From C++ to C:
What You Need to Know for Systems Programming

Ann Ford Tyson

Florida State University
January 2007

Dedication

This book is dedicated to my Pembroke Welsh Corgis, Dancer, Nicky, Beck and Snickers. Pembrokes are a very smart breed. In fact, my dogs have all learned that if they place a furry paw on my laptop keyboard, an interrupt signal is immediately generated. They must have read chapter 8.

Preface

In a nutshell, this book's overriding goal is to tell you what you need to know about C - by no means about *ALL* of C, but what's most useful and important to get a jump start in the language, and to do this in the form of a quick read. Your time is valuable, so we'll get right to the point on all topics discussed.

Why is this book needed? More and more often, universities teach C++ or JAVA as the core programming language students learn in the first one or two years of college. Then, as computer science majors and other students are required to take upper-level courses in operating systems, compiler writing, graphics, security, image processing and other advanced tasks, they are often required to learn C to write their own code and to study and even modify legacy code in these areas. Instructors often tell students to learn C "on your own!" Similarly, professionals in the field experienced in C++ or JAVA may suddenly need to learn C. Or maybe you've heard that killer-app recent games such as Madden NFL were written primarily in C, and you're wondering, hey, what makes C so useful? C has had an extremely large and faithful following of programmers for many years.

Why C? C has gained an enormous following over the years for many reasons: it's highly portable, C compilers and the code they generate tend to be compact and extremely efficient, C expressions and operations are extensive and powerful, and C libraries are broad and provide enormous functionality. C provides fundamental and direct support for pointers, memory allocation and deallocation, and machine level operations, allowing programmers to maximize memory efficiency. The standard C input/output library provides perhaps the most direct control and broad flexibility of any high-level programming language.

C is both rich and complex, and learning the language can be a daunting task. And often, the requirement is also not just to learn C, it is to learn many of the low level, advanced details of C programming typically needed at the systems level. This book exists to bridge the learning gap for both students and professionals.

We make these assumptions about your background prior to reading this book: that you have had two courses in programming, typically in C++, but possibly in JAVA (JAVA was based on C++), or the equivalent experience at work or elsewhere.

The platform used to compile, test and execute example C and C++ programs in this text is UNIX/LINUX using the gcc or g++ compilers.

The most recent version of the C language as of this writing is the 1999 ANSI C standard, which we will call C99. The 1989 standard, C89, and even earlier versions of C are represented in a great deal of existing code. The most widely used version of C at this time is still the 1989 version, which the vast majority of this text adheres to and discusses. C99 is brought up in particular where it provides new features of interest.

It is not the intent of this book to cover *all* of the C language. The goal is to help you learn what you need to know to get going quickly using the C language in advanced programming tasks.

A final note: you might think of the current situation for programmers with a car analogy: driving a car with an automatic transmission is great. It's easy because the car does everything for you. No clutch, no shifter to worry about. But, to drive a sports car or a racing car, who wants an automatic transmission? As programmers, we often want closer access to the machine level, and we want to control exactly what happens, and when it happens. C was created originally by Dennis Ritchie of Bell Labs for systems-level programming. The tools of C, happily, are available to us when programming with C as our major language, and from within C++ programs, and from within programs based on other languages as well.

Let's start driving.

Ann Ford Tyson

January 2007

Please send any comments to: aftyson@cs.fsu.edu

Table of Contents

Chapter 1: Fundamentals

In this chapter we will explore the basics of C as compared to C++. Since the C language was used as the kernel of C++, there is a great deal of overlap. However, as you quickly will see, there are also enormous and important differences.

1.1 First Program Example

No programming text would be complete without a "hello world" program. So let's look at one, first in C++, and then in C. This example will help us get started discussing the important differences between these two languages.

First, the C++ version:

```
#include <iostream>
using namespace std;
int main ( )
{
   for (int i = 1; i <= 5; i++)               // for loop
      cout << i << "hello world!" << endl;
   return 0;
}
```

Program 1.1

Now, the same task written in C:

```
#include <stdio.h>
int main ( )
{
   int i;
   for (i = 1; i <= 5; i++)                   /* for loop */
      printf("%d%s\n",  i, "hello world!");
```

```
   return 0;
}
```

Program 1.2

Already we can see several differences:

- header file names
- C does not use a namespace
- scope rules for variable declarations vary
- output is handled quite differently
- comments may be written differently

Let's take a look at each of these items now.

1.1.1 Header files

The following table summarizes what you need to know regarding major headers.

C++ header file name	**Usage**	**Which header file to use in C**
iostream	major header file for IO stream classes, e.g. cin and cout	stdio.h
iomanip	for formatting IO, e.g. function setw to control field widths	stdio.h
fstream	needed for file IO	stdio.h
cmath, cstdlib, etc.	standard header files inherited from C	math.h, stdlib.h, etc.

Table 1.1

The great news is that you can access any standard C header file from within a C++ program.

1.1.2 Namespaces and other C++ concepts

The C language does not support namespaces. Importantly, it also does not support classes or the concepts of inheritance and polymorphism. More on all of this later.

1.1.3 Basic Scope Rules

In C++ you may declare a variable inside any block. One example is contained in the for loop loop shown in the first program. Another example:

```
#include <iostream>
using namespace std;
int main ( )
{
```

```
   int num = 3,
       status = 1;
   if (status)
   {
      double num = 3.1423;
      cout << num;            // prints the INNER block num
   }
   return 0;
}
```

Program 1.3

This is OK in C++, but illegal in C. Variables simply *cannot* be declared in interior blocks in the C language.

1.1.4 Output (and input) is handled differently

For now we will preview major concepts in IO, and treat them in greater detail in the chapter devoted to that topic. The table below summarizes fundamental points:

C++ concept	**Corresponding C concept**
cin (input stream)	stdin
cout (output stream)	stdout
cerr (output stream for error messages)	stderr
cin >> (perform input)	scanf function
cout << (perform output)	printf function

Table 1.2

1.1.5 Internal Program Comments

You have probably heard that C++ supports both of the following styles of comments, but that C does not:

```
/*
   here is a comment
   this code is the most elegant and efficient anywhere!
*/

// I would also add that I write code faster than anybody else!
```

Be aware that C compilers which do not comply to the most recent ANSI standard of 1999 only support the first type of comment. The ANSI standard of 1999 does accept both types in C, and makes C and C++ the same in this regard. Most C legacy code will only contain the first type of comment shown.

Next we will go on to some other topics, exploring ideas beyond the first program example.

1.2 Macros

Macros have often been widely used in C programming. Note that most usage of macros is frowned upon as a style violation in modern C++ programming.

The macro, or #define directive, is a pre-processor directive which instructs the pre-processor to make a direct text substitution of one snippet of C code for another snippet. As usual, considering examples is the best way to understand this idea.

1.2.1 Defining Symbolic (Named) Constants

In C++ we typically define a constant as follows:

```
const int MINUTES_PER_HOUR = 60;
```

This causes the compiler to allocate memory space for an integer with value 60, which is then held constant throughout the program.

In C we do this instead:

```
#define MINUTES_PER_HOUR 60
```

This has a very different effect. When the program is pre-processed, the pre-processor takes every instance of the string "MINUTES_PER_HOUR" in the source code, with the exception of those instances found inside comments or output strings, and it substitutes the string "60" into the source code at that point.

#define directives are typically listed in a global position in front of the main function. This allows usage of the identifiers throughout the remainder of the file the directive occurs in.

1.2.2 Macros with and Without Parameters

A macro can also serve a purpose like the inline function call one sometimes uses in C++. For example:

```
#define SEMICOLON_ERROR \
printf("missing semicolon on this line or above")
```

The '\' allows us to continue the macro onto an additional line. Later in our code we could write

```
if (<error condition occurs>)
      SEMICOLON_ERROR;
```

The pre-processor, as before, will substitute the entire printf string when it sees SEMICOLON_ERROR in the code.

Macros can also take parameters. Note that no type checking of parameters is done so major errors can occur if care is not taken to use macros correctly. Function calls are normally preferred over macros in most situations except for those containing very simplistic bits of code and no parameters.

Here is one example of a macro with a parameter:

```
#define DOCALC(X, Y, Z) (X)Z(Y)
```

now if we have this later in the source code

```
int i = 2, j = 3, k;
k = CALC(i+j, j, *);
```

the pre-processor expands the second line to k = (i+j)*(j).

1.3 Simple Data Types

Fortunately for those of us trying to master both of these useful languages, simple data types are largely identical. We have, in both

integral types	char, enum, short, int, long, unsigned
real types	float, double, long double
addressing types	pointer

Table 1.3

C++ also provides an addressing type called a *reference*, which C does not. We will discuss the difference between pointers and references later in this text.

We need to look at some special considerations about enum and bool.

We will also discuss the use of unsigned integers for bitwise operations at a later point.

1.3.1 bool

Prior to the 1999 standard, C did not contain a Boolean type. Current C++ does contain a standard Boolean type, named bool. C provides a Boolean type as of C99, however most existing C code does not utilize any standardized Boolean type. The traditional way to implement logical values in C is by using macros like this:

```
#define TRUE 1
#define FALSE 0

int flag;

flag = TRUE;
```

etc.

Alternatively, C programmers often defined their own Boolean type within programs using enumerations.

To be more specific about the C99 implementation of a Boolean type, the type defined in the core language is named _Bool and may hold the values 0 and 1. The header file stdbool.h defines a macro named bool as a synonym for the type _Bool.

1.3.2 enum

There is one thing you need to know about enum types in C that makes their use different in C than in C++. C supports implicit (automatic) conversion from enum to int type and from int to enum type, but C++ does not support the second as an implicit conversion. Declarations are also somewhat different.

In C++, we might have

```
enum CardSuit {CLUBS, DIAMONDS, HEARTS, SPADES} CardSuit;

CardSuit card;

// the cast is required; see the section on type casts next if needed
for (card = CLUBS; card <= SPADES, static_cast<CardSuit>(card++))
```

in C this would be written as

```
typedef enum CardSuit {CLUBS, DIAMONDS, HEARTS, SPADES} CardSuit;
CardSuit card;

// no cast required!
for (card = CLUBS; card <= SPADES, card++)
```

The type cast itself is treated differently in C and C++. This is what we will look at next.

1.4 Type casting

The C++ language supports four forms of type casting with four different syntaxes. C performs various forms of casts based on contextual information in the code using only one syntax. Here we consider a type cast in C which corresponds essentially to a static_cast in C++.

Example in C++:

```
int i;
double x = 2.1234;

i = static_cast<int>(x);    // the cast form is specified by syntax
```

what to do in C:

```
i = (int) x;                // here is the cast in C
```

Note that C++ compilers generally also support this C syntax to be backward compatible.

1.5 The big picture: procedural and object-oriented programming

We will finish this chapter by talking about the big picture. So far, you have seen that C and C++ have some basic syntax differences. This is important as you write new code in C and study existing C code for operating systems, compilers and applications. However, it is critically important to take special note of the fundamental philosophical difference between these two languages.

When Bjarne Stroustrup created the C++ language, a major goal was to retain the C language kernel and add object-oriented capabilities beyond this kernel to form C++. At the heart of this language development lies a paradigm shift, from *procedural* to *object oriented* programming.

C is a procedural language and does not provide OO tools such as the class, inheritance, or polymorphism. C programs typically begin with what is called a *top-down design*, or a *structured programming* approach. All of this boils down to, TASKS come first. We think about what has to be done, what order to do it in, and what tasks are subtasks of what other tasks. These tasks are then set up as functions.

Using C++ for object-oriented programming means an OBJECTS (data) first approach. The objects will contain member functions, or tasks. Data and functions that act upon it are *encapsulated* together. Many programmers will say that procedural programming is a subset of OO programming, and there is merit to that point of view.

Let's take a simple example. Assume we are writing a program such as an appointment manager, which by necessity has a time component. If we think of our time module from a procedural perspective, we would most likely start off with a structure chart and decompose our time module into functions and sub-functions. In the back of our minds we realize that we will be setting up a data structure to store a time, and passing variables of that data structure type to our various functions. The data structure might eventually be some form or combination of arrays and/or structs. We most likely focus on the tasks first. What needs to be done? Figure 1.1 displays our plan in structure chart form.

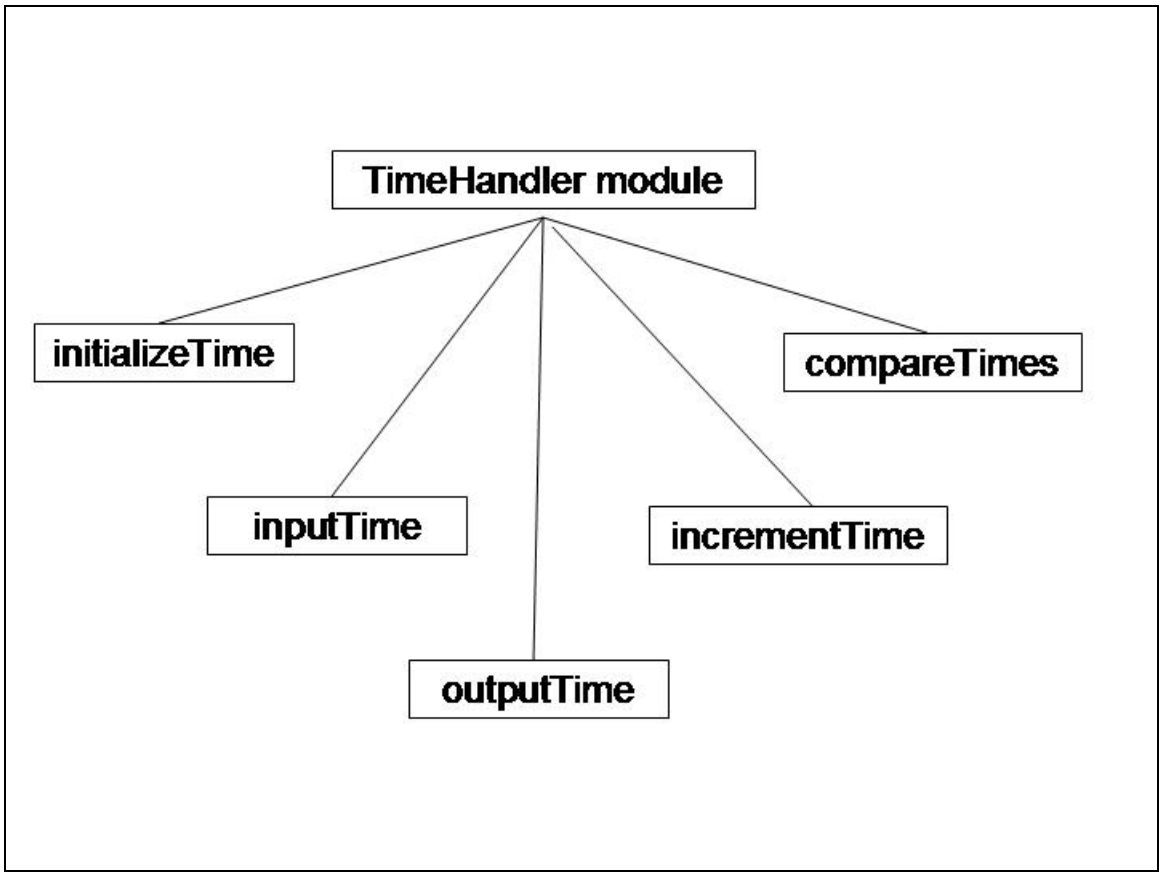

Figure 1.1

From an OO perspective, we would start by thinking of an ADT (abstract data type) to represent a time and its member functions. Eventually this would be implemented as a class. We would plan to create objects and have the objects call their member functions when something needed to happen. Figure 1.2 illustrates our class object and its member functions.

