# C Language Tutorial

**Table of Contents:**


- 1. A First Program
- 2. Let's Compute
- 3. Loops
- 4. Symbolic Constants
- 5. Conditionals
- 6. Pointers
- 7. Arrays
- 8. Character Arrays
- 9. I/O Capabilities
- 10. Functions
- 11. Command-line Arguments
- 12. Graphical Interfaces: Dialog Boxes


---

This section contains a brief introduction to the C language. It is intended as a tutorial on the language, and aims at getting a reader new to C started as quickly as possible. It is certainly *not* intended as a substitute for any of the numerous textbooks on C.

The best way to learn a new ``human" language is to speak it right from the outset, listening and repeating, leaving the intricacies of the grammar for later. The same applies to computer languages--to learn C, we must start writing C programs as quickly as possible.

An excellent textbook on C by two well-known and widely respected authors is:

**The C Programming Language -- ANSI C**
*Brian W. C. Kernighan & Dennis M. Ritchie*
*Prentice Hall, 1988*

Dennis Ritchie designed and implemented the first C compiler on a PDP-11 (a prehistoric machine by today's standards, yet one which had enormous influence on modern scientific computation). The C language was based on two (now defunct) languages: BCPL, written by Martin Richards, and B, written by Ken Thompson in 1970 for the first UNIX system on a PDP-7. The original ``official" C language was the ``K & R" C, the nickname coming from the names of the two authors of the original ``The C Programming Language". In 1988, the American National Standards Institute (ANSI) adopted a ``new and improved" version of C, known today as ``ANSI C". This is the version described in the current edition of ``The C Programming Language -- ANSI C". The ANSI version contains many revisions to the syntax and the internal workings of the language, the major ones being improved calling syntax for procedures and standarization of most (but, unfortunately, not quite all!) system libraries.

# 1. A First Program

Let's be polite and start by saluting the world! Type the following program into your favorite editor:

---

```
#include < stdio.h>

void main()
{
```

```
    printf("\nHello World\n");
}
```

Save the code in the file `hello.c`, then compile it by typing:

```
gcc hello.c
```

This creates an *executable* file `a.out`, which is then executed simply by typing its name. The result is that the characters `` Hello World'' are printed out, preceded by an empty line.

A C program contains *functions* and *variables*. The functions specify the tasks to be performed by the program. The ``main'' function establishes the overall logic of the code. It is normally kept short and calls different functions to perform the necessary sub-tasks. All C codes must have a ``main'' function.

Our `hello.c` code calls `printf`, an output function from the I/O (input/output) library (defined in the file `stdio.h`). The original C language did not have any built-in I/O statements whatsoever. Nor did it have much arithmetic functionality. The original language was really not intended for "scientific" or "technical" computation.. These functions are now performed by standard libraries, which are now part of ANSI C. The K & R textbook lists the content of these and other standard libraries in an appendix.

The `printf` line prints the message ``Hello World'' on ``stdout'' (the output stream corresponding to the X-terminal window in which you run the code); ``\n'' prints a ``new line'' character, which brings the cursor onto the next line. By construction, `printf` never inserts this character on its own: the following program would produce the same result:

```
#include < stdio.h>

void main()
{
    printf("\n");
    printf("Hello World");
    printf("\n");
}
```

Try leaving out the ``\n'' lines and see what happens.

The first statement ``#include < stdio.h>'' includes a specification of the C I/O library. All variables in C must be explicitly defined before use: the ``.h'' files are by convention ``header files'' which contain definitions of variables and functions necessary for the functioning of a program, whether it be in a user-written section of code, or as part of the standard C libaries. The directive ``#include'' tells the C compiler to insert the contents of the specified file at that point in the code. The ``< ...>'' notation instructs the compiler to look for the file in certain ``standard'' system directories.

The `void` preceeding ``main'' indicates that main is of ``void'' type--that is, it has *no* type associated with it, meaning that it cannot return a result on execution.

The ``;'' denotes the end of a statement. Blocks of statements are put in braces {...}, as in the definition of functions. All C statements are defined in free format, i.e., with no specified layout or column assignment. Whitespace (tabs or spaces) is never significant, except inside quotes as part of a character string. The following program would produce exactly the same result as our earlier example:

```
#include < stdio.h>
void main(){printf("\nHello World\n");}
```

The reasons for arranging your programs in lines and indenting to show structure should be obvious!

