and responding to each other.
- Neither programmer is holding back their opinion(s) or question(s).
- Criticism/suggestions are made constructively.
- Both programmers own the project.

LAB 01 Name: _____

Before you start these exercises, you should read the tutorial named l**ab01_tutorial.pdf**, located in the class directory,

/data/biocs/b/student.accounts/cs135_sw/tutorials

It explains everything you need to know to so this assignment. It is also on the course website http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci135/csci135_36_fall12.php for your convenience.

Getting Around the File System

Let's make sure that you know how to get around the Unix/Linux file system before you start writing code. All of the exercises below are supposed to be done in the command line, unless otherwise stated. When I write a command, I always put the dollar sign **$** to represent the shell prompt. You do not type a dollar sign!!

The easy way to figure out "where you are" in the file system is to ask what your present working directory is: type the **pwd** command:

```
$ pwd
```

In the space below, write the output of that command.

Before continuing, try running the following commands to see what they output: try typing

```
$ who
```

and

```
$ who am I.
```

Now type the following *compound command*

```
$ date ; hostname
```

and write the output in the space below:

If you are not in your home directory, you can get there by simply typing **cd** followed by enter (with no additional arguments) – this will always bring you home.

Now, you will create several directories and files. Use the **mkdir** command to create the following

directories in your home directory:
cs135, **cs136**, **old_files**, **scratch**.

If you type **ls**, you should see the list of all the files in your home directory, including the directories that you just created. If you see a long list of files zooming by, then you probably need to do some cleanup in your directory: it is not the best idea to keep all of your files at the top level of your directory; after this class you should be able to organize your files better.
Change your PWD to **cs136** and create two more directories there: **labs** and **testing**. Within **labs**, create a subdirectory named **lab01**.

Change your current working directory to **lab01** and type

```
$ ls -l
```

to see its contents. Write its output below.

What single command will change your PWD to **testing**?

Make sure that your PWD is now **testing**. You can create an empty text file by typing **touch** followed by the file name. Enter

```
$ touch test1
```

and then use **ls** to see if a file named **test1** was created. You can edit the content of this file in many ways. **gedit** is one of the friendlier simple editors on Linux and I suggest you use that for editing your text files, including your C++ source code files. Type

```
$ gedit test1 &
```

and a window with an empty file should pop-up (the **&** at the end of the last command runs the command in the background, i.e. it allows you to continue using the terminal window while the **gedit** window is open – if you omit **&** your terminal will be blocked until you close the **gedit** window). You will see some output in the terminal such as

```
[1] 1276
```

Ignore it for now. Later I will explain it.

Enter some text into the body of the file (at this point it does not matter what text you type), save it and close it. This is exactly what you will do to write your code. After you close **gedit** and enter another command, you will see a message like you might see a message appear on the screen like

```
[1]+  Done            gedit test1
```

Ignore this as well. It says that **gedit** exited.
What command will remove the file **test1** from the directory **testing**? Write it in the space below and delete **test1** with it.

Now, create a new, empty text file called **test1** inside the directory **testing** again (yes, the one that I just told you to delete). Change your PWD to **lab01** (parent of **testing**) and try to remove the directory **testing** itself. What command should do this? What happens when you try? What do you need to do first to remove a directory? Write the answer in the space below.

You can now move back to your home directory and remove some other directories that you created before if you want, or you can leave them there.

Using the g++ Compiler

Let's make sure you can use the **g++** compiler. The class directory

```
/data/biocs/b/student.accounts/cs135_sw/cs136demos/lab01/
```

contains a C++ source file named **johnnybgoode.cp**p. Assuming that you read the tutorial and when doing so, you created a link to the class directory in your home directory that is called **cs135link** (if you did not, now is the time to do it), you can copy this file to the directory **cs136/labs/lab01/** in your home directory using the **cp** command:

```
$ cp ~/cs135link/cs136demos/lab01/johnnybgoode.cpp  ~/cs136/labs/lab01
```

(~ is another shortcut notation that stands for your home directory: whatever you type after is relative to your home directory.) There are other ways to copy the file, but this is relatively simple. Once the file is in your directory you can compile it, build executable code and run it (make sure that at the end of this lab you understand the difference between these three terms).

Change your PWD to the directory **cs136/labs/lab01** and make sure that the file is really there. A simple way to compile and build this program is to run

```
$ g++ -o johnnybgoode johnnybgoode.cpp
```

The **-o** option to **g++** tells **g++** that the string that follows it is what it should name the executable file that it creates for your program. If you omit this option, **g++** will use the default name which is **a.out**. Run **g++** again, but without specifying the name of the executable file. Look at the list of the files in your PWD, what do you see? Write it here.

To run the program type **./johnnybgoode** (it is the name of the executable file preceded by dot and forward slash) or **./a.out**. These programs are identical since they were created from the same source code.

Now edit the file **johnnybgoode.cpp** and save it as **johnnybgoode2.cpp**. Remove the line from the file (the line that contains **using namespace std;**), save and compile this new code. Write what happens when you run **g++**?

The compiler performs multiple steps when it is taking your source code and creating the executable program. Let's look at these steps:

1) ***Preprocessing***. Remember all those lines in the program that start with #? These are called ***preprocessor directives***. Their purpose is to insert the contents of the included file into the program at that point. When you type **#include <iostream>** in your code, the compiler replaces this line with the contents of the **iostream** library header file **iostream.h**. **g++** runs only the preprocessor[1] if you use **-E** option:

   ```
   g++ -E johnnybgoode.cpp > johnnybgoode.i
   ```

   By default the output is written to the screen; the bash redirection

   > **johnnybgoode.i**

   redirects the output to a file named **johnnybgoode.i**. You can use **gedit** to look at this file (it is much larger than the original **johnnybgoode.cpp**). Look at last sixty lines of this file. What do you find there? Describe what is there in the space below.

1 The preprocessor is actually a program named **cpp**. You can read about it by typing **man cpp**.

2) ***Compilation***. After preprocessing, the compiler actually reads the code and makes sure that it is syntactically correct. Compilation itself has a few sub-steps: parsing your code to validate it syntactically, creating assembly code, optimizing it, and then creating the executable code. **`g++`** stops after the creating the assembly code if you use **`-S`** option:

```
g++ -S johnnybgoode.cpp
```

By default the output is written to a file with **`.s`** suffix, so you should have a file called **`johnnybgoode.s`**. (Yes, it is a bit confusing that the output of **`g++ -E`** goes to the screen but the output of **`g++ -S`** goes to a file.) Look at this file. This is the assembler code for your program. The command **`wc`** (word count) followed by the name of the file tells you how many newlines, words and bytes the file has, in that order. Use this command to find out how many newlines are in your file **`johnnybgoode.s`** . How many lines does it have?

The second part of compilation is creation of the executable code of your program alone (this code will rarely be all that is needed to actually run the program). **`g++`** stops after this part if you use **`-c`** option:

```
g++ -c johnnybgoode.cpp
```

By default the output is written to a file with **`.o`** suffix, so you should have a file called **`johnnybgoode.o`**. This is not a text file anymore, so you cannot look at it with a text editor such as **`gedit`**. It is not an executable program yet; it does not have definitions for the symbols defined outside of itself such as **`cout`** and **`cin`**. They are in the **`iostream`** library. Files produced by this stage are called ***object code***, or ***object files***. The next step produces the final runnable program.

3) ***Linking***. At this point the object code for your source code exists, but the missing definitions need to be linked to it. These definitions usually come from the C++ libraries that the program uses; for now we are only using **`iostream`**, but soon we will use other ones as well. The last

step that is performed is called linking[2]. The command

```
g++ -o johnnybgoode  johnnybgoode.cpp
```

runs all the steps including the linking.

Display the list of all the files in your directory now:

Finally, write your own small C++ program: it should ask user for his or her age and then print a greeting using that information. The sample output of such program is:

```
Enter your age: 75

++++++++++++++++++++++++++++++++++
 Hi Stranger, You are 75 years old.
   Welcome to CSci 136, section 2
++++++++++++++++++++++++++++++++++
```

Name this file ***username*_welcome.cpp**, where ***username*** is replaced by your username on our system, i.e., the name with which you login. Compile and run your program.

How To Submit Your Work

The exercise has two parts, this document and your program. To submit this lab document, with everything you filled in, use Acrobat Reader's Print to File feature. Select Print, and in the dialog box, check the Print To File check-box. Then click the Browse button and name the file ***username*_lab01.ps**. where ***username*** is replaced by your username on our system.

2 The linker program used by g++ is called **ld**. You can read about it by typing **man ld**.

To submit both this file and the file ***username*_welcome.cpp** that you created above,

1. Create a directory in
   /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab01/submissions
   whose name is your username. For example, I would create the directory **sweiss**.
2. Copy the two files, ***username*_lab01.ps** and ***username*_welcome.cpp**, to this directory.
3. Change your `PWD` to
   /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab01/submissions
4. Execute the command
   $ chmod 700 *username*
   where ***username*** is the name of the directory you created. I would type "**chmod 700 sweiss**" for example. This prevents everyone else from reading your files.

You must do all of this by the end of the class on Thursday, August 30. If you do not complete all the work, you have until the end of the day on Friday, August 31, to receive full credit.

Lab 2: Bakhshali Approximation for Computing Square Roots

Finding the positive square root of a positive number (e.g., $\sqrt{5}$) is an example of a problem that requires a *numerical* solution. There is no formula that expresses the square root of an arbitrary number S in an exact way as a function of S. However, there are many methods for *numerically* computing square roots. A *numerical computation* is an approximation to the solution. Algorithms that compute successively better approximations to the solution are called *approximation methods*, or *approximation algorithms*.

In this lab exercise you will write a program that implements one particular square root approximation algorithm. It is a variation of a method known as the *Bakhshali approximation algorithm*. The Bakhshali method for finding an approximation to a square root was described in an ancient Indian mathematical manuscript called the *Bakhshali manuscript* (see the Wikipedia page on Methods of computing square roots for details). The Bakhshali method is really just two iterations of a more general, and ancient, algorithm dating back to the Babylonians. In this assignment, we will extend the Bakhshali method so that it can be iterated indefinitely, like the Babylonian method. The description of the algorithm follows.

The Bakhshali algorithm to compute the square root of a number S is defined by the following steps:

1. Make an initial *guess* at the square root by picking any number smaller than S.
2. Compute the difference d between the square of your *guess* and S:
$$d = S - guess^2.$$
3. Compute the ratio r of d and twice the *guess*:
$$r = \frac{d}{2 \cdot guess}.$$
4. Compute sum of the *guess* and the ratio r:
$$a = guess + r.$$
5. Compute your next *guess* as
$$guess = a - \frac{r^2}{2a}.$$
6. Go back to step 2 using the *guess* just computed.