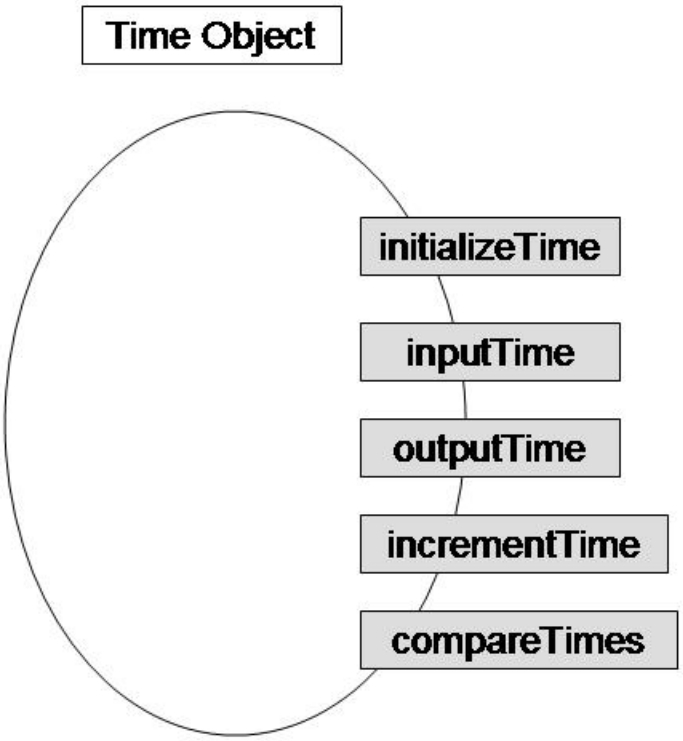

Figure 1.2

We can't thoroughly discuss procedural and object-oriented design in just a few pages in a small book. We leave that to major courses and much larger texts. For now, awareness of this fundamental difference is what we're after.

Let's explore more about C...

Chapter 2: Pointers and Addressing

There are some programming languages which a programmer can use and "get by" without understanding pointers. C is *not* one of those languages!

To use C effectively, especially to do any kind of systems or other advanced programming, we must understand pointers and addressing. This chapter will discuss pointers and memory addressing in the C language, and the material presented here will be needed for all future chapters in this book.

2.1 Pointer Fundamentals and Related Operators

A computer's memory consists of a series of storage locations or cells, typically one byte in size, where each byte has a unique address. A pointer is a program entity which can store a memory address. The pointer is said to "point at" that location. Memory addresses begin at zero and increase as positive integral values, usually expressed in hexidecimal format (base-16). Note that low addresses are normally reserved for operating system use.

A zero-value address is treated as a special case in C. The zero address, represented by the standard identifier NULL, is not viewed as a physical memory address when used in a C program. NULL means *no address*. Another way to word this is to say that a NULL pointer *points at nothing*.

First let's consider a few simple examples, assuming that an integer occupies 4 bytes of memory and that addresses are represented using 32 bits:

```
/* declare an integer variable and give it the value 5 */
int i = 5;

/* declare a pointer to an integer */
/* the data type int * is read as "pointer to int" */
int *p;

/* set the pointer p to contain the address of integer i */
p = &i;
```

What we have in memory in the runtime stack is now something like this (exact values are always platform dependent; the example address values here were obtained from a run of this code on a typical UNIX machine):

associated identifier	address	contents
i	0xbff6f894	5
p	0xbff6f890	0xbff6f894

Table 2.1

2.1.2 Address Operator &

The address operator is written using a single &, and is read as "the address of." Thus in the example code above, we are setting the value of p to "the address of i."

We could also have declared p as follows with the same result:

```
int *p = &i;
```

2.1.3 De-referencing Operator *

The dereferencing operator is written as a single *. We can access the value stored at location 0xbff6f894 in one of two ways: directly, by using the name i, or indirectly, using the notation *p. If we write

```
*p = 7;
```

this will change the value of i to 7. There are various ways to pronounce this syntax, with the most common being "star p." Roughly, we could say "follow the pointer p to the memory location it refers to, and access the contents of that location."

When we access this cell using *p, we say we are using one level of indirection.

2.2 Multiple Levels of Indirection

The C language allows programmers to use more than one level of indirection. For example, we could declare

```
int **q;
```

where q is a "pointer to a pointer to an int."

Now, we can have q point at p, where p already points at i.

```
q = &p;
```

This gives us three ways to access the value of i. All of the following statements set the value of i to 11:

```
i = 11;          // directly
*p = 11;         // one level of indirection
```

```
**p = 11;        // two levels of indirection
```

Levels of indirection can continue beyond two.

Are we having fun yet? You bet. Understanding pointers and addressing in C is fundamental to using the language. Many a programmer learning C has said to him or herself, "What on earth are all these asterisks doing in this code?" The better grip you can get on what we have just been talking about, the more comfortable you will be with the C language and the code you will be seeing and writing.

This can seem confusing at first - but keep in mind, it's all very logical and totally based on the memory map, so keep at it, and you'll get it. Consider how memory is set up and you will use C like a pro.

One of the reasons C is greatly appreciated by systems and other advanced programmers is that C allows you to access memory locations in this particular way.

Note: it is possible to attempt to access a location you are not allowed to go to. In this case the operating system will typically halt your program run with a message such as "segmentation fault" or "addressing exception."

2.3 Comparing Pointers and References

The C language supports only pointers. C++ supports both pointers and references. JAVA supports only references. Since you are probably already familiar with using references in C++, it is probably best to compare the two concepts to avoid being overly confused.

The main idea is this: a reference is essentially a *constant* and *automatically dereferenced* pointer.

Here is an example program in C++ using both pointers and references:

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

int main ( )
{
      int number = 5;         // number is an int
      int* myPtr = &number;   // myPtr is a pointer to an int
      int& myRef = number;    // myRef is a reference to an int

      *myPtr = 12;            // sets number to 12
      myRef = 17;             // sets number to 17

      return (0);
}
```

Program 2.1

The notation can be confusing, no doubt about it. Keep in mind that operators in C and C++ are *overloaded*, that is, they can have different meaning in different contexts. Overloading contributes to the power and flexibility of both languages.

In C, the * may be used in declaring a pointer, and in dereferencing a pointer (as well as for multiplication!).

The & may be used in C for the address operator. However in C++ it can be used as the address operator and also to indicate a reference. It is also a bitwise operator, which we will consider later on in this textbook.

Chapter 3: Input and Output

The core C language does not provide for IO. IO data types and functions are provided in the standard C library *stdio.h*, and sometimes additionally in non-standard libraries associated with particular platforms. In this text we will discuss standard IO only.

Many C++ programmers actually prefer to use C IO constructs to perform IO, because of the incredible flexibility and control C provides. In this chapter we will study some of the major characteristics of IO in C.

3.1 Files

When we talk about IO in C, we use the term *file* to refer to sources of input and targets for output. The general concepts involved are essentially the same as those you have learned about C++ IO *streams*. However, in C++, IO streams are set up as classes and we call class member functions for various operations. That is not the case in the C language, primarily because C does not support the class data structure. In C, IO streams are represented by files.

The data structure **FILE** defined in stdio.h is represented by a structure ("struct" data type in C), dynamically allocated at runtime. At the time this struct is created, a pointer to it is also created and we use the pointer to access the file.

When a C program begins executing, three standard files and constant pointers to these files are automatically opened:

file source/target	**file pointer name**
standard input: keyboard	stdin
standard output: console window	stdout
standard error output: console window	stderr

Table 3.1

These standard files may be re-directed if needed, and of course, we can open other files for input and output, for example those stored on disk. We will discuss these options shortly.

In this chapter our focus will be the standard IO capabilities that C provides for both interactive and file IO.

IO of strings and string related operations will be discussed later in this textbook in the chapter on C strings.

3.2 the printf function

The *printf* function is used for general formatted output and by default, prints to stdout. Here is its prototype:

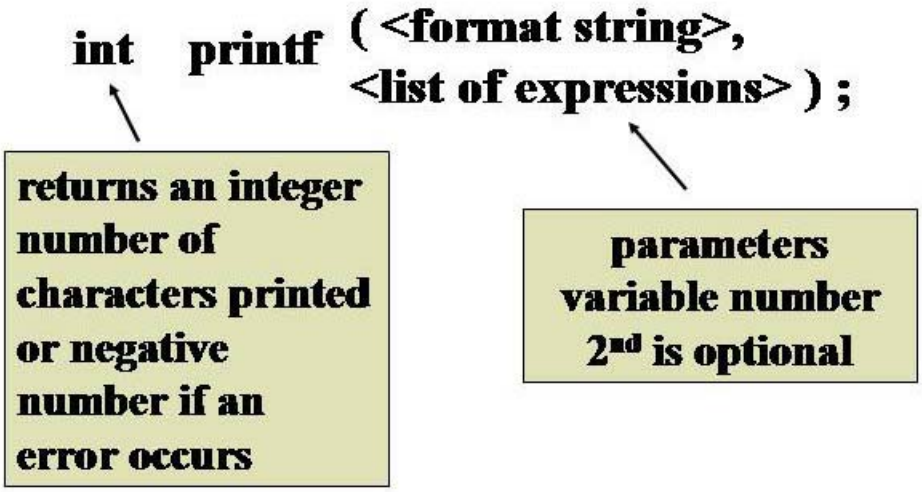

Figure 3.1

To start our discussion of printf, let's look at some example code. The following example C++ program illustrates basic formatted output of numbers and characters in C++:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main ( )
{
   int i = 5;
   double x = 3.21754;
   char ch = 'Q';

   cout << fixed << showpoint << setprecision(2);
   cout << "i is:" <<  setw(2) << i << endl;
   cout << "x is:" << setw(5) << x << endl;
```

```
   cout << "ch is: " << ch << endl;

   return 0;
}
```

Program 3.1

The output of this program when run looks like this:

```
aftyson@diablo:~/testcpp>a.out
i is: 5
x is: 3.22
ch is: Q
```

The following C program will produce exactly the same output:

```
#include <stdio.h>
int main ( )
{
   int i = 5;
   double x = 3.21754;
   char ch = 'Q';

   printf("i is:%2d\n", i);
   printf("x is:%5.2f\n", x);
   printf("ch is:%2c\n", ch);

   return 0;
}
```

Program 3.2

Lots of differences! Let's examine what's going on here.

First, when you write something like

```
   cout << "i is:" <<  setw(2) << i << endl;
```

cout is an instance of an IO stream class in C++. Each occurrence of "<<" will initiate a call to a function related to this class, causing output to occur. The class output function corresponding to the data type of the item being printed is correctly determined by the compiler and then the item of that type, in this case i, is printed using the rules for that data type. Items such as "setw" etc. are further functions calls to class related functions that act on the stream cout. setw(2) tells the program to print the value of i in a field width of 2. Unoccupied print spaces are padded with blanks by default.

Using C,

```
   printf("i is:%2d\n", i);
```

we call the printf function, causing the value of i to first be copied into a CPU register. We must tell the printf function to interpret this value as an integer using the %d type specifier. Using %2d tells printf to not only interpret the value as an integer, but to print the value in a field width of 2, right-justified [note that we can switch to left-justify using negative field width values, as in %-2d].

If the programmer specifies the wrong data type specifier for the printf function, printf will print the value using the type given by the programmer, whether right or wrong, sometimes giving bizarre results in the output!

If a field width value is given which is too small, printf will automatically use the minimum field width required to display the value.

The character escape code '\n' is used in C to print the newline.

For output of the real value of x, we have specified %4.2f. The %f indicates that the data type is float or double, and the 4.2 indicates 4 spaces for the field width and a precision of 2.

The following table provides frequently used output conversion specifiers for C:

Conversion Specifier	**Data Type**
%d (or %i)	signed decimal integer
%u	unsigned decimal integer
%o	unsigned octal integer
%x, %X	unsigned hexidecimal integer
%c	character
%s	string
%f, %F	signed decimal floating point
%e, %E	signed decimal floating point
%g,%G	signed decimal floating point
%a, %A	signed hexidecimal floating point
%%	prints a single percent sign

Table 3.2

The following table provides commonly used special characters which can be printed using escape sequences:

escape code	**prints**
\a	bell (alert)
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\'	single quote
\"	double quote
\?	question mark
%%	percent sign

Table 3.3

It is recommended that the reader see a full-size reference text for a complete discussion of output features and options.

3.2 The scanf Function

The scanf function is used for general formatted output and by default, reads from stdin. Here is its prototype:

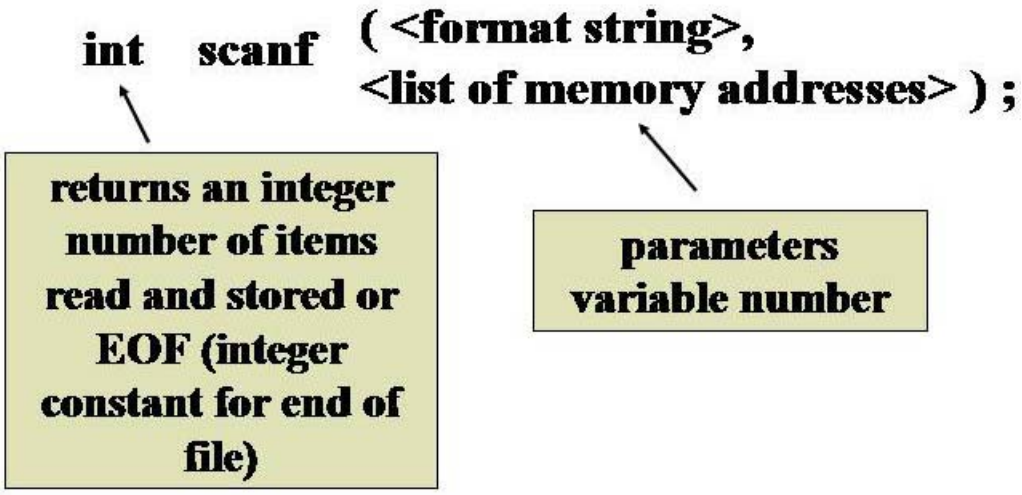

Figure 3.2

Let's start off by considering a program example which reads an integer, a double, and a character. First, the C++ version:

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

int main()
{
   int i;
   double x;
   char ch;

   cout << "Enter an integer now: ";
   cin >> i;
   cout << "Enter a double now: ";
   cin >> x;
   cout << "Enter a character now: ";
   cin >> ch;

   cout << endl << i << x << ch;
   return 0;
}
```

Program 3.3

Here is a sample run, using data values 5, 4.321 and y. The user types enter after each value is input. Here is out sample output:

```
aftyson@diablo:~/testcpp>a.out
Enter an integer now: 5
Enter a double now: 4.321
Enter a character now: y
```

```
54.321y
```

Ok, now for the equivalent functionality in C (first try, we'll have to fix this!):

```
#include <stdio.h>

int main()
{
   int i;
   double x;
   char ch;

   printf("Enter an integer now: ");

   /* you must give a type specifier for the value read, and pass */
   /* the address of the variable to read into, or serious errors */
   /* will occur! */
   scanf("%d", &i);
   printf("Enter a double now: ");
   scanf("%lf", &x);
   printf("Enter a character now: ");
   scanf("%c", &ch);

   printf("\n%d%.3f%c\n", i, x, ch);
   return 0;
}
```

Program 3.4

When this program is run, we have the following interaction:

```
aftyson@diablo:~/testc>a.out
Enter an integer now: 5
Enter a double now: 4.321
Enter a character now:
54.321
```

Oops! Something is wrong! The program never asked us for a value for ch, and is actually printing a newline for ch.

This example points out a very significant different between C++ input and C input: when you read with cin >>, the input operation will skip over any leading whitespace (blanks, newlines and tabs) to locate non-whitespace data to read. In C, this is true for numeric reads, but it is not true for character reads as we have here trying to read ch using %c.

We can fix this easily, as C provides a truly simple way to skip whitespace during input: just place a blank in front of the %c, as follows:

```
scanf(" %c", &ch);
```

If we run the program with this one change, our program will work correctly and the output will be identical to that shown above after the C++ program.

Why does the erroneous C program read and print a newline for ch? Because there is a newline in the input stream as follows (the '\n' character occurs wherever the user types enter):

```
5\n4.321\n
```

and only after that 2nd newline is either read or deliberately skipped are we able to type in a value for ch.

A blank may be inserted into the format string for scanf at other positions and in other situations as well, causing whitespace to be skipped.

Two important things to remember about using scanf:

- you must provide a correct type specifier to match the input value entered, or serious errors may often occur
- you must pass scanf the memory address of what it is to read into; if you do not pass a valid address, you are likely to see runtime error occur (e.g. segmentation fault or addressing exception)

The following table provides frequently used input specifiers for C:

Conversion Specifier	**Argument Type**
%d	int *
%u	unsigned *
%c	char * (single characters)
%s	char * (strings)
%f	float *
%lf	double *
%Lf	long double *

Table 3.4

It is recommended that the reader see a full-size reference text for a complete discussion of input options. The C language provides enormous flexibility and control with IO, and we are only covering the basics here.

3.3.1 Using the return value from scanf

The scanf function returns a value which is extremely useful in testing and debugging. Specifically, it returns an integer value giving the number of items it successfully read and stored in memory. This will be zero if nothing was read and stored successfully. The value will be the standard constant EOF if end of file was encountered.

As an example, let's consider the following code:

