A Little C Primer/Print version

A Little C Primer

The current, editable version of this book is available in Wikibooks, the open-content textbooks collection, at
https://en.wikibooks.org/wiki/A_Little_C_Primer

Contents

An Introductory C Program

Here's a simple C program to calculate the volume of a sphere:

```c
/* sphere.c */

#include <stdio.h>                  /* Include header file for printf. */

#define PI 3.141592654f             /* Define a constant. */

static float sphere(float);         /* Function prototype. sphere is given internal linkage with the
keyword
                                        'static' since it isn't used outside of this translation unit. */

int main(void)                      /* Main function. */
{
    float volume;               /* Define a float variable. */
    float radius = 3;           /* Define and initialize variable. */

    volume = sphere(radius);    /* Call sphere and copy the return value into volume. */
    printf( "Volume: %f\n", volume );  /* Print the result. */
    return 0;
}

float sphere(float rad)             /* Definition of a volume calculating function. */
{
    float result;               /* "result" is defined with an automatic storage duration and block
scope. */

    result = rad * rad * rad;
    result = 4 * PI * result / 3;
    return result;              /* "result" is returned to the function which called sphere. */
}
```

The first feature that can be observed in this program is that comments are enclosed by "/*" and "*/". Comments can go almost anywhere, since the compiler ignores them. Another obvious feature is that individual statements in the program end with a ";". Either forgetting a ";" or adding one where it isn't needed is a common C programming bug. Lines of code are grouped into blocks by using curly brackets ("{ }").

C is case-sensitive. All C keywords are in lower-case. Program variable and function names in whatever case desired, but by convention they should be in lower case. "Constants", to be discussed a bit later, are upper case by convention.

Not all the lines in a C program are executable statements. Some of the statements shown above are "preprocessor directives".

C compilation is a multi-pass process. To create a program, a C compiler system performs the following steps:

- It runs the source file text through a "C preprocessor". All this does is perform various text manipulations on the source file, such as "macro expansion", "constant expansion", "file inclusion", and "conditional compilation", which are also explained later. The output of the preprocessor is a second-level source file for actual compilation. You can think of the C preprocessor as a sort of specialized automatic "text editor".
- Next, it runs the second-level source file through the compiler proper, which actually converts the source code statements into their binary equivalents. That is, it creates an "object file" from

the source file.

- The object file still cannot be executed, however. If it uses C library functions, such as "printf()" in the example above, the binary code for the library functions has to be merged, or "linked", with the program object file. Furthermore, some addressing information needs to be linked to the object file so it can actually be loaded and run on the target system.

These linking tasks are performed by a "linker", which takes one or more object files and links them to binary library files to create an "executable" file that can actually be run.

C has a large number of libraries and library functions. C by itself has few statements, so much of its functionality is implemented as library calls.

Commands intended for the C preprocessor, instead of the C compiler itself, start with a "#" and are known as "preprocessor directives" or "metacommands". The example program above has two such metacommands:

```
#include <stdio.h>

#define PI 3.14
```

The first statement, "#include <stdio.h>", simply merges the contents of the file "stdio.h" into the current program file before compilation. The "stdio.h" file contains declarations required for use of the standard-I/O library, which provides the "printf()" function.

Incidentally, the "stdio.h" file, or "header file", only contains declarations. The actual code for the library is contained in separate library files that are added at link time. Custom header files loaded with custom declarations can be created if needed, and then included in a program as follows:

```
#include "mydefs.h"
```

Angle brackets ("< >") are not used with custom header files. They only are used to define the default header files provided standard with a C compiler (and which, on Unix/Linux systems, are found in directory "/usr/include").

A C program is built up of one or more functions. The program above contains two user-defined functions, "main()" and "sphere()", as well as the "printf()" library function.

The "main()" function is mandatory when writing a self-contained program. It defines the function that is automatically executed when the program is run. All other functions will be directly or indirectly called by "main()".

A C function is called simply by specifying its name, with any arguments enclosed in following parentheses, with commas separating the arguments. In the program above, the "printf()" function is called as follows:

```
printf( "Volume: %f\n", volume );
```

This invocation provides two arguments. The first -- "Volume: %f\n"—supplies text and some formatting information. The second -- "volume"—supplies a numeric value.

A function may or may not return a value. The "sphere()" function does, and so is invoked as follows:

```
    volume = sphere( radius );
```

A function uses the "return" keyword to return a value. In the case of "sphere", it returns the volume of the sphere with the statement:

```
    return result;
```

All variables in a C program must be "declared" by specifying their name and type. The example program declares two variables for the "main" routine:

```
    float volume;
    float radius = 3;
```

—and one in the "sphere" routine:

```
    float result;
```

The declarations of "volume", "radius", and "result" specify a floating-point variable. The declaration allows variables to be initialized when declared if need be, in this case declaring "radius" and assigning it a value of "3".

All three of these declarations define variables block scope. Variables with block scope exist only within the blocks that declare them. Variables of the same name could be declared in different blocks without interference. It is also possible to declare variables with file scope that can be shared by all functions by declaring them outside of all blocks within a translation unit. By default, all identifiers with file scope are given external linkage. It is only with the keyword "static" that an identifier is given internal linkage. Declarations like this:

```
 extern int a;
```

identify "a" as an int whose storage is defined elsewhere; preferably in another translation unit.

```
    /* global.c */

    #include <stdio.h>

    void somefunc( void );
    int globalvar;

    void main()
    {
      extern int globalvar;
      globalvar = 42;
      somefunc();
      printf( "%d\n", globalvar );
    }

    void somefunc( void )
    {
      extern int globalvar;
```

```
    printf( "%d\n", globalvar );
    globalvar = 13;
}
```

Besides the variable declarations, the example programs above also feature a function declaration, or "function prototype", that allows the C compiler to perform checks on calls to the function in the program and flag an error if they are not correct:

```
float sphere(float);
```

The function prototypes declare the type of value the function returns (the type will be "void" if it does not return a value), and the arguments that are to be provided with the function.

Finally, the "printf()" library function provides text output capabilities for the program. The "printf()" function can be used to print a simple message as follows:

```
printf( "Hello, world!" );
```

—displays the text:

```
Hello, world!
```

Remember that "printf()" doesn't automatically add a "newline" to allow following "printf()"s to print on the next display line. For example:

```
printf( "Twas bryllig " );
printf( "and the slithy toves" );
```

-- prints the text:

```
Twas bryllig and the slithy toves
```

A newline character ("\n") must be added to force a newline. For example:

```
printf( "Hello,\nworld!" );
```

—gives:

```
Hello,
world!
```

These examples only print a predefined text constant. It is possible to include "format codes" in the string and then follow the string with one or more variables to print the values they contain:

```
printf( " Result = %f\n", result );
```

This would print something like:

```
Result = 0.5
```

The "%f" is the format code that tells "printf" to print a floating-point number. For another example:

```
printf( "%d times %d = %d\n", a, b, a * b );
```

—would print something like:

```
4 times 10 = 40
```

The "%d" prints an integer quantity. Math or string expressions and functions can be included in the argument list.

To simply print a string of text, there is a simpler function, "puts()", that displays the specified text and automatically appends a newline:

```
puts( "Hello, world!" );
```

Just for fun, let's take a look at what our example program would be like in the pre-ANSI versions of C:

```
/* oldspher.c */

#include <stdio.h>
#define PI 3.141592654

float sphere();        /* Parameters not defined in function prototype. */

main()
{
  float volume;
  int radius = 3;

  volume = sphere( radius );
  printf( "Volume: %f\n", volume );
}

float sphere( rad )
int rad;               /* Parameter type not specified in function header. */
{
  float result;

  result = rad * rad * rad;
  result = 4 * PI * result / 3;
  return result;
}
```

The following sections elaborate on the principles outlined in this section.

C Functions in Detail

As noted previously, any C program must have a "main()" function to contain the code executed by default when the program is run.

A program can contain as many functions as needed. All functions are "visible" to all other functions. For example:

```
/* fdomain.c */

#include <stdio.h>;

void func1( void );
void func2( void );

int main()
{
  puts( "MAIN" );
  func1();
  func2();
}

void func1( void )
{
  puts( "FUNC1" );
  func2();
}

void func2( void )
{
  puts( "FUNC2" );
  func1();
}
```

In this example, "main()" can call "func1()" and "func2()"; "func1()" can call "func2()"; and "func2()" can call "func1()". In principle, even

"main()" could be called by other functions, but it's hard to figure out why anyone would want to do so. Although "main()" is the first function in the listing above, there's no particular requirement that it be so, but by convention it always should be.

Functions can call themselves recursively. For example, "func1()" can call "func1()" indefinitely, or at least until a stack overflow occurs. You cannot declare functions inside other functions.

Functions are defined as follows:

```
float sphere( int rad )
{
   ...
}
```

They begin with a function header that starts with a return value type declaration ("float" in this case), then the function name ("sphere"), and finally the arguments required ("int rad").

ANSI C dictates that function prototypes be provided to allow the compiler to perform better checking on function calls:

```
float sphere( int rad );
```

For an example, consider a simple program that "fires" a weapon (simply by printing "BANG!"):

```
/* bango.c */

#include <stdio.h>

void fire( void );

void main()
{
  printf( "Firing!\n" );
  fire();
  printf( "Fired!\n" );
}

void fire( void )
{
  printf( "BANG!\n" );
}
```

This prints:

```
Firing!
BANG!
Fired!
```

Since "fire()" does not return a value and does not accept any arguments, both the return value and the argument are declared as "void"; "fire()" also does not use a "return" statement and simply returns automatically when completed.

Let's modify this example to allow "fire()" to accept an argument that defines a number of shots. This gives the program:

```
/* fire.c */

#include <stdio.h>

void fire( int n );

void main()
{
  printf( "Firing!\n" );
  fire( 5 );
  printf( "Fired!\n" );
}

void fire( int n )
{
  int i;
  for ( i = 1; i <= n ; ++i )
  {
    printf( "BANG!\n" );
  }
}
```

This prints:

```
Firing!
BANG!
BANG!
BANG!
BANG!
BANG!
Fired!
```

This program passes a single parameter, an integer, to the "fire()" function. The function uses a "for" loop to execute a "BANG!" the specified number of times—more on "for" later.

If a function requires multiple arguments, they can be separated by commas:

```
printf( "%d times %d = %d\n", a, b, a * b );
```

The word "parameter" is sometimes used in place of "argument". There is actually a fine distinction between these two terms: the calling routine specifies "arguments" to the called function, while the called function receives the "parameters" from the calling routine.

When a parameter is listed in the function header, it becomes a local variable to that function. It is initialized to the value provided as an argument by the calling routine. If a variable is used as an argument, there is no need for it to have the same name as the parameter specified in the function header.

For example:

```
fire( shots );
...
void fire( int n )
...
```

The integer variable passed to "fire()" has the name "shots", but "fire()" accepts the value of "shots" in a local variable named "n". The argument and the parameter could also have the same name, but even then they would remain distinct variables.

Parameters are, of course, matched with arguments in the order in which they are sent:

```
/* pmmatch.c */

#include <stdio.h>

void showme( int a, int b );

void main()
{
  int x = 1, y = 100;
  showme( x, y );
}

void showme( int a, int b )
{
  printf( "a=%d  b=%d\n", a, b );
}
```

This prints:

```
a=1  b=100
```

This program can be modified to show that the arguments are not affected by any operations the function performs on the parameters, as follows:

```c
/* noside.c */

#include <stdio.h>

void showmore( int a, int b );

void main()
{
   int x = 1, y = 100;
   showmore( x, y );
   printf( "x=%d  y=%d\n", x, y );
}

void showmore( int a, int b )
{
   printf( "a=%d  b=%d\n", a, b );
   a = 42;
   b = 666;
   printf( "a=%d  b=%d\n", a, b );
}
```

This prints:

```
a=1  b=100
a=42  b=666
x=1  y=100
```

Arrays can be sent to functions as if they were any other type of variable:

```c
/* fnarray.c */

#include <stdio.h>
#define SIZE 10

void testfunc( int a[] );

void main()
{
  int ctr, a[SIZE];
  for( ctr = 0; ctr < SIZE; ++ctr )
  {
    a[ctr] = ctr * ctr;
  }
  testfunc( a );
}

void testfunc( int a[] )
{
  int n;
  for( n = 0; n < SIZE; ++ n )
  {
    printf( "%d\n", a[n] );
  }
}
```

It is possible to define functions with a variable number of parameters. In fact, "printf()" is such a function. This is a somewhat advanced issue and we won't worry about it further in this document.