Steps 2 through 6 are repeated as many times as the implementer sees fit. It can be proven mathematically that the *guess* gets closer and closer to the actual square root of n as the number of iterations increases[1]. But the process needs to stop somewhere for it to be useful.

[1] In fact, one can show mathematically how much more accurate it gets each time, but that fact will be a secret for now.

Exercises

Problem 1. Write a program that reads a positive integer for S from the user. You can assume that the user will always enter an integer, but the program does have to detect if the user entered a negative integer, and if she did, it should exit with an error message. The program then applies five iterations of the above Bakhshali algorithm to compute the square root of S. Your program should display the answer with ten decimal places (if there are fewer than ten digits after the decimal point, your program should display zeros). How will you choose the initial guess?

Problem 2. Modify the program from *Problem 1* so that the computed answer is more accurate. Execute as many iterations as are needed until the values computed in two consecutive iterations differ in absolute value by less than 0.0000001. For example, when oen iteration is 2.23606783 and the next is 2.23606791 the difference is 0.00000008. Add a counter to your program that keeps track of the number of iterations needed before the loop terminates. Output the number of iterations together with the answer. What happens as you choose to compute square roots of larger and larger integers? How can you make this second program more general? If you can think of a way, add that to a comment in the program. If not, do not worry about answering it.

Testing Your Work

Before you hand in your programs, you have to make sure that they work correctly. For these two programs, this means that when they are given a number, their output should be an approximation to its square root. One way to test is to manually square the answer to make sure it is close to the original number. Another way is to use a calculator to verify the answer. Still a third way is to pick perfect squares as input, so that their square roots are whole numbers. For example, when given the input 144, your program ought to produce something close to 12.

The second program is easier to test than the first, because it is supposed to be accurate to a specific number of decimal places. If your program's result is not that accurate, it is not correct. The first is harder, because the accuracy is not specified, only the number of iterations. In this case just make sure it really iterates five times as specified.

What to Submit

Submit both programs by the end of today's lab, i.e., before the end of the class at 2:00 P.M. To do this,

1. Create a directory in

   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab02/submissions`

   whose name is your *username*. For example, I would create the directory `sweiss`.

2. Copy the two files, which should be named *username*`_lab02a.cpp` and *username*`_lab02b.cpp` to this directory. The first program is *username*`_lab02a.cpp` and the second is *username*`_lab02b.cpp`. It is critical that you name these properly. You will lose 5% of the grade if you misname either file!

3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

```
$ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab02/submissions
$ chmod 700 username
where username is the name of the directory you created.
```

Do not submit executable files. (Make sure that your files have the .cpp extension.) Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be judged using the rubric I outline in the Programming Rules document.

If you do not complete all the tasks, submit whatever you have by the end of class. There are absolutely no extensions to the deadline. You can submit a revised version of either program by 8:00 P.M. on Friday, Sept. 7. If you do, you must name it with a different name than what you first posted, e.g., *username*`_lab02a_v2.cpp`. and put it in the same directory as the original programs. Code submitted after the end of class will receive only partial credit and must look like a revision of the original file. If you do this, you must send email to me before 8:00 P.M. at myHunter address, telling me to look there!

Lab 3: Generating and Analyzing Random Strings

Scientists routinely need random data for experiments and simulations. Natural events appear to be random, and in order to study and simulate them, scientists use randomization. Computer scientists especially need to know how to generate random data since they are the ones who are often called upon to design the software. It is sufficient to be able to generate ***random numbers***, because if we have a sequence of random numbers, we can generate random data of other types from it.[1]

There is a contradiction in the idea that a computer program can generate random numbers, because if a computer program generates anything, it is the result of an algorithm and is therefore not random! A computer program can generate numbers *that look random* because no mere mortal can detect a pattern in them, nor could another very smart computer, even if given a very long time. Such "random looking numbers" are called ***pseudo-random numbers***.

In this exercise you will write a program that generates pseudo-random text strings of a specific form, consisting only of the lowercase letters 'a', 'c', 'g', and 't'. In other words, the program will generate a string that contains just these letters, but the order of the letters and the quantity of each type of letter is pseudo-random. Such a string will be called a ***pseudo-random DNA string*** here. Once you have successfully completed this first program, you will enhance it a bit.

Exercises

Problem 1. Write a program that begins by prompting a user to enter a positive integer N. You can assume that the user will always enter an integer, but the program must detect if the user entered a negative integer or zero, and if she did, it should exit with an error message. The program then prompts the user to enter the name of file into which it will store the generated DNA string. The program then generates a pseudo-random DNA string of length N and writes it into a file with the chosen name, in its current working directory.

Problem 2. Once the program from *Problem 1* is working correctly, and only when it is working correctly, you will make a small modification to it. In addition to writing the pseudo-random DNA string to the file, it must display on the terminal window the fractions of 'a's, 'c's, 'g's, and 't's in the string. For example, if the string is length 50 and it contains exactly 20 'a's, 10 'c's, 5 'g's, and 15 't's, then it would display something like:

```
a: 0.4000 ; c: 0.2000 ; g: 0.1000 ; t: 0.3000
```

Notice that the sum of these fractions must be 1.0. The program should display the fractions with four digits of precision to the right of the decimal point.

[1] In fact it is a theorem of computer science that all data, however complex it may appear, is representable as a set of non-negative integers.

Testing Your Work

Before you submit your program, you have to make sure it is correct. It is not easy to test a program that generates random data, because in order to test, one needs to reproduce a result repeatedly. The `srand()` function is useful here. If `srand()` is given the same value, the sequence of random numbers produced by `rand()` will be the same. You should make sure also that the length of the string is correct, that the data looks random, and that your fractions add up to 1. If you give `srand()` a new value in each run of the program, then you should see a different DNA string each time.

What to Submit

Unlike the previous lab assignment, ***you will submit just one program***. If you finished the second problem and it is working correctly, submit that program and not the first. I will not look at the first program and it will not count toward your grade.

If you did not get the second program working, but the first was working, you may either submit the first, or submit the partially working second one. The first program is worth only 6 out of 10 points. The second is 10 points. If the second is almost working, and you think you will lose fewer than 4 points on it, submit it. If you know it needs much more work, submit the first instead.

Submit your program, whichever it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. The instructions are just like the previous lab's:

1. Create a directory in `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab03/submissions` whose name is your *username*. For example, I would create the directory `sweiss`.

2. Copy your program, which should be named either *username*`_lab03a.cpp` or *username*`_lab03b.cpp` to this directory. You will lose 5% of the grade if you misname the file!

3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

   ```
   $ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab03/submissions
   $ chmod 700 username
   ```

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be judged using the rubric I outline in the Programming Rules document.

There are absolutely no extensions to the deadline. You can submit a revised version of the program by 10:00 P.M. on Thursday, Sept. 13 for partial credit. If you do, you must name it with a different name than what you first posted, e.g., *username*`_lab03b_v2.cpp`. and put it in the same directory as the original. It must be a *revision* of the original file, with only fixes to things that were not working before. I will adjust the grade based on how well the first version worked and how many changes you made.

Lab 4: An Interactive 3D Distance Calculator

This lab will give you practice in writing your own functions and in designing an interactive, menu-driven program.

Math Background

You should know what the distance between two points in the plane is: the distance between the points (x_1, y_1) and (x_2, y_2) is given by the formula

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

When we deal with points in space, it is defined similarly. The distance between the points (x_1, y_1, z_1) and (x_2, y_2, z_2) is given by the formula

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

This kind of distance is called ***Euclidean distance***. It is not the only way to measure distance.

In Manhattan, you cannot walk on a diagonal[1]. If you are at the intersection of 120th Street and Fifth Avenue and need to walk to 86th Street and First Avenue, you have to walk down $120 - 86 = 34$ blocks and across $5 - 1 = 4$ blocks, for a total of 38 blocks. (We will ignore the fact that blocks are not square.) Therefore the distance between these two corners is 38 blocks. This type of distance is sometimes called ***Manhattan distance***[2]. The Manhattan distance between the points (x_1, y_1) and (x_2, y_2) is given by the formula

$$|x_2 - x_1| + |y_2 - y_1|$$

It can be generalized to three dimensions easily.

A third way to measure distance is known as the ***Chebyshev distance***[3]. The Chebyshev distance between the points (x_1, y_1, z_1) and (x_2, y_2, z_2) is the maximum of the distances between the x, y, and z coordinates respectively:

$$\max(|x_2 - x_1|, |y - y_1|, |z_2 - z_1|)$$

Exercise

You will write a program that displays the following menu to the user:

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^  3D Distance Calculator ^^^^^^^^^^^^^^^^^^^^^
[1]    Change the first point
[2]    Change the second point
[S/s]  Display the current two points
[E/e]  Calculate and display the Euclidean distance between the two points
[M/m]  Calculate and display the Manhattan distance between the two points
[C/c]  Calculate and display the Chebyshev distance between the two points
[Q/q]  Quit the program
```

[1] We ignore streets like Broadway.
[2] It is actually based on what mathematicians call the L1 norm.
[3] In two dimensions it is known as *chessboard distance*.

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Enter your choice:
```

The characters in square brackets are what the user is supposed to type to select the menu item. The notation "`[E/e]`" means enter either an upper or lowercase `e`. Once the user presses a valid key, the program should perform the indicated action. The selections to change a point (`1` and `2`) should prompt the user to enter the x, y, and z coordinates of that point and not display anything else. After the action is performed, the prompt

```
Enter your choice:
```

should be redisplayed, *but not the menu*. If the user enters an invalid choice, the program should just ignore it. Points 1 and 2 should each have the initial value (0,0,0). All numbers should be printed to the screen with four digits of precision.

When you first write the program, you may use the math library's `sqrt()` function, but to receive full credit for this program, you must write your own `sqrt()` function by converting the second version of the Bakhshali program of Lab 2 into a function. Without your own `sqrt()` function, the program is worth at most 8 out of 10 points. There is a correct version of the Bakhshali program in the lab04 directory on the server.

Note. Your main program should use a switch statement, not nested if-statements.

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. The instructions are just like the previous lab's:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab04/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.
2. Copy your program, which should be named *`username`*`_lab04.cpp`. You will lose 5% of the grade if you misname the file!
3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

   `$ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab04/submissions`

   `$ chmod 700 username`

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document.

There are absolutely no extensions to the deadline. You can submit a revised version of the program by 10:00 P.M. on Thursday, Sept. 20 for partial credit. If you do, you must name it with a different name than what you first posted, e.g., *`username`*`_lab04_v2.cpp`. and put it in the same directory as the original. It must be a *revision* of the original file, with only fixes to things that were not working before. I will adjust the grade based on how well the first version worked and how many changes you made.