```
#include <stdio.h>
int main()
{
   int i;

   printf("Please enter an integer: ");
   scanf("%d", &i);

   printf("The value entered is: %d\n", i);

   return 0;
}
```

Program 3.5

What if the user enters something that is not a valid integer when asked for input? Here is an example:

```
aftyson@diablo:~/testc>a.out
Please enter an integer: x
The value entered is: 6970528
```

Ooops, this isn't good. However, we can easily detect the error and recover from it using the return value from scanf as follows:

```
#include <stdio.h>
int main()
{
   int i;

   printf("Please enter an integer: ");

   /* scanf returns a zero if it cannot read an int successfully */
   /* scanf returns a one if it does read an int successfully */
   while(!scanf("%d", &i))
   {
      /* first scan and discard the remainder of the input line */
      /* getchar reads one character from the keyboard - we'll cover */
      /* it in more detail shortly */
      while(getchar() != '\n');
      printf("\nPlease try again: ");
   }

   printf("The value entered is: %d\n", i);

   return 0;
}
```

Program 3.6

3.3.2 Other useful techniques with scanf

We can prevent input from being stored into memory by using the *assignment suppression operator*, the *****. For example:

```
scanf("%*d%c", &ch);
```

will read an integer from the input, but not store it anywhere, and then read a character from the input and store it in ch.

We can specify a *field width* for input values, similar to the way in which we did so for printf. For example:

```
scanf("%2d", &i);
```

will read a 2 digit integer from the input stream and store the value in i.

The concept of a *scanset*, using [and] to delineate a set of characters, allows us to read or exclude characters from a specified set. We will discuss scansets when we cover strings later in this book.

3.4 Redirecting Standard IO

What if we have an existing program using standard IO, that is, using scanf to read from the keyboard, and using printf to write to the video monitor's console window, but we have a need to run the program using external files on disk for IO? It is very simple to *redirect*. All systems provide a mechanism for this; we will consider UNIX.

Assume the executable file for our program is named *runprog*.

If we use

```
% runprog
```

at the UNIX % prompt, scanf will read from the keyboard and printf will write to the console window.

If we type instead

```
% runprog <data.txt
```

then scanf will read from the file data.txt, and printf will write to the console.

Using

```
% runprog >out.txt
```

causes printf to write to the file out.txt.

If we want to redirect both input and output, we could use something this

```
% runprog <data.txt>out.txt
```

scanf would now read from data.txt, and printf would write to out.txt.

It's that easy.

3.5 File IO: Fundamentals with Text Files

Many programs need to read from or write to files on disk or other storage media, not just from the keyboard and to the console window. In programming we encounter two major types of stored files, text and binary. Text files consist of entirely ASCII characters and are easily typed in by a user with various text editors. Binary files must be created by programs writing to files in a specified way, and can only be read by a program which knows how the file was created. In this section we will discuss IO with text files.

These are the basic steps when performing file IO using non-standard files:

(1) declare a file pointer for each file to read from or write to using type FILE *
(2) open the files using function fopen
(3) read from the files using appropriate functions; here we will use fscanf and fprintf
(4) close any file when done with it using function fclose

Here are our first two function prototypes:

```
FILE *fopen (<filename>, <mode>);
int fclose (FILE * stream);
```

The function fopen attempts to open the file specified by the file name. We will specify a filename as a literal constant string for this discussion, e.g. something like "data.txt." The *mode* determines whether the file is open read-only, write-only, or with other capabilities. Function fopen returns a pointer to the struct containing information for the newly opened file, if the open was successful. If it cannot open the file, it returns the NULL pointer (zero).

Function fclose closes the stream passed to it, and returns zero if it is successful, and EOF if an error occurs.

The modes are as follows:

Mode	What it means
"r"	open existing file for input read-only
"w"	open existing file for output (erasing previous contents) or create new file if needed
"a"	append to existing file for output or create new one if needed
"r+"	open an existing file for update (both input and output); start at beginning of file
"w+"	open an existing file for update or create a new one if needed (erases any previous contents)
"a+"	open an existing file for update or create a new one one if needed; start at end of file

Table 3.5

We need two more functions to get us started, fscanf and fprintf. Here are their prototypes:

```
int fscanf (FILE *stream, <format string>, <list of memory addresses>);
int fprintf (FILE * stream, <format string>, <list of expressions to print>);
```

These functions behave just like scanf and printf, except that they require an additional argument, the first argument, which gives a pointer to the FILE stream being used.

As an example, here is a program which copies from one text file and writes to another. We assume that the file data.txt already exists.

```
#include <stdio.h>
int main()
{
   char ch;

   /* declare two file pointers */
   FILE *infile;
   FILE *outfile;

   /* open input file read-only */
   infile = fopen("data.txt", "r");

   /* open output file write-only; if the file exists already it will */
   /* be erased; if it does not exist it will be created */
   outfile = fopen("newfile.txt", "w");

   /* infile will be NULL, or zero, if the file is not open */
   if (!infile)
      printf("ERROR: input file could not be opened!\n");
   else
   {
      /* fscanf returns EOF when encounters end of file */
      while(fscanf(infile, "%c", &ch) != EOF)
```

```
         fprintf(outfile, "%c", ch);
      fclose(infile);
      fclose(outfile);
   }

   return 0;
}
```

Program 3.7

3.6 Character IO

3.6.1 fgetc, getc, getchar, ungetc

The functions fgetc, getc, getchar and ungetc deal with input of a single character. Note that whitespace characters are read as single characters (not skipped or ignored!). The prototypes are as follows:

```
int fgetc (FILE *stream);
int getc (FILE *stream);
int getchar(void);
int ungetc(int ch, FILE *stream);
```

Notice that the more general data type int is used to represent a character value being passed into a function or being returned as its result.

The function fgetc reads one character from the specified input stream argument, returns it as the function result, and advances the read pointer for the stream by one position. The function getc is identical to fgetc except that it is typically implemented as a macro.

The function getchar behaves like fgetc and getc, except that stdin is assumed for the input stream.

Function ungetc is quite a bit different from the preceding functions: instead of reading a character from the input stream, it pushes the most recently read character back onto the input stream. If fgetc is called and reads 'A', then a subsequent call to ungetc will put the 'A' back into the stream. A subsequent call to fgetc would then read the 'A' as if it had not been read before.

3.6.2 fputc, putc, putchar

The functions fputc, putc and putchar deal with single character output. As with fgetc etc., these functions utilize type int for the character values passed in and result value generated. Their prototypes are:

```
int fputc (int ch, FILE *stream);
int putc (int ch, FILE *stream);
int putchar(int ch);
```

The function fputc simply prints the character argument to the specified output stream. Function putc behaves just like fputc, except that it is usually implemented as a macro. The function putchar prints its argument to stdout.

All of these functions return the value of the character argument using type int. If an error occurs, the functions return the standard value EOF.

3.7 The main Function and Command Line Arguments

The main function in C allows for two possible argument lists:

```
(1) int main (void);

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

We are already accustomed to using (1). Option (2) allows us to use *command line arguments* to main to send information to the main function at the time that main is invoked. One very common use of this is to pass filenames to a program for IO, and we will look at an example program which does exactly that.

When main is invoked using command line arguments, argc specifies the number of arguments. The second parameter, argv, is an array of pointers to strings. Note that argv[0] is the name of the program itself.

Here's an example. Let's say that we wish to copy the contents of a source text file to a target text file. Consider the following program:

```
#include <stdio.h>
#define NUMARGS 3

int main (int argc, char *argv[])
{
   char ch;               /* char used for input and output */
   FILE *infile;          /* source file */
   FILE *outfile;         /* target file */

   /* first check to see if have required number for argc */
   /* if not, error has occurred, don't do anything else */
   if (argc != NUMARGS)
   {
     printf ("Copy program was not invoked correctly.\n");
     printf ("Invoke using \'copyFile sourceFile targetFile\'\n");
   }
   else
   /* we have correct number of arguments, try to open both files */
   {
      infile = fopen (argv[1], "r");  /* source file name string is */
                                      /* in argv[1] */
      outfile = fopen (argv[2], "w"); /* target file name string is */
                                      /* argv[2] */
```

```
      /* if both files are open, perform the file copy until EOF */
      if (infile && outfile)
      {
         while (fscanf(infile, "%c", &ch) != EOF)
            fprintf(outfile, "%c", ch);
         printf ("SUCCESS: file copied.\n");
         fclose (infile);
         fclose (outfile);
      }
      else  /* if reach here, problem with file(s) */
      {
         printf ("ERROR: one or both files did not open.\n");
      }
   }

   return (0);
}
```

Program 3.8

The program does not provide the file names internal to the code. We send the file names in as command line arguments by invoking our program with the following command at the unix prompt:

```
% copyFile data1.txt data2.txt
```

The string "copyFile" is stored in argv[0] and must be the name of the executable. The strings "data1.txt" and "data2.txt" are stored in argv[1] and argv[2], respectively. When we run the program, data1.txt is opened using file pointer infile and data2.txt is opened using file pointer outfile. If we run the program with the command line above, we will see this output, assuming both files do open:

```
aftyson@diablo:~/testc>copyFile data1.txt data2.txt
SUCCESS: file copied.
```

If we run the program with a command line giving the wrong number of arguments, we will see output something like this:

```
aftyson@diablo:~/testc>copyFile data1.txt
Copy program was not invoked correctly.
Invoke using 'copyFile sourceFile targetFile'
```

Command line arguments can also be used to pass other types of information to the main function.

Chapter 4: Functions and Parameter Passing

4.1 Call by value
4.2 FindMax program example
4.3 Example using multiple levels of indirection: passing a pointer to a pointer

Fortunately, once you have learned to utilize functions and parameters in C++, you can jump right in and use them the same way in C for the most part. There is one enormous difference, however, and that is that C supports passing parameters ONLY using call by value, and C++ supports both call by value and call by reference. You *cannot* pass parameters by reference in C. Instead, we pass pointers to arguments when we want to send a potentially modified parameter value back from a function to its caller.

4.1 Call by Value

The basic principle of call by value parameter passing is that the value of an argument is COPIED into the corresponding parameter. Since the callee receives a copy of the original argument, the callee cannot change the value of the argument in the caller. We will illustrate the necessary concepts using the example program in the next section.

This chapter assumes that you have already read and understood chapter 2 on pointers and memory addressing, so if you have not read chapter 2 as of yet, please do so now.

4.2 FindMax Program Example

The following C program illustrates basic parameter passing. The main function reads in three values for variables one, two and three. It then calls function FindMax to determine the maximum of the three values, and returns this maximum via the fourth argument.

Keep firmly in mind that ALL arguments are passed by value in C. Here, the values of arguments one, two and three are copied into the corresponding parameters first, second and third. The value of the address of max, passed as &max, is copied into parameter maximum.

Note that you can utilize pointers this way in C++ functions as well.

```
/* ============================================================= */

#include <stdio.h>
void FindMax (int, int, int, int*);

int main ()
{
      int   one, two, three,        /* stores three integers */
                                    /* entered by the user */
            max;                    /* the maximum value of the 3 */
                                    /* integers */
```

```
      /* prompt for three integers and read them */
      /* notice we pass pointers to scanf as well as to FindMax */
      printf ("\nEnter three values => ");
      scanf ("%d%d%d", &one, &two, &three);

      /* find the maximum of the three and print */
      /* pass three ints and one pointer to an int to FindMax */
      FindMax (one, two, three, &max);        /* LINE 1 */
      printf ("The maximum is: %d\n", max);
      printf ("\nExecution Terminated.\n");

      return (0);

}     /* main */

/* ================================================================ */

void FindMax (    int first,        /* the three values for which */
                  int second,       /* the max must be found */
                  int third,
                  int* maximum)     /* pass a pointer here so */
                                    /* can return value to caller */

/* This function takes as input three integer values, determines */
/* the maximum of the three values, and then returns that max to the */
/* caller. */

{     /* FindMax */

      if (first > second)
            if (first > third)        /* we deference the pointer */
                  *maximum = first;   /* to change the value of the */
            else                      /* argument max */
                  *maximum = third;
      else
            if (second > third)
                  *maximum = second;
            else
                  *maximum = third;

      return;                         /* LINE 2 */

}     /* FindMax */

/* ================================================================ */
```

Program 4.1

Here is a sample output from this program:

```
aftyson@diablo:~/testc>a.out

Enter three values => 5 4 3
The maximum is: 5
```

```
Execution Terminated.
```

Let's consider an example of what is on the runtime stack at various points during this example run, to aid in understanding the use of pointers in this calling mechanism. We will simplify the stack frames a bit and only look at the values of declared variables, which includes parameters. Note that actual address values and stack layout are highly platform dependent. These example values were generated from a run on a typical UNIX machine.

Here is an example stack snapshot at the point where the main function is executing, just before LINE 1 executes:

Identifier	**Address**	**Value**
one	0xbfea7804	5
two	0xbfea7800	4
three	0xbfea77fc	3
max	0xbfea77f8	unitialized

Table 4.1

Here is a stack snapshot just before LINE 2 executes in function FindMax:

Identifier	**Address**	**Value**
one	0xbfea7804	5
two	0xbfea7800	4
three	0xbfea77fc	3
max	0xbfea77f8	5
first	-	5
second	-	4
third	-	3
maximum	-	0xbfea77f8

Table 4.2

Note that in this example, the addresses of one, two, three, first, second and third are irrelevant to understanding the parameter passing mechanism.

In our next program example, we will pass a pointer to a pointer.

4.3 Example using multiple levels of indirection: passing a pointer to a pointer

We can and often must, use more than one level of indirection when passing parameters.

The following program declares an integer variable in main, and then passes it through two levels of sub-functions for modifications.

```
/* ================================================================ */

#include <stdio.h>
void f1 (int*);
void f2 (int**);

int main()
{
   int i = 3;

   /* pass a pointer to i to f1 so that f1 can change the value of i */
   f1(&i);
   printf("i is now: %d\n", i);

   return 0;
}

/* ================================================================ */

void f1 (int* j)   /* j is a pointer to an int */
{
   *j = 5;         /* this will change the value of i to 5 */
   printf("j is now: %d\n", *j);

   /* f1 can pass its ability to change i down to f2 */
   f2(&j);         /* here we pass the address of j */

   return;
}

/* ================================================================ */

void f2 (int **k)   /* k is a pointer to a pointer to an int */
{
   **k = 7;         /* this will change the value of i to 7 */

   printf("k is now: %d\n", **k);

   /* we could also change the value of j here, but that would lead */
   /* to trouble */

   return;
}

/* ================================================================ */
```

Program 4.2

Here is a sample output from a run of this program:

```
aftyson@diablo:~/testc>a.out
j is now: 5
k is now: 7
i is now: 7
```

Based on what we have seen in this chapter, you probably realize that the ability to pass pointers gives the programmer a huge amount of power to access various memory locations as needed.

Chapter 5: More About Operators and Bit Manipulation

One of the most popular strengths of the C language is the ability it provides programmers to manipulate bits in memory directly. In this chapter we will explore what C offers in this regard.

5.1 The sizeof Operator

In the C language *sizeof* is a reserved word (a.k.a. keyword) and is an operator. The operation returns the integral number of bytes of storage required for a particular data type or variable.

The basic syntax is

```
sizeof (<data type name or variable name>)
```

Keep in mind that this is not a function call, even though it looks like one.

For example, consider the following program example:

```
#include <stdio.h>
int main ( )
{
   char ch = 'A';
   double x = -45.672;

   /* print number of bytes required to store a char on current platform */
   printf("The size of a char on this platform is: %d\n", sizeof(char));

   /* print number of bytes for x, also that required for a double */
   printf("The size of x on this platform is: %d\n", sizeof(x));

   return 0;
}
```

Program 5.1

When this program is run on a typical UNIX machine, the output (which is platform dependent) is

```
aftyson@diablo:~/testc>a.out
The size of a char on this platform is: 1
The size of x on this platform is: 8
```

The sizeof operator is used frequently when allocating memory dynamically. We will discuss its use with the *malloc* and *calloc* functions later in this text.

5.2 Bitwise Operations

The bitwise operators allow a programmer to manipulate the individual bits in a storage cell, thus interacting very directly with a particular platform's hardware.