The normal way to get a value out of a function is simply to provide it as a return value. This neatly encapsulates the function and isolates it from the calling routine. In the example in the first section, the function "sphere()" returned a "float" value with the statement:

```
    return( result );
```

The calling routine accepted the return value as follows:

```
    volume = sphere( radius );
```

The return value can be used directly as a parameter to other functions:

```
    printf( "Volume: %f\n", sphere( radius ) );
```

The return value does not have to be used; "printf()", for example, returns the number of characters it prints, but few programs bother to check.

A function can contain more than one "return" statement::

```
    if( error == 0 )
    {
      return( 0 );
    }
    else
    {
      return( 1 );
    }
```

A "return" can be placed anywhere in a function. It doesn't have to return a value; without a value, "return" simply causes an exit from the function. However, this does imply that the data type of the function must be declared as "void":

```
    void ftest( int somevar )
    {
       ...
       if( error == 0 )
       {
         return();
       }
       ...
    }
```

If there's no "return" in a function, the function returns after it executes its last statement. Again, this means the function type must be declared "void".

The "return" statement can only return a single value, but this value can be a "pointer" to an array or a data structure. Pointers are a complicated subject and will be discussed in detail later.

C Control Constructs

C contains a number of looping constructs, such as the "while" loop:

```
/* while.c */

#include <stdio.h>

void main()
{
  int test = 10;
  while( test > 0 )
  {
    printf( "test = %d\n", test );
    test = test - 2;
  }
}
```

If "test" starts with an initial value less than or equal to 0 the "while" loop will not execute even once. There is a variant, "do", that will always execute at least once:

```
/* do.c */

#include <stdio.h>

void main()
{
  int test = 10;
  do
  {
    printf( "test = %d\n", test );
    test = test - 2;
  }
  while( test > 0 );
}
```

The most common looping construct is the "for" loop, which creates a loop much like the "while" loop but in a more compact form:

```
/* for.c */

#include <stdio.h>

void main()
{
  int test;
  for( test = 10; test > 0; test = test - 2 )
  {
    printf( "test = %d\n", test );
  }
}
```

Notice that with all these loops, the initial loop statement does *not* end with a ";". If a ";" was placed at the end of the "for" statement above, the "for" statement would execute to completion, but not run any of the statements in the body of the loop.

The "for" loop has the syntax:

```
for( <initialization>; <operating test>; <modifying expression> )
```

All the elements in parentheses are optional. A "for" loop could be run indefinitely with:

```
for( ; ; )
{
  ...
}
```

—although using an indefinite "while" is cleaner:

```
while( 1 )
{
  ...
}
```

It is possible to use multiple expressions in either the initialization or the modifying expression with the "," operator:

```
/* formax.c */

#include <stdio.h>

void main()
{
  int a, b;
  for ( a = 256, b = 1; b < 512 ; a = a / 2, b = b * 2 )
  {
    printf( "a = %d  b = %d\n", a, b );
  }
}
```

The conditional tests available to C are as follows:

```
a == b    equals
a != b    not equals
a < b     less than
a > b     greater than
a <= b    less than or equals
a >= b    greater than or equals
```

The fact that "==" is used to perform the "equals" test, while "=" is used as the assignment operator, often causes confusion and is a common bug in C programming:

```
a == b    Is "a" equal to "b"?
a = b     Assign value of "b" to "a".
```

C also contains decision-making statements, such as "if":

```
/* if.c */

#include <stdio.h>
#define MISSILE 1

void fire( int weapon );
```

```
void main()
{
  fire( MISSILE );
}

void fire( int weapon )
{
  if( weapon == MISSILE )
  {
    printf( "Fired missile!\n" );
  }
  if( weapon != MISSILE )
  {
    printf( "Unknown weapon!\n");
  }
}
```

This example can be more easily implemented using an "else" clause:

```
/* ifelse.c */

void fire( int weapon )
{
  if( weapon == MISSILE )
  {
    printf( "Fired missile!\n" );
  }
  else
  {
    printf( "Unknown weapon!\n");
  }
}
```

Since there is only one statement in each clause the curly brackets aren't really necessary. This example would work just as well:

```
void fire( int weapon )
{
  if( weapon == MISSILE )
    printf( "Fired missile!\n" );
  else
    printf( "Unknown weapon!\n" );
}
```

However, the brackets make the structure more obvious and prevent errors if you add statements to the conditional clauses. The compiler doesn't care one way or another, it generates the same code.

There is no "elseif" keyword, but "if" statements can be nested:

```
/* nestif.c */

#include <stdio.h>
#define MISSILE 1
#define LASER 2

void fire( int weapon );

void main()
{
  fire( LASER );
```

```
    }

    void fire( int weapon )
    {
      if( weapon == MISSILE )
      {
        printf( "Fired missile!\n" );
      }
      else
      {
        if( weapon == LASER )
        {
          printf( "Fired laser!\n" );
        }
        else
        {
          printf( "Unknown weapon!\n");
        }
      }
    }
```

This is somewhat clumsy. The "switch" statement does a cleaner job:

```
    /* switch.c */

    void fire( int weapon )
    {
      switch( weapon )
      {
      case MISSILE:
        printf( "Fired missile!\n" );
        break;
      case LASER:
        printf( "Fired laser!\n" );
        break;
      default:
        printf( "Unknown weapon!\n");
        break;
      }
    }
```

The "switch" statement tests the value of a single variable; unlike the "if" statement, it can't test multiple variables. The optional "default" clause is used to handle conditions not covered by the other cases.

Each clause ends in a "break", which causes execution to break out of the "switch". Leaving out a "break" can be another subtle error in a C program, since if it isn't there, execution flows right through to the next clause. However, this can be used to advantage. Suppose in our example the routine can also be asked to fire a ROCKET, which is the same as a MISSILE:

```
    void fire( int weapon )
    {
      switch( weapon )
      {
      case ROCKET:
      case MISSILE:
        printf( "Fired missile!\n" );
        break;
      case LASER:
        printf( "Fired laser!\n" );
        break;
      default:
        printf( "Unknown weapon!\n");
```

```
      break;
    }
  }
```

The "break" statement is not specific to "switch" statements. It can be used to break out of other control structures, though good program design tends to avoid such improvisations:

```
/* break.c */

#include <stdio.h>

void main()
{
  int n;
  for( n = 0; n < 10; n = n + 1 )
  {
    if( n == 5 )
    {
      break;  /* Punch out of loop at value 5. */
    }
    else
    {
      printf( "%d\n", n );
    }
  }
}
```

If the "for" loop were nested inside a "while" loop, a "break" out of the

"for" loop would still leave you stuck in the "while" loop. The "break" keyword only applies to the control construct that executes it.

There is also a "continue" statement that skips to the end of the loop body and continues with the next iteration of the loop. For example:

```
/* continue.c */

#include <stdio.h>

void main()
{
  int n;
  for( n = 0; n < 10; n = n + 1 )
  {
    if( n == 5 )
    {
      continue;
    }
    else
    {
      printf( "%d\n", n );
    }
  }
}
```

Finally, there is a "goto" statement:

```
goto punchout;
...
punchout:
```

—that jumps to an arbitrary tag within a function, but the use of this

statement is generally discouraged and it is rarely seen in practice.

While these are the lot of C's true control structures, there is also a special "conditional operator" that performs a simple conditional assignment of the form:

```
    if( a == 5)
    {
      b = -10;
    }
    else
    {
      b = 255;
    }
```

—using a much tidier, if more cryptic, format:

```
    b = ( a == 5 ) ? -10 : 255 ;
```

the ?: construct is called a ternary operator—or the ternary operator—as it takes 3 arguments.

C Variables, Declarations and Constants

C supports a flexible set of variable types and structures, as well as common arithmetic and math functions along with a few interesting operators that are unique to C. This chapter explains them in detail, and ends with a short discussion of preprocessor commands.

C includes the following fundamental data types:

Type	Use	Size (bits)	Range
char	Character	8	-128 to 127
unsigned char	Character	8	0 to 255
short	Integer	16	-32,768 to 32,767
unsigned short	Integer	16	0 to 65,535
int	Integer	32	–2,147,483,648 to 2,147,483,647
unsigned int	Integer	32	0 to 4,294,967,295
long	Integer	32	-2,147,483,648 to 2,147,483,647
unsigned long	Integer	32	0 to 4,294,967,295
float	Real	32	1.2×10^{-38} to 3.4×10^{38}
double	Real	64	2.2×10^{-308} to 1.8×10^{308}
long double	Real	128	3.4×10^{-4932} to 1.2×10^{4932}

These are representative values. The definitions tend to vary between implementations. For example, in some systems an "int" is 16 bits, and a "long double" could be 64 bits. The only thing that is guaranteed is the precedence:

```
short <= int <= long
float <= double <= long double
```

One peculiarity of C that can lead to maddening problems is that while there is an "unsigned char" data type, for some reason many functions that deal with individual characters require variables to be declared "int" or "unsigned int".

Declarations are of the form:

```
int myval, tmp1, tmp2;
unsigned int marker1 = 1, marker2 = 10;
float magnitude, phase;
```

Variable names can be at least 31 characters long, though modern compilers will always support longer names. Variables names can be made up of letters, digits, and the "_" (underscore) character; the first character must be a letter. While it is possible to use uppercase letters in variable names, conventional C usage reserves uppercase for constant names. A leading "_" is also legal, but is generally reserved for marking internal library names.

C allows several variables to be declared in the same statement, with commas separating the declarations. The variables can be initialized when declared. Constant values for declarations can be declared in various formats:

```
128        decimal int
256u       decimal unsigned int
512l       decimal long int
0xAF       hex int
0173       octal int
0.243      float
0.1732f    float
15.75E2    float
'a'        character
"giday"    string
```

There are a number of special characters defined in C:

```
'\a'     alarm (beep) character
'\p'     backspace
'\f'     formfeed
'\n'     newline
'\r'     carriage return
'\t'     horizontal tab
'\v'     vertical tab
'\\'     backslash
'\?'     question mark
'\''     single quote
'\"'     double quote
'\0NN'   character code in octal
'\xNN'   character code in hex
'\0'     null character
```

"Symbolic constants" can be specified using the "define" C preprocessor declaration:

```
#define PI 3.141592654
```

There is also a "const" declaration that defines a read-only variable, such as a memory location in ROM:

```
const int a;
```

- Arrays can be declared and initialized:

```
int myarray[10];
unsigned int list[5] = { 10, 15, 12, 19, 23 };
float rdata[128], grid[5][5];
```

All C arrays have a starting index of 0, so "list" has the indexes 0 through 4. Elements in "rdata" would be accessed as follows:

```
for( i = 0; i <= 127; i = i + 1 )
{
   printf ( "%f\n", rdata[i] );
}
```

C does not perform rigorous bounds checking on array access. It is easy to overrun the bounds of the array, and the only symptom will be that the program acts very strangely.

- Of particular importance are arrays of characters, which are used to store

strings:

```
    char s[128];
    strcpy( s, "This is a test!");
```

The string "This is a test!" is used to initialize "s" through the "strcpy()" function, discussed in a later chapter. The stored string will contain a terminating "null" character (the character with ASCII code 0, represented by '\0'). The null is used by C functions that manipulate strings to determine where the end of the string is, and it is important to remember the null is there.

The curious reader may wonder why the "strcpy()" function is needed to initialize the string. It might seem to be easier to do:

```
    char s[128] = "This is a test!";
```

In fact, this is an absurd operation, but to explain why, the concept of "pointers" must be introduced.