Lab 5: Simulations

Overview

In a software development company, programmers are often asked to write the implementation of a function, given its prototype and a precise description of what it is supposed to do. To make it easier for them, they are also given a ***driver program***. A driver program is one that calls their function, so that they can test it and make sure it works.

In this lab assignment you are given a few function prototypes and precise descriptions of what these functions must do. These prototypes and descriptions are in a ***header file***. A header file is simply a file containing function prototypes, possibly constant and type definitions, and sufficiently detailed comments so that a programmer can make calls to those functions in his or her own program. Your job will be to create a file called an ***implementation file***, which contains the actual definitions of the functions in the header file. The names of the header file and the implementation file are usually coordinated so that they differ only in the file extension. If the header file is named `duel.h`, then its implementation file is named `duel.cpp` in C++ (or `duel.c` in C). Header files are always given the extension "`.h`", whether in C, C++, or another programming language.

Once you have written the function definitions in the file `duel.cpp`, you will be able to compile the file into an object file using the command

```
g++ -c duel.cpp
```

This will create the file `duel.o`.

You will not be given the source code to the driver program. You don't need it. But you will have its object file, which will be named `lab05_main.o`. With your `duel.o` file and `lab05_main.o`, can create the executable program with the command

```
g++ -o lab05 lab05_main.o duel.o
```

If your code has no syntax errors, the file `lab05` will be created in your working directory. If it has errors you will have to look at the messages and correct them. It is better to use the command

```
g++ -Wall -g -o lab05 lab05_main.o duel.o
```

turning on all warning messages and compiling code that can be run under the GDB debugger. For a more detailed explanation, see my tutorial, http:///~stewart/resources/separateCompilation.pdf.

Exercises

You will be writing functions that are used in a game. This exercise will also give you practice with random number generation. The game simulates three people with different shooting skill levels trying to duel each other. In the imaginary land of Moronia, Aaron, Bob, and Charlie had an argument over which one of them was the greatest logician of all time. To end the argument once and for all, they agreed on a duel to the death, which seemed quite logical to all of them.

The facts surrounding the duel and their skill levels are that

- Aaron is a poor shooter and only hits his target with a probability of 1/3.

- Bob is a bit better and hits his target with a probability of 1/2.
- Charlie is an expert marksman and never misses.

A ***hit*** means a kill and the person hit drops out of the duel. A ***duel*** is carried out in a specific order. To compensate for the inequities in their marksmanship skills, they decided that they would take turns firing, starting with Aaron, then Bob, and then Charlie. Since each is a smart person, they each adopt the strategy to shoot at the most accurate shooter still alive, on the grounds that this shooter is the deadliest and has the best chance of hitting back. The duel is carried out until only one of them is left alive. It is not possible for all to die because only one person shoots at a time and either kills or does not kill the intended target.

There is a header file named `duel.h` and an object file named `lab05_main.o` in the directory

```
/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab05
```

You are to copy these files into your working directory and create a file there named `duel.cpp`. The header file contains the prototypes of four functions that you must implement in `duel.cpp`. After you have successfully written `duel.cpp` and compiled without error, create the executable program using the instructions that I wrote above. Run the executable, `lab05`. It will display a prompt to enter the number of duels to run. Enter a number and look at the output. Aaron should win about 36 to 37% of the time, Bob, about 41 to 42%, and Charlie, about 21 to 22%. If you make the number of duels large enough, this is what you should see. If not, your implementation is flawed.

Advice: To simplify writing the `.cpp` file, and avoid typing mistakes, copy `duel.h` into `duel.cpp` and replace the semicolons by curly braces.

Hint: How can you make a program simulate an event that happens with probability p, where $0 \leq p \leq 1.0$? Suppose you generate a random number between 0 and 1. If it is less than or equal to p, then the event happened. If it is greater than p, then it did not.

What to Submit

Submit your implementation file, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. The instructions are just like the previous lab's:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab05/submissions`
   whose name is your *username*.
2. Copy your file, renaming it to *`username`*`_duel.cpp`. You will lose 5% of the grade if you misname the file!
3. Change the permission on the directory that you created so that no one else can read it. You do this with the commands
   ```
   $ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab05/submissions
   $ chmod 700 username
   ```

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document.

There are absolutely no extensions to the deadline. You can submit a revised version of the program by 10:00 P.M. on Thursday, Sept. 27 for partial credit. If you do, you must name it with a different name than what you first posted, e.g., *`username`*`_duel_v2.cpp`. and put it in the same directory as the original. It must be a *revision* of the original file, with only fixes to things that were not working before. I will adjust the grade based on how well the first version worked and how many changes you made.

Lab 6: Software Testing and Debugging

Overview

The purpose of these exercises is threefold:

1. to give you practice in identifying and correcting ***syntax errors*** in programs,
2. to give you practice in identifying and correcting obvious ***runtime errors*** in programs, and
3. to give you practice in devising ***software tests***.

The first problem asks you to compile a program, look at the error messages produced by the compiler, and then read the program and correct the mistakes. The faster you can do this, the more efficient you will become at writing syntactically correct software. You do not need to know what the program is supposed to do because the errors are not in logic but in using the language properly.

The second problem asks you to compile a program, which will compile without error, and then run it. When it runs, there will be obvious errors. You will not need to know what the program is supposed to do because the program's behavior will be unacceptable no matter what it is. For example, it might loop forever or produce no output or produce garbled output. You will have access to the source code so that you can correct the mistake(s) or at least suggest what is wrong.

The third problem asks you to test a program to determine whether it has any faults, and if so, to determine what they are. You will not be able to see the source code of the program, but you will be able to run it. This type of testing is called ***black box testing***, because the program is essentially like a black box, an object whose inside is invisible to you. You will be given a precise description of what the program is supposed to do, which is called a requirements specification. This specification can be used to decide what tests to use on the program.

A Bit About Software Testing

A ***test*** is an input to a program. A ***test case*** is an input together with the expected output. Running a program on a test input is meaningless unless you know what the output for that test input should be. For example, if the program's requirements specification states that the program computes the positive square root of any non-negative number, then a test case could be (input = 25.0, expected output = 5.0), because 5.0 is the positive square root of 25.0. Therefore, your job as a program tester is to devise a suitable set of test cases. Such a set is often called a ***test suite***.

Usually, before you start testing the program, you would create the list of all test cases – inputs and expected outputs – and put that list into a file, perhaps into a spreadsheet. When you run the program on a test, either it passes or fails the test. If a program fails the test, we say that the test ***exposed an error*** in the program. A spreadsheet might contain information such as when you tested it, whether the error was reported and corrected, and so on.

If a program passes all of your tests, you have to decide whether the program is error-free. Just because it passed all of the tests does not mean that it has no errors. It might mean that your test cases were not very good. If you think your tests were thorough enough to detect all possible errors in the program, then you can conclude that the program is error-free. You might decide that the tests were not thorough enough and continue testing further.

How can you decide whether the tests were thorough enough? Many people have spent much time trying to answer this question, and there are hundreds of scientific articles about the subject. We cannot begin to tackle that problem in very much depth here. However, there is some common sense that you can apply.

If the program's specification says that for all inputs that satisfy some condition *C*, the output is supposed to be *X*, and for inputs that satisfy a different condition *D*, the output should be *Y*, then clearly any good set of test cases would include some inputs that satisfy condition *C* and some that satisfy condition *D*. This type of thinking will suffice in the exercises that you have to do today.

Exercises

This is a three-part lab.

Problem 1. There is a directory named `syntaxbugs` in the `cs136labs/lab06` directory. Within it are five files named `syntaxbug01.cpp`, `syntaxbug02.cpp`, ..., `syntaxbug05.cpp`. None of these compile without error. Study the error messages and find a fix, then make sure your corrected file compiles. Name the corrected version of `syntaxbug0x.cpp` `syntaxbug0x_fixed.cpp`, x=1,2,3,4,5. In the corrected program, put a comment line like this:

```
/* ERROR FIXED HERE: */
```

to the right of the code on the line that you fixed. If there is more than one line, put this on each such line. Each corrected program is worth 0.5 out of 10 points for this lab.

Problem 2. There is a directory named `runtimebugs` in the `cs136labs/lab06` directory. Within it are three files named `runtimebug01.cpp`, `runtimebug02.cpp`, and `runtimebug03.cpp`. Each of these compiles without error but does not run correctly. Determine what is wrong with each and try to correct the code. Make sure that your corrected code compiles and runs correctly. ***Follow the same instructions as in Problem 1 and add comment lines to indicate where you fixed the program.*** Once it is correct, name the corrected version `runtimebug0x_fixed.cpp`, x =1,2,3. Each corrected program is worth 1 out of 10 points for this lab.

Problem 3. The third directory is named `softwaretesting`. It contains the following files:

program_specification A precise description of what the program `classifytriangle` is supposed to do.
classifytriangle1 A version of `classifytriangle` that does not work correctly.
classifytriangle2 A version of `classifytriangle` that does not work correctly.
classifytriangle3 A version of `classifytriangle` that does not work correctly.
angleof A program that can be useful in checking whether the output of `classifytriangle` is correct.

Your job is to find tests of the three `classifytriangle` programs that expose their errors. Each of these programs has one bug, i.e., a single mistake in its code that causes one or more inputs to result in incorrect output. Therefore, for each it is enough to find a single test case that exposes that bug. When you find the test case, your job is to characterize what the program is doing wrong. In other words, it is not enough to say, for example, that the program does not work when given the input (5,12,15). You also have to say something like, "it appears that the program sometimes says a triangle is acute when it is not", or "it appears that it sometimes accepts inputs that are not valid triangles." You will get partial credit if you find a bug but cannot describe it accurately. You will get full credit if you describe it accurately as well.

In case you do not remember your geometry, a triangle is ***acute*** if all angles in it are less than 90 degrees, it is ***obtuse*** if there is a single angle greater than 90 degrees, and it is a ***right*** triangle if there is 90 degree angle in it. These are mutually exclusive possibilities. A triangle is ***scalene*** if no two sides are equal, ***isosceles*** if exactly two sides are equal, and ***equilateral*** if all three sides are equal. The program `angleof` accepts command line arguments like `triangleclassify`.

```
angleof a b c
```

will output the number of degrees in the angle between the sides whose lengths are `a` and `b`. You might find this useful when you need to know whether the `classifytriangle` program correctly identified the type of triangle.

For this problem, you should create a single file named `triangletests`. For each of the buggy `classifytriangle` programs, write the test case that exposes an error and a description of what you believe is the error. There should be three short paragraphs in this file. Each correctly identified error is worth 1.5 out of 10 points for this lab.