Bitwise operations also allow for extremely efficient storage. For example, a bit pattern of 8 bits might be used to store 8 individual flags representing the states of an IO file or stream. For example, let's say that our 8 bit pattern looks like this

```
01001010
```

The leftmost bit might represent whether the input stream has encountered EOF, and 0, meaning *off* or *false*, would represent the idea that it has not. The 2nd bit from the left, set to 1 meaning *on* or *true*, might represent a fail state, and so on.

Bitwise operations can only be performed on integral data types, for example char, int and unsigned int. Unsigned int is often used.

As our first example manipulating individual bits, assume we are working on a platform using the unsigned int type and that sizeof(unsigned int) is 4 bytes. This gives us 4 * 8, or 32 bits, to work with. We could declare an appropriate variable and store the bit pattern above in the rightmost bits as follows:

unsigned int flags = 0x0000004a;

Using this declaration and initialization, flag will now contain the 32-bit pattern

```
00000000000000000000000001001010
```

In the following discussion we will assume a a platform using an 8-bit byte, a char type with a sizeof 1, the ASCII character set, and two's complement representation of integers.

5.2.1 AND, OR, XOR, NOT Operations

These bitwise operations act similarly to the corresponding logical operations, but they operate on each pair of individual bits.

Let's assume we begin with the following declarations:

```
char ch1 = 'A';   /* ASCII code is decimal 65 */
char ch2 = 'B';   /* ASCII code is decimal 66 */
char ch3;
```

The bit pattern for ch1 is 01000001 and for ch2 it is 01000010.

The bitwise AND operation is specified by the operator &. Example:

```
ch3 = ch1 & ch2;
```

sets ch3 to the bit pattern 01000000:

```
01000001
01000010
--------
01000000
```

The bitwise OR operation is specified by the operator |. Example:

```
ch3 = ch1 | ch2;
```

sets ch3 to the bit pattern 01000011:

```
01000001
01000010
--------
01000011
```

The bitwise XOR operation is specified by the operator ^. Example:

```
ch3 = ch1 ^ ch2;
```

sets ch3 to the bit pattern 00000011:

```
01000001
01000010
--------
00000011
```

The bitwise NOT operation takes one argument and is specified by the operator ~. Example:

```
ch3 = ~ch1;
```

sets ch3 to the bit pattern 10111110:

```
01000001
--------
10111110
```

The bitwise NOT is also known as the *logical complement* or the *one's complement* operator. Note that it generates the one's complement of the original bit pattern.

5.2.2 Example Mask: Set

Using the bitwise operations just discussed, we can use a very powerful technique in bit manipulation: the mask. A *mask* is a bit pattern which is applied to another bit pattern using some bitwise operation, in order to achieve a desired result.

For example, let's say that we have the following 8 bit pattern set up and we are using it to store 8 flags representing the condition of an input stream:

```
char IOflags = 0;
```

Here we have initialized all of the 8 flags to zero, meaning they are all *off*, represent *false*, and are not *set*.

If the leftmost bit represents the EOF condition, and in executing some code, the input stream encounters EOF, we can then *set* the crucial bit using this mask and the OR operation:

```
00000000       original set of flags
10000000       the mask with a 1 in the position to set
--------
10000000       end result after OR operation
```

After applying the mask using OR, the flag is set to 1, meaning *true* or *on*.

5.2.3 Example Mask: Reset

In other situations we may want to *reset* a bit to 0, meaning *false* or *off*.

Assume our IOflags variable contains 10000000 after a set has occurred, and we now want to *reset* the leftmost bit to zero. We can apply the following mask and use AND:

```
10000000       starting set of flags
00000000       the mask with a 0 in the position to reset
--------
00000000       end result after AND operation
```

Masking techniques are incredibly simple, yet also incredibly elegant and useful.

5.2.4 Left Shift <<

The shift operators are used to move bits to the left or to the right within a storage cell. Again, integral types are normally used to store bit patterns. The basic syntax is

```
i << j
```

where i is the integral value to be shifted. The operation will left shift the bits in i by j positions to the left. Note that j should be a positive integral value which is not greater than the length of i in bits. Vacated positions on the right are automatically filled with zeroes. If the data type of i is int, unsigned int, long, or unsigned long, then the data type is left as it is. If the data type of i is unsigned char or unsigned short, the value is converted to an int before the shift is performed, if type int is adequate to represent the original value; if int is not adequate, then type unsigned int is used. The value of the evaluated expression is of the original type or of the converted type if a conversion is done.

Example: let's assume we are using the unsigned int type to store machine level instructions, and that the leftmost 4 bits are required to contain the opcode. If we are in the process of building an instruction, and we currently have this situation

```
unsigned int instruction;
```

where instruction's current value is

```
00000000000000000000000000001101
```

we can move the opcode where it needs to go using the following code:

```
/* sets instruction to 11010000000000000000000000000000 */
instruction = instruction << 28;
```

It is also possible to use logical operations and a mask to perform this task. There is almost always more than one way to accomplish the same task in C. The purpose of the examples in this section is to show you the mechanics of how these shift operations work. You must use your own judgment in determining what is the best option for a specific situation.

5.2.5 Right Shift >>

The right shift moves bits to the right. The basic syntax is

```
i >> j
```

where again i is the bit pattern to be shifted and j gives the numbers of positions to shift.

Let's say that we have the following machine instruction stored in the variable *instruction* declared above:

```
11010000000000000000000000000000
```

Now, let's say we are converting this instruction to another format, which requires the opcode to be located in the 2nd group of four bits, starting from the left. We can do this:

```
instruction = instruction >> 4;
```

This will change the value in instruction to

```
00001101000000000000000000000000
```

Note that if other bits are set, of course, they will have to be taken care of appropriately.

The right shift will pad vacated bits with zeroes consistently across platforms only if the value to be shifted is of type unsigned int. Note that the original data type is not converted as it is for left shift operations, and if the original data type is not unsigned int, platforms do vary in how they pad vacated positions. Some systems always pad with zeroes (called a *logical shift*) but some systems will pad with zeroes if the original value is positive, and ones if the original value is negative (called an *arithmetic shift*).

5.3 Combined Assignment and Bitwise Operators

As with basic operators such as + and *, the bitwise operators can be combined with assignment as follows:

Operator	**Effect**
<<=	left shift and assign
>>=	right shift and assign
&=	bitwise AND and assign
|=	bitwise OR and assign
^=	bitwise XOR and assign

Table 5.1

For example, the following two statements are equivalent:

```
instruction = instruction >> 4;

instruction >>= 4;
```

5.4 A Note on Bit Fields

It is possible to store bit patterns in C in a programmer-specified number of bits. For example, we could store a bit pattern in 2 bits or 12 bits. This is very useful for storing tightly *packed* information.

Bit fields must be declared inside of structures in C. For this reason, we will discuss this topic later in this textbook, along with C structs.

Chapter 6: Data Structures in C

The fundamental data structures in C are the array, the struct and the union. In C++, we have these same types, and as an addition, the class.

In C++, you may have used the string and vector classes extensively. These classes have at their core what is called a *dynamic array*. It is critically important to know how the C array, which is the same array type in the core C++ language, operates. As part of this we will learn about how strings are stored and manipulated in C, which is enormously different from how string class objects are set up and used in C++.

In this chapter we will look at how arrays, strings and structs are used in the C language.

Arrays and pointers are heavily intertwined in C, and it is important that you have mastered the concepts in chapters 2 and 4 before reading this chapter.

6.1 Arrays and Pointers

The following concepts are critical with respect to arrays in C:

(1) the name of the array represents the address of the first element, is a pointer to the first element, and is a constant
(2) the address of the first element is the *base address*, and the address of every other element in the array is calculated using the *offset* from this base, as in

base address + (index * sizeof(data type of array element))

(2) array elements are stored in sequence in memory
(3) array indices start at zero and go up from there to the number of elements - 1

(4) the C language does not perform any range checking on array indices, thus it is critical to prevent array bounds errors in your code; this is especially important to prevent worm and viral attacks, as well as nasty program crashes!

The following program sets up a one-dimensional array in C and demonstrates some fundamental operations and concepts.

```
#include <stdio.h>
#define SIZE 10       /* number of array elements declared as constant */
int main()
{
   int i;
   int intArray [SIZE];

   /* initialize all array elements to index*2 */
   /* valid indices range from 0 to SIZE-1 */
   for (i = 0; i < SIZE; i++)
      intArray[i] = i * 2;

   /* print the name of the array: the address of the first element */
   printf("intArray is: %x\n", intArray);

   /* print each array element address, index, and value */
   for (i = 0; i < SIZE; i++)
     printf("%x   %d   %d\n", &intArray[i], i, intArray[i]);

  return 0;
}
```

Program 6.1

Here is a sample run on a UNIX machine on which the sizeof(int) is 4:

```
aftyson@diablo:~/testc>a.out
intArray is: bff06bb0
bff06bb0   0   0
bff06bb4   1   2
bff06bb8   2   4
bff06bbc   3   6
bff06bc0   4   8
bff06bc4   5   10
bff06bc8   6   12
bff06bcc   7   14
bff06bd0   8   16
bff06bd4   9   18
```

We can observe very easily that the name of the address is indeed the address of the first element, and that the elements themselves are stored in consecutive int cells, each occupying 4 bytes on this machine. We can also see for the element with index 2

```
&intArray[2] = intArray + (2 * sizeof(int))
```

When we write something like `intArray[2]` in our code, the compiler essentially translates the brackets and array index into a pointer dereference to the proper address.

The C language was set up this way to provide very fast and efficient access to array elements.

6.1.1 Passing One-Dimensional Arrays to Functions

When we pass an array to a function, as in this example call

```
DoSomething (intArray);
```

we are passing a pointer to the first element, using call by value. The prototype for this example could be any of these three options:

```
void DoSomething (int arrayParam [SIZE]);

/* ok because compiler ignores the SIZE */
/* the SIZE can be ignored because there is no range checking */
void DoSomething (int arrayParam [ ]);

/* ok because the name of array is of type int * */
void DoSomething (int *arrayParam);
```

The function called, DoSomething, receives a pointer to the first element of the array and then can access the array elements just as the caller could, e.g. this statement inside DoSomething

```
arrayParam[i] = 17;
```

would set the (i + 1)th element of intArray to 17.

Using this method of passing arrays, C saves memory space and time due to not copying the elements in the original array. Only the address of the first element is copied.

6.1.2 Using a Pointer Variable to Access Array Elements

Using array intArray established above, we could also access the array purely using a pointer variable. For example, to print out the entire array, we could use instead

```
int *p;
p = intArray;
for (i = 0; i < SIZE; i++)
{
   printf("%d\n", *p);
   p++;
}
```

This illustrates an important concept about pointer arithmetic. Note that we still need to make sure the code does access any memory past the end of the array.

```
p++;
```

does not "add one" to the pointer, as in the sense of adding one to an integer. It increments the pointer p by **1 * sizeof(<the basetype of the pointer>)**. In this case, it increments p by sizeof(int).

Pointer variables are often used to access array elements to increase efficiency in certain coding situations. For example, two-dimensional image elements may be stored in a one-dimensional array and then accessed using a single pointer very efficiently.

6.1.3 A Bit About Two-Dimensional Arrays and Beyond

The C language does not set a limit for the number of array dimensions. We can set up a two-dimensional array as in the following code:

```
#include <stdio.h>
#define NUMROWS 2
#define NUMCOLS 3
int main ()
{
   /* set up a 2 x 3 array of doubles */
   /* here we see how to initialize an entire array to zero as well */
   double floatTable [NUMROWS][NUMCOLS] = {0};

   /* access one element using row index and column index */
   floatTable[1][3] = 5.12;

   /* etc. */

}
```

We will not go into the details of how two-dimensional arrays are stored in this book. But, it is worth noting that most compilers will display the data type of floatTable as double** (pointer to a pointer to a double). If we pass floatTable to a function, as in

```
ProcessTable (floatTable);
```

we are still passing a pointer to the array, and function ProcessTable will be able to access the array elements using two indices, just as in the caller. Possible prototypes for ProcessTable are

```
void ProcessTable (double tableParam[NUMROWS][NUMCOLS]);
```

and

```
void ProcessTable (double tableParams[ ][NUMCOLS]);
```

When passing an array of more than one dimension, the size of every dimension beyond the first MUST be provided!!!

Note also that is it possible to use a single index into this array, such as

```
floatTable[i]
```

as a valid expression. Specifically, this is a pointer to the first element in the (i + 1)th row, and is of type double * (pointer to double).

Arrays of three, four, and more dimensions can be declared and used similarly.

6.2 Strings

You most likely have used the C++ string class in your earlier studies. Suffice to say, the C string is a very different animal. Without further ado, we will cover the important topics you need to understand to effectively use C strings. When C strings are used correctly and appropriately, they are an *extremely* efficient data structure for storing and processing data, both in terms of memory storage and in processing time.

In essence, a string in C is a one-dimensional array of basetype char which contains a special character called the terminator to represent the end of the string. The terminator, or NULL character, is represented by ASCII code zero, 00000000 for a 1-byte char. As with arrays in general, the string array is declared with a specified maximum size. String data stored in a particular array may be, and often is, shorter than this maximum size.

Along with the enormous benefits in storage and time efficiency that come with C strings, however, come some programming pitfalls to be aware of. There are two critical items to remember when using C strings:

- Always be sure that no operation in your program will cause a string to exceed its declared maximum size. Note that the size must guarantee availability of one space for the NULL character. To prevent security breaches, write your code so that there is nothing the user can do to cause a string to overflow. Be particularly careful with input of strings.

- Always be sure that any string you use contains a terminator character. It is possible to lose the terminator, to strip it off, or to read in a string and omit the terminator in various ways.

The core C language, because it provides the array type, provides what you need to perform basic string operations. To utilize the standard C string functions which provide a great deal of the capabilities of C strings, we must also

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

6.2.1 String Fundamentals

First we will take a look the basics.

To declare a string constant, we use a macro as in the following:

```
#define MISSING_SEMICOLON_MSG "Semicolon missing on this line or above"
```

String constants like this are often used for error messages; for example a compiler might have a few hundred such string constants declared.

To declare a string variable, we typically declare a maximum size and then set up a type using typedef. We often utilize multiple string types of different sizes in a single program, as in

```
#define NAME_SIZE 26
#define ADDRESS_SIZE 51

typedef char NameString [NAME_SIZE];
typedef char AddressString [ADDRESS_SIZE];
```

This code sets up two string types, one for an array containing a maximum of 25 characters for a name plus allowing one space for the NULL character, and a second array containing a maximum of 51 characters for an address and the NULL. We could go on and declare variables using

```
NameString name = "Doctor Who";    // shows an optional initialization
AddressString address;             // this string is not initialized
```

Be careful! While in the *initialization* above we can successfully use the assignment operator =, we cannot use the = to assign to a string at <u>any</u> other point in a program. More on this later.

Every string must be terminated with the NULL character. For example, for the name above, what we have stored in memory is

'D'
'o'
'c'
't'
'o'
'r'
' '
'W'
'h'
'o'
\0
uninit
uninit

uninit
uninit
uninit
uninit
uninit
uninit
uninit
uninit
uninit
uninit
uninit
uninit

Table 6.1

The *size* of the name string is 26, and we are not using all of the available space. The *length* of the name string is 10, counting all of the characters in the string's value starting with the first character and counting up through the last one prior to the '\0'. If there is more than one NULL character in a string storage area, the first instance is the string's terminator.

We can access an individual array element just like in any array.

```
printf ("%c\n", name[7]);
```

would print the single character 'W'.

```
name[3] = '\0';
```

would change the string's value to "Doc" and its length to 3.

As with any one-dimensional array, the name of a string array is a constant and a pointer to the first element in the array. The data type of name is therefore char*.

6.2.2 String IO

6.2.2.1 Output of Strings

Let's look at the easy part first, output. We have several options to output a string. Here are some examples which illustrate how to print strings:

```
printf ("%s\n", name);
```

will print the value of name using the current length of the name for its field width.

```
printf("%30s", name);
```

will print the name right-justified using 30 for the field width and padding on the left with blanks.

When the %s specification is used, the printf function will print characters in the string starting at the first character, and stopping when it reaches the NULL character (which is not printed).

```
puts(name);
```

will print the name to the standard output using the length of the string for the field width and automatically adding a newline character in the output after the string.

That was pretty easy. Input is more complex.

6.2.2.2 Input of Strings

The following table summarizes several major options for string input.