- C programs can define pointers that contain the address of a variable or

an array. For example, a pointer could be defined named:

```
    int *ptr;
```

-- that gives the address of a variable, rather than the variable itself. A value could then be put into that location with the statement:

```
    *ptr = 345;
```

In an inverse fashion, the address of a variable can be obtained with "&":

```
    int tmp;
    somefunc( &tmp );
```

To sum up:

- A pointer is declared in the form: "*myptr".
- If "myvar" is a variable, then "&myvar" is a pointer to that variable.
- If "myptr" is a pointer, then "*myptr" gives the variable data for that pointer.

Pointers are useful because they allow a function to return a value through a parameter variable. Otherwise, the function will simply get the data the variable contains and have no access to the

variable itself.

One peculiar aspect of C is that the name of an array actually specifies a pointer to the first element in the array. For example, given the string declaration:

```
char s[256];
```

-- then the function call:

```
somefunc( s )
```

-- will actually pass the address of the character array to the function, and the function will be able to modify it. However:

```
s[12]
```

-- gives the value in the array value with index 12. Remember that this is the 13th element, since indexes *always* start at 0.

There are more peculiarities to strings in C. Another interesting point is that a string literal actually evaluates to a pointer to the string it defines. This means that in the following operation:

```
char *p;
p = "Life, the Universe, & Everything!";
```

-- "p" ends up being a pointer to the memory in which the C compiler stored the string literal, and "p[0]" would evaluate to "L". In a similar sense, the following operation:

```
char ch;
ch = "Life, the Universe, & Everything!"[0];
```

-- would put the character "L" into the variable "ch".

This is very well and good, but why care? The reason to care is because this explains why the operation:

```
char s[128] = "This is a test!";
```

-- is absurd. This statement tells the C compiler to reserve 128 bytes of memory and set a pointer named "s" to point to them. Then it reserves another block of memory to store "This is a test!" and points "s" to that. This means the block of 128 bytes of memory that were originally allocated is now sitting empty and unusable, and the program is actually accessing the memory that stores "This is a test!".

This will seem to work for a while, until the program tries to store more bytes into that block than can fit into the 16 bytes reserved for "This is a test!". Since C is poor about bounds checking, this

may cause all kinds of trouble.

This is why "strcpy()" is usually necessary. It isn't needed for a string that won't be modified or will not be used to store more data than it is initialized to, and under such circumstances the following statements will work fine:

```
char *p;
p = "Life, the Universe, & Everything!                    ";
```

These issues become particularly tricky when passing strings as parameters to functions. The following example shows how to get around the pitfalls:

```
/* strparm.c */

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

char *strtest( char *a, char *b );

int main ()
{
  char a[256],
       b[256],
       c[256];

  strcpy( a, "STRING A: ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" );
  strcpy( b, "STRING B: ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" );
  strcpy( c, "STRING C: ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" );

  printf( "Initial values of strings:\n" );
  printf( "\n" );
  printf( "   a = %s\n", a );
  printf( "   b = %s\n", b );
  printf( "   c = %s\n", c );
  printf( "\n" );

  strcpy( c, strtest( a, b ));

  printf( "Final values of strings:\n" );
  printf( "\n" );
  printf( "   a = %s\n", a );
  printf( "   b = %s\n", b );
  printf( "   c = %s\n", c );
  printf( "\n" );

  return 0;
}

char *strtest( char *x, char *y )
{
  printf( "Values passed to function:\n" );
  printf( "\n" );
  printf( "   x = %s\n", x );
  printf( "   y = %s\n", y );
  printf( "\n" );

  strcpy( y, "NEWSTRING B: abcdefghijklmnopqrstuvwxyz0123456789" );
  return "NEWSTRING C: abcdefghijklmnopqrstuvwxyz0123456789";
}
```

- It is possible to define "structures" in C, which are collections of different data elements:

```
/* struct.c */

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

struct person                           /* Define structure type. */
{
   char name[50];
   int age;
   float wage;
};

void display( struct person );

int main()
{
  struct person m;                      /* Declare an instance of it. */
  strcpy( m.name, "Coyote, Wile E." );  /* Initialize it. */
  m.age = 41;
  m.wage = 25.50f;
  display( m );
  return 0;
}

void display( struct person p )
{
  printf( "Name: %s\n", p.name );
  printf( "Age:  %d\n", p.age );
  printf( "Wage: %4.2f\n", p.wage );
}
```

This program has a few interesting features:

- The structure has to be defined by a "struct" declaration before it can declare any structures themselves. In this case we define a struct of type "person".
- Instances of the struct ("m") are then declared as by defining the structure type ("struct person").
- Elements of the structure are accessed with a "dot" notation ("m.name", "m.age", and "m.wage").

A structure can be copied to another structure with a single assignment statement, as long as the structures are of the same type:

```
struct person m, n;
...
m = n;
```

It is also possible to declare arrays of structures:

```
struct person group[10];
...
strcpy( group[5].name, "McQuack, Launchpad" );
```

-- or even embed structures inside structure declarations:

```
struct trip_rec
```

```
    {
       struct person traveler;
       char dest[50];
       int date[3];
       int duration;
       float cost;
    }
```

-- in which case the nested structure would be accessed as follows:

```
    struct trip_rec t1;
    ...
    strcpy( t1.traveler.name, "Martian, Marvin" );
```

The name of a structure defines a variable, not an address. If the name of a structure is passed to a function, the function works only on its local copy of the structure. To return values, an address must be specified:

```
    setstruct( &mystruct );
```

There is a shorthand way to get at the elements of a structure with the pointer to the structure instead of the structure itself. If "sptr" is a pointer to a structure of type "person", its fields can be accessed as follows:

```
    strcpy( sptr->name, "Leghorn, Foghorn" );
    sptr->age = 50;
    sptr->wage = 12.98;
```

- C contains a concept similar to a structure known as a "union". A union is declared in much the same way as a structure. For example:

```
    union usample
    {
      char ch;
      int x;
    }
```

The difference is that the union can store either of these values, but not both at the same time. A "char" value or an "int" value can be stored in an instance of the union defined above, but it's not possible to store both at the same time. Only enough memory is allocated for the union to store the value of the biggest declared item in it, and that same memory is used to store data for all the declared items. Unions are not often used and will not be discussed further.

- The following example program shows a practical use of structures. It tests a set of functions that perform operations on three-dimensional vectors:

```
    vadd():     Add two vectors.
    vsub():     Subtract two vectors.
    vdot():     Vector dot product.
    vcross():   Vector cross product.
    vnorm():    Norm (magnitude) of vector.
    vangle():   Angle between two vectors.
```

```
    vprint():   Print out vector.
```

The program follows:

```
    /* vector.c */

    #include <stdio.h>
    #include <math.h>

    #define PI 3.141592654

    struct v
    {
       double i, j, k;
    };

    void vadd( struct v, struct v, struct v* );
    void vprint( struct v );
    void vsub( struct v, struct v, struct v* );
    double vnorm( struct v );
    double vdot( struct v, struct v );
    double vangle( struct v, struct v );
    void vcross( struct v, struct v, struct v* );

    int main()
    {
      struct v v1 = { 1, 2, 3 }, v2 = { 30, 50, 100 }, v3;
      double a;

      printf( "Sample Vector 1: " );
      vprint( v1 );
      printf( "Sample Vector 2: " );
      vprint( v2 );

      vadd( v1, v2, &v3 );
      printf( "Vector Add:      " );
      vprint( v3 );

      vsub( v1, v2, &v3 );
      printf( "Vector Subtract: " );
      vprint( v3 );

      vcross( v1, v2, &v3 );
      printf( "Cross Product:   " );
      vprint( v3 );

      printf( "\n" );
      printf( "Vector 1 Norm:  %f\n", vnorm( v1 ) );
      printf( "Vector 2 Norm:  %f\n", vnorm( v2 ) );
      printf( "Dot Product:    %f\n", vdot( v1, v2 ) );
      a = 180 * vangle( v1, v2) / PI ;
      printf( "Angle:          %3f degrees.\n", a );

      return 0;
    }

    void vadd( struct v a, struct v b, struct v *c )  /* Add vectors. */
    {
      c->i = a.i + b.i;
      c->j = a.j + b.j;
      c->k = a.k + b.k;
    }

    double vangle( struct v a, struct v b )  /* Get angle between vectors. */
    {
      double c;
      c = vdot( a, b ) / ( vnorm( a ) * vnorm( b ) );
      return acos( c );
    }
```

```
void vcross( struct v a, struct v b, struct v *c )  /* Cross product. */
{
  c->i = a.j * b.k - a.k * b.j;
  c->j = a.k * b.i - a.i * b.k;
  c->k = a.i * b.j - a.j * b.i;
}

double vdot( struct v a, struct v b ) /* Dot product of vectors. */
{
  return a.i * b.i + a.j * b.j + a.k * b.k;
}

double vnorm ( struct v a )  /* Norm of vectors. */
{
  return sqrt( a.i * a.i + a.j * a.j + a.k * a.k );
}

void vprint ( struct v a )  /* Print vector. */
{
  printf( " I = %6.2f   J = %6.2f   K = %6.2f\n", a.i, a.j, a.k );
}

void vsub ( struct v a, struct v b, struct v *c )  /* Subtract vectors. */
{
  c->i = a.i - b.i;
  c->j = a.j - b.j;
  c->k = a.k - b.k;
}
```

- The concept of local and global variables should be clear by now. It is

also possible to declare a local variable as "static", meaning it retains its value from one invocation of the function to the next. For example:

```
#include <stdio.h>

void testfunc( void );

int main()
{
  int ctr;
  for( ctr = 1; ctr <= 8; ++ctr )
  {
    testfunc();
  }
  return 0;
}

void testfunc( void )
{
  static int v;
  printf( "%d\n", 2*v );
  ++v;
}
```

This prints:

```
0
2
4
6
8
10
```

```
    12
    14
```

-- since the initial value of a integer is 0 by default. It is *not* a good idea to rely on a default value!

- There are two other variable declarations that should be recognized, though

there's little reason to use them: "register", which declares that a variable should be assigned to a CPU register, and "volatile", which tells the compiler that the contents of the variable may change spontaneously.

There is more and less than meets the eye to these declarations. The "register" declaration is discretionary: the variable will be loaded into a CPU register if it can, and if not it will be loaded into memory as normal. Since a good optimizing compiler will try to make the best use of CPU registers anyway, this is not in general all that useful a thing to do.

The "volatile" declaration appears ridiculous at first sight, something like one of those "joke" computer commands like "halt and catch fire". Actually, it's used to describe a hardware register that can change independently of program operation, such as the register for a realtime clock.

- C is fairly flexible in conversions between data types. In many cases, the

type conversion will happen transparently. If a "char" is converted to a "short" data type, or an "int" is converted to a "long" data type, for example, the converted data type can easily accommodate any value in the original data type.

Converting from a bigger to a smaller data type can lead to odd errors. The same is true for conversions between signed and unsigned data types. For this reason, type conversions should be handled carefully, and it is usually preferable to do them explicitly, using a "cast" operation. For example, a cast conversion can be performed from an "int" value to a "float" value as follows:

```
    int a;
    float b;
    ...
    b = (float)a;
```

- It is possible to define custom "enumerated" types in C. For example:

```
   enum day
   {
      saturday, sunday, monday, tuesday, wednesday, thursday, friday
   };
```

-- defines enumerated type "day" to consist of the values of the days of the week. In practice, the values are merely text constants associated to a set of consecutive integer values. By default, the set begins at 0 and counts up, so here "saturday" has the value 0, "sunday" has the value 1, and so on. Any set of number assignments can be specified if desired:

```
    enum temps
    {
      zero = 0, freeze = 32, boil = 220
    };
```

```
```

Obviously much the same could be done with sets of "#define" directives, but this is a much cleaner solution. Once the type is defined, for example, variables of that type can be declared as follows:

```
    enum day today = wednesday;
```

The variable "today" will act as an "int" variable and will allow the operations valid for "int" variables. Once more, remember that C doesn't do much in the way of bounds checking, and it is not wise to rely on the C compiler to give warnings.