What and How to Submit

Submit your work, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. The instructions are just like the previous lab's:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab06/submissions`
   whose name is your *username*.
2. Put all of your files into this directory. Make sure they are named correctly. There should be five files from Problem 1, three from Problem 2, and one from Problem 3. You will lose 5% of the grade if you misname the file!
3. Change the permission on the directory that you created so that no one else can read it. You do this with the commands
   `$ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab05/submissions`
   `$ chmod 700` *`username`*

Remember to add the comment lines described in the problems above. ***There are absolutely no extensions to the deadline for this lab.***

Lab 7: Array processing

Overview

This lab will give you practice with arrays, sorting and searching, and file I/O. The program will read input data from a file and produce its output in a file. The only interaction with the user will be limited to asking for the names of the files[1].

More About Opening Files

So far you have only used the `open()` member function of the `fstream` class to open files with fixed names, as in

```
ifstream fin;
fin.open("dna.txt");
```

which will attempt to open a file named `dna.txt` in the program's current working directory when it runs. If you want to ask the user for the name of a file to open, store it into a `string` variable, and then use it to open that file, you cannot write this:

```
ifstream fin;
string filename;
cin >> filename;
fin.open(filename); // error -- this will not work
```

because the `open()` function expects a different type of string called a ***C string***. C strings are the progenitors of C++ strings – they were defined in the C language originally and became part of C++ by the fact that they are part of C. People call them C strings to distinguish them from C++'s `string` class. You are not going to learn about C strings here[2], but you will learn how to circumvent this problem.

The `string` class has several functions that can be called on its `string` objects. You know about the `size()` function already. It also has a `c_str()` function. This function, whose name is pronounced "C string", does exactly what the doctor ordered for this problem – it returns the C string equivalent of the string itself.

For example, if `filename` is declared to be a `string`:

```
string filename;
```

and if the string "`../../data/mydata.txt`" is stored into it via an input instruction such as

```
cin >> filename;
```

then `filename.c_str()` will be the C string "`../../data/mydata.txt`". This means you can use it as the argument of `open()` as in the following:

```
ifstream fin;
string filename;
cin >> filename;
fin.open(filename.c_str() ); // this will open the file whose name the user entered
```

This is how you can ask the user to enter a file name and then try to open that file. Of course you need to check that the `open()` succeeded. This expands your programming toolkit for opening files.

[1] In a future lab exercise, you will learn how to write programs that can get these file names directly from the command line.

[2] You first need to learn about pointers.

Exercises

Problem 1. You will write a program that prompts the user to enter the names of an input file and an output file, and then reads input from the input file, processes the input, and sends its output to the output file. If the input files cannot be opened, or if the output file cannot be opened, the program returns with an error value (1). The input file will consist of real numbers separated by white space (blanks, tabs, newlines). These numbers should be thought of as sample values from some experiment. The numbers in the file are not in any particular order. The program will compute various statistics about the file's data and write these one per line into the output file in the order listed below:

1. the *smallest* number in the file,
2. the *largest* number in the file,
3. the *median* of the numbers in the file,
4. the *mean* of the numbers in the file, and
5. the *standard deviation* of the numbers in the file.

The *median* of a set of sample values (i.e., numbers) is the numerical value for which half of the numbers in the set are larger and half of the numbers in the set are lower. If there is an odd number of samples in the set, the median is one of the values. For example, if the list of numbers is 2.3, 8.4, 9.3, 10.0, 12.0, then the median is 9.3, since there are two values below 9.3 and two above it. If the set has an even number of values, the median is halfway between the two middle values. For example, if the list is 2.3, 8.4, 9.3, 10.0, then the median is halfway between 8.4 and 9.3, i.e., the average of 8.4 and 9.3, which is 8.85.

The *mean* is the sum of the values divided by the number of values. (It is the same as the average value.)

The *standard deviation* is defined as $\sqrt{\left(\sum_{i=0}^{N-1}(x_i-a)^2\right)/N}$, where $x_0,\ldots,x_{N-1}$ are the N values, and a is their mean. In order to compute the standard deviation, it is easiest to first compute the mean. If you have somehow made it into this class never having seen the summation symbol $\sum_{i=0}^{N-1}(x_i)$ it means the sum of the values $x_0, x_1, ..., x_{N-1}$.

You can assume that the file contains at most 1000 numbers. It might contain fewer, but never more, than that. It is possible that the file is empty, in which case the output file should be created and should be empty as well, but an error message should be written to the standard error stream (`cerr`).

Advice. Although there are algorithms that can find the median of a list of unordered numbers, the simplest solution is to first sort the numbers. Your program should sort the numbers before doing anything else. This will make the other computations much easier.

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. ***There is no grace period for this. Programs submitted after 2:00 PM will not be accepted.*** The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab07/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.
2. Copy your program, which should be named *`username`*`_lab07.cpp`. You will lose 5% of the grade if you misname the file!
3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

   ```
   $ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab07/submissions
   $ chmod 700 username
   ```

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document.

There are absolutely no extensions to the deadline. You can submit a revised version of the program by 10:00 P.M. on Thursday, Oct. 11 for partial credit. If you do, you must name it with a different name than what you first posted, e.g., *username*`_lab07_v2.cpp`. and put it in the same directory as the original. It must be a *revision* of the original file, with only fixes to things that were not working before. I will adjust the grade based on how well the first version worked and how many changes you made.

Lab 8: Arrays and Reusable Code

Overview

Large projects are usually not written as a single source code file, for various reasons:

- When more than one person works on a project, each person works on a different piece of it and it is more efficient if these pieces are in separate files, otherwise they would have to take turns editing the file.
- Putting frequently needed portions of code into distinct files makes it possible to incorporate this code into multiple projects without actually having to rewrite the code or have multiple copies of it, making that code reusable.
- If all of the code is in one large file, then each time any small part of it is changed, the entire file has to be recompiled. These days with such fast computers, compiling takes very little time so this is not an issue for small programs, but for very large programs, this can still take some time. If the program is composed of several small files, then only the changed file needs to be recompiled, and the program re-linked to the changed file. This is usually much faster than recompiling the entire program.
- Maintaining the code is easier because logically related code is all in a single small file, making it easier to find the parts of the code that need to be revised.

This exercise is designed to give you practice creating a multiple-file program whose parts can be compiled separately. It also gives you more practice with arrays, sorting, and file input and output.

Exercise

You will write a program that prompts the user to enter the names of an input file and an output file, and then reads input from the input file, processes the input, and sends its output to the output file. If the input files cannot be opened, or if the output file cannot be opened, the program returns with the error value 1. The input file will consist of lines containing single words, one per line. You can think of the words as the usernames of people who have used some online service. The service provider is interested in seeing usage patterns for its users, so they have commissioned you to do some very simple analysis of this data. Specifically, your program must read the words and store them into an array. It must sort that array using the usual comparison operation for strings (i.e., `str1 < str2` for strings `str1` and `str2`). The data will probably have many words that are the same, and when the data is sorted, these lines will be adjacent. For example, if the data file has the names

```
haley, sue, bingo, sam, sue, bingo, cranky, mel, sam, sue
```

(one per line, but here written separated by commas to save space), then after sorting it will look like

```
bingo, bingo, cranky, haley, mel, sam, sam, sue, sue, sue
```

Your program must create a new array that contains the data from the original array but without any repeats. In other words, it must create a new array that is also sorted, but such that every word in that new array is unique. For the above example, the array would contain

```
bingo, cranky, haley, mel, sam, sue
```

It must write the contents of this array to the output file whose name the user specified.

Program Design

The program must consist of exactly five files, named as follows:

- A header file named `file_io.h`
- An implementation file for this header file, named `file_io.cpp`
- A header file named `list.h`
- An implementation file for this header file, named `list.cpp`
- A main program file named `lab08_main.cpp`

The `file_io.h` header file must contain the prototypes of the two functions needed to do file I/O: one that reads the contents of an input file stream into an array, and one that writes the contents of an array into an output file stream. The following listing uses DOxygen-style comments to describe what these functions must do and what their prototypes should look like.

```
/** readlist(inputstream, names, limit, length)
 *  @param istream& inputstream [in/out]  the stream to read from
 *                                        one string per line
 *  @param string   names[]     [out]     the array to fill with strings
 *  @param int      limit       [in]      max size of the array
 *  @param int&     length      [out]     number of names stored in names[]
 *  @pre   inputstream is open && limit > 0 && names is size limit
 *  @post  0 <= length <= limit && names[0]..[length-1] is filled with strings
 *         from inputstream
 */

/** writelist(outputstream, names, length)
 *  @param ostream& outputstream [in/out] the stream to write to
 *  @param string   names[]      [in]     the array of strings to write
 *  @param int&     length       [in]     number of names stored in names[]
 *  @pre   outputstream is open && length >= 0 && names[0]..names[length-1]
 *                                        contains words to write, one per line
 *  @post  none
 */
```

`file_io.cpp` must contain the definitions of those two functions.

The `list.h` header file must contain the prototypes of the two functions needed to do array processing: one that sorts an array of strings, and one that, given a sorted array of strings, fills a second array with the unique list of words contained in the first array. Because the size of the second array may be smaller than the first, this function must also fill a parameter with the size of the second array. The following listing describes what these functions must do and what their prototypes should look like.

```
/** sort (list, length)
 *  @param string  list[]   [in/out] the array of strings to be sorted
 *  @param int     length   [in]     number of strings stored in list
 *  @pre   list[] has 0 or more strings
 *  @post  list[0] <= list[1] <= ... <= list[length-1] and every string
 *         that was in list before is in list exactly once after sort finishes
 *         (none got deleted or replicated)
 */

/** uniq (origlist, newlist, origlength, newlength)
 *  @param string  origlist[] [in]  a sorted array of strings
```

```
 *  @param string  newlist[]  [out] an array of strings containing no duplicates
 *  @param int     origlength [in]  number of strings stored in origlist
 *  @param int&    newlength  [out] number of strings stored in newlist
 *  @pre   origlist[] is sorted (and may have duplicate adjacent strings)
 *  @post  newlist is sorted and has unique strings && newlength <= origlength
 *         && newlength is the number of elements in newlist.
 */
```

The file `list.cpp` must contain the definitions of those two functions.

The main program, `lab08_main.cpp`, will be very small. It must prompt the user to enter the names of the files, open streams for these files, and then call the functions in the other files to (1) read the input file data into the array, (2) sort it, (3) create the unique word array, and (4) print that unique word array to the output file stream, one word per line.