Option	***Basic Action***
%Nc	• used with scanf and related functions • starts reading at current read pointer position and and loads characters read into string • reads and includes *all* characters including any whitespace • stops reading after N chars are read or when <eof> is encountered, whichever occurs first • leaves read pointer just after Nth character read or at <eof> • does *not* put '\0' in string
%s	• used with scanf and related functions • starts reading at current read pointer position and loads characters read into string, skipping over any leading whitespace • stops reading when a trailing whitespace is encounterd • leaves read pointer in front of the trailing whitespace • adds '\0' automatically
char *gets (char *s);	• reads from standard input stream • starts reading at current read pointer position and and loads characters read into string

	• reads and includes *all* characters including any whitespace • stops reading when <eoln> is encountered • does *not* store <eoln> in the string • leaves read pointer just after the <eoln> which stopped the read • adds '\0' to string automatically • returns NULL if fails to read any characters, or if successful, a pointer to the first character in string s
char *fgets (char *s, int N, FILE *infile);	• reads from input stream specified • starts reading at current read pointer position and and loads characters read into string • reads and includes *all* characters including any whitespace • stops reading when <eoln> is encountered OR when N-1 characters have been read, whichever occurs first • does store <eoln> in the string • leaves read pointer just after the <eoln> which stopped the read • adds '\0' to string automatically • returns NULL if fails to read any characters, or if successful, a pointer to the first character in string s

Table 6.2

Trying to memorize these routines and their subtle differences is enough to give anyone a headache. But, on the bright side, you as the programmer have the power and flexibility to perform string input in lots of extremely useful, and different, ways.

One VERY important note: the safest method of reading a string, of those methods listed here, is considered by many to be the *fgets* function, because it allows you to specify an integer giving the maximum number of characters to read, and it provides the '\0' automatically. The value of N is normally provided as the size of the string.

CODE TO PREVENT BUFFER OVERFLOWS WITH STRING READS, ESPECIALLY WHEN DOING INTERACTIVE INPUT!

Many virus and worm writers would love to take advantage of any buffer vulnerabilities that they can find, and they have provably done so in the past.

Let's look at some examples.

Using this declaration

```
char s[10];
```

this input line

```
Oh say, can you C?<eoln>
```

and this call to scanf

```
scanf("%s", s);
```

the value of s will be set to "Oh", including a terminating '\0'. Note that the NULL character is implicit in front of the last double quote.

Using the same declaration and input line, this call to scanf

```
scanf("%5c", s);
```

sets s to the value "Oh sa" but with no terminating '\0' in the string!

We must put it there explicity using

```
s[5] = '\0';
```

Using the same declaration and input line, this call to gets

```
gets(s);
```

causes an array overflow. This is bad news. The size of s is 10, but gets will read 18 characters and load them into the memory space for s, overwriting 9 spaces beyond the storage space for s. *Groan*.

Much safer: let's read the line using fgets; same declaration for s and same input:

fgets(s, 10, stdin);

Using this call, a maximum of 9 characters can be read from the input. The value of s will be set to "Oh say, c" and will include the '\0' to terminate the string.

We will see more examples of input when we look at an example program in the next section.

6.2.3 String Functions

For many of the most important string operations we must first

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

and then utilize the functions provided in this important header file. We must use standard functions for string copying ("assignment") and for string comparison. We will look at examples of a few standard functions in this section. A list of string function prototypes is provided in Appendix B.

To get us started, here are a few prototypes and discussion for some of the most useful string functions:

`size_t strlen (const char *s1);`

The strlen function returns the length of the string argument as an integral value. The data type size_t is platform dependent. String s1 must be null-terminated, as characters are counted starting from the character pointed at by s1 and continuing until the '\0' is encountered.

For example, if s1 has the value "hello!", then

```
printf ("%d\n", strlen(s1));
```

prints a 6.

How do you say the name of this function? Some prefer "string length", some "stir len", etc.

`char *strcpy (char *s1, const char *s2);`

The strcpy function copies characters from s2 into s1; s2 is unchanged and previous contents of s1 are overwritten. For this operation to work correctly, both strings must be null-terminated and s1 must be large enough to hold the result. The function returns the address of the first character in s1, which is often ignored.

```
strcpy (s1, "algorithm");
```

places the string "algorithm" including its '\0', into s1.

Note that we cannot write

```
s1 = "algorithm";
```

because s1 is the name of an array, it is a pointer, and it is a constant! We cannot assign to it.

```
strcpy (s1, s2);
```

copies the contents of s2 into string s1.

int strcmp (const char *s1, const char *s2);

This function compares two strings for their ordering in the character set of the platform in use. Strings are compared left to right, one character at a time. This function returns a negative value if s1 is lexicographically less than s2, a zero if they are equal, and a positive value if s1 is greater than s2.

For example, to check if s1 is less than s2 (as in the string "apple" is less than "banana") we would write this

```
if (strcmp(s1, s2) < 0)
```

To see if s1 is equal to s2 (as in "quit" is equal to "quit", but "quit" is not equal to "Quit" or " quit") we could write

```
if (!strcmp(s1, s2))
```

int strncmp (const char *s1, const char *s2, int N);

Many string functions in C have an alternate version, where an integer N is provided to determine how many characters to process. For example, strncmp compares only the first N characters of the two strings.

char *strcat (char *s1, const char *s2);

The strcat function concatentates s1 and s2. String s1 holds the result, and s2 is left unchanged. The arguments s1 and s2 must be null-terminated and s1 must be large enough to hold the result of the concatenation. The function returns the address s1.

There are many more string functions, and you can see a list of prototypes in Appendix B.

At this point in our discussion, we know quite a bit of practical information about strings. Let's look at a program in C++ using the string class, and a program with the same functionality in C using the C string, so that you can compare C and C++ string usage. First, the C++ version:

```
/* ============================================================ */
/* C++ string class example program
/* ============================================================ */

#include <iostream>
#include <string>         // header file for string class
using namespace std;

/* ============================================================ */

// a string constant
```

```
const string FIRST_MOTTO = "Go Noles!";

/* ============================================================= */

int main( )
{
    // declare string class objects
    string firstName, lastName, clientName;
    string motto = FIRST_MOTTO;

    // interactive input using >> will not read whitespace into string
    cout << "Enter your first name, with no blanks: ";
    cin >> firstName;
    cout << "Enter your last name, with no blanks: ";
    cin >> lastName;

    // concatenate using overloaded + operator
    clientName = lastName + ", " + firstName;

    // output and call member function length
    cout << endl << "Your name in our client list is: " << clientName
         << endl;
    cout << "This name has " << clientName.length() << " characters!"
         << endl << endl;

    // print current motto and ask user for a new one
    cout << "Our motto is: " << motto << endl;
    cout << "A motto must be entered on one line." << endl;
    cout << "Please suggest a new one: ";

    // do input using getline to pick up blanks in string
    // first ignore rest of old input line
    cin.ignore(80, '\n');
    getline (cin, motto);
    cout << endl << "Our new motto will be: " << motto << endl;

    // do a comparison of old and new mottos for equality
    // using overloaded == operator
    if (motto == FIRST_MOTTO)
        cout << endl << "You picked the same motto!" << endl << endl;

    return 0;
}

/* ============================================================= */
```

Program 6.2

When this program is run and the new motto is different from the old, the output looks like this:

```
aftyson@diablo:~/testcpp>a.out
Enter your first name, with no blanks: Dennis
Enter your last name, with no blanks: Ritchie
```

```
Your name in our client list is: Ritchie, Dennis
This name has 15 characters!

Our motto is: Go Noles!
A motto must be entered on one line.
Please suggest a new one: Go Dolphins!!!

Our new motto will be: Go Dolphins!!!
```

When this program is run and the new motto is identical to the old, the output looks like this:

```
aftyson@diablo:~/testcpp>a.out
Enter your first name, with no blanks: Dennis
Enter your last name, with no blanks: Ritchie

Your name in our client list is: Ritchie, Dennis
This name has 15 characters!

Our motto is: Go Noles!
A motto must be entered on one line.
Please suggest a new one: Go Noles!

Our new motto will be: Go Noles!

You picked the same motto!
```

Now, let's look at a program with the same runtime functionality using C strings and related operations. In this version, we must tell the user there is a maximum size for the motto.

```
/* ================================================================ */
/* C string example
/* ================================================================ */

#include <stdio.h>
#include <string.h>     // header file for C string functions

/* ================================================================ */

#define NAME_SIZE 26    // declare sizes for string arrays
#define MOTTO_SIZE 81

#define FIRST_MOTTO "Go Noles!"        // declare a string constant

typedef char NameString [NAME_SIZE];   // declare 2 string types
typedef char MottoString [MOTTO_SIZE];

void nextLine( );   // we do not have ignore function in C

/* ================================================================ */

int main( )
{
    // declare string variables, which are arrays
    NameString firstName, lastName, clientName;
    MottoString motto = FIRST_MOTTO;
```

```
    // interactive input using %s will not read whitespace into string
    printf ("Enter your first name: ");
    scanf ("%s", firstName);
    printf ("Enter your last name: ");
    scanf ("%s", lastName);

    // copy and concatenate to form client name
    strcpy (clientName, lastName);
    strcat (clientName, ", ");
    strcat (clientName, firstName);

    // output and call standard function strlen
    printf ("\nYour name in our client list is: %s\n", clientName);
    printf ("This name has %d characters!\n\n", strlen(clientName));

    // print current motto and ask user for a new one
    printf ("Our motto is: ");
    puts(motto);
    printf
      ("A motto must be on one line and 80 characters or fewer.\n");
    printf ("Please suggest a new one: ");

    // do input using fgets to pick up blanks in string
    // first ignore rest of old input line
    nextLine ( );
    fgets (motto, MOTTO_SIZE, stdin);

    // must strip off the <eoln> mark which was read into the string
    // by fgets, then print it
    motto[strlen(motto) - 1] = '\0';
    printf ("\nOur new motto will be: %s\n", motto);

    // do a comparison of old and new mottos for equality
    // using strcmp function
    if (!strcmp (motto, FIRST_MOTTO))
        printf ("\nYou picked the same motto!\n\n");

    return 0;
}

/* ================================================================ */

void nextLine( )
/*  ignore the rest of an input line */
/*  alternatively we could use fgets to do this task and read in */
/*  a dummy string to read and ignore the rest of the line */
{
    char nextChar;
    while (nextChar = getchar( ) != '\n');
      return;
}

/* ================================================================ */
```

Program 6.3

6.2.4 Using sscanf and sprintf

The sscanf and sprintf functions are used like scanf and printf, except that the "input" source is a string variable and the "output" target is a string variable.

It is sometimes more convenient to read an entire input line using fgets, and then "read" the individual components from the string using sscanf. For example, let's say that we ask the user to enter three integer values separated by blanks, and then hit return. The user enters

```
   17      335  -1<eoln>
```

We can use the following code to perform the needed input and storage of values:

```
char buffer1[81];
char buffer2[81];
int num1, num2, num3;

fgets (buffer1, 81, stdin);
sscanf (buffer1, "%d%d%d", &num1, &num2, &num3);
```

The value 17 will be put in num1, 335 into num2, and -1 into num3. Note that sscanf is converting string data into numeric data here. The standard atoi function can also be used to do a string to int conversion.

In other situations we might want to place data of various types into a string for convenience. If we use this call

```
// note there are 3 blanks between each pair of %ds
sprintf (buffer2, "%d   %d   %d", num1, num2, num3);
```

we will place the string "17 335 -1" into buffer2, including a NULL terminator. The sprintf function does automatically place the '\0' at the end of thestring.

6.2.5 Scansets

Now that we have learned about strings, we can talk about using scansets with scanf and related functions.

The code

```
[<characters>]
```

provided with a corresponding argument of type char*, tells scanf to read all characters in the input, load those in the specified set into the string, and stop the reading process when a character not in the set is encountered.

For the following discussion please assume this declaration:

```
char buffer[81];
```

As our first example, let's say the user input is

```
dolphins21bills28
```

This call to scanf will cause "dolp" to be read into buffer (and will include the '\0'):

```
scanf ("%[adlmnop]", buffer);
```

Basically, we can use %[ ] as an alternative to %s to gain fine tuning over our string input.

The code

```
[^<characters>]
```

provided with a corresponding argument of type char*, tells scanf to read all characters in the input, load those that are NOT in the specified set into the string, and stop the reading process when a character in the set is encountered.

Let's say the user input is the same as before, but we use this call to scanf:

```
scanf ("%[^123456789]", buffer);
```

The string "dolphins" will be read into buffer.

To read all characters into buffer and stop at the end of line mark, we could use this:

```
scanf ("%[^\n]", buffer);
```

Similarly, we could use this mechanism to move the read pointer to the next TAB stop in a data file with fscanf, and so on.

6.3 Structs

If you have programmed in C++ or JAVA, you have almost certainly used the class data structure. The essential idea behind the class is to encapsulate data and functions inside one data structure. Data and function members in a class can be public or private.

The C struct is used to store data only, and all members are public. While some platforms do allow structs to contain functions and private data, it is generally considered bad programming style to use them that way.

The struct is typically used to represent the abstract data type *record*, containing member data often of different data types.

6.3.1 Struct Fundamentals

Let's declare a useful struct type to store data about clients of a computer system, perhaps something like an online store. Assume the following declarations:

```
#define NAMESIZE 26
#define PASSWORDSIZE 9

typedef char NameString [NAMESIZE];
typedef char PasswordString [PASSWORDSIZE];

typedef struct
{
   int number;
   NameString name;
   PasswordString password;
} ClientRecord;
```

The data type we have set up here has name *ClientRecord*, and contains three members. We can declare a variable using this type as in

```
ClientRecord client;
```

The client struct will be stored in memory in a series of consecutive locations.

Here's example code initializing all of the members of client:

```
client.number = 111111;
strcpy(client.name, "New Client");
strcpy(client.password, "X1Y2Z3");
```

As you can see, the member or dot operator **.** is used to access members. Because the members are public, any code for which client is in scope can access the members directly.

When we pass a struct to a function, we can pass it by value if we do not want it to be modified. If we want the called function to be able to modify the struct, we must pass a pointer to it. We can use this single pointer to access any data in the struct.

The following example program uses the ClientRecord above and illustrates passing structs to functions as well as accessing members via a pointer:

```
/* ============================================================== */

#include <stdio.h>
#include <string.h>

#define NAMESIZE 26
#define PASSWORDSIZE 9
```

```
typedef char NameString [NAMESIZE];
typedef char PasswordString [PASSWORDSIZE];

/* declare the struct type in a global position */
typedef struct
{
   int number;
   NameString name;
   PasswordString password;
} ClientRecord;

void InitClient (ClientRecord *inClient);
void PrintClient (ClientRecord inClient);

/* ============================================================== */

int main ()
{
   ClientRecord client;

   /* pass the address of the struct so that it can be modified */
   InitClient (&client);

   /* pass the struct by value so that it is protected from change */
   PrintClient (client);

   return 0;
}

/* ============================================================== */

void InitClient (ClientRecord *inClient)

/* this function initializes all members of the client parameter */

{
   /* we can access struct members via the pointer using */
   /* one of two alternative notations */
   /* inClient->number = 123456; */
   /*    sets the client's number to the value using most common syntax */
   /* (*inClient).number = 123456; */
   /*    is the alternative and is also correct here */

   inClient->number = 111111;
   strcpy(inClient->name, "New Client");
   strcpy(inClient->password, "X1Y2Z3");

   return;
}

/* ============================================================== */

void PrintClient (ClientRecord inClient)

/* this function prints all members of the client parameter */
```

```
{
   /* this function has a complete copy of the original client record */

   printf("Number: %d\n", inClient.number);
   printf("Name: %s\n", inClient.name);
   printf("Password: %s\n", inClient.password);

   return;
}

/* ============================================================== */
```

Program 6.4

In the next section we will demonstrate how to access the characters inside the string elements inside these structs, as well as how to combine arrays and structs in general.

6.3.2 Combining Arrays and Structs

We can set up very complex data structures by combining arrays and structs in virtually any manner we wish. Arrays can occur inside of structs, structs inside of arrays, structs inside of structs... the possibilities are endless.

First, note that in the ClientRecord struct we declared above, there are two array members. We can access the array elements in these structs using array subscripts:

```
client.name[i]
```

will access the (i + 1)th character in the name string of client.