# 2. Let's Compute

The following program, `sine.c`, computes a table of the sine function for angles between 0 and 360 degrees.

```
                                /************************/
                                /*   Table of           */
                                /*   Sine Function      */
                                /************************/

                                /* Michel Vallieres     */
                                /* Written: Winter 1995 */
#include < stdio.h>
#include < math.h>

void main()
{
    int    angle_degree;
    double angle_radian, pi, value;

                                        /* Print a header */
    printf ("\nCompute a table of the sine function\n\n");

                                        /* obtain pi once for all */
                                        /* or just use pi = M_PI, where
                                           M_PI is defined in math.h    */
    pi = 4.0*atan(1.0);
    printf ( " Value of PI = %f \n\n", pi );

    printf ( " angle     Sine \n" );

    angle_degree=0;                     /* initial angle value           */
                                        /* scan over angle               */

    while (  angle_degree <= 360 )      /* loop until angle_degree > 360 */
    {
       angle_radian = pi * angle_degree/180.0 ;
       value = sin(angle_radian);
       printf ( " %3d      %f \n ", angle_degree, value );

       angle_degree = angle_degree + 10; /* increment the loop index     */
    }
}
```

The code starts with a series of comments indicating its the purpose, as well as its author. It is considered good programming style to identify and document your work (although, sadly, most people only do this as an afterthought). Comments can be written anywhere in the code: any characters between /* and */ are ignored by the compiler and can be used to make the code easier to understand. The use of variable names that are meaningful within the context of the problem is also a good idea.

The `#include` statements now also include the header file for the standard mathematics library `math.h`. This statement is needed to define the calls to the trigonometric functions `atan` and `sin`. Note also that the compilation must include the mathematics library explicitly by typing

```
gcc sine.c -lm
```

Variable names are arbitrary (with some compiler-defined maximum length, typically 32 characters). C uses the following standard variable types:

```
int    -> integer variable
short  -> short integer
long   -> long integer
float  -> single precision real (floating point) variable
double -> double precision real (floating point) variable
char   -> character variable (single byte)
```

The compilers checks for consistency in the types of all variables used in any code. This feature is intended to prevent mistakes, in particular in mistyping variable names. Calculations done in the math library routines are usually done in double precision arithmetic (64 bits on most workstations). The actual number of bytes used in the internal storage of these data types depends on the machine being used.

The `printf` function can be instructed to print integers, floats and strings properly. The general syntax is

```
printf( "format", variables );
```

where `"format"` specifies the converstion specification and `variables` is a list of quantities to print. Some useful formats are

```
%.nd    integer (optional n = number of columns; if 0, pad with zeroes)
%m.nf   float or double (optional m = number of columns,
                                n = number of decimal places)
%ns     string (optional n = number of columns)
%c      character
\n \t   to introduce new line or tab
\g      ring the bell (``beep'') on the terminal
```

# 3. Loops

Most real programs contain some construct that loops within the program, performing repetitive actions on a stream of data or a region of memory. There are several ways to loop in C. Two of the most common are the `while` loop:

```
while (expression)
{
    ...block of statements to execute...
}
```

and the `for` loop:

```
for (expression_1; expression_2; expression_3)
{
    ...block of statements to execute...
}
```

The `while` loop continues to loop until the conditional `expression` becomes false. The condition is tested upon entering the loop. Any logical construction (see below for a list) can be used in this context.

The `for` loop is a special case, and is equivalent to the following while loop:

```
expression_1;

while (expression_2)
{
    ...block of statements...

    expression_3;
}
```

For instance, the following structure is often encountered:

```
i = initial_i;

while (i <= i_max)
{
    ...block of statements...

    i = i + i_increment;
}
```

This structure may be rewritten in the easier syntax of the `for` loop as:

```
for (i = initial_i; i <= i_max; i = i + i_increment)
{
    ...block of statements...
}
```

Infinite loops are possible (e.g. `for(;;)`), but not too good for your computer budget! C permits you to write an infinite loop, and provides the `break` statement to ``breakout '' of the loop. For example, consider the following (admittedly not-so-clean) re-write of the previous loop:

```
angle_degree = 0;

for ( ; ; )
{
    ...block of statements...

    angle_degree = angle_degree + 10;
    if (angle_degree == 360) break;
}
```

The conditional `if` simply asks whether `angle_degree` is equal to 360 or not; if yes, the loop is stopped.