- Finally, a "typedef" declaration can be used to define custom data types:

```
    typedef ch[128] str;
```

Then variables of this type could be declared as follows:

```
    str name;
```

C Operators

C supports the following arithmetic operators:

```
c = a * b     multiplication
c = a / b     division
c = a % b     mod (remainder division)
c = a + b     addition
c = a - b     subtraction
```

It also supports the following useful (if cryptic) "increment" and "decrement" operators:

```
++a           increment
--a           decrement
```

These operators can also be expressed as "a++" and "a--". If the only thing that's needed is an increment or decrement, the distinction between the two forms is irrelevant. However, if a variable is being incremented or decremented as a component of some expression, then "++a" means "increment the variable first, then get its value", while "a++" means "get the value of the variable first, then increment it". Failing to remember this distinction can lead to subtle programming errors.

Finally, C supports a set of bitwise operations:

```
a = ~a         bit complement
a = b << c     shift b left by number of bits stored in c
a = b >> c     shift b right by number of bits stored in c
a = b & c      b AND c
a = b ^ c      b XOR c
a = b | c      b OR c
```

C allows math operations to be performed in a shortcut fashion:

```
operation        shortcut
_________________________

a = a * b         a *= b
a = a / b         a /= b
a = a % b         a %= b
a = a + b         a += b
a = a - b         a -= b
a = a << b        a <<= b
a = a >> b        a >>= b
a = a & b         a &= b
a = a ^ b         a ^= b
a = a | b         a |= b
_________________________
```

The C relational operations were discussed in the previous chapter and are repeated here for completeness:

```
a == b    equals
a != b    not equals
a < b     less than
```

```
    a > b     greater than
    a <= b    less than or equals
    a >= b    greater than or equals
```

These are actually math operations that yield 1 if true and 0 if false. The following operation actually makes syntactic sense:

```
    a = b * ( b < 2 ) + 10 * ( b >= 2 );
```

This would give "a" the value "b" if "b" is less than 2, and the value "10" otherwise. This is cute, but not recommended. It's cryptic; may make porting to other languages more difficult; and in this case, it can be done much more effectively with the conditional operator discussed in the previous chapter:

```
    a = ( b < 2 ) ? b : 10;
```

This conditional operator is also known as the "triadic" operator.

There are similar logical operators:

```
    !     logical NOT
    &&    logical AND
    ||    logical OR
```

Remember that these are *logical* operations, not bitwise operations—don't confuse "&&" and "||" with "&" and "|". The distinction is that while the bitwise operators perform the operations on a bit-by-bit basis, the logical operations simply assess the values of their operands to be either 0 or 1 (any nonzero operand value evaluates to 1 in such comparisons) and return either a 0 or a 1:

```
    if(( A == 5 ) && ( B == 10 ))
    {
       ...
    }
```

Finally, there is a "sizeof" operand that returns the size of a particular operand in bytes:

```
    int tvar;
    ...
    printf ( "Size = %d\n", sizeof( int ) );
```

This comes in handy for some mass storage operations. The "sizeof()" function can be given a data type name or the name of a variable, and the variable can be an array, in which case "sizeof()" gives the size of the entire array.

The precedence of these operators in math functions—that is, which ones are evaluated before others—are defined as follows, reading from the highest precedence to the lowest:

```
    ()     []     ->     .
```

```
!      ~      ++     --     (cast)      &      sizeof   - (minus prefix)
     /      %
+      -
<<     >>
<      <=     >      >=
==     !=
&

^
|
&&
||
?:
=      +=     -=     *=     /=     %=     >>=     <<=     &=
^=     |=
,
```

Of course, parentheses can be used to control precedence. If in doubt about the order of evaluation of an expression, add more parentheses. They won't cause any trouble and might save some pain.

Advanced math operations are available as library functions. These will be discussed in a later chapter.

C Preprocessor Directives

We've already seen the "#include" and "#define" preprocessor directives. The C preprocessor supports several other directives as well. All such directives start with a "#" to allow them to be distinguished from C language commands.

As explained in the first chapter, the "#include" directive allows the contents of other files to be included in C source code:

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

Notice that the standard header file "stdio.h" is specified in angle brackets. This tells the C preprocessor that the file can be found in the standard directories designated by the C compiler for header files. To include a file from a nonstandard directory, use double quotes:

```
#include "\home\mydefs.h"
```

Include files can be nested. They can call other include files.

Also as explained in the first chapter, the "#define" directive can be used to specify symbols to be substituted for specific strings of text:

```
#define PI 3.141592654
...
a = PI * b;
```

In this case, the preprocessor does a simple text substitution on PI throughout the source listing. The C compiler proper not only does not know what PI is, it never even sees it.

The "#define" directive can be used to create function-like macros that allow parameter substitution. For example:

```
#define ABS(value)  ( (value) >=0 ? (value) : -(value) )
```

This macro could then be used in an expression as follows:

```
printf( "Absolute value of x = %d\n", ABS(x) );
```

Beware that such function-like macros don't behave exactly like true functions. For example, suppose "x++" is as an argument for the macro above:

```
val = ABS(x++);
```

This would result in "x" being incremented twice because "x++" is substituted in the expression twice:

```
val = ( (x++) >=0 ? (x++) : -(x++) )
```

Along with the "#define" directive, there is also an "#undef" directive that undefines a constant that has been previously defined:

```
#undef PI
```

Another feature supported by the C preprocessor is conditional compilation, using the following directives:

```
#if
#else
#elif
#endif
```

These directives can test the values of defined constants to define which blocks of code are passed on to the C compiler proper:

```
#if WIN == 1
  #include "WIN.H"
#elif MAC == 1
  #include "MAC.H"
#else
  #include "LINUX.H"

#endif
```

These directives can be nested if needed. The "#if" and "#elif" can also test to see if a constant has been defined at all, using the "defined" operator:

```
#if defined( DEBUG )
   printf( "Debug mode!\n");
#endif
```

—or test to see if a constant has *not* been defined:

```
#if !defined( DEBUG )
   printf( "Not debug mode!\n");
#endif
```

Finally, there is a "#pragma" directive, which by definition is a catch-all used to implement machine-unique commands that are not part of the C language. Pragmas vary from compiler to compiler, since they are by definition nonstandard.

C Console IO

This chapter covers console (keyboard/display) and file I/O. You've already seen one console-I/O function, "printf()", and there are several others. C has two separate approaches toward file I/O, one based on library functions that is similar to console I/O, and a second that uses "system calls". These topics are discussed in detail below.

Console I/O in general means communications with the computer's keyboard and display. However, in most modern operating systems the keyboard and display are simply the default input and output devices, and user can easily redirect input from, say, a file or other program and redirect output to, say, a serial I/O port:

```
type infile > myprog > com
```

The program itself, "myprog", doesn't know the difference. The program uses console I/O to simply read its "standard input (stdin)"—which might be the keyboard, a file dump, or the output of some other program—and print to its "standard output (stdout)"—which might be the display or printer or another program or a file. The program itself neither knows nor cares.

Console I/O requires the declaration:

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

Useful functions include:

```
printf()      Print a formatted string to stdout.
scanf()       Read formatted data from stdin.
putchar()     Print a single character to stdout.
getchar()     Read a single character from stdin.
puts()        Print a string to stdout.
gets()        Read a line from stdin.
```

Windows-based compilers also have an alternative library of console I/O functions. These functions require the declaration:

```
#include <conio.h>
```

The three most useful Windows console I/O functions are:

```
getch()    Get a character from the keyboard (no need to press Enter).
getche()   Get a character from the keyboard and echo it.
kbhit()    Check to see if a key has been pressed.
```

The "printf()" function, as explained previously, prints a string that may include formatted data:

```
printf( "This is a test!\n" );
```

—which can include the contents of variables:

```
printf( "Value1:  %d   Value2:  %f\n", intval, floatval );
```

The available format codes are:

```
%d    decimal integer
%ld   long decimal integer
%c    character
%s    string
%e    floating-point number in exponential notation
%f    floating-point number in decimal notation
%g    use %e and %f, whichever is shorter
%u    unsigned decimal integer
%o    unsigned octal integer
%x    unsigned hex integer
```

Using the wrong format code for a particular data type can lead to bizarre output. Further control can be obtained with modifier codes; for example, a numeric prefix can be included to specify the minimum field width:

```
%10d
```

This specifies a minimum field width of ten characters. If the field width is too small, a wider field will be used. Adding a minus sign:

```
%-10d
```

—causes the text to be left-justified.

A numeric precision can also be specified:

```
%6.3f
```

This specifies three digits of precision in a field six characters wide.

A string precision can be specified as well, to indicate the maximum number of characters to be printed. For example:

```
/* prtint.c */

#include <stdio.h>

int main(void)
{
  printf( "<%d>\n", 336 );
  printf( "<%2d>\n", 336 );
  printf( "<%10d>\n", 336 );
  printf( "<%-10d>\n", 336 );
  return 0;
}
```

This prints:

```
<336>
<336>
<       336>
<336       >
```

Similarly:

```
/* prtfloat.c */

#include <stdio.h>

int main(void)
{
  printf( "<%f>\n", 1234.56 );
  printf( "<%e>\n", 1234.56 );
  printf( "<%4.2f>\n", 1234.56 );
  printf( "<%3.1f>\n", 1234.56 );
  printf( "<%10.3f>\n", 1234.56 );
  printf( "<%10.3e>\n", 1234.56 );
  return 0;
}
```

—prints:

```
<1234.560000>
<1.234560e+03>
<1234.56>
<1234.6>
<  1234.560>
< 1.234e+03>
```

And finally:

```
/* prtstr.c */

#include <stdio.h>

int main(void)
{
  printf( "<%2s>\n", "Barney must die!" );
  printf( "<%22s>\n", "Barney must die!" );
  printf( "<%22.5s>\n", "Barney must die!" );
  printf( "<%-22.5s>\n", "Barney must die!" );
  return 0;
}
```

—prints:

```
<Barney must die!>
<      Barney must die!>
<                 Barne>
<Barne                >
```

Just for convenience, the table of special characters listed in chapter 2 is repeated here. These characters can be embedded in "printf" strings:

```
    '\a'     alarm (beep) character
    '\p'     backspace
    '\b'     backspace
    '\f'     formfeed
    '\n'     newline
    '\r'     carriage return
    '\t'     horizontal tab
    '\v'     vertical tab
    '\\'     backslash
    '\?'     question mark
    '\''      single quote
    '\"'     double quote
    '%%'     percentage
    '\0NN'   character code in octal
    '\xNN'   character code in hex
    '\0'     null character
```

The "scanf()" function reads formatted data using a syntax similar to that of "printf", except that it requires pointers as parameters, since it has to return values. For example:

```
    /* cscanf.c */

    #include <stdio.h>

    int main(void)
    {
      int val;
      char name[256];

      printf( "Enter your age and name.\n" );
      scanf( "%d %s", &val, name );
      printf( "Your name is: %s -- and your age is: %d\n", name, val );
      return 0;
    }
```

There is no "&" in front of "name" since the name of a string is already a pointer. Input fields are separated by whitespace (space, tab, or newline), though a count, for example "%10d", can be included to define a specific field width. Formatting codes are the same as for "printf()", except:

- There is no "%g" format code.
- The "%f" and "%e" format codes work the same.
- There is a "%h" format code for reading short integers.

If characters are included in the format code, "scanf()" will read in the characters and discard them. For example, if the example above were modified as follows:

```
    scanf( "%d,%s", &val, name );
```

—then "scanf()" will assume that the two input values are comma-separated and swallow the comma when it is encountered.

If a format code is preceded with an asterisk, the data will be read and discarded. For example, if the example were changed to:

```
scanf( "%d%*c%s", &val, name );
```

—then if the two fields were separated by a ":", that character would be read in and discarded.

The "scanf()" function will return the value EOF (an "int"), defined in "stdio.h", when its input is terminated.

The "putchar()" and "getchar()" functions handle single character I/O. For example, the following program accepts characters from standard input one at a time:

```
/* inout.c */

#include <stdio.h>

int main (void)
{
  unsigned int ch;

  while ((ch = getchar()) != EOF)
  {
    putchar( ch );
  }
  return 0;
}
```

The "getchar" function returns an "int" and also terminates with an EOF. Notice the neat way C allows a program to get a value and then test it in the same expression, a particularly useful feature for handling loops.

One word of warning on single-character I/O: if a program is reading characters from the keyboard, most operating systems won't send the characters to the program until the user presses the "Enter" key, meaning it's not possible to perform single-character keyboard I/O this way.