Let's change our struct so that it also contains a nested struct. Consider the following declaration for a struct containing information on a date:

```
typedef struct
{
   int month;
   int day;
   int year;
} DateRecord;
```

We could utilize this inside the ClientRecord:

```
typedef struct
{
   int number;
   DateRecord dateJoined;
   NameString name;
   PasswordString password;
} ClientRecord;
```

Now, let's declare a variable using the new type and look at how to access the nested struct members:

```
ClientRecord client;

/* initialize the date members */
client.dateJoined.month = 1;
client.dateJoined.day = 1;
client.dateJoined.year = 2006;
```

Let's go one more step and declare an array of these ClientRecords. Here is a program which declares such an array and initializes all of the members to defaults:

```
/* ================================================================ */

#include <stdio.h>
#include <string.h>

#define NAMESIZE 26
#define PASSWORDSIZE 9
#define NUM_CLIENTS 100

typedef char NameString [NAMESIZE];
typedef char PasswordString [PASSWORDSIZE];

/* declare the DateRecord struct type */
typedef struct
{
   int month;
   int day;
   int year;
} DateRecord;

/* declare the ClientRecord struct type */
typedef struct
{
   int number;
   DateRecord dateJoined;
   NameString name;
   PasswordString password;
} ClientRecord;

/* declare a type for an array of ClientRecords */
typedef ClientRecord ClientList [NUM_CLIENTS];

void InitAllClients (ClientList clients);

/* ================================================================ */

int main ()
{
   /* clients is an array of structs */
   ClientList clients;

   /* note that here we are passing the name of an array, which is */
```

```
   /* a pointer to the first element in that array */
   InitAllClients (clients);

   return 0;
}

/* ============================================================== */

void InitAllClients (ClientList clients)
{
   int i;

   /* initialize all values in all records to defaults */
   /* clients[i] selects (i + 1)th client record */
   /* then select the member using the dot operator */
   for (i = 0; i < NUM_CLIENTS; i++)
   {
      clients[i].number = 111111;
      strcpy(clients[i].name, "New Client");
      strcpy(clients[i].password, "X1Y2Z3");
      clients[i].dateJoined.month = 1;
      clients[i].dateJoined.day = 1;
      clients[i].dateJoined.year = 2006;
   }
   return;
}

/* ============================================================== */
```

Program 6.5

For some fun mental exercise, think about how you would set up things like arrays of structs containing two or three dimensional arrays and whatever else you can come up with. Data in the real world is often extremely complex.

In the next chapter we will look at how to allocate arrays and structs *dynamically*.

6.3.2.1 Passing An Array "By Value" By Hiding it Inside a Struct

We have noted than when an array is passed, including an array of structs, the C language provides that the address of the first element in the array is passed as the parameter. What if you would like to pass an array and protect its contents from being changed? A common convention is to declare the array inside a struct, and then pass the struct by value.

For example, we could declare an array of ints inside a struct as follows:

```
struct hiddenArray
{
   int myArray [10];
};
```

When we pass hiddenArray by value, we protect the array inside it from being modified by the called function.

Note that placing arrays inside of structs also allows you to copy from one array to another using the assignment = operator.

6.3.3 Bit Fields

Bit fields provide a manner in which we can utilize structs so as to achieve great economy of storage space. In essence, we can *pack* a struct's members to achieve storage efficiency and also utilize bitwise operations on these members if we wish.

If you have not read Chapter 5 as of yet, which covers bitwise operations, now would be a great time to do so.

Here are some major points to remember about bit fields:

- bit fields can only be set up as members of structs
- the data types of the members used as bit fields must typically be of type *int, unsigned int,* or *signed int;* C99 also allows bit fields of type _Bool
- bit fields are declared with a particular size (number of bits); this size cannot exceed the size of the type in use; the bit field size must be a constant, zero or positive integer expression
- the address operator cannot be applied to a bit field
- a bit field cannot be referenced via a pointer
- a struct can contain both regular members (which we have discussed earlier in this chapter) and bit fields
- bit fields are often used to maximize storage efficiency in machine-dependent programs and code using them is thus subject to portability concerns

Here is an example of a short program declaring a struct and utilizing bit fields in that struct. We are assuming a platform where sizeof(unsigned int) == 32.

```
#include <stdio.h>

/* all of the information for one of these structs will be stored */
/* in one 32 bit unsigned int storage cell */
typedef struct
{
   unsigned int color : 16;  /* number after the : specifies */
                             /* the width of the field in bits */
   unsigned int hue : 12;
   unsigned int saturation : 4;
} ColorRecord;

int main ()
{
   ColorRecord myColor;
```

```
   myColor.color = 242;     /* stored in first 16 bits of the unsigned int */
   myColor.hue = 11;        /* next 12 bits */
   myColor.saturation = 5;  /* next 4 bits */

   printf("color: %d\n", myColor.color);
   printf("hue: %d\n", myColor.hue);
   printf("saturation: %d\n", myColor.saturation);

   return 0;
}
```

Program 6.6

This code, when run, prints 242, 11 and 5 as you would expect.

We could, as another example, declare a very large array of these color structures and store a great deal of information in a minimal amount of storage. Contrast this to the storage required if we declared each of the three struct members separately as regular integers.

Bit field storage is highly platform dependent and it is recommended that you carefully study the specifications of the platform you are working with.

Chapter 7: Dynamic Memory Allocation

C provides several functions to manage memory allocation and deallocation. In C, we typically use the *malloc, calloc,* and *free* functions in contrast to the *new* and *delete* operators found in C++. Memory allocated dynamically is stored in an area of memory called the *heap*, also called the *free store*.

7.1 Fundamentals: malloc, calloc, free, heap

First, let's look at the important functions C provides for memory allocation and deallocation. The malloc, calloc and free functions are declared in stdlib.h, so we must **#include <stdlib.h>** in order to use them.

`void *malloc(size_t size);`

The malloc ("memory allocation") function allocates a region of memory consisting of *size* bytes. The memory cells are not initialized. The size is usually provided by using the sizeof operator in the call to malloc. The data type size_t is implementation dependent and is an integral type. A pointer to the first element of the memory allocated is returned. The return type "void *" refers to a generic pointer type which is typically cast to the type required. If malloc cannot successfully allocate memory, it returns the NULL pointer.

`void *calloc (size_t nElements, size_t size);`

The calloc function allocates a region of memory with nElements elements, with each the size in bytes specified by the second argument. The memory cells allocated are initialized to zero. Like malloc, calloc returns a NULL pointer if it cannot allocate memory successfully, and if successful it returns a pointer to the first element in the newly allocated space.

`void free (void *pointer);`

The free function deallocates memory which was previously allocated by a call to malloc or calloc. The data type of *pointer* should be the specific type of pointer originally used to allocate the space.

Note: to practice safe programming, it is considered best after any call to free to explicitly set the pointer involved to NULL. The call to free by itself does *not* do that.

C provides the additional functions *realloc* and *relalloc* which we will not discuss in this text.

First, let's look at an example program to allocate and then deallocate a dynamic integer variable using a pointer named p. We can picture the situation in memory like this:

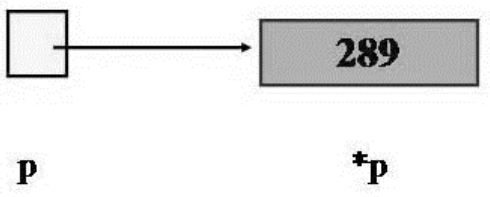

Figure 7.1

```
/* ================================================================ */
/* C first dynamic variable example                                 */

#include <stdio.h>
#include <stdlib.h>   /* needed for malloc and free */

int main ()
{
   /* declare a pointer to an int which is uninitialized at first */
   int *p;

   /* call malloc using sizeof(int) to provide the correct number of */
   /* bytes to allocate.  p will be set to the address of the first */
   /* byte of the memory allocated or to NULL if the allocation */
   /* fails */
   p = (int *)malloc(sizeof(int));

   /* check for error before using the integer */
   if (p != NULL)
   {
      /* if allocation ok, set dynamic integer to a value and print */
      *p = 289;
      printf("%x  %d\n", p, *p);
   }
   else
      /* memory could not be allocated for some reason */
      printf ("memory allocation failed\n");

   /* deallocate the memory for the dynamic int and reset pointer */
   free (p);
   p = NULL;

   return 0;
}
```

```
/* ============================================================= */
```

Program 7.1

When this program is run, it produces the following output (some of which is platform dependent):

```
aftyson@diablo:~/testc>a.out
8464008  289
```

7.2 Dynamic Arrays

When we discussed arrays earlier in this text, we always declared a constant for the size, which is the specific number of elements in the array. In some cases it is more useful to set up an array with a number of elements determined by a variable value generated at runtime. We can do this using malloc to allocate the array space.

7.2.1 A Dynamic One-Dimensional Array

Keep in mind that when we declare an array like this

```
#define SIZE 10
int myArray [SIZE];
```

The identifier myArray represents a constant pointer of type int *. The array size is fixed at compile-time.

We can set up a dynamic array of ints named dynArray something like this:

```
/* declare a pointer to refer to the array */
int *dynArray;

/* call malloc to allocate the actual array in the heap */
dynArray = (int *) malloc (<number of elements> * sizeof(int));
```

If we do this, dynArray is a pointer of type int*, but it is *not* a constant, it is a variable. This gives us the ability to allocate memory for it on the fly, free it, allocate it again with a different size, and so on.

Consider the following program which sets up and then deallocates a dynamic array:

```
/* ============================================================= */
/* C dynamic array example */
/* ============================================================= */

#include <stdio.h>
#include <stdlib.h>         /* needed for malloc and free */
```

```
/* ============================================================= */

int main ()
{
      int* p;          /* we will use this pointer for the array */
      int arraySize,
            lcv;

      /* set up a dynamic array and work with it a bit */
      /* read in the size desired from the user */
      printf ("How many cells do you want in this array? ");
      scanf ("%d", &arraySize);
      p = malloc (arraySize * sizeof(int));

      /* we can use [ ] just like with our previous arrays */
      /* initialize array elements, then print them */
      for (lcv = 0; lcv < arraySize; lcv++)
            p[lcv] = lcv * 2;
      for (lcv = 0; lcv < arraySize; lcv++)
            printf ("%d ", p[lcv]);
      printf ("\n\n");

      /* deallocate the entire dynamic array */
      free (p);
      p = NULL;

      return 0;
}

/* ============================================================= */
```

Program 7.2

When we run this program, we see output like this (which again is platform dependent):

```
aftyson@diablo:~/testc>a.out
How many cells do you want in this array? 12
0 2 4 6 8 10 12 14 16 18 20 22
```

Note that the dynamic array is the core data structure in the C++ *string* and *vector* classes.

7.2.2 A Ragged Array of Strings

Let's think about memory efficiency for our next discussion. What if we'd like to store an array of strings, but we know that they are all different lengths. We could declare all strings using one maximum size which would handle them all, but this would result in a lot of wasted space. A *ragged* array provides a better alternative. In a ragged array, the array elements can vary in size. Up to this point, we have only discussed *smooth* arrays, in which all elements are identical in size.

We can set up our array of strings most efficiently as an array of pointers to strings, where each string may have its own size, with that size the minimum needed for each individual string. This requires that we allocate each string dynamically.

For example, let's assume we need to store the names of college majors. We have three majors (wow, this is a small school!), math, chemistry and computer science. To store these names as strings in a smooth array, we would have to use 17 as the size for all the name strings, since the longest string, "computer science", requires it. If we declare our array as

```
char major [3][17];
```

we are using 51 bytes, many of which are wasted space. In the following diagram the grey boxes indicate wasted cells:

m	a	t	h	\0												
c	h	e	m	i	s	t	r	y	\0							
c	o	m	p	u	t	e	r	''	s	c	i	e	n	c	e	\0

Figure 7.2

Let's set up a ragged array of strings instead and eliminate all of these grey boxes. We can start by declaring an array of 3 pointers:

```
char *majors [3];
```

Be careful! All we have set up so far is 3 pointers! We do not have any space allocated for the strings themselves. The following program will allocate memory for the strings and read in the string values appropriately, while avoiding any wasted bytes:

```
/* ============================================================= */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main ()
{
   char tempstring[17];    /* temp string to use for input */
   int templen;            /* length of temporary string */
   char *majors [3];       /* the array of pointers */
   int i;

   /* for each string, read in temp string, allocate space */
   /* and then link the new string into the ragged array */
   for (i = 0; i < 3; i++)
   {
      fgets(tempstring, 17, stdin);
      templen = strlen (tempstring);
```

```
      /* allocate space for the string just read in, and link it */
      /* we allocate the minimum bytes required by this string */
      majors[i] = (char *) malloc (templen * sizeof(char));
      strcpy (majors[i], tempstring);
   }

   /* print out the strings */
   for (i = 0; i < 3; i++)
      puts(majors[i]);

   /* deallocate the strings to clean up */
   for (i = 0; i < 3; i++)
   {
      free (majors[i]);
      majors[i] = NULL;
   }

   return 0;
}

/* ================================================================ */
```

Program 7.3

The memory savings here can be substantial with large amounts of data. Here we are using 32 bytes (5 + 10 + 17) with the ragged array as compared to 51 with the smooth array. Our new array looks like this:

m	a	t	h	\0												
c	h	e	m	i	s	t	r	y	\0							
c	o	m	p	u	t	e	r	''	s	c	i	e	n	c	e	\0

Figure 7.3

Imagine an array of, say 500,000 product names in an online business inventory, and you get the idea that the memory savings can be very significant.

We can also access either array using two sets of brackets and array indices as with any two-dimensional array. For example,

```
majors[2][0] = 'C';
```

would set the first character of the third string to 'C', regardless of how we initially set up the array.

7.3 A Linked List in C

In C++ there are a variety of ways to set up a linked list: we can use dynamic structs and pointers, we can use classes and/or template classes, or we can use the list and iterators provided in the C++ STL. In C, we use dynamic structs and pointers.

In this section we will look at a C program which builds a linked list in alphabetical order.

First, let's assume we have a data file containing words and definitions, something like this:

```
fast
characterized by quick motion
clamourous
marked by confused din or outcry
rat
any of numerous rodents
pointer
one that points
```

This "mini-dictionary" contains a word on one line, and its definition on the following line. The data is not alphabetized, but we will build our linked list so that it is.

Each node of our linked list will be a dynamically allocated struct and will contain a word, its definition, and a pointer to the next word in the list, as follows:

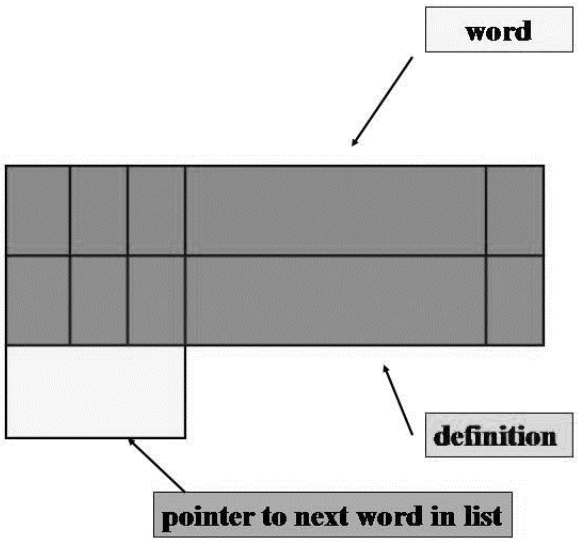

Figure 7.4

To set up an ordered linked list, recall that when a new node is added, we must execute code for one of two cases:

(1) the new node is attached at the head of the list

(2) the new node is attached in the middle of the list or at the end

In either case, we must do a sequential search to find the correct insertion point.