Advice. First create the two header files. Then create your main program. Then create the two implementation files. You can check the main program file by compiling with the `g++ -c` option. This will not attempt to do any linking, so you can see programming errors in `main()` even if you do not have the implementation files.

There are files in the lab08 directory containing a list of names that you use for testing your program. The command

```
$ sort -u filename
```

will sort the contents of the file `filename`, removing duplicates, and display the result on the standard output. You can redirect it to a file using

```
$ sort -u filename > sortedfile
```

You can check if your program is correct by comparing your output file to the file produced by the above command using the `diff` command. diff has no putput if two files are identical:

```
$ diff file1 file2
```

will produce no output if `file1` is an exact copy of `file2`.

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. ***There is no grace period for this. Programs submitted after 2:00 PM will not be accepted.*** The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab08/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.
2. Copy your five files, which should be named as described above, to that directory. You will lose 5% of the grade if you misname the files! Make sure that each file has a preamble with your name and other appropriate information in it.
3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

   ```
   $ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab08/submissions
   $ chmod 700 username
   ```

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document. There are absolutely no extensions to the deadline.

Lab 9: Structures and Data Abstraction

Overview

This exercise is a prelude to the design of classes. C++ structures can be used, with discipline, to design abstract data types. They do not provide the protections that classes do, but they are a good means to warm up to classes. The exercise also gives you more practice creating a multiple-file program whose parts can be compiled separately, and still more practice with file I/O.

In this lab you will define and implement an abstract data type that represents a student record. It will consist of a structure and associated functions for manipulating the data in the student record.

Review of Data Abstraction and Abstract Data Types

Procedural abstraction is the separation of what a function does from how it does it. The idea is to write descriptions of what functions do without actually writing the functions, and separate the *what* from the *how*. The client software, i.e., the software calling the function, only needs to know the parameters to pass to it, the return value, and what the function does; it should not know how the function works. In this sense, functions become like black boxes that perform tasks.

Data abstraction separates what can be done to data from how it is actually done. It focuses on the operations of data, not on the implementation of the operations. For example, in data abstraction, you might specify that a set of numbers provides functions to find the largest or smallest values, or the average value, but never to display all of the data in the set. It might also provide a function that answers yes or no to queries such as, "is this number present in the set?" In data abstraction, data and the operations that act on it form an entity, an object, inseparable from each other, and the implementation of these operations is hidden from the client.

Information hiding takes data abstraction one step further. Not only are the implementations of the operations hidden within the module, but the data itself can be hidden. The client software does not know the form of the data inside the black box, so clients cannot tamper with the hidden data.

An **abstract data type** is a representation of an object. It is a collection of data and a set of operations that act on the data. An ADT's operations can be used without knowing their implementations or how data is stored, as long as the interface to the ADT is precisely specified.

There are two views of an abstract data type: its **public view**, called its **interface**, and its **private view**, called its **implementation**. The parts of a class that are in its public view are said to be *exposed by the class*.

Example

A soft drink vending machine is a good analogy. From a consumer's perspective, a vending machine contains soft drinks that can be selected and dispensed when the appropriate money has been inserted and the appropriate buttons pressed. The consumer sees its interface alone. Inputs are money and button-presses. The vending machine outputs soft drinks and change. The user does not need to know how a vending machine works, only what its abstract data type is.

The vending machine company's support staff need to know the vending machines internal workings, its data structure. The person that restocks the machine has to know the internal structure, i.e., where the spring water goes, where the sodas go, and so on. That person is like the programmer who implements the ADT with data structures and other programming constructs.

The ADT is sometimes characterized as a wall with slits in it for data to pass in and out. The wall prevents outside users from accessing the internal implementation, but allows them to request services of the ADT.

ADT operations can be broadly classified into the following types:

- operations that add new data to the ADT's data collection,
- operations that remove data from the ADT's data collection,
- operations that modify the data in the ADT's collection,
- operations to query the ADT about the data in the collection.

The `student` ADT

The student abstract data type contains the student's last name, first name, id (9 digit number with possible leading zeros), GPA (a real number in the range of 0.0 to 4.0, a whole non-negative number of credits completed to date, and a single letter status indicating full time ('F') or part time ('P').

The functions that the ADT should make available to client software (i.e., should expose) are:

1. Given a first name, last name, id, and status, create a new student record with those values, setting the GPA and number of completed credits to zero.
2. Given a student record, return the student's id.
3. Given a student record, return the student's last name.
4. Given a student record, return the student's status.
5. Given a student record and a new status, change the student's status.
6. Given a student record and a number of credits, change number of completed credits for a student by adding the new number to the current number of credits.
7. Given a student record and a GPA, change the student's GPA to the new value.
8. Given a student record and an output stream, print the members of the record on that output stream in the format:

   ```
   lastname  firstname  id  status  GPA  credits_earned
   ```

   separated by tab characters (`'\t'`).

Exercise

You will write a small program that implements the `student` ADT described above. Specifically, you will first design a header file that contains a structure definition that represents the student data, as well as the prototypes of all functions described above. Each function prototype must be thoroughly documented. Name this header file `student.h`.

You will then create an implementation file named `student.cpp` in which you implement each of the above eight functions.

This exercise is a bit unusual because the main program does not do anything particularly logical. It exists just to exercise your student ADT. Specifically, it must do the following:

- The main program must read a file containing at most ten lines of the form

  ```
  lastname  firstname  id  status
  ```

  and for each line, create a student record with that data and a GPA and total credits earned of 0. These records should be elements of an array of student records. For simplicity assume that the file is named `studentdata`. You may hard-code its name into the main program.

- After the data is read in, the main program must display on the screen the last name and the id of every student record.

- It must then display the last name of every student whose status is 'F'.
- The program must then change the GPA of every student to 4.0, the credits earned to 16, and the status to 'P'.
- Finally, the main program must write all of the records to a file named `newstudentrecords`.

Remember that the main program is not allowed to access the data members of the structure. It must do everything only by calling the functions whose prototypes are in your header file. Name the main program `lab09_main.cpp`.
In short, the program must consist of exactly three files:

- A header file named `student.h`
- An implementation file for this header file, named `student.cpp`
- A main program file named `lab09_main.cpp`

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. ***There is no grace period for this. Programs submitted after 2:00 PM will not be accepted.*** The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab09/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.
2. Copy your three files, which should be named as described above, to that directory. You will lose 5% of the grade if you misname the files! Make sure that each file has a preamble with your name and other appropriate information in it.
3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

   ```
   $ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab09/submissions
   $ chmod 700 username
   ```

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document. There are absolutely no extensions to the deadline.

Lab 10: Classes and Data Abstraction

Overview

This lab will give you practice designing and implementing a simple class. It is similar to Lab 9, except that I have written most of a main program that you need to solve this problem. I include the same review of data abstraction that was contained in Lab 9's description.

Review of Data Abstraction and Abstract Data Types

Procedural abstraction is the separation of what a function does from how it does it. The idea is to write descriptions of what functions do without actually writing the functions, and separate the *what* from the *how*. The client software, i.e., the software calling the function, only needs to know the parameters to pass to it, the return value, and what the function does; it should not know how the function works. In this sense, functions become like black boxes that perform tasks.

Data abstraction separates what can be done to data from how it is actually done. It focuses on the operations of data, not on the implementation of the operations. For example, in data abstraction, you might specify that a set of numbers provides functions to find the largest or smallest values, or the average value, but never to display all of the data in the set. It might also provide a function that answers yes or no to queries such as, "is this number present in the set?" In data abstraction, data and the operations that act on it form an entity, an object, inseparable from each other, and the implementation of these operations is hidden from the client.

Information hiding takes data abstraction one step further. Not only are the implementations of the operations hidden within the module, but the data itself can be hidden. The client software does not know the form of the data inside the black box, so clients cannot tamper with the hidden data.

An **abstract data type** is a representation of an object. It is a collection of data and a set of operations that act on the data. An ADT's operations can be used without knowing their implementations or how data is stored, as long as the interface to the ADT is precisely specified.

There are two views of an abstract data type: its **public view**, called its **interface**, and its **private view**, called its **implementation**. The parts of a class that are in its public view are said to be *exposed by the class*.

The `student` class

The `student` class represents a student record in a single college class. It contains the student's last name, first name, email address, a set of five homework grades, the number of graded homeworks, a set of three exam scores, and the number of graded exams. The homework and exam scores are whole numbers. If the number of graded homeworks is N < 5, then the first N homeworks are the ones with grades, and the same holds true for exams.

The member functions that the `student` class should make available to client software (i.e., should expose) are:

1. A function which, when given a first name, last name, and email address, initializes the student record with those values, setting all homework and exam scores to zero.
2. A function which returns the student's first name.
3. A function which returns the student's last name.
4. A function which returns the student's email address.
5. A function which returns the list of graded homework scores.
6. A function which returns the list of graded exam scores.

7. A function which returns the student's current composite grade, which is 0.6 times the average of all graded exams plus 0.4 times the average of all graded homeworks

8. A function to add a new homework grade to the student's record, provided the total number of homework grades is no more than 5. If the total is already 5, it should return an error value.

9. A function to add a new exam grade to the student's record, provided the total number of exam grades is no more than 3. If the total is already 3, it should return an error value.

10. A function to output to a given stream, the student data in the following format:

    ```
    lastname firstname email N hwkscore1 ...hwkscoreN M examscore1 ... examscoreM
    ```

    where `N` is the number of graded homeworks and `M` is the number of graded exams, and `hwkscore1`, etc are the scores on the homeworks and `examscore1`, etc are the exam scores.

Exercise

You will write a program that implements the `student` class described above. Specifically, you will design the `student` class header and implementation files, named `student.h` and `student.cpp` respectively. Each member function must be thoroughly documented. in the header file.

I will supply most of a main program. This program will prompt the user to enter the name of an input file, and then read input from that file. The file will consist of lines that represent student records, and these will be stored in an array of such records by the main program. The program will display a menu-driven interface that will allow the user to interactively add new grades to a given student's record, display a student's grades and/or grade average, and display the set of all student records. The main program will be missing various pieces, which have comments as placeholders. You will have to fill in the missing pieces.

The

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. ***There is no grace period for this. Programs submitted after 2:00 PM will not be accepted.*** The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab10/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.

2. Copy your three files, which should be named `student.h`, `student.cpp`, and `lab10_main.cpp`, to that directory. You will lose 5% of the grade if you misname the files! Make sure that each file has a preamble with your name and other appropriate information in it.

3. Change the permission on the directory that you created so that no one else can read or modify it.

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document.