The little program above is the essential core of a character-mode text "filter", a program that can perform some transformation between standard input and standard output. Such a filter can be used as an element to construct more sophisticated applications:

```
type file.txt > filter1 | filter2 > outfile.txt
```

The following filter capitalizes the first character in each word in the input. The program operates as a "state machine", using a variable that can be set to different values, or "states", to control its operating mode. It has two states: SEEK, in which it is looking for the first character, and REPLACE, in which it is looking for the end of a word.

In SEEK state, it scans through whitespace (space, tab, or newline), echoing characters. If it finds a printing character, it converts it to uppercase and goes to REPLACE state. In REPLACE state, it converts characters to lowercase until it hits whitespace, and then goes back to SEEK state.

The program uses the "tolower()" and "toupper()" functions to make case conversions. These two functions will be discussed in the next chapter.

```
/* caps.c */
```

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

#define SEEK 0
#define REPLACE 1

int main(void)
{
  int ch, state = SEEK;
  while(( ch = getchar() ) != EOF )
  {
    switch( state )
    {
    case REPLACE:
      switch( ch )
      {
      case ' ':
      case '\t':
      case '\n':   state = SEEK;
                   break;
      default:     ch = tolower( ch );
                   break;
      }
      break;
    case SEEK:
      switch( ch )
      {
      case ' ':
      case '\t':
      case '\n':   break;
      default:     ch = toupper( ch );
                   state = REPLACE;
                   break;
      }
    }
    putchar( ch );
  }
  return 0;
}
```

The "puts()" function is like a simplified version of "printf()" without format codes. It prints a string that is automatically terminated with a newline:

```
puts( "Hello world!" );
```

The "fgets()" function is particularly useful: it reads a line of text terminated by a newline. It is much less finicky about its inputs than "scanf()":

```
/* cgets.c */

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

#include <stdlib.h>

int main(void)
{
  char word[256],
       *guess = "blue\n";
  int i, n = 0;

  puts( "Guess a color (use lower case please):" );
  while( fgets(word, 256, stdin) != NULL )
  {
```

```
        if( strcmp( word, guess ) == 0 )
        {
           puts( "You win!" );
           break;
        }
        else
        {
           puts( "No, try again." );
        }
      }
      return 0;
    }
```

This program includes the "strcmp" function, which performs string comparisons and returns 0 on a match. This function is described in more detail in the next chapter.

These functions can be used to implement filters that operate on lines of text, instead of characters. A core program for such filters follows:

```
    /* lfilter.c */

    #include <stdio.h>

    int main (void)
    {
      char b[256];
      while (( fgets(b, 256, stdin) ) != NULL )
      {
        puts( b );
      }
      return 0;
    }
```

The "fgets()" function returns NULL, defined in "stdio.h", on input termination or error.

The Windows-based console-I/O functions "getch()" and "getche()" operate much as "getchar()" does, except that "getche()" echoes the character automatically.

The "kbhit()" function is very different in that it only indicates if a key has been pressed or not. It returns a nonzero value if a key has been pressed, and zero if it hasn't. This allows a program to poll the keyboard for input, instead of hanging on keyboard input and waiting for something to happen. As mentioned, these functions require the "conio.h" header file, not the "stdio.h" header file.

C File-IO Through Library Functions

The file-I/O library functions are much like the console-I/O functions. In fact, most of the console-I/O functions can be thought of as special cases of the file-I/O functions. The library functions include:

```
    fopen()      Create or open a file for reading or writing.
    fclose()     Close a file after reading or writing it.

    fseek()      Seek to a certain location in a file.
    rewind()     Rewind a file back to its beginning and leave it open.
    rename()     Rename a file.
    remove()     Delete a file.

    fprintf()    Formatted write.
    fscanf()     Formatted read.
    fwrite()     Unformatted write.
    fread()      Unformatted read.
    putc()       Write a single byte to a file.
    getc()       Read a single byte from a file.
    fputs()      Write a string to a file.
    fgets()      Read a string from a file.
```

All these library functions depend on definitions made in the "stdio.h" header file, and so require the declaration:

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

C documentation normally refers to these functions as performing "stream I/O", not "file I/O". The distinction is that they could just as well handle data being transferred through a modem as a file, and so the more general term "data stream" is used rather than "file". However, we'll stay with the "file" terminology in this document for the sake of simplicity.

The "fopen()" function opens and, if need be, creates a file. Its syntax is:

```
    <file pointer> = fopen( <filename>, <access mode> );
```

The "fopen()" function returns a "file pointer", declared as follows:

```
    FILE *<file pointer>;
```

The file pointer will be returned with the value NULL, defined in "stdio.h", if there is an error. The "access modes" are defined as follows:

```
    r     Open for reading.
    w     Open and wipe (or create) for writing.
    a     Append -- open (or create) to write to end of file.
    r+    Open a file for reading and writing.
    w+    Open and wipe (or create) for reading and writing.
    a+    Open a file for reading and appending.
```

The "filename" is simply a string of characters.

It is often useful to use the same statements to communicate either with files or with standard I/O. For this reason, the "stdio.h" header file includes predefined file pointers with the names "stdin" and "stdout". There's no need to do an "fopen()" on them—they can just be assigned to a file pointer:

```
    fpin = stdin;
    fpout = stdout;
```

—and any following file-I/O functions won't know the difference.

The "fclose()" function simply closes the file given by its file pointer parameter. It has the syntax:

```
    fclose( fp );
```

The "fseek()" function call allows the byte location in a file to be selected for reading or writing. It has the syntax:

```
    fseek( <file_pointer>, <offset>, <origin> );
```

The offset is a "long" and specifies the offset into the file, in bytes. The "origin" is an "int" and is one of three standard values, defined in "stdio.h":

```
    SEEK_SET    Start of file.
    SEEK_CUR    Current location.
    SEEK_END    End of file.
```

The "fseek()" function returns 0 on success and non-zero on failure.

The "rewind()", "rename()", and "remove()" functions are straightforward. The "rewind()" function resets an open file back to its beginning. It has the syntax:

```
    rewind( <file_pointer> );
```

The "rename()" function changes the name of a file:

```
    rename( <old_file_name_string>, <new_file_name_string> );
```

The "remove()" function deletes a file:

```
    remove( <file_name_string> )
```

The "fprintf()" function allows formatted ASCII data output to a file, and has the syntax:

```
fprintf( <file pointer>, <string>, <variable list> );
```

The "fprintf()" function is identical in syntax to "printf()", except for the addition of a file pointer parameter. For example, the "fprintf()" call in this little program:

```
/* fprpi.c */

#include <stdio.h>

void main()
{
  int n1 = 16;
  float n2 = 3.141592654f;
  FILE *fp;

  fp = fopen( "data", "w" );
  fprintf( fp, "  %d   %f", n1, n2 );
  fclose( fp );
}
```

—stores the following ASCII data:

```
 16   3.14159
```

The formatting codes are exactly the same as for "printf()":

```
%d    decimal integer
%ld   long decimal integer
%c    character
%s    string
%e    floating-point number in exponential notation
%f    floating-point number in decimal notation
%g    use %e and %f, whichever is shorter
%u    unsigned decimal integer
%o    unsigned octal integer
%x    unsigned hex integer
```

Field-width specifiers can be used as well. The "fprintf()" function returns the number of characters it dumps to the file, or a negative number if it terminates with an error.

The "fscanf()" function is to "fprintf()" what "scanf()" is to "printf()": it reads ASCII-formatted data into a list of variables. It has the syntax:

```
fscanf( <file pointer>, <string>, <variable list> );
```

However, the "string" contains only format codes, no text, and the "variable list" contains the addresses of the variables, not the variables themselves. For example, the program below reads back the two numbers that were stored with "fprintf()" in the last example:

```
/* frdata.c */

#include <stdio.h>
```

```
void main()
{
  int n1;
  float n2;
  FILE *fp;

  fp = fopen( "data", "r" );
  fscanf( fp, "%d %f", &n1, &n2 );
  printf( "%d %f", n1, n2 );
  fclose( fp );
}
```

The "fscanf()" function uses the same format codes as "fprintf()", with the familiar exceptions:

- There is no "%g" format code.
- The "%f" and "%e" format codes work the same.
- There is a "%h" format code for reading short integers.

Numeric modifiers can be used, of course. The "fscanf()" function returns the number of items that it successfully read, or the EOF code, an "int", if it encounters the end of the file or an error.

The following program demonstrates the use of "fprintf()" and "fscanf()":

```
/* fprsc.c */

#include <stdio.h>

void main()
{
  int ctr, i[3], n1 = 16, n2 = 256;
  float f[4], n3 = 3.141592654f;
  FILE *fp;

  fp = fopen( "data", "w+" );

  /* Write data in:   decimal integer formats
                      decimal, octal, hex integer formats
                      floating-point formats  */

  fprintf( fp, "%d %10d %-10d \n", n1, n1, n1 );
  fprintf( fp, "%d %o %x \n", n2, n2, n2 );
  fprintf( fp, "%f %10.10f %e %5.4e \n", n3, n3, n3, n3 );

  /* Rewind file. */

  rewind( fp );

  /* Read back data. */

  puts( "" );
  fscanf( fp, "%d %d %d", &i[0], &i[1], &i[2] );
  printf( "   %d\t%d\t%d\n", i[0], i[1], i[2] );
  fscanf( fp, "%d %o %x", &i[0], &i[1], &i[2] );
  printf( "   %d\t%d\t%d\n", i[0], i[1], i[2] );
  fscanf( fp, "%f %f %f %f", &f[0], &f[1], &f[2], &f[3] );
  printf( "   %f\t%f\t%f\t%f\n", f[0], f[1], f[2], f[3] );

  fclose( fp );
}
```

The program generates the output:

```
16         16         16
256        256        256
3.141593   3.141593   3.141593   3.141600
```

The "fwrite()" and "fread()" functions are used for binary file I/O. The syntax of "fwrite()" is as follows:

```
fwrite( <array_pointer>, <element_size>, <count>, <file_pointer> );
```

The array pointer is of type "void", and so the array can be of any type. The element size and count, which give the number of bytes in each array element and the number of elements in the array, are of type "size_t", which are equivalent to "unsigned int".

The "fread()" function similarly has the syntax:

```
fread( <array_pointer>, <element_size>, <count>, <file_pointer> );
```

The "fread()" function returns the number of items it actually read.

The following program stores an array of data to a file and then reads it back using "fwrite()" and "fread()":

```
/* fwrrd.c */

#include <stdio.h>

#include <math.h>

#define SIZE 20

void main()
{
  int n;
  float d[SIZE];
  FILE *fp;

  for( n = 0; n < SIZE; ++n )                /* Fill array with roots. */
  {
    d[n] = (float)sqrt( (double)n );
  }
  fp = fopen( "data", "w+" );                /* Open file. */
  fwrite( d, sizeof( float ), SIZE, fp );    /* Write it to file. */
  rewind( fp );                              /* Rewind file. */
  fread( d, sizeof( float ), SIZE, fp );     /* Read back data. */
  for( n = 0; n < SIZE; ++n )                /* Print array. */
  {
    printf( "%d: %7.3f\n", n, d[n] );
  }
  fclose( fp );                              /* Close file. */
}
```

The "putc()" function is used to write a single character to an open file. It has the syntax:

```
putc( <character>, <file pointer> );
```

The "getc()" function similarly gets a single character from an open file. It has the syntax:

```
<character variable> = getc( <file pointer> );
```

The "getc()" function returns "EOF" on error. The console I/O functions "putchar()" and "getchar()" are really only special cases of "putc()" and "getc()" that use standard output and input.

The "fputs()" function writes a string to a file. It has the syntax:

```
fputs( <string / character array>, <file pointer> );
```

The "fputs()" function will return an EOF value on error. For example:

```
fputs( "This is a test", fptr );
```

The "fgets()" function reads a string of characters from a file. It has the syntax:

```
fgets( <string>, <max_string_length>, <file_pointer> );
```

The "fgets" function reads a string from a file until if finds a newline or grabs <string_length-1> characters. It will return the value NULL on an error.

The following example program simply opens a file and copies it to another file, using "fgets()" and "fputs()":