The following C program creates an ordered linked list as described.

```
/* ============================================================= */
/* This program sets up a linked list using dynamic structs. */
/* Reads data into the list and then prints it out. */
/* The list is set up in sorted (alphabetical) order. */
/* ============================================================= */
/* GLOBAL HEADER FILES */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* ============================================================= */
/* GLOBAL CONSTANTS */

#define STR_SIZE 81                /* size for all strings */
#define NULL_CHAR '\0'             /* the null character */

/* ============================================================= */
/* GLOBAL DATA TYPES */

typedef char string [STR_SIZE];     /* a string type */

typedef struct WordRec              /* the struct type for each word */
{                                   /* node in our linked list */
      string
            word,
            definition;
      struct WordRec* next;         /* pointer to next node in */
                                    /* linked list */
} WordRec;

typedef WordRec* WordRecPtr;        /* a type for the pointers */

typedef enum bool {false, true} bool;     /* for logical values */

/* ============================================================= */
/* GLOBAL FUNCTION PROTOTYPES */

WordRecPtr BuildSortedList ( );
void FindInsertionPoint(WordRecPtr firstWord, string wordToInsert,
                        WordRecPtr* insertAt, WordRecPtr*
predecessor);
void InsertBefore(WordRecPtr insertAt, WordRecPtr predecessor,
                  WordRecPtr temp, WordRecPtr* firstWord);
void PrintAllWords ( WordRecPtr );

/* ============================================================= */

int main ( )
```

```
{
      /* create pointer for head of list */
      WordRecPtr firstWord;

      firstWord = BuildSortedList ( );    /* build the list */

      PrintAllWords ( firstWord );        /* print data in list */

      return (0);
}

/* ============================================================== */

WordRecPtr BuildSortedList ( )

/* Builds a linked list in sorted order */
/*
   Method
      - use tempWord to create a new node
      - call FindInsertionPoint to find out where the node goes
      - call InsertBefore to do the insertion into the list
*/

{  /* function BuildSortedList */

   FILE* infile;        /* input data file */
   WordRecPtr
      tempWord,         /* used to create a new node */
      firstWord,        /* pointer to first node in list */
      insertAt,         /* pointer to node to insert before */
      predecessor;      /* pointer to its predecessor */

   string inWord, inDefn;     /* used to read data */

   char fileName[] = "words.txt";

   /* open data file read-only */
   infile = fopen ( fileName, "r" );

   /* if file does not open, abort the run */
   /* we will discuss the use of the exit function in the next */
   /* chapter */
   if(!infile)
   {
      printf("FATAL ERROR: file will not open!\n");
      exit (-1);
   }

   firstWord = NULL;          /* initialize list pointer to NULL */

   /* keep reading and entering words until eof encountered */
   /* read data into temporary storage first */
   while ( (fgets ( inWord, STR_SIZE, infile ) != NULL ) &&
            (fgets ( inDefn, STR_SIZE, infile) != NULL))
   {
      /* strip off EOLNS */
      inWord[strlen(inWord) - 1] = NULL_CHAR;
```

```
      inDefn[strlen(inDefn) - 1] = NULL_CHAR;

      /* set up new word node */
     tempWord = malloc (sizeof(WordRec)); /* allocate new node */
     strcpy(tempWord->word, inWord);      /* store data read */
     strcpy(tempWord->definition, inDefn);      /* in new node */

      /* to maintain alphabetical order, must find correct insertion */
      /* point and insert node at that point */
      FindInsertionPoint (firstWord, inWord, &insertAt, &predecessor);
            InsertBefore (insertAt, predecessor, tempWord, &firstWord);

   }

   fclose ( infile );
   return ( firstWord );      /* return external list ptr */

}  /* end of function BuildSortedList */

/* ================================================================ */

void FindInsertionPoint
(WordRecPtr firstWord,      /* pointer to first node in list */
string wordToInsert,        /* the word being inserted */
WordRecPtr* insertAt,       /* returns ptr to node to insert before */
WordRecPtr* predecessor)    /* returns ptr to node before that */
                            /* NOTE: this is a pointer to a */
                            /* pointer ! */

{
   bool found = false;      /* initially, location not found */

   *insertAt = firstWord;   /* start search at first node */
   *predecessor = NULL;     /* pred follows one behind */

   /* keep searching until hit end of list or find point to insert */
   while ((*insertAt != NULL) && (!found))
   {
      if ((strcmp(wordToInsert, (*insertAt)->word) < 0))
         found = true;                    /* found insert point */
      else
      {
         *predecessor = *insertAt;        /* advance to next node */
         *insertAt = (*insertAt)->next;
      }
   }

}  /* end of function FindInsertionPoint */

/* ================================================================ */

void InsertBefore
(WordRecPtr insertAt,     /* pointer to node to insert before */
WordRecPtr predecessor,   /* pointer to node before that */
WordRecPtr temp,          /* pointer to node being inserted */
WordRecPtr* firstWord)    /* pointer to first node in list */
                          /* NOTE: this is a pointer to a */
```

```
                          /* pointer ! */
{
   if (insertAt == *firstWord)
   {                                  /* insert at front */
      temp->next = *firstWord;
      *firstWord = temp;
   }
   else
   {                                  /* insert at middle or end */
      temp->next = insertAt;
      predecessor->next = temp;
   }

}     /* end of function InsertBefore */

/* ================================================================ */

void PrintAllWords
( WordRec* firstWord )             /* pointer to first node in list */

{  /* function    PrintAllWords */

   WordRec* current;
   current  =  firstWord;          /* initialize traversal pointer */

   while ( current !=  NULL ) /* keep visiting nodes until hit NULL */
   {
      /* print the contents of current node */
      printf("%s\n%s\n\n",
      current -> word, current -> definition);

      /* advance pointer to scan next node */
      current  =  current -> next;
   }

}  /* end of function PrintAllWords */

/* ================================================================ */
/* END OF PROGRAM */
/* ================================================================ */
```

Program 7.4

If we run this program using the data file described above, we will see output something like the following:

```
aftyson@diablo:~/testc>a.out
clamourous
marked by confused din or outcry

fast
characterized by quick motion

pointer
```

```
one that points

rat
any of numerous rodents
```

Programs using dynamic data structures are especially vulnerable to serious runtime errors. In the next chapter, we will discuss some good options for error handling in C.

Chapter 8: Error Handling

In C, as in C++, we have many options for handling errors which occur during a program run. We are not talking here about typical flow-of-control changes such as setting up a while loop to ask a user for input of a correct value after he/she has entered something invalid, or creating an if-else statement to handle a correct case in one branch and an error case in another. Here we are talking about more advanced tools.

C does not provide the "try-throw-catch" syntax of exception handling which C++ does, however we have equivalent ability to handle problems at runtime using the constructs that C provides instead.

First we will look at the simplest techniques for aborting a program run.

8.1 Aborting a Run: return, exit and assert

We have three basic options to exit a program immediately and completely upon an error condition:

- use a return from within the main function
- use a call to the exit function from anywhere in the program
- invoke the assert facility from anywhere in the program

We will look at each of these in turn.

First, note that the header file stdlib.h contains two important and often used constants, EXIT_FAILURE and EXIT_SUCCESS.

For example, if we write

```
return (EXIT_SUCCESS);
```

in the main function, we terminate the program and return a success code, normally 0, to the operating system, implying that everything went ok in the program run.

If we use

```
return (EXIT_FAILURE);
```

in main, we terminate the program and return a failure code indicating that something has gone wrong. This constant is often -1.

8.1.1 return

Let's look at a simple C program which opens a file and echoprints the contents to the screen. We'll assume that we have a text data file named *data.txt* for input.

```
#include <stdio.h>
#include <stdlib.h>   /* need for EXIT constants */

int main ()
{
   char ch;
   FILE *infile;

   /* open the file */
   infile = fopen ("data.txt", "r");

   /* echoprint all of its contents until EOF */
   while (fscanf (infile, "%c", &ch) != EOF)
      printf("%c", ch);

   fclose (infile);

   return (EXIT_SUCCESS);   /* alternative to return (0) */
}
```

Program 8.1

This program assumes everything is going to go fine, and does no error handling. What if the file does not open? We could do the following to terminate the program immediately:

```
#include <stdio.h>
#include <stdlib.h>      /* need for EXIT constants */

int main ()
{
   char ch;
   FILE *infile;

   infile = fopen ("data.txt", "r");

   /* if file is not open, use a return to terminate the run */
   if (!infile)
   {
      printf ("FATAL ERROR: file will not open!\n");
      return (EXIT_FAILURE);
   }

   while (fscanf (infile, "%c", &ch) != EOF)
      printf("%c", ch);
```

```
   fclose (infile);

   return (EXIT_SUCCESS);   /* alternative to return (0) */
}
```

Program 8.2

The main function now has two return statements. The first terminates the program if the file does not open. The second terminates the program after a successful run.

Note: regarding style, some programming environments have objections to multiple returns or the use of exit. Be sure to check your local style conventions to determine what are and are not acceptable and desirable techniques in a particular location.

8.1.2 exit

To abort a program run from anywhere other than main, we cannot use return, but we can use the exit function. The exit function is declared in stdlib.h.

Here is a new version of our program, terminating the run upon an error from within a function other than main:

```
#include <stdio.h>
#include <stdlib.h>      /* need for EXIT constants and exit function */
FILE *openFile();

int main ()
{
   char ch;
   FILE *infile;

   /* call another function to try to open the file */
   infile = openFile();
   while (fscanf (infile, "%c", &ch) != EOF)
      printf("%c", ch);

   fclose (infile);

   return (EXIT_SUCCESS);   /* alternative to return (0) */
}

FILE *openFile ()
{
   FILE *tempFile;
   tempFile = fopen ("data.txt", "r");

   /* if file is not open, abort the run using exit call */
   if (!tempFile)
   {
      printf ("FATAL ERROR: file will not open!\n");
      exit (EXIT_FAILURE);   /* abort entire run if file not open */
   }
```

```
   return (tempFile);   /* only return normally if file opens */
}
```

Program 8.3

This program behaves like our last example, but illustrates how to abort the run from within a function other than main.

8.1.3 assert

The assert facility is a very useful debugging tool as well as an error-handling routine. In most cases, it is only invoked while a programmer is in the act of testing and debugging a program; it is normally NOT invoked when a program is in production mode or sold to users. This is because it produces error messages which are only meant for programmers to see and to learn from.

The assert facility can be viewed as a function declared in the header file assert.h with the following prototype:

void assert (int <expression>);

Most compilers normally define assert as a macro, however that is not important for our discussion. When we invoke assert, the program run will continue normally if the expression is true. If the expression passed to assert is false, the program will immediately terminate with an error message. For example, let's change program 8.1 to the following, which uses assert to terminate the run if an error occurs:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>   /* need for assert facility */

int main ()
{
   char ch;
   FILE *infile;

   infile = fopen ("data.txt", "r");

   /* program will terminate on next line if file pointer is NULL */
   assert (infile);

   while (fscanf (infile, "%c", &ch) != EOF)
      printf("%c", ch);

   fclose (infile);

   return (EXIT_SUCCESS);   /* alternative to return (0) */
}
```

Program 8.4

If we run this program, and the file does not open, we will see an error message like this:

```
aftyson@diablo:~/testc>a.out
a.out: prog.c:13: main: Assertion `infile' failed.
Abort (core dumped)
```

This tells us that the program aborted at line 13 due to the assertion being false.

Professional programmers in practice use assert calls liberally while testing and debugging their code. Then, before the program is sold or put into production use, we can "turn off" the assert calls, so that they do not execute, VERY simply, by doing the following:

```
#include <stdio.h>
#include <stdlib.h>
#define NDEBUG        /* to pre-processor: turn off ALL asserts */
#include <assert.h>   /* need for assert facility */

int main ()
etc...
```

If we place this #define NDEBUG in our program 8.4 before the #include <assert.h>, all assert calls will be ignored.

8.2 signal

You have probably seen an error window pop up at some point displaying the word "exception," for example, an "addressing exception" during a program run. An exception is an unusual event, often a significant error, which occurs at runtime. The system may generate what is called a *signal* when an exception occurs. In a C program, we can detect these signals and execute an *exception handler*, which may be a standard function or a function we write ourselves.

Standard signals are represented by integer codes declared in the header file *signal.h*. The macros are as follows:

Macro	***What it Means***
SIGABRT	abnormal termination
SIGFPE	erroneous arithmetic operation; e.g. divide by zero
SIGILL	invalid instruction
SIGINT	interrupt; e.g. user typing certain keystrokes
SIGSEGV	illegal memory access
SIGTERM	program termination signal

Table 8.1

Additional non-standard signals may be defined for specific platforms.

C provides the library function *signal* to catch a generated signal. The first argument to signal is one of the macros from the preceding table. The second argument is the address of a macro to handle the exception.

In signal.h, two standard exception handlers are declared. These are:

Macro for Handler	***What it Does***
SIG_DFL	terminate the program run
SIG_IGN	ignore the signal and continue the run

Table 8.2

First, let's look at an example program in which the user is performing some task (we don't much care what the task is for the current discussion), and if the user types the CONTROL-C key sequence, we want the program to simply ignore it and keep going.

```
#include <stdio.h>
#include <signal.h>

int main ()
{
   int i;

   /* set up to ignore interrupts from the user */
   /*    signal is SIGINT */
   /*    handler is standard macro SIG_IGN */
   signal (SIGINT, SIG_IGN);

   /* ask user for a number, then read it and print it */
   /* ignoring interrupts if any occur */
   printf ("Enter an integer value: ");
   scanf ("%d", &i);
   printf ("The value entered was: %d\n", i);
   printf("Execution terminated.\n");

   return 0;
}
```

Program 8.5

When this program is run, and CONTROL-C typed repeatedly prior to typing an integer 5, the following output is produced with the code as given:

```
aftyson@diablo:~/testc>a.out
Enter an integer value: 5
The value entered was: 5
Execution terminated.
```

If the call to signal with SIG_IGN is removed from the above program, the program run instead terminates abruptly if the user types CONTROL-C.

In our second program example using signal, we will look at a programmer-defined handler function. Perhaps we would like to change the program above so that it does terminate on a user-interrupt, but it also prints out a message to the user telling them what has happened. We could do the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void myHandler (int sig);

int main ()
{
   int i;

   /* set up to catch interrupts from the user */
   /*    signal is SIGINT */
   /*    handler function is myHandler */
   signal (SIGINT, myHandler);

   /* ask user for a number and read */
   /* if an interrupt occurs, myHandler is invoked */
   printf ("Enter an integer value: ");
   scanf ("%d", &i);
   printf ("The value entered was: %d\n", i);
   printf("Execution terminated normally.\n");
   return (EXIT_SUCCESS);
}
```

```
void myHandler (int sig)  /* SIG_INT is automatically passed in */
{
   /* print a message for the user */
   printf
      ("\nYou typed an interrupt code and the program will now stop.\n");
   printf
      ("Any data entered during this run will be lost.\n");
   printf("Execution terminated abnormally due to interrupt.\n");

   /* terminate the program right now with exit call */
   exit (EXIT_FAILURE);
}
```

Program 8.5

Now if we run this program and *don't* type an interrupt, we see this:

```
aftyson@diablo:~/testc>a.out
Enter an integer value: 5
The value entered was: 5
Execution terminated normally.
```

If we run the program and *do* type an interrupt, we would see this instead:

```
aftyson@diablo:~/testc>a.out
Enter an integer value:
You typed an interrupt code and the program will now stop.
Any data entered during this run will be lost.
Execution terminated abnormally due to interrupt.
```

In the second run, the program terminates as soon as the interrupt signal is caught.

8.3 setjmp and longjmp

The setjmp and longjmp facilities in C behave somewhat like the catch and throw mechanisms of exception handling in C++. However, they are not as sophisticated in their functionality.

In normal program execution, a called function always returns to its caller. The setjmp and longjmp functions allow us to alter this standard flow of control by creating a *non-local jump*. Note that setjmp and longjmp should be used judiciously - that is *cautiously* - because they alter this standard flow. Typically this mechanism is only used in very long and complex programs when more standard error handling may not suffice.

For this discussion it is a good idea to think about our machine. The CPU registers, among other things, store the current state of a program, or its *environment*, at any particular moment. This includes information such as the stack pointer (SP) and program counter (PC). If you know what these terms mean, that's good; if not, that's fine too, just think about the *environment* as being the *current state of the program* at a particular time.

The setjmp function is declared in <setjmp.h> and has the following prototype:

```
int setjmp (jmp_buf environment);
```

When we call setjmp as in

```
jmp_buf env;     /* must declare an environment variable */
setjmp (env);    /* call to setjmp */
```

the contents of the CPU registers at the time setjmp is called are saved into the data structure env. You can think of the jmp_buf data type as a data structure which is capable of storing a snapshot of current register contents.

The call to setjmp above has a return value of zero. This indicates that setjmp was called directly, without any call to longjmp having occurred.

The longjmp function is also declared in <setjmp.h> and has this prototype:

```
void longjmp (jmp_buf environment, int returnValue);
```

We can call longjmp only after we have already called setjmp. A call to longjmp will cause setjmp to restore the previously saved environment back into the registers (including the PC and SP), and then setjmp will return using the int value passed into longjmp as its return value (which should not be zero). For example, if after our setjmp call above we call longjmp with

```
longjmp (env, -1);
```

the environment saved at the time of the setjmp call will be restored, and the flow of control will go to just after the setjmp call. Note that it is as if longjmp itself does not return, only setjmp returns.