Lab 11: Write your own string class

In this lab you will get practice designing and implementing a class with constructors, and working with C strings. This project is a pair programming project. You will be working in pairs following the guidelines and procedures described in http://ecee.colorado.edu/~ecen2120/Manual/pair.html. I have taken excerpts from this University of Colorado webpage as the basis for a short tutorial on pair programming, which is included in the lab's directory on the server. Before you begin this assignment, make sure you know who is the driver and who is the navigator, and know what your respective roles are.

Exercise

Write a program that defines a class called `String`. The class should have a data member that is an array of 80 characters to store the characters of the string (yes, the largest number of characters in an object of type `String`, including the newline character at the end, is 80). A proper empty object of type `String` should have a single character it in (this character is `'\0'`, and is located in the first array location. You should have the following functions for this class. The functions are described in plain sentences. You must determine their signatures and create their prototypes.

- A default constructor that creates an empty string;
- A constructor that initializes the object to the C string passed as its argument;
- `copy()`, which copies its C string argument into the current object, overwriting whatever was in the current object before;
- `concat()`, which appends its C string argument at the end of the string in the current object, provided that there is storage available;
- `size()`, which returns the current size of the string in the object;
- `str()`, which returns a constant pointer to the current C string in the object, exactly as the `c_str()` member function of the C++ `string` class does.

If the C string passed to either `copy()` or `concat()` would cause the internal `String` array to be overflowed, then the function should copy whatever can be copied and make sure that the last character is set to the `NULL` character.

You must write a `main()` function that demonstrates the correctness of your `String` class. The `main()` function must ensure that every function of the class is called, and it must demonstrate that the function works properly. It should purposely try to overflow the array and show what happens as a result.

You are free to use the C string library functions if you like, or you may implement the functionality without using any library functions. It is your choice.

For full credit, make sure all documentation meets the standard rules. Think of ways to make your program efficient in its use of time and its use of storage.

Extra Credit

If you complete all the above, try to add a few other features to the class. You can add the following functions and modify the `main()` function to demonstrate their correctness. You will not get credit for any one of them unless it is correct and shown to be so in `main()`, so try them one at a time.

- An overload of `copy()` that has a parameter of type `String` and has the same action as the original `copy()`; (10% extra)

- An overload of `concat()` that has a parameter of type `String` and has the same action as the original `concat()`; (10% extra)
- A `comparedto()` function that has a parameter that is either a C string or an object of type `String` and performs comparison, returning the same result as the C library `strcmp()` would. I.e., if `mystring` and `yourstring` are `String` objects, then `mystring.comparedto(yourstring)` would return a negative value if `mystring` precedes `yourstring`, zero if they are identical, and positive if `mystring` follows `yourstring`. (10% extra)

What to Submit

Because this is a pair programming project, the submission requirement is slightly different. As usual, you will submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. But what you submit is differrent. The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab10/submissions`
   whose name is *username*1_*username2*, where these are the usernames of the two people working on the project.
2. Copy all files, which should be named `String.h`, `String.cpp`, and `lab11_main.cpp`, to that directory. You will lose 5% of the grade if you misname the files! Make sure that each file has a preamble with *both names* and other appropriate information in it. Make sure both partners have a copy of these files in their home directories.
3. Change the permission on the submitted directory that you created so that no one else can read or modify it. It does not matter who the owner of that directory is as long as both names are part of its name.

Do not submit executable files. Your work will be graded based on the rubric outlined in the Programming Rules document.

Lab 12: Extending the String class

In the last lab, you developed your own version of the C++ string class, with a very small set of member functions. In this lab you will get more practice designing and implementing a class and working with C strings and character data. This project is again a pair programming project. You will be working in pairs following the guidelines and procedures described in http://ecee.colorado.edu/~ecen2120/Manual/pair.html. I have taken excerpts from this University of Colorado webpage as the basis for a short tutorial on pair programming, which is included in the lab's directory on the server. Before you begin this assignment, make sure you know who is the driver and who is the navigator, and know what your respective roles are.

Exercise

Write a program that extends the `String` class that was developed in Lab 11. The class should have a data member that is an array of 80 characters to store the characters of the string including the `NULL` character at the end. A proper empty object of type `String` should have a single character it in (this character is `'\0'`, and is located in the first array location). The `String` class specified in the previous lab was defined to have the following functions:

- A default constructor that creates an empty string;
- A constructor that initializes the object to the C string passed as its argument;
- `copy()`, which copies its C string argument into the current object, overwriting whatever was in the current object before;
- `concat()`, which appends its C string argument at the end of the string in the current object, provided that there is storage available;
- `size()`, which returns the current size of the string in the object;
- `str()`, which returns a constant pointer to the current C string in the object, exactly as the `c_str()` member function of the C++ `string` class does.

This extension to the `String` class must also have the additional functions described in plain English below. You must determine their signatures and create their prototypes.

- A constructor that initializes the object to the `String` object passed as its argument;
- `find()`, which returns the index of the first occurrence of its string argument in the string in the `String` object, or -1 if it is not found;
- `find()`, which returns the index of the first occurrence of its string argument in the `String` object after a given position in the string in the `String` object specified as its second argument, or -1 if it is not found; e.g. `find(str,6)` returns the position of the first occurrence of `str` in the `String` object starting at or after position 6;
- `toupper()`, which changes the string in the object by changing all lowercase letters in it to uppercase letters;
- `tolower()`, which changes the string in the object by changing all uppercase letters in it to lowercase letters;
- `replace()`, which replaces every occurrence of its second character argument by its first character argument in the string in the object. In other words, `replace('a', '1')` would replace all `'1'`s by `'a'`s. As a special case, if `replace()` has just a single argument, it replaces every character in the string by that character, so `replace(' ')` replaces all characters by the blank.

You must write a `main()` function that demonstrates the correctness of the new features of the `String` class. The `main()` function must ensure that every new function of the class is called, and it must demonstrate that the function works properly.

You are free to use the C string library and C character handling functions if you like, or you may implement the functionality without using any library functions. It is your choice. You are free to use my solution to Lab 11 as a starting point.

For full credit, make sure all documentation meets the standard rules. Think of ways to make your program efficient in its use of time and its use of storage.

What to Submit

Because this is a pair programming project, the submission requirement is slightly different. As usual, you will submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. But what you submit is differrent. The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab12/submissions`
   whose name is *username*1_*username2*, where these are the usernames of the two people working on the project.
2. Copy all files, which should be named `String.h`, `String.cpp`, and `lab12_main.cpp`, to that directory. You will lose 5% of the grade if you misname the files! Make sure that each file has a preamble with *both names* and other appropriate information in it. Make sure both partners have a copy of these files in their home directories.
3. Change the permission on the submitted directory that you created so that no one else can read or modify it. It does not matter who the owner of that directory is as long as both names are part of its name.

Do not submit executable files. Your work will be graded based on the rubric outlined in the Programming Rules document.

Lab 13: Using Vectors

In this lab you will get practice working with ***vectors***. The project will use a class similar to the `student` class that was created in Lab 10. You will write a main program that uses my versions of the `student.h` and `student.cpp` files.

Exercise

Write a program that creates a `vector` whose base type is the class `student`. The vector is initially empty. It should be filled with the data coming from an input file. The program should have a single argument on the command line that is the name of the input file. For this lab assume that the file contains valid data. Each line consists of a record for a single student and has the following format:

```
last_name first_name id status gpa number_of_credits
```

The last two fields on the line are optional (they are either both there, or both missing). The entries on a single line are separated by any number of space characters, meaning spaces or tabs.

Once the list of students from the file is read into your vector, your program should display a menu that gives the user the following options:

```
[a] Display all records, in alphabetical order by last name.
[g] Display all records sorted in descending order of gpa.
[q] Exit.
```

and repeatedly respond to the user's choice and then redisplay the menu until the user quits. In all cases, when the records are displayed, the fields must be tab-separated. For option `'a'`, if last names are the same, first names should be used to break ties. In option `'g'`, if gpas are the same, any order is acceptable.

Implementation Suggestions

Since the number of entries on a line in the input file is variable, you cannot use the >> stream extraction operator for reading one element at a time. You need to read an entire line and then parse that line. There are a few different ways that you can do this, such as using the global `getline()` function to read into a C++ string, or the `istream::getline()` member function to read into a C string. Once your data is in a string of either kind you have many ways to extract the fields.

If you store into a C string, you could use one of the C string library functions to get the words within the string.

If you store into a C++ string, you could also do this by extracting the C string from it, or you could use one of several `string` class member functions, such as:

`find_first_of()` Find character in `string` (public member function)

`find_last_of()` Find character in `string` from the end (public member function)

`find_first_not_of()` Find absence of character in `string`

`find_last_not_of()` Find absence of character in `string` from the end (public member function)

You can find a more detailed explanation and the exact parameter list on the website for the `string` class. Regardless of which method you choose, remember that the gpa and number of credits either are both present or both absent from the line.

Extra Credit

- If the gpa is present but not the number of credits, catch this. (5%)
- If the gpa or number of credit is not numeric, catch this (5%)
- In the gpa sort, use the last name in increasing alphabetic order as the tie-breaker. (10%)

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. ***There is no grace period for this. Programs submitted after 2:00 PM will not be accepted.*** The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab13/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.
2. Copy your file, which should be named `lab13_main.cpp`, and the `student.h` and `student.cpp` files, to that directory. You will lose 5% of the grade if you misname the file! Make sure that each file has a preamble with your name and other appropriate information in it.
3. Change the permission on the directory that you created so that no one else can read or modify it.

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document.

Assignment 1: Computing with DNA sequences

Summary

Deoxyribonucleic acid (DNA) is a nucleic acid containing the genetic instructions used in the development and functioning of all known living organisms. In this assignment you will be extracting information from strands of DNA, which you can simply think of as sequences of the letters `a`, `c`, `g`, and `t` in any order (assume that the letters are always in lower case). Although you do not need to know anything else about DNA other than what is written here, it would be useful to read my tutorial containing biological background information (found here: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci135/biobackground.pdf) to get a better perspective on the project.

You will write a program that performs several computations given strands of DNA read from a file. The results of computations should be written to the standard output stream and any error messages should be sent to the standard error stream.

Input

The input DNA strings will come from a file that the program is expected to open and read. The program should expect that the file is named "`dna.txt`" and is in its current working directory. However, if the program cannot open this file for any reason, it should report this error on the standard error stream and then exit gracefully. This might happen for a number of reasons, including that the name is wrong, the file is not there, or the permissions on the file are wrong.

The file will contain two DNA strings, on separate lines. There will be no blank lines in the file. The program should not assume that the strings are valid DNA strings. If for some reason, any of the strings have characters other than `a`, `c`, `g`, or `t` it should report the error on the standard error stream and gracefully exit.

The DNA strings in the file will not be larger than 4095 characters long. This is an arbitrary length chosen just for the assignment.