```
/* fcopy.c */

#include <stdio.h>


#define MAX 256

void main()
{
  FILE *src, *dst;
  char b[MAX];

  /* Try to open source and destination files. */

  if ( ( src = fopen( "infile.txt", "r" )) == NULL )
  {
     puts( "Can't open input file." );
     exit();
  }
  if ( (dst = fopen( "outfile.txt", "w" )) == NULL )
  {
     puts( "Can't open output file." );
     fclose( src );
     exit();
  }

  /* Copy one file to the next. */
```

```
    while( ( fgets( b, MAX, src ) ) != NULL )
    {
       fputs( b, dst );
    }

    /* All done, close up shop. */

    fclose( src );
    fclose( dst );
  }
```

C File-IO Through System Calls

File-I/O through system calls is simpler and operates at a lower level than making calls to the C file-I/O library. There are seven fundamental file-I/O system calls:

```
    creat()     Create a file for reading or writing.
    open()      Open a file for reading or writing.
    close()     Close a file after reading or writing.
    unlink()    Delete a file.

    write()     Write bytes to file.
    read()      Read bytes from file.
```

These calls were devised for the UNIX operating system and are not part of the ANSI C spec.

Use of these system calls requires a header file named "fcntl.h":

```
    #include <fcntl.h>
```

The "creat()" system call, of course, creates a file. It has the syntax:

```
    <file descriptor variable> = creat( <filename>, <protection bits> );
```

This system call returns an integer, called a "file descriptor", which is a number that identifies the file generated by "creat()". This number is used by other system calls in the program to access the file. Should the "creat()" call encounter an error, it will return a file descriptor value of -1.

The "filename" parameter gives the desired filename for the new file. The "permission bits" give the "access rights" to the file. A file has three

"permissions" associated with it:

- Write permission:

```
   Allows data to be written to the file.
```

- Read permission:

```
   Allows data to be read from the file.
```

- Execute permission:

```
   Designates that the file is a program that can be run.
```

These permissions can be set for three different levels:

- User level:

```
Permissions apply to individual user.
```

- Group level:

```
Permissions apply to members of user's defined "group".
```

- System level:

```
Permissions apply to everyone on the system.
```

For the "creat()" system call, the permissions are expressed in octal, with an octal digit giving the three permission bits for each level of permissions. In octal, the permission settings:

```
0644
```

—grant read and write permissions for the user, but only read permissions

for group and system. The following octal number gives all permissions to everyone:

```
0777
```

An attempt to "creat()" an existing file (for which the program has write permission) will *not* return an error. It will instead wipe the contents of the file and return a file descriptor for it.

For example, to create a file named "data" with read and write permission for everyone on the system would require the following statements:

```
#define RD_WR 0666
...
int fd;                                 /Define file descriptor. */
fd = creat( "data", RD_WR );
```

The "open()" system call opens an existing file for reading or writing. It has the syntax:

```
<file descriptor variable> = open( <filename>, <access mode> );
```

The "open()" call is similar to the "creat()" call in that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code. There are three modes (defined in the "fcntl.h" header file):

```
O_RDONLY    Open for reading only.
O_WRONLY    Open for writing only.
O_RDWR      Open for reading and writing.
```

For example, to open "data" for writing, assuming that the file had been created by another program, the following statements would be used:

```
    int fd;
    fd = open( "data", O_WRONLY );
```

A few additional comments before proceeding:

- A "creat()" call implies an "open()". There is no need to "creat()" a file and then "open()" it.
- There is an operating-system-dependent limit on the number of files that a program can have open at any one time.
- The file descriptor is no more than an arbitrary number that a program uses to distinguish one open file for another. When a file is closed, re-opening it again will probably not give it the same file descriptor.

The "close()" system call is very simple. All it does is "close()" an open file when there is no further need to access it. The "close()" system call has the syntax:

```
    close( <file descriptor> );
```

The "close()" call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

The "unlink()" system call deletes a file. It has the syntax:

```
    unlink( <file_name_string> );
```

It returns 0 on success and -1 on failure. Note: Even after unlink, you will be able to read / write using fd.

The "write()" system call writes data to an open file. It has the syntax:

```
    write( <file descriptor>, <buffer>, <buffer length> );
```

The file descriptor is returned by a "creat()" or "open()" system call. The "buffer" is a pointer to a variable or an array that contains the data; and the "buffer length" gives the number of bytes to be written into the file.

While different data types may have different byte lengths on different systems, the "sizeof()" statement can be used to provide the proper buffer length in bytes. A "write()" call could be specified as follows:

```
    float array[10];
    ...
    write( fd, array, sizeof( array ) );
```

The "write()" function returns the number of bytes it actually writes. It will return -1 on an error.

The "read()" system call reads data from a open file. Its syntax is exactly the same as that of the "write()" call:

```
    read( <file descriptor>, <buffer>, <buffer length> );
```

The "read()" function returns the number of bytes it *actually* returns. At the end of file it returns 0, or returns -1 on error.

C Math Library

This chapter discusses some useful standard C libraries:

- math library
- standard utility library
- the "sprintf()" function
- string function library
- character class test library

And the following minor topics:

- command-line arguments
- dynamic memory allocation
- pointers to functions
- PC memory model and other declarations
- troubleshooting hints

The math library requires the declaration:

```
#include <math.h>
```

The math functions consist of:

```
sin( x )       Sine of x.
cos( x )       Cosine of x.
tan( x )       Tangent of x.
asin( x )      Inverse sine of x.
acos( x )      Inverse cosine of x.
atan( x )      Inverse tangent of x.
sinh( x )      Hyperbolic sine of x.
cosh( x )      Hyperbolic cosine of x.
tanh( x )      Hyperbolic tangent of x.
exp( x )       Exponential function -- e^x.
log( x )       Natural log of x.
log10( x )     Base 10 log of x.
pow( x, y )    Power function -- x^y.
sqrt( x )      Square root of x.
ceil( x )      Smallest integer not less than x, returned as double.
floor( x )     Greatest integer not greater than x, returned as double.
fabs( x )      Absolute value of x.
```

All values are "doubles", and trig values are expressed in radians.

C Standard Utility Library & Time Library

The utility functions library features a grab-bag of functions. It requires the declaration:

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

Useful functions include:

```
atof( <string> )     Convert numeric string to double value.
atoi( <string> )     Convert numeric string to int value.
atol( <string> )     Convert numeric string to long value.
rand()               Generates pseudorandom integer.
srand( <seed> )      Seed random-number generator -- "seed" is an "int".
exit( <status> )     Exits program -- "status" is an "int".
system( <string> )   Tells system to execute program given by "string".
abs( n )             Absolute value of "int" argument.
labs( n )            Absolute value of long-int argument.
```

The functions "atof()", "atoi()", and "atol()" will return 0 if they can't convert the string given them into a value.

The time and date library includes a wide variety of functions, some of them obscure and nonstandard. This library requires the declaration:

```
#include <time.h>
```

The most essential function is "time()", which returns the number of seconds since midnight proleptic Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds. It returns a value as "time_t" (a "long") as defined in the header file.

The following function uses "time()" to implement a program delay with resolution in seconds:

```
/* delay.c */

#include <stdio.h>

#include <time.h>

void sleep( time_t delay );

void main()
{
  puts( "Delaying for 3 seconds." );
  sleep( 3 );
  puts( "Done!" );
}

void sleep( time_t delay )
{
  time_t t0, t1;
  time( &t0 );
  do
  {
    time( &t1 );
  }
```

```
    while (( t1 - t0 ) < delay );
  }
```

The "ctime()" function converts the time value returned by "time()" into a time-and-date string. The following little program prints the current time and date:

```
  /* time.c */

  #include <stdio.h>
  #include <time.h>

  void main()
  {
    time_t *t;
    time( t );
    puts( ctime( t ) );
  }
```

This program prints a string of the form:

```
  Tue Dec 27 15:18:16 1994
```

Further reading

- C Programming/Standard libraries
- C Programming/C Reference
- C++ Programming/Code/Standard C Library

The C sprintf Function

The "sprintf" function creates strings with formatted data. Technically speaking, this is part of the standard-I/O library, and requires the declaration:

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

However, it is really a string function and needs to be discussed along with the other string functions. The syntax of "sprintf()" is exactly the same as it is for "printf()", except there is an extra parameter at the beginning, which is a pointer to a string. Instead of outputting to standard output, sprintf outputs to the string. For example:

```
/* csprntf.c */

#include <stdio.h>

int main()
{
   char b[100];
   int i = 42;
   float f = 1.1234f;
   sprintf( b, "Formatted data:  %d / %f", i, f );
   puts( b );
}
```

—prints the string:

```
Formatted data:  42 / 1.1234
```

There is also an "sscanf()" function that similarly mirrors "scanf()" functionality.

C STRING FUNCTION LIBRARY

moved to A Little C Primer/C String Function Library

C String Function Library

The string-function library requires the declaration:

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

The most important string functions are as follows:

```
strlen()    Get length of a string.
strcpy()    Copy one string to another.
strcat()    Link together (concatenate) two strings.
strcmp()    Compare two strings.
strchr()    Find character in string.
strstr()    Find string in string.
strlwr()    Convert string to lowercase.
strupr()    Convert string to uppercase.
```

strlen()

The "strlen()" function gives the length of a string, not including the NUL character at the end:

```
/* strlen.c */

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

int main()
{
  char *t = "XXX";
  printf( "Length of <%s> is %d.\n", t, strlen( t ));
}
```

This prints:

```
Length of <XXX> is 3.
```

strcpy()

The "strcpy" function copies one string from another. For example:

```
/* strcpy.c */

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

int main()
{
  char s1[100],
       s2[100];
  strcpy( s1, "xxxxxx 1" );
  strcpy( s2, "zzzzzz 2" );

  puts( "Original strings: " );
```

```
    puts( "" );
    puts( s1 );
    puts( s2 );
    puts( "" );

    strcpy( s2, s1 );

    puts( "New strings: " );
    puts( "" );
    puts( s1 );
    puts( s2 );
}
```

This will print:

```
Original strings:

xxxxxx 1
zzzzzz 2

New strings:

xxxxxx 1
xxxxxx 1
```

Please be aware of two features of this program:

- This program assumes that "s2" has enough space to store the final string. The "strcpy()" function won't bother to check, and will give erroneous results if that is not the case.
- A string constant can be used as the source string instead of a string variable. Using a string constant for the destination, of course, makes no sense.

These comments are applicable to most of the other string functions.

strncpy()

There is a variant form of "strcpy" named "strncpy" that will copy "n" characters of the source string to the destination string, presuming there are that many characters available in the source string. For example, if the following change is made in the example program:

```
strncpy( s2, s1, 5 );
```

—then the results change to:

```
New strings:

xxxxxx 1
xxxxxz 2
```

Notice that the parameter "n" is declared "size_t", which is defined in "string.h". Because there is no null byte among the first 5 characters, strncpy does not add '\0' after copying.

strcat()

The "strcat()" function joins two strings:

```
/* strcat.c */

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

int main()
{
  char s1[50],
       s2[50];
  strcpy( s1, "Tweedledee " );
  strcpy( s2, "Tweedledum" );
  strcat( s1, s2 );
  puts( s1 );
}
```

This prints:

```
Tweedledee Tweedledum
```

strncat()

There is a variant version of "strcat()" named "strncat()" that will append "n" characters of the source string to the destination string. If the example above used "strncat()" with a length of 7:

```
strncat( s1, s2, 7 );
```

—the result would be:

```
Tweedledee Tweedle
```

Again, the length parameter is of type "size_t".

strcmp()

The "strcmp()" function compares two strings:

```
/* strcmp.c */

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

#define ANSWER "blue"

int main()
{
  char t[100];
  puts( "What is the secret color?" );
  gets( t );
  while ( strcmp( t, ANSWER ) != 0 )
  {
    puts( "Wrong, try again." );
    gets( t );
  }
```

```
    puts( "Right!" );
  }
```

The "strcmp()" function returns a 0 for a successful comparison, and nonzero otherwise. The comparison is case-sensitive, so answering "BLUE" or "Blue" won't work.

There are three alternate forms for "strcmp()":

strncmp()

- A "strncmp()" function which, as might be guessed, compares "n" characters in the source string with the destination string:

```
"strncmp( s1, s2, 6 )".
```

stricmp()

- A "stricmp()" function that ignores case in comparisons.

strnicmp()

- A case-insensitive version of "strncmp" called "strnicmp".

strchr()

The "strchr" function finds the first occurrence of a character in a string. It returns a pointer to the character if it finds it, and null if not. For example:

```
  /* strchr.c */

  #include <stdio.h>

  #include <string.h>

  int main()
  {
    char *t = "MEAS:VOLT:DC?";
    char *p;
    p = t;
    puts( p );
    while(( p = strchr( p, ':' )) != NULL )
    {
      puts( ++p );
    }
  }
```

This prints:

```
  MEAS:VOLT:DC?
  VOLT:DC?
  DC?
```

The character is defined as a character constant, which C regards as an "int". Notice how the example program increments the pointer before using it ("++p") so that it doesn't point to the ":" but to the character following it.

strrchr()

The "strrchr()" function is almost the same as "strchr()", except that it searches for the *last* occurrence of the character in the string.

strstr()

The "strstr()" function is similar to "strchr()" except that it searches for a string, instead of a character. It also returns a pointer:

```
char *s = "Black White Brown Blue Green";
...
puts( strstr( s, "Blue" ) );
```

strlwr() and strupr()

The "strlwr()" and "strupr()" functions simply perform lowercase or uppercase conversion on the source string. For example:

```
/* casecvt.c */

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

int main()
{
  char *t = "Hey Barney hey!";
  puts( strlwr( t ) );
  puts( strupr( t ) );
}
```

—prints:

```
hey barney hey!
HEY BARNEY HEY!
```

These two functions are only implemented in some compilers and are not part of ANSI C.

Further reading

- C Programming/Strings
- C++ Programming/Code/IO/Streams/string

C Character Class Test Library

These functions perform various tests on characters. They require the declaration:

```
#include <ctype.h>
```

The character is represented as an "int" and the functions return an "int". They return 0 if the test is false and non-0 if the test is true:

```
isalnum( c )     Character is alpha or digit.
isalpha( c )     Character is alpha.
iscntrl( c )     Character is control character.
isdigit( c )     Character is decimal digit.
isgraph( c )     Character is printing character (except space).
islower( c )     Character is lower-case.
isprint( c )     Character is printing character (including space).
ispunct( c )     Character is printing character but not space/alpha-digit.
isspace( c )     Character is space, FF, LF, CR, HT, VT.
isupper( c )     Character is upper-case.
isxdigit( c )    Character is hex digit.
```

The library also contains two conversion functions that also accept and return an "int":

```
tolower( c )    Convert to lower case.
toupper( c )    Convert to upper case.
```

C Command Line Arguments

C allows a program to obtain the command line arguments provided when the executable is called, using two optional parameters of "main()" named "argc (argument count)" and "argv (argument vector)".

The "argc" variable gives the count of the number of command-line parameters provided to the program. This count includes the name of the program itself, so it will always have a value of at least one. The "argv" variable is a pointer to the first element of an array of strings, with each element containing one of the command-line arguments.

The following example program demonstrates:

```
/* cmdline.c */

#include <stdio.h>

void main( int argc, char *argv[] )
{
  int ctr;
  for( ctr=0; ctr < argc; ctr++ )
  {
    puts( argv[ctr] );
  }
}
```

If this program is run from the command line as follows:

```
stooges moe larry curley
```

—the output is:

```
stooges
moe
larry
curley
```

In practice, the command line will probably take a number of arguments, some of which will indicate options or switches, designated by a leading "-" or "/". Some of the switches may be specified separately or together, and some may accept an associated parameter. Other arguments will be text strings, giving numbers, file names, or other data.

The following example program demonstrates parsing the command-line arguments for an arbitrary program. It assumes that the legal option characters are "A", "B", "C", and "S", in either upper- or lower-case. The "S" option must be followed by some string representing a parameter.

```
/* cparse.c */

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

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

```
{
  int m, n,                             /* Loop counters. */
      l,                                /* String length. */
      x,                                /* Exit code. */
      ch;                               /* Character buffer. */
  char s[256];                          /* String buffer. */

  for( n = 1; n < argc; n++ )           /* Scan through args. */
  {
    switch( (int)argv[n][0] )           /* Check for option character. */
    {
    case '-':
    case '/': x = 0;                  /* Bail out if 1. */
              l = strlen( argv[n] );
              for( m = 1; m < l; ++m ) /* Scan through options. */
              {
                ch = (int)argv[n][m];
                switch( ch )
                {
                case 'a':             /* Legal options. */
                case 'A':
                case 'b':
                case 'B':
                case 'C':
                case 'd':
                case 'D': printf( "Option code = %c\n", ch );
                          break;
                case 's':             /* String parameter. */
                case 'S': if( m + 1 >= l )
                          {
                            puts( "Illegal syntax -- no string!" );
                            exit( 1 );
                          }
                          else
                          {
                            strcpy( s, &argv[n][m+1] );
                            printf( "String = %s\n", s );
                          }
                          x = 1;
                          break;
                default:  printf( "Illegal option code = %c\n", ch );
                          x = 1;      /* Not legal option. */
                          exit( 1 );
                          break;
                }
                if( x == 1 )
                {
                  break;
                }
              }
              break;
    default:  printf( "Text = %s\n", argv[n] ); /* Not option -- text. */
              break;
    }
  }
  puts( "DONE!" );
}
```

For a more practical example, here's a simple program, based on an example from the previous chapter, that attempts to read the names of an input and output file from the command line. If no files are present, it uses standard input and standard output instead. If one file is present, it is assumed to be the input file and opens up standard output. This is a useful template for simple file-processing programs.

```
/* cpfile.c */
```

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 256

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

  FILE *src, *dst;
  char b[MAX];

  /* Try to open source and destination files. */

  switch (argc)
  {
  case 1:          /* No parameters, use stdin-stdout. */
    src = stdin;
    dst = stdout;
    break;

  case 2:          /* One parameter -- use input file & stdout. */
    if ( ( src = fopen( argv[1], "r" )) == NULL )
    {
       puts( "Can't open input file.\n" );
       exit( 0 );
    }
    dst = stdout;
    break;

  case 3:         /* Two parameters -- use input and output files. */
    if ( ( src = fopen( argv[1], "r" )) == NULL )
    {
       puts( "Can't open input file.\n" );
       exit( 0 );
    }
    if ( ( dst = fopen( argv[2], "w" )) == NULL )
    {
       puts( "Can't open output file.\n" );
       exit( 0 );
    }
    break;

  default:        /* Too many parameters. */
    puts( "Wrong parameters.\n" );
    exit( 0 );

  }

  /* Copy one file to the next. */

  while( ( fgets( b, MAX, src ) ) != NULL )
  {
     fputs( b, dst );
  }

  /* All done, close up shop. */

  fclose( src );
  fclose( dst );
}
```

Pointers to C Functions

This document has explained how to declare pointers to variables, arrays, and structures in C. It is also possible to define pointers to functions. This feature allows functions to be passed as arguments to other functions. This is useful for, say, building a function that determines solutions to a range of math functions.

The syntax for declaring pointers to functions is obscure, and so let's start with an idiot example: declaring a pointer to the standard library function "printf()":

```
/* ptrprt.c */

#include <stdio.h>

void main()
{
  int (*func_ptr) ();                  /* Declare the pointer. */
  func_ptr = printf;                   /* Assign it a function. */
  (*func_ptr) ( "Printf is here!\n" ); /* Execute the function. */
}
```

The function pointer has to be declared as the same type ("int" in this case) as the function it represents.

Next, let's pass function pointers to another function. This function will assume the functions passed to it are math functions that accept double and return double values:

```
/* ptrroot.c */

#include <stdio.h>

#include <math.h>

void testfunc ( char *name, double (*func_ptr) () );

void main()
{
  testfunc( "square root", sqrt );
}

void testfunc ( char *name, double (*func_ptr) () )
{
  double x, xinc;
  int c;

  printf( "Testing function %s:\n\n", name );
  for( c=0; c < 20; ++c )
  {
    printf( "%d: %f\n", c,(*func_ptr)( (double)c ));
  }
}
```

It is obvious that not all functions can be passed to "testfunc()". The function being passed must agree with the expected number and type of parameters, as well as with the value returned.

C Dynamic Memory Allocation & Deallocation

For simple programs, it is OK to just declare an array of a given size:

```
   char buffer[1024]
```

In more sophisticated programs, this leads to trouble. There may be no way of knowing how big an array needs to be for the specific task the program is performing, and so allocating an array in a fixed size will either result in wasted memory or in not having enough to do the job.

The answer to this problem is to have the program allocate the memory at runtime, and that's what the "malloc()" library function does. For example, let's use "malloc()" to allocate an array of "char":

```
    /*malloc.c */

    #include <stdio.h>
    #include <stdlib.h>                    /*For "malloc", "exit" functions. */

    int main()
    {
      char *p;                             /*Pointer to array. */
      unsigned count;                      /*Size of array. */

      puts( "Size of array?" );
      scanf( "%d", &count );                /*Get size in bytes. */
      p = malloc( (size_t)count );         /*Allocate array. */
      if( p == NULL )                      /*Check for failure. */
      {
        puts( "Can't allocate memory!" );
        exit( 0 );
      }
      puts( "Allocated array!" );
      free( p );                           /*Release memory. */
      return 0;
    }
```

The header file "stdlib.h" must be included, and a pointer to the memory block to be allocated must be declared. The "malloc()" function sets the pointer to the allocated memory block with:

```
    p = malloc( (size_t)count );
```

The count is in bytes and it is "cast" to the type of "size_t", which is defined in "stdio.h". The pointer returned by "malloc()" is assigned to the variable "p", which is of type "char *". In ANSI C, "malloc()" returns a pointer of type "void *", which can be assigned to any other pointer type without a cast.

If the "malloc()" fails because it can't allocate the memory, it returns the value NULL (as defined in "stdio.h").

It is simple to allocate other data types:

```
    int *buf;
    ...
    buf = malloc( (size_t)sizeof( int ) );
```

The "sizeof()" function is used to determine the number of bytes in the "int" data type.

When the programs finished using the memory block, it get rids of it using the "free" function:

```
    free( p );
```

C also contains two other memory-allocation functions closely related to "malloc()": the "calloc()" function, which performs the same function as "malloc()" but allows the block allocated to be specified in terms of number of elements:

```
    void *calloc( size_t <number_elements>, size_t <sizeof_element_type> );
```

—and the "realloc()" function, which reallocates the size of an array

that's already been allocated:

```
    void *realloc( void *<block_pointer>, size_t <size_in_bytes> );
```

Common Programming Problems in C

There are a number of common programming pitfalls in C that trap even experienced programmers:

1: Confusing "=" (assignment operator) with "==" (equality operator). For example:

```
    if ( x = 1 ){ /* wrong! */
    }
    ...
    if ( x == 1 ){ /* good. */
    }
```

and

```
    for ( x == 1; ... /* wrong! */
    ...
    for ( x = 1; ... /* good. */
```

2: Confusing precedence of operations in expressions. When in doubt, use parentheses to enforce precedence. It never hurts to use a few extra parenthesis.

3: Confusing structure-member operators. If "struct_val" is a structure and "struct_ptr" is a pointer to a structure, then:

```
    struct_val->myname
```

—is wrong and so is:

```
    struct_ptr.myname
```

4: Using incorrect formatting codes for "printf()" and "scanf()". Using a "%f" to print an "int", for example, can lead to bizarre output.

5: Remember that the actual base index of an array is 0, and the final index is 1 less than the declared size. For example:

```
    int data[20];
    ...
    for ( x = 1; x <= 20; ++x )
    {
      printf( "%d\n", data[x] );
    }
```

—will give invalid results when "x" is 20. Since C does not do bounds

checking, this one might be hard to catch.

6: Muddling syntax for multidimensional arrays. If:

```
data[10][10]
```

—is a two-dimensional array, then:

```
data[2][7]
```

—will select an element in that array. However:

```
data[ 2, 7 ]
```

—will give invalid results but not be flagged as an error by C.

7: Confusing strings and character constants. The following is a string:

```
"Y"
```

—as opposed to the character constant:

```
'Y'
```

This can cause troubles in comparisons of strings against character constants.

8: Forgetting that strings end in a null character ('\0'). This means that a string will always be one character bigger than the text it stores. It can also cause trouble if a string is being created on a character-by-character basis, and the program doesn't tack the null character onto the end of it.

9: Failing to allocate enough memory for a string—or, if pointers are declared, to allocate any memory for it at all.

10: Failing to check return values from library functions. Most library functions return an error code; while it may not be desirable to check every invocation of "printf()", be careful not to ignore error codes in critical operations.

Of course, forgetting to store the value returned by a function when that's the only way to get the value out of it is a bonehead move, but people do things like that every now and then.

11: Having duplicate library-function names. The compiler will not always catch such bugs.

12: Forgetting to specify header files for library functions.

13: Specifying variables as parameters to functions when pointers are supposed to be specified, and the reverse. If the function returns a value through a parameter, that means it must be specified as a pointer:

```
myfunc( &myvar );
```

The following will not do the job:

```
   myfunc( myvar );
```

Remember that a function may require a pointer as a parameter even if it doesn't return a value, though as a rule this is not a good programming practice.

14: Getting mixed up when using nested "if" and "else" statements. The best way to avoid problems with this is to always use brackets. Avoiding complicated "if" constructs is also a good idea; use "switch" if there's any choice in the matter. Using "switch" is also useful even for simple "if" statements, since it makes it easier to expand the construct if that is necessary.

15: Forgetting semicolons—though the compiler usually catches this—or adding one where it isn't supposed to be—which it usually doesn't. For example:

```
   for( x = 1; x < 10; ++x );
   {
      printf( "%d\n", x )
   }
```

—never prints anything.

16: Forgetting "break" statements in "switch" constructs. As commented earlier, doing so will simply cause execution to flow from one clause of the "switch" to the next.

17: Careless mixing and misuse of signed and unsigned values, or of different data types. This can lead to some insanely subtle bugs. One particular problem to watch out for is declaring single character variables as "unsigned char". Many I/O functions will expect values of "unsigned int" and fail to properly flag EOF. It is recommended to cast function arguments to the proper type even if it appears that type conversion will take care of it on its own.

18: Confusion of variable names. To ensure portability of code, it is recommended that the first six characters of such identifiers be unique.

19: In general, excessively tricky and clever code. Programs are nasty beasts and even if it works, it will have to be modified and even ported to different languages. Maintain a clean structure and do the simple straightforward thing, unless it imposes an unacceptable penalty.

C Quick Reference

This section contains a sample program to give syntax examples for fundamental C statements, followed by a list of library routines. This list is very terse and simply provides reminders. If more details are needed, please refer to the previous chapters.

```
/* sample.c:  a silly program to give syntax examples. */

#include <stdio.h>           /* Include header file for console I/O. */

int f1( int p );             /* Function prototypes. */
long f2( void );
long g;                      /* Global variable. */

void main( int argc, char *argv[] )
{
  float f;                   /* Declare variables. */
  int ctr;
  extern long g;

  printf( "Arguments:\n\n" );
  for( ctr = 0; ctr < argc; ctr++ )
  {
    puts( argv[ctr] );
  }

  printf( "\nFunction 1:\n\n" );
  ctr = 0;
  while( ctr < 5 )
  {
    printf( "%d\n", f1( ctr++ ) );
  }

  printf( "\nFunction 2:\n\n" );
  ctr = 0;
  do
  {
    g = ctr++;
    printf( "%d\n", f2( ) );
  }
  while( ctr < 5 );

  exit( 0 );
}

int f1( int p )
{
  return( ( p < 3 ) ? p : p p );
}

long f2( void )
{
  extern long g;
  return( g g );
}
```

Console I/O -- #include <stdio.h>:

```
int printf( char *s, <varlist> ) > 0      Print formatted string to stdout.
int scanf( char *s, *<varlist> ) != EOF   Read formatted data from stdin.
int putchar( int ch )                     Print a character to stdout.
int getchar() != EOF                      Read a character from stdin.
int puts( char *s )                       Print string to stdout, add \n.
```

```
char *gets() != NULL                    Read line from stdin (no \n).
```

PC console routines -- #include <conio.h>:

```
int getch() != 0          Get a character from the keyboard (no Enter).
int getche() != 0         Get a character from the keyboard and echo it.
int kbhit() != 0          Check to see if a key has been pressed.
```

Format codes:

```
%h       short int (scanf() only)
%d       decimal integer
%ld      long decimal integer
%c       character
%s       string
%e       exponential floating-point
%f       decimal floating-point
%g       use %e or %f, whichever is shorter (printf() only)
%u       unsigned decimal integer
%o       unsigned octal integer
%x       unsigned hex integer

%10d     10-character field width.
%-10d    Left-justified field.
%6.3f    6-character field width, three digits of precision.

'\0NN'   character code in octal.
'\xNN'   character code in hex.
'\0'     null character.
```

File-I/O -- #include <stdio.h>:

```
FILE *fopen( char *f, char *mode ) != NULL          Create or open file.
int fclose( FILE *f )                               Close a file.

rewind( FILE *f )                                   Rewind.
rename( char *old, char *new )                      Rename a file.
remove( char *name )                                Delete a file.

fseek( FILE *f, long offset, int origin) == 0       Seek.

fprintf( FILE *f, char *fmt, <varlist> ) > 0        Formatted write.
fscanf( FILE *f, char *fmt, &<varlist> ) != EOF     Formatted read.
fwrite( void *b, size_t s, size_t c, FILE *f ) > 0  Unformatted write.
fread( void *b, size_t s, size_t c, FILE *f ) > 0   Unformatted read.

putc( int c, FILE *f )                              Write character.
int getc( FILE *f ) != EOF                          Read character.
fputs( char *s, FILE *f )                           Write a string.
fgets( char *s, int max, FILE *f) != NULL           Read a string.

sprintf( char *b, char *fmt, <varlist> )            Print into string.
sscanf( char *b, char *fmt, &<varlist> ) > 0        Scan string.
```

File modes:

```
r     Open for reading.
w     Open and wipe (or create) for writing.
a     Append -- open (or create) to write to end of file.
r+    Open a file for reading and writing.
```

```
    w+    Open and wipe (or create) for reading and writing.
    a+    Open a file for reading and appending.
```

Offset values:

```
    SEEK_SET    Start of file.
    SEEK_CUR    Current location.
    SEEK_END    End of file.
```

Math library -- #include <math.h>:

```
    double sin( double x )             Sine of x (in radians).
    double cos( double x )             Cosine of x.
    double tan( double x )             Tangent of x.
    double asin( double x )            Inverse sine of x.
    double acos( double x )            Inverse cosine of x.
    double atan( double x )            Inverse tangent of x.
    double sinh( double x )            Hyperbolic sine of x.
    double cosh( double x )            Hyperbolic cosine of x.
    double tanh( double x )            Hyperbolic tangent of x.
    double exp( double x )             Exponential function -- e^x.
    double log( double x )             Natural log of x.
    double log10( double x )           Base 10 log of x.
    double pow( double x, double y )   Power function -- x^y.
    double sqrt( double x )            Square root of x.
    double ceil( double x )            Integer >= x (returned as double).
    double floor( double x )           Integer <= x (returned as double).
    double fabs( x )                   Absolute value of x.
```

Standard utility library -- #include <stdlib.h>:

```
    double atof( char *nvalstr ) != 0   Convert numeric string to double.
    int atoi( char *nvalstr )    != 0   Convert numeric string to int.
    long atol( char *nvlastr )   != 0   Convert numeric string to long.
    int rand()                          Generates pseudorandom integer.
    srand( unsigned seed )              Seed random-number generator.
    exit( int status )                  Exits program.
    int system( char *syscmd )   == 0   Execute system program.
    int abs( int n )                    Absolute value of int.
    long labs( long n )                 Absolute value of long.
```

Time & date library -- #include <time.h>:

```
    time_t time( time_t *timeptr )      Current time count as long int.
    char *ctime( time_t *timeptr )      Current time & date string.
```

String function library -- #include <string.h>:

```
    int strlen( char *s )                          Length.
    strcpy( char *dst, char *src )                 Copy.
    strncpy( char *dst, char *src, size_t n )      Copy n characters max.
    strcat( char *dst, char *s )                   Concatenate.
    strncat( char *d, char *s, size_t n )          Concatenate n characters.
    strcmp( char *s1, char *s2 )             == 0  Compare.
    strncmp( char *s1, char *s2, size_t n )  == 0  Compare n characters.
    stricmp( char *s1, char *s2 )            == 0  Compare, no case.
```

```
    strnicmp( char *s1, char *s2, size_t n ) == 0   Compare, no case, n chars.
    char *strchr( char *s, int ch )       != NULL   Find first character.
    char *strrchr( char *s, int ch )      != NULL   Find last character.
    char *strstr( char *dst, char *src)   != NULL   Find string.
    char *strlwr( char *s )                         Lowercase.
    char *strupr( char *s )                         Uppercase.
```

Character class test library -- #include <ctype.h>:

```
    int isalnum( int c )  != 0         Alpha / digit.
    int isalpha( int c )  != 0         Alpha.
    int iscntrl( int c )  != 0         Control character.
    int isdigit( int c )  != 0         Decimal digit.
    int isgraph( int c )  != 0         Printing character (except space).
    int islower( int c )  != 0         Lower-case.
    int isprint( int c )  != 0         Printing character (including space).
    int ispunct( int c )  != 0         Printing character but not space/alnum.
    int isspace( int c )  != 0         Space, FF, LF, CR, HT, VT.
    int isupper( int c )  != 0         Upper-case.
    int isxdigit( int c ) != 0         Hex digit.

    int tolower( int c )               Convert to lower case.
    int toupper( int c )               Convert to upper case.
```

Dynamic memory allocation -- #include <malloc.h>:

```
    buf = (<type> *)malloc( (size_t)sizeof( <type> ) <array size>) != NULL
    free( <type> *buf )
```

Comments and Revision History

I wrote this document, not because I am an expert on this subject, but because I'm not. I don't use C often, and the combination of infrequent use and relatively obscure syntax makes them frustrating to deal with. So I threw these notes together to make sure that what I did know was in an accessible form.

This document originally contained two chapters on C++, but they were sketchy and I had little interest in the language. C++ is really not so much an enhancement of C as it is a different language based on C. For the very small software projects I tend to work on, its additional features are much more bother than they're worth. Since I couldn't maintain that material, I deleted it.

Revision history:

```
   v1.0   / 01 jan 95 / gvg
   v2.0   / 01 may 95 / gvg / Added chapters on C++, reformatted.
   v2.1   / 30 may 95 / gvg / Minor corrections and changes.
   v2.2   / 01 jul 95 / gvg / Added quick reference and minor changes.
   v2.3   / 09 jul 95 / gvg / Corrected bug in "strlen()" description.
   v2.4   / 25 aug 95 / gvg / Added command-line parser example.
   v2.5   / 13 oct 95 / gvg / Web update.
   v2.5   / 01 feb 99 / gvg / Minor cosmetic update.
   v2.0.7 / 01 feb 02 / gvg / Minor cosmetic update.
   v3.0.0 / 01 aug 03 / gvg / Eliminated C++ material.
   v3.0.1 / 01 aug 05 / gvg / Minor proofing & cleanup.
   v3.0.2 / 01 jul 07 / gvg / Minor cosmetic update.
```

Resources

Wikimedia Resources

- C Programming
- C++ Programming/Code/Standard C Library

Other Resources

- http://www.vectorsite.net
- Learn C Language (http://www.studiesinn.com/learn/Programming-Languages/C-Language.html)

Licensing

Licensing

The original text of this book was released into the public domain by its author, Greg Goebel. The text of the book is available at http://www.vectorsite.net/tscpp.html, and that original version is still available in the public domain.

The version here at Wikibooks is released under the following license:

Permission is granted to copy, distribute and/or modify this document under the terms of the **GNU Free Documentation License**, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License."