Here is a very simple program example:

```
#include <setjmp.h>
#include <stdlib.h>

jmp_buf env;  /* env is often declared globally */
              /* any call to longjmp must have env in scope */

int main ()
{
   int returnVal;

   returnVal = setjmp(env);
   printf ("here at first output line\n");
   if (returnVal == 0)
      printf ("setjmp return value is zero\n");
   else
   {
      printf("setjmp return value is nonzero\n");
      exit (EXIT_FAILURE);
```

```
   }

   longjmp(env, -1);
   printf("here at second output line\n");

   return (EXIT_SUCCESS);
}
```

Program 8.6

If we run this program, we will see the following output:

```
aftyson@diablo:~/testc>a.out
here at first output line
setjmp return value is zero
here at first output line
setjmp return value is nonzero
```

A longjmp call can occur deep into a program's function call chain, and often does. Also, we can call longjmp from within a signal handler function that we have written, jump back into our main program, and thus be able to catch the same signal multiple times.

BE CAREFUL: what if you call setjmp from within a function, and then that function exits? The saved environment no longer exists! If you try to longjmp after such a call to setjmp, a serious error will occur, often a segmentation fault.

Let's look at one more program using the setjmp/longjmp mechanism. Keep in mind that our goal here is not to look at long complex systems programs in this short textbook, but to look at short programs which will illustrate critical concepts quickly.

```
#include <stdio.h>
#include <setjmp.h>

double calc1 (double, double);
double calc2 (double, double);

jmp_buf env;

int main ()
{
   /* read in two double values and try some simple calculations */
   /* we are assuming user inputs two valid doubles */
   double a, b, result;
   printf ("Enter values for a and b -> ");
   scanf ("%lf%lf", &a, &b);

   switch (setjmp(env))
   {
      case 0: /* first call to setjmp returns zero, all ok */
              /* call functions to do tasks, return to switch on error */
              result = calc1 (a, b);
```

```
               /* this result only prints if no longjmps occur */
               printf ("result is: %.3f\n", result);
               break;

      case 1: /* if get here, calc1 produced an error */
               printf ("cannot obtain result, calc1 failure\n");
               break;

      case 2: /* if get here, calc2 produced an error */
               printf ("cannot obtain result, calc2 failure\n");
               break;

      default: /* just in case of any surprises */
               printf ("unknown error occurred\n");

   }   /* end of switch */

   printf ("Execution terminated.\n");

   return (0);
}

double calc1 (double c, double d)
{
   /* this routine fails if c or d is negative */
   if (c < 0.0 || d < 0.0)
      longjmp (env, 1);
   else
      return (calc2 (c + d, c * d));
}

double calc2 (double e, double f)
{
   /* this routine fails if f is zero */
   if (f == 0.0)
      longjmp (env, 2);
   else
      return (e/f);
}
```

Program 8.7

The following output is produced when no errors are caught:

```
aftyson@diablo:~/testc>a.out
Enter values for a and b -> 5.1 6.23
result is: 0.357
Execution terminated.
```

This output is produced when calc1 calls longjmp on catching an error:

```
aftyson@diablo:~/testc>a.out
Enter values for a and b -> -1.234  78.9
cannot obtain result, calc1 failure
Execution terminated.
```

This output is produced when calc1 finds no problems, but calc2 calls longjmp on catching an error:

```
aftyson@diablo:~/testc>a.out
Enter values for a and b -> 5.678 0.0
cannot obtain result, calc2 failure
Execution terminated.
```

As you can see, calling setjmp inside a switch while returning appropriate values with longjmp provides a very useful mechanism for error handling.

Appendices

Appendix A: C Precedence Rules

Operator	Associativity	Precedence
( ) [ ] -> ·	Left to right	***Highest***
++ -- (type) + - ! & * sizeof ~	Right to left (unary)	
* / %	Left to right	
+ -	Left to right (binary)	
<< >>	Left to right	
< <= > >=	Left to right	
== !=	Left to right	
&	Left to right (bitwise)	
^	Left to right (bitwise)	
|	Left to right (bitwise)	
&&	Left to right	
||	Left to right	
?:	Right to left	
= += -= *= /= %= &= ^= |= <<= >>=	Right to left	
,	Left to right	***Lowest***

Appendix B: Prototypes and Header Files for Commonly Used C Functions

In this section we will briefly list many of the widely-used functions available in various standard header files.

#include <assert.h>

function name	***prototype***	***usage***
assert	void assert (int expression); {usually implemented as a macro}	continue program run if expression true, abort if false

#include <ctype.h>

function name	***prototype***	***usage***
isalnum	int isalnum (in ch);	returns true (nonzero integer) if ch is an alphanumeric character otherwise returns 0 (false)
isalpha	int isalpha (in ch);	returns true (nonzero integer) if ch is an alphabetic character otherwise returns 0 (false)
iscntrl	int iscntrl (in ch);	returns true (nonzero integer) if ch is a control character otherwise returns 0 (false)
isdigit	int isdigit (in ch);	returns true (nonzero integer) if ch is a decimal digit otherwise returns 0 (false)
isgraph	int isgraph (in ch);	returns true (nonzero integer) if ch is a nonblank printable character otherwise returns 0 (false)
islower	int islower (in ch);	returns true (nonzero integer) if ch is a lowercase alphabetic character otherwise returns 0 (false)
isprint	int isprint (in ch);	returns true (nonzero integer) if ch is a printable character otherwise returns 0 (false)
ispunct	int ispunct (in ch);	returns true (nonzero integer) if ch is a punctuation character otherwise returns 0 (false)
isspace	int isspace (in ch);	returns true (nonzero integer) if ch is a whitespace character otherwise returns 0 (false)

isupper	int isupper (in ch);	returns true (nonzero integer) if ch is an uppercase alphabetic character otherwise returns 0 (false)
isxdigit	int isxdigit (in ch);	returns true (nonzero integer) if ch is a hexidecimal digit otherwise returns 0 (false)
tolower	int tolower (int ch);	converts ch from uppercase to lowercase returning the converted value; if ch is not lowercase alpha, simply returns original ch
toupper	int toupper (int ch);	converts ch from lowercase to uppercase returning the converted value; if ch is not uppercase alpha, simply returns original ch

#include <math.h>

function name	***prototype***	***usage***
acos	double acos (double realnum);	returns arccosine in radians of the argument
asin	double asin (double realnum);	returns arcsine in radians of the argument
atan	double atan (double realnum);	returns arctangent in radians of the argument
ceil	double ceil (double realnum);	returns the least integer which is greater than or equal to realnum
cos	double cos (double realnum);	returns the cosine of realnum (realnum must be provided in radians)
cosh	double cosh (double realnum);	returns the hyperbolic cosine of realnum
exp	double exp (double realnum);	returns e to the power realnum
fabs	double fabs (double realnum);	returns the absolute value of realnum
floor	double floor (double realnum);	returns the greatest integer which is less than or equal to realnum
log	double log (double realnum);	returns the natural logarithm of realnum
log10	double log10 (double realnum);	returns the base 10 logarithm of realnum
pow	double pow	returns realnum to the power

	(double realnum1, double realnum2);	realnum2
sin	double sin (double realnum);	returns the sine of realnum (realnum must be provided in radians)
sinh	double sinh (double realnum);	returns the hyperbolic sine of realnum
sqrt	double sqrt (double realnum);	returns the square root of realnum
tan	double tan (double realnum);	returns the tangent of realnum (realnum must be provided in radians)
tanh	double tanh (double realnum);	returns the hyperbolic tangent of realnum

#include <stdio.h>

function name	***prototype***	***usage***
clearerr	void clearerr (FILE* filePointer);	clears EOF and error flags in file indicated
fclose	int fclose (FILE* filePointer);	closes the file indicated and flushes all buffers; returns 0 on success otherwise EOF
feof	int feof (FILE* filePointer);	returns nonzero integer (true) if EOF was encountered on last input operation, 0 (false) otherwise
fgetc	int fgetc (FILE* filePointer);	returns next character from file, or EOF if EOF is encountered or if error occurs
fgets	char* fgets (char* stringAddress, int maxChars, FILE* filePointer);	reads next line from file, storing characters at stringAddress, returns address of string or NULL if fails
fopen	FILE* fopen (const char* string, const char* mode);	opens file with name given by string using mode specified; returns pointer to file or NULL if fails
fprintf	int fprintf (FILE* filePointer, const char* string, ...);	writes formatted output to file see scanf
fputc	int fputc (int ch, FILE* filePointer);	writes ch to file, returns character written or EOF if fails
fputs	int fputs (const char* string, FILE* filePointer);	writes string to file specified and does not add newline to output; returns non-negative int if

		successful and EOF if not
fread	size_t fread (void* buffer, size_t itemSize, size_t numItems, FILE* filePointer);	reads up to numItems of size itemSize from the specified file, storing them at buffer; returns number of items successfully read
fscanf	int fscanf (FILE* filePointer, const char* string, ...);	reads formatted input from specified file (see scanf)
fseek	int fseek (FILE* filePointer, long offset, int base);	moves file pointer in specified file
ftell	long ftell (FILE* filePointer);	returns position of file pointer
fwrite	size_t fwrite (const void* buffer, size_t itemSize, size_t numItems, FILE* filePointer);	writes numItems of size itemSize to the file specified; returns number of items successfully written
getc	int getc (FILE* filePointer);	returns next character from file, or EOF if EOF is encountered or if error occurs; usually implemented as a macro
getchar	int getchar (void);	returns next character from standard input stream, or EOF if EOF is encountered or if error occurs
gets	char* gets (char* string);	reads next line from standard input up to next newline, stores this in s and does not includes newline in s; returns s or NULL if fails; adds NULL terminator
printf	int printf (const char* string, ...);	writes formatted output to standard output stream
putc	int putc (int ch, FILE* filePointer);	writes ch to file, returns character written or EOF if fails; usually implemented as a macro
putchar	int putchar (int ch);	writes ch to standard input stream; returns ch or EOF if an error occurs
puts	int puts (const char* string);	writes string to standard output stream followed by a newline; returns nonnegative value if successful or EOF if not
rewind	void rewind (FILE* filePointer);	places file pointer at beginning of specified file
scanf	int scanf	reads formatted input from

	(const char* string, ...);	standard input stream; returns number of items successfully read and stored or EOF if no items are read successfully
sprintf	int sprintf (char* buffer, const char* string, ...);	writes formatted output to memory at buffer (see printf)
sscanf	int sscanf (const char* s1, const char* s2, ...);	reads formatted input from s1 (see scanf)
ungetc	int ungetc (int ch, FILE* filePointer);	writes ch to the buffer of the file specified; returns ch if successful and EOF if not

#include <setjmp.h>

function name	***prototype***	***usage***
longjmp	void longjmp (jmp_buf environment, int returnValue);	restores the environment set by the last call to setjmp; makes it appear as though setjmp returned with value given by returnValue of longjmp; setjmp must be called before longjmp may be called
setjmp	int setjmp (jmp_buf environment);	sets destination for a nonlocal jump; returns zero or value passed via longjmp if the return is from longjmp

#include <signal.h>

function name	***prototype***	***usage***
signal	void (*signal (int sig, void (*handler) (int))) (int);	catches a signal and invokes a function to handle it

#include <stdlib.h>

function name	***prototype***	***usage***
abs	int abs (int number);	returns absolute value of number
atof	double atof (const char* s);	converts a floating point number stored in string s to a double
atoi	int atoi (const char* s);	converts an integer number stored in string s to an int

atol	long atol (const char* s);	converts an integer number stored in string s to a long
bsearch	void* bsearch (const void* target, void* arrayAddress, size_t arraySize, size_t elementSize, int (*cmp) (const void*, const void*));	searches for target in array at address arrayAddress of size arraySize, with elements of size elementSize (in bytes), using cmp to perform comparison of elements; returns pointer to target if it is found (using binary search)
calloc	void* calloc (size_t numElements, size_t elementSize);	allocates numElements * elementSize bytes of consecutive storage and sets all bits to zero; returns address of first byte allocated or NULL if fails
exit	void exit (int status);	terminates program run and returns status to calling process; flushes all buffers and closes all files
free	void free (void* storage);	frees area previously allocated by malloc or calloc
labs	long labs (long number);	returns absolute value of number
lsearch	void* lsearch (const void* target, void* arrayAddress, size_t arraySize, size_t elementSize, int (*cmp) (const void*, const void*));	searches for target in array at address arrayAddress of size arraySize, with elements of size elementSize (in bytes), using cmp to perform comparison of elements; returns pointer to target if it is found (using linear search)
malloc	void* malloc (size_t numBytes);	allocates numBytes bytes of consecutive storage; returns address of first byte allocated or NULL if fails
qsort	void qsort (void* arrayAddress, size_t arraySize, size_t elementSize, int (*cmp)(const void*, const void*));	sorts array of size arraySize starting at address arrayAddress with elementSize in bytes; uses cmp function to compare two elements (using quicksort)
rand	int rand (void);	returns a pseudorandom integer in the range 0 through RAND_MAX
system	int system (const char* s);	executes the system command given by s

#include <string.h>

function name	***prototype***	***usage***
memchr	void* memchr (const void* buffer, int ch, size_t nBytes);	returns address of first occurrence of ch in the first nBytes at address buffer, or NULL if ch does not occur
memcmp	int memcmp (const void* buffer1, const void* buffer2, size_t nBytes);	compares first nBytes of buffer1 with buffer2; returns a negative value if contents of buffer1 are less than contents of buffer2, zero if equal and positive if greater
memcpy	void* memcpy (void* buffer1, const void* buffer2, size_t nBytes);	copies first nBytes of contents at buffer2 into block1 and returns address block1; may not work if contents overlap
memmove	void* memmove (void* buffer1, const void* buffer2, size_t nBytes);	copies first nBytes of contents at buffer2 into block1 and returns address block1; ok if contents overlap
strcat	char* strcat (char* s1, const char* s2);	copies s2 to the end of s1 and returns s1
strchr	char* strchr (const char* s, int ch);	returns address of first occurence of ch in s1, or NULL if it does not occur
strcmp	int strcmp (const char* s1, const char* s2);	compares characters in s1 and s2; returns negative value if contents of s1 are lexicographically less than contents of s2, zero if equal and positive value if greater
strcpy	char* strcpy (char* s1, const char* s2);	copies contents of s2 into s1 and returns s1
strcspn	size_t strcspn (const char* s1, const char* s2);	returns number of consecutive characters in s1 that do not occur in s2
strlen	int strlen (const char* s);	returns length of s (number of characters up to and not including first NULL)
strncat	char* strncat (char* s1, const char* s2, size_t n);	copies up to n characters from s2 to the end of s1 and returns s1
strncmp	int strncmp (const char* s1, const char* s2, size_t n);	compares first n characters of s1 and s2 (see strcmp)
strncpy	char* strncpy (char* s1, const char* s2, size_t n);	copies n characters from s2 to s1 (counting NULL)
strpbrk	char* strpbrk (const char * s1, const char* s2);	returns address of first character in s1 that occurs anywhere in s2, or NULL if there is none

strrchr	char * strrchr (const char* s, int ch);	returns address of last occurrence of ch in s, or NULL if there is no occurrence
strspn	size_t strspn (const char* s1, const char* s2);	returns the number of consecutive characters in s1 that occur in s2
strstr	char *strstr (const char* s1, const char* s2);	returns address of first occurence of s1 in s2, or NULL if there is none

#include <time.h>

function name	***prototype***	***usage***
difftime	double difftime (time_t endtime, time_t begintime);	returns the difference in seconds endtime - begintime
time	time_t time (time_t* memAddress);	returns the time in seconds since midnight, January 1, 1970; if memAddress is not NULL, stores the value at that location

References and Bibliography

Harbison, Samuel P. III and Steele, Guy L. Jr., C: A Reference Manual, Fifth Edition, Prentice-Hall Inc., 2002

Johnsonbaugh, Richard and Kalin, Martin, Applications Programming in ANSI C, Third Edition, Prentice-Hall Inc., 1996

Johnsonbaugh, Richard and Kalin, Martin, C for Scientists and Engineers, Prentice-Hall Inc., 1997

Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, 2nd Edition, Prentice-Hall Inc., 1988

Kernighan, Brian W. and Pike, Rob, The Practice of Programming, Addison Wesley Longman Inc., 1999

Sutter, Herb and Alexandrescu, Andrei, C++ Coding Standards: 101 Rules, Guidelines and Best Practices, Pearson Education Inc., 2005

van der Linden, Peter, Expert C Programming: Deep C Secrets, Prentice-Hall Inc., 1994