Computational Tasks

The program must be able to perform the following computational tasks. Each of these must be implementing by a user-defined function.

GC Content: The *GC content* of a single DNA strand is the ratio of the total number of `c`'s and `g`'s to the length of the strand. For example, the sequence '`atcgtttgga`' is of length 10 and has a total of 4 `c`'s and `g`'s, so its GC content is 0.4.

Given a DNA string, count the GC content of the string and report this as the fraction (number of `c`'s and `g`'s)/(length of string). The result of this computation is therefore a number between 0 and 1 inclusive.

Poly-T sequences: A *poly-T sequence* of length at least N is a sequence of N or more consecutive `t` nucleotides. It is maximal if it is not contained in a longer poly-T sequence.

Given a DNA string, count the number of occurrences of maximal poly-T sequences of length at least N in the string and report this as a whole number.

For example, `actttaattttactttcctta` has 3 poly-t sequences of length 3 or more:

```
actttaattttactttcctta
```

and only one of length 4 or more:

```
actttaattttactttcctta.
```

Note: The sequence `tttt` has only one maximal poly-T sequence of length at least 3. Similarly, the sequence `ttttttt` has only one maximal poly-T sequence of length at least 3. This is because we are looking for sequences of 3 or more consecutive `t`'s.

Mutations: A *mutation* is a change in one or more bases in the DNA sequence. Mutations are essential to evolution.

Given two DNA strands, not necessarily of the same length, count and report the positions at which they differ. If one is longer than the other, then treat each extra positions in the longer as a mutation. For example, the two strings '`aaaataa`' and '`aaaaaaaccc`' differ in just one common position, the fifth letter, but the second has three extra letters, so the result should be 4 (1 plus 3).

Phylogenetic distance: The *phylogenetic distance* between two DNA strands is the number of mutations that occurred in the transition from one strand to the other.

Given two DNA strands of equal size, compute and output the phylogenetic distance between them, measured as the number of mutations between them divided by the length of the strings, producing a value between 0 and 1. For example the phylogenetic distance between '`aaaataa`ggg' and '`aaaaaaaccc`' is $4/10 = 0.4$.

User Interface

This program provides a menu driven user interface. It should display the following menu on the screen:

1 compute GC content of DNA string 1
2 compute GC content of DNA string 2
3 compute number of poly-T sequences of length N or more in DNA string 1
4 compute number of poly-T sequences of length N or more in DNA string 2
5 compute number of mutations between the two DNA strings
6 compute phylogenetic distance between the two DNA strings
7 quit

After displaying the menu the program should display a prompt and wait for the user to enter one of the above numbers. When the user enters a valid choice (a digit 1, 2, 3, 4, 5, 6, or 7) the task should be performed and then menu should be redisplayed unless the user chooses to quit.

When the user enters either 3 or 4 to compute the number of poly-T sequences of length N or more, he/she should be prompted for the value of N. N can be any integer greater than zero. Your program should verify that the number entered is greater than zero. If the user types anything other than one of the numbers 1 through 6, the program should display the message "`Invalid choice. Try again.`"

Programming Rules

Your program must conform to the programming rules described in the Programming Rules document on the course website. Remember in particular that:

- You must write your program yourself. I will on occasion ask you to explain the code to me, so you need to make sure that you understand the code that you write. If you want to use features of C++ that we did not cover in class, you need to understand them thoroughly.
- You must document your program (preamble, function pre- and post-conditions, comments throughout the code, appropriately choosing variable and function names).
- Input from the standard input stream and output to the standard output and error streams should be performed by the `main` function only, i.e. no other functions should perform any I/O operations).
- Late programs lose 20% for each 24 hours they are late.

Also, you must implement and use at least four functions that perform the four computational tasks described above. Each of these functions has to take at least one parameter, i.e. the DNA string. It may take other parameters if you want. It is up to you whether the main program or your functions should validate that the string is a valid DNA string. (Remember that a valid DNA string contains only the letters `a`, `c`, `g`, and `t` in any order and in lower case.)

Grading

The program will be graded based on the following rubric.

- Does the program compile (on a cslab machine): 15%
- Correctness and implementation of the four functions: 45%
  - compute GC content (10%)
  - compute number of poly-T sequences of length N or more (15%)
  - compute mutation between two strings (10%)
  - compute phylogenetic distance (10%)
- Correctness and implementation of the main function and any other functions in the program: 20%
- Documentation: 15%
- Style and proper naming: 5%

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on October 4, 2012. You need to submit a single file for this assignment. Its name should be *username*`_hwk1.cpp.` where *username* is to be replaced by your username on our system. Do not name it anything else. If it is named anything else, it loses 3% of the program's value. Before you submit the assignment, make sure that it compiles and runs correctly on one of the cslab machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to put your file into the directory

```
/data/biocs/b/student.accounts/cs135_sw/cs135projects/project1
```

Give it permission 600 so that only you have access to it. To do this, cd to the above directory and run the command

```
chmod 600 yourfilename
```

where *yourfilename* is the name of the file you put in that directory.

If you put a program there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date.

Assignment 2: A Library Catalog

Summary

This program manages the catalog of an old-fashioned library, i.e., one that actually has books. A catalog entry for a book consists of a unique catalog number, the author's last name, the author's first name, the name of the book, the genre, and whether it is available for checkout. The program is not very useful or robust because that would be too hard for a second project, but it will give you practice in many different aspects of computing. The program will open two input files specified on the command line, one containing a database of library holdings (books), and a second containing a list of requested books. The list of requested books is something like a batch processing system for the library. Imagine that at the end of every day, all requests made that day, which were recorded in a file, are then processed. This is what that file represents.

The program will produce three reports, one for each of the computational tasks described below, and these will be saved in three different output files whose names are also specified on the command line. The order of the file names on the command line is as follows:

1. library holdings input file
2. book request input file
3. author-sort output file
4. genre-sort output file
5. checkout output file

Input

There are two input files. If an input file is missing, the program should output an error message and quit. You can assume the input files will have the formats described below; there is no need to error check the format.

Library Holdings File

The ***library holdings file*** contains a list of all of the library's books in a specific format. Each line represents a single book, and will have six tokens (fields). Tokens are separated by tab characters in each line. Tokens can have non-tab whitespace characters in them. Let <T> denote a tab character. Then a line has the form

catalog number<T>*author last name*<T>*author first name*<T>*book title*<T>*genre*<T>*availability*

A sample input file might look like the following, in which tabs are also denoted by <T>:

```
C0594<T>Woolf<T>Virginia<T>A Room of One's Own<T>essay<T>available
C7495<T>Gibson<T>William<T>Neuromencer<T>science fiction<T>available
C1934<T>Heidegger<T>Martin<T>Being and Time<T>philosophy<T>unavailable
C8394<T>Kundera<T>Milan<T>The Unbearable Lightness of Being<T>novel<T>available
```

There will be no more than 10000 entries in the library holdings file.

Book Request File

The ***book request file*** contains a list of catalog numbers, one per line. These are the books being requested by various library users, and the program should check if they are available. A sample input file might look like the following:

```
C0594
C7495
C1934
C2237
```

There will be no more than 500 entries in the book request file.

Computational Tasks

There are three different tasks that the program must perform on the input files.

author-sort. List all of the books in the library in ascending alphabetical order of author last name. If two authors have the same last name, they should be listed in ascending order by first name.

genre-sort. List all the books in the library in ascending alphabetical order of genre. Within a genre, books should be ordered by author last name in ascending alphabetical order.

checkout. Given a list of requested catalog numbers, check if those books are available for checkout, and if so, for each one, indicate availability.

User Interface

There is no user interface to this program. The arguments are given on the program command line, and they are the names of files that the program uses for its input.

Output

The program will produce three output files with the names specified on the command line. If those files exist already they will be overwritten by the program.

Author-Sort Output File

The output file for the author-sort task will list the library holdings sorted by author last name, in ascending alphabetical order. The format is the same as that of the "library holdings input file". If two authors have the same last name, the books should be listed in ascending order by first name within that set of last names.

Genre-Sort Output File

The output file for the genre-sort task will list the library holdings sorted by genre, in ascending alphabetical order. The format is the same as that of the "library holdings input file". Within a genre, books should be ordered by author last name in ascending alphabetical order.

Checkout Output File

The output file for the checkout task will list the catalog numbers appearing in the "book request input file" followed by the corresponding title and availability. The books should be listed in the same order in which they appear in the input file. If a requested catalog number does not appear in the library holdings, the program should list the catalog number followed by the word unavailable.

A sample output file for computational task 3 might be:

```
C0594 A Room of One's Own available
C7495 Neuromencer available
C1934 Being and Time unavailable
C2237 unavailable
```

Programming Rules

Your program must conform to the programming rules described in the Programming Rules document on the course website. Remember in particular that:

- You must write your program yourself. I will on occasion ask you to explain the code to me, so you need to make sure that you understand the code that you write. If you want to use features of C++ that we did not cover in class, you need to understand them thoroughly.
- You must document your program (preamble, function pre- and post-conditions, comments throughout the code, appropriately choosing variable and function names).
- All input and output must be performed by the `main` function only, i.e. no other functions should perform any I/O operations. The functions that perform the computational tasks must pass all data through their parameter lists.
- Late programs lose 20% for each 24 hours they are late.

Grading

The program will be graded based on the following rubric.

- Does the program compile (on a cslab machine): 15%
- Correctness and implementation of the book class: 10%
- Correctness and implementation of the sort and search functions: 15%
- Correctness and implementation of I/O, `main()`, and any other features in the program: 35%
- Correctness of multiple file implementation and modularity: 10%
- Documentation: 10%
- Style and proper naming: 5%

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on November 5, 2012. You need to submit multiple files for this assignment, so they must be placed into a directory and zipped up as the instructions below describe. The name of your directory must be ***username*`_hwk2.`** where ***username*** is to be replaced by your username on our system. Do not name it anything else. If it is named anything else, you lose 3% of the program's value. Before you submit the assignment, make sure that it compiles and runs correctly on one of the cslab machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to create the above named directory and place all of your source code files into it. Do not place any executable files or object files into this directory. You will lose 1% for each file that does not belong there. With all files in your directory, run the command

```
zip -r username_hwk2.zip ./username_hwk2
```

This will compress all of your files into the file named `username_hwk2.zip`. Put this zip file into the directory

```
/data/biocs/b/student.accounts/cs135_sw/cs135projects/project2
```

Give it permission 600 so that only you have access to it. To do this, `cd` to the above directory and run the command

```
chmod 600 username_hwk2.zip
```

If you put a file there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date.

Assignment 3: Playlist Creator

Summary

Many software audio players let the user organize his or her music in various playlists that are saved as separate files without duplicating the music files themselves. In this assignment you will design and develop a program named `jukebox` that implements a simple playlist creator. It will be able to open a database of songs and create a new playlist consisting of a subset of the songs in the database. This project will exercise your knowledge of classes, string processing, and general program organization. It is the largest of the programs you will have written. The program will be run with a single comment line argument that specifies the name of the song database file that the program will open. If there is no command line argument, the program will attempt to open a file in its current working directory named `songs.csv`. If there is no command line argument and no file named `songs.csv` in the current working directory, the program will exit with a useful error message to the user.

Program Description

The program should open a file that contains a database of songs. The input file will contain zero or more songs, one per line. The format of this file is specified below in the section entitled *Input*. If the program is unable to open the song database for one reason or another, it will display an error message and exit. The error handling is described below in the *Input* section. If the program successfully opens the song database file, it will then prompt the user to enter the name of the playlist that he or she wishes to create. Once the user enters the name of the playlist, the program will display a menu of choices to the user. The user will be able to view the songs from the database in various ways and select songs from among the listed songs to add to the playlist. The user will also be able to save the playlist to an output file.

User Interface

The program will display the following menu:

```
[A/a] List all.
[R/r] List all matching an artist keyword ...
[T/t] List all matching a title keyword ...
[V/v] View the playlist.
[S/s] Save
[Q/q] Exit.
```

The details follow.

`List all.` All songs in the database should be printed on the screen , 20 songs at a time, followed by the prompt string "`action:`". The user should either type `next` to see the next 20 songs, or `add` followed by one or more numbers $n_1, n_2, ..., n_k$ to add the song with indices $n_1, n_2, ..., n_k$ to the playlist, or `stop` to stop the display of songs and redisplay the main menu.

`List all by matching an artist keyword ...` The user should be prompted for the keyword to match. A keyword cannot contain blanks or tabs. Then the program should display on the screen, 20 songs at a time, all the songs in the database whose artist matches or contains the keyword, followed by the prompt string "`action:`". The user should either type `next` to see the next 20 songs, or `add` followed by one or more numbers $n_1, n_2, ..., n_k$ to add the song with indices $n_1, n_2, ..., n_k$ to the playlist, or `stop` to stop the display of songs and redisplay the main menu.

`List all by matching a title keyword ...` The user should be prompted for the keyword to match. A keyword cannot contain blanks or tabs. Then the program should display on the screen, 20 songs

at a time, all the songs in the database whose title matches or contains the keyword, followed by the prompt string "`action:`". The user should either type `next` to see the next 20 songs, or `add` followed by one or more numbers $n_1, n_2, ..., n_k$ to add the song with indices $n_1, n_2, ..., n_k$ to the playlist, or `stop` to stop the display of songs and redisplay the main menu.

`View the playlist.` This is how the user can delete songs from the playlist. All songs in the playlist should be printed on the screen , 20 songs at a time, followed by the prompt string "`action:`". The user should either type `next` to see the next 20 songs, or `remove num` to remove the song with index `num` from the playlist.

`Save.` The playlist should be saved to a file whose name matches the name of the playlist. (See the format of the output file below). If the file exists, it will be overwritten by the new playlist[1].

Exit. The program exits.

Displaying a list of songs. When the user chooses any of the first four menu items above, the program should display the list of songs, 20 songs at a time (a smaller number of songs should only be printed on the last screen). Each song should be displayed on a single line using no more than 80 characters (since this is the typical width of the terminal window). You should organize the information as follows:

```
NNNN. AAAAAAAAAAAAAAAAAAAA TTTTTTTTTTTTTTTTTTTTTTTTTTTTTT BBBBBBBBBB MM:SS YYYY
```

`NNNN.` is a four digit, right justified index number of a song in the playlist followed by a single dot.

`AAAAAAAAAAAAAAAAAAAA` is a twenty character wide, left justified name of the artist, possibly with spaces.

`TTTTTTTTTTTTTTTTTTTTTTTTTTTTTT` is a thirty character wide, left justified title, possibly with spaces.

`BBBBBBBBBB` is ten character wide, left justified name of an album containing the song.

`MM:SS` is the duration of the song displayed using exactly two digits for minutes, followed by the colon, followed by exactly two digits for seconds.

`YYYY` is the year associated with the song.

If the exact artist name, title, or album name does not fit in those limits, they should be truncated. If it is too short, the rest of the allotted characters should be filled with spaces.

Each entry on the line is separated by a single space.

A sample display may appear as follows:

```
   1. Adele               Melt My Heart To Stone         19          03:23 2008
   2. Adele               First Love                     19          03:10 2008
  13. Norah Jones         Wish I Could                   Not Too La  04:18 2007
  44. Norah Jones         Not Too Late                   Not Too La  03:31 2007
 137. Joni Mitchell       One Week Last Summer           Shine       04:59 2007
 139. Joni Mitchell       This Place                     Shine       03:54 2007
 231. Sara Bareilles      Love Song                      Little Voi  04:20 2007
5202. Sia                 Little Black Sandals           Some Peopl  04:14 2007
```

Note that the index numbers may not be consecutive if the search results are displayed.

A typical terminal is 24 rows tall. If 20 songs are displayed followed by 3 blank lines, the prompt will appear on bottom line. If the user has resized the terminal so that it is shorter than 24 rows, the program will not display nicely. Your program does not have to contend with this problem.

After the user enters an action in response to the prompt, if the program again displays 20 songs followed by 3 blank lines, the prompt will again be on the bottom line. This is how you can control to a limited extent how "nice" your program's display looks.

[1] In C++ using the iostream library, it is not easy to alter this behavior. It is much easier when using the C library's file I/O functions to prevent a file from being overwritten.

Input

As noted above, the input file is either a command line argument or the default `songs.csv` file in the current working directory. Whichever it is, when the program tries to open it, if the open fails, it must exit and report this error on the standard error stream.

If the file is opened successfully, it must be read into an appropriate data structure in the program's memory.

The input file is a tab-separated text file. This means that each field on a line is separated from the adjacent fields by a single tab character. In addition, each field is delimited by double quotes.

Each line, except for the first one, should contain a description of a single song. The first row in the file will contain a set of headings. The headings are always the same in a valid input file:

```
"Name"  "Artist"    "Album" "Genre" "Size"  "Time"  "Year"  "Comments"
```

If the heading in the input file does not match the above ones, the program should detect such an input file as invalid and report on the standard error stream that the input file heading is not valid.

The remaining lines contain descriptions of songs. Each line contains eight fields, corresponding to the above named headings. Although your program must display the time using the format `mm:ss`, the time stored in a file is given in seconds. An example might look like:

```
"Melt My Heart To Stone" "Adele" "19" "R&B" "6772679" "203" "2008" "my favorite"
"First Love" "Adele" "19" "R&B" "6651186" "190" "2008" "it's ok"
"Wish I Could" "Norah Jones" "Not Too Late" "Jazz" "8536495" "258" "2007" "from Otis"
"Not Too Late" "Norah Jones" "Not Too Late" "Jazz" "6913447" "211" "2007" "from Otis"
"One Week Last Summer" "Joni Mitchell" "Shine" "Pop" "9936667" "299" "2007" ""
"This Place" "Joni Mitchell" "Shine" "Pop" "7695743" "234" "2007" ""
"Love Song" "Sara Bareilles" "Little Voice" "Pop" "4210858" "260" "2007" "send to Joe"
"Little Black Sandals" "Sia" "Some People Have Real Problems" "Pop" "" "254" "2007" ""
```

Only the first two fields: *Name* and *Artist* are mandatory on the line. Any field past the first two can be left blank. A blank field is indicated by an empty set of quotes (not even white spaces should be allowed). For example, if the Album, Genre, Size and Comments are all blank, an entry could look as follows (with tabs separating the entries):

```
"One Week Last Summer"  "Joni Mitchell" ""  ""  ""  "299"   "2007"  ""
```

Validation of the input file. Summarizing the validity of an input file, a valid file must satisfy all of the following conditions:

- the headings must be valid and all present;
- the first two fields on each line must be non empty;
- each line has to contain exactly eight fields;
- the fields are separated by tabs (and therefore there are seven tabs in each line).

No other checks on the validity must be performed. You can assume that the following will be true in every input file that is valid according to the above conditions:

- the time is always specified as the number of seconds;
- the size is the number of bytes;
- there are no empty lines, but each line ends with a newline character (specifically, the last line in the file will have a newline at the end).

Output

The format of the output file should be identical to the format of the input file. It should contain the same heading and each line should be in exactly the same format as an input line's format. This way, an output file from one run of the program can be used as an input file for another run of the program.

Programming Rules

Your program must conform to the programming rules described in the Programming Rules document on the course website. Remember in particular that:

- You must write your program yourself. I will on occasion ask you to explain the code to me, so you need to make sure that you understand the code that you write. If you want to use features of C++ that we did not cover in class, you need to understand them thoroughly.
- You must document your program (preamble, function pre- and post-conditions, comments throughout the code, appropriately choosing variable and function names).
- All input and output must be performed by the `main` function only, i.e. no other functions should perform any I/O operations. The functions that perform the computational tasks must pass all data through their parameter lists.
- You must use a class to represent a single song.
- You must use a class to represent a playlist.
- Late programs lose 20% for each 24 hours they are late.

Grading

The program will be graded based on the following rubric.

- Does the program compile (on a cslab machine): 10%
- Correctness and implementation of the song class: 10%
- Correctness and implementation of the playlist class: 20%
- Correctness and implementation of the user interface: 15%
- Correctness and implementation of the file I/O operations: 10%
- Correctness and implementation of `main()` and any other features in the program: 15%
- Correctness of multiple file implementation and modularity: 10%
- Documentation: 10%

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on December 10, 2012. You need to submit multiple files for this assignment, so they must be placed into a directory and zipped up as the instructions below describe. The name of your directory must be *`username`*`_hwk3.` where *`username`* is to be replaced by your username on our system. Do not name it anything else. If it is named anything else, you lose 3% of the program's value. Before you submit the assignment, make sure that it compiles and runs correctly on one of the cslab machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to create the above named directory and place all of your source code files into it. Do not place any executable files or object files into this directory. You will lose 1% for each file that does not belong there. With all files in your directory, run the command

```
zip -r username_hwk3.zip ./username_hwk3
```

This will compress all of your files into the file named `username_hwk3.zip`. Put this zip file into the directory

```
/data/biocs/b/student.accounts/cs135_sw/cs135projects/project3
```

Give it permission 600 so that only you have access to it. To do this, `cd` to the above directory and run the command

```
chmod 600 username_hwk3.zip
```

If you put a file there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date.