# 4. Symbolic Constants

You can define constants of any type by using the `#define` compiler directive. Its syntax is simple--for instance

```
#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

would define `ANGLE_MIN` and `ANGLE_MAX` to the values 0 and 360, respectively. C distinguishes between lowercase and uppercase letters in variable names. It is customary to use capital letters in defining global constants.

# 5. Conditionals

Conditionals are used within the `if` and `while` constructs:

```
if (conditional_1)
{
    ...block of statements executed if conditional_1 is true...
}
else if (conditional_2)
{
    ...block of statements executed if conditional_2 is true...
}
else
{
    ...block of statements executed otherwise...
}
```

and any variant that derives from it, either by omitting branches or by including nested conditionals.

Conditionals are logical operations involving comparison of quantities (of the same type) using the conditional operators:

```
<               smaller than
<=              smaller than or equal to
==              equal to
!=              not equal to
>=              greater than or equal to
>               greater than
```

and the boolean operators

```
&&              and
||              or
!               not
```

Another conditional use is in the `switch` construct:

```
switch (expression)
{
    case const_expression_1:
    {
        ...block of statements...
        break;
    }
    case const_expression_2:
    {
        ...block of statements...
        break;
    }
    default:
    {
        ...block of statements..
```

```
        }
    }
```

The appropriate block of statements is executed according to the value of the expression, compared with the constant expressions in the case statement. The `break` statements insure that the statements in the cases following the chosen one will not be executed. If you would want to execute these statements, then you would leave out the `break` statements. This construct is particularly useful in handling input variables.

# 6. Pointers

The C language allows the programmer to ``peek and poke" directly into memory locations. This gives great flexibility and power to the language, but it also one of the great hurdles that the beginner must overcome in using the language.

All variables in a program reside in memory; the statements

```
float x;
x = 6.5;
```

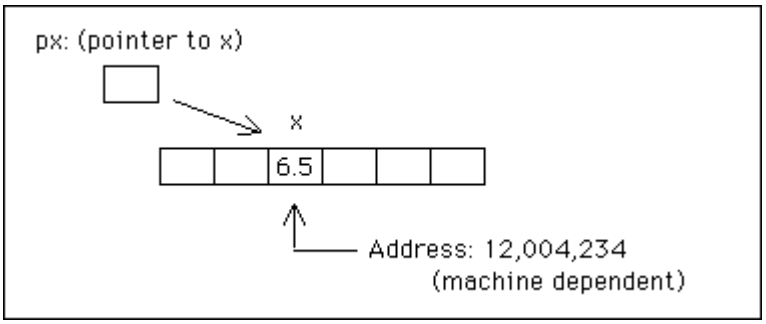
request that the compiler reserve 4 bytes of memory (on a 32-bit computer) for the floating-point variable `x`, then put the ``value" 6.5 in it.

Sometimes we want to know where a variable resides in memory. The address (location in memory) of any variable is obtained by placing the operator ```&`" before its name. Therefore `&x` is the address of `x`. C allows us to go one stage further and define a variable, called a *pointer*, that contains the address of (i.e. ``points to") other variables. For example:

```
float x;
float* px;

x = 6.5;
px = &x;
```

defines `px` to be a pointer to objects of type float, and sets it equal to the address of `x`:




Pointer use for a variable

The content of the memory location referenced by a pointer is obtained using the ``*" operator (this is called *dereferencing* the pointer). Thus, `*px` refers to the value of `x`.

C allows us to perform arithmetic operations using pointers, but beware that the ``unit" in pointer arithmetic is the size (in bytes) of the object to which the pointer points. For example, if `px` is a pointer to a variable `x` of type

`float`, then the expression `px + 1` refers not to the next bit or byte in memory but to the location of the next `float` after `x` (4 bytes away on most workstations); if `x` were of type `double`, then `px + 1` would refer to a location 8 bytes (the size of a `double`)away, and so on. Only if `x` is of type `char` will `px + 1` actually refer to the next byte in memory.

Thus, in

```
    char* pc;
    float* px;
    float x;

    x = 6.5;
    px = &x;
    pc = (char*) px;
```

(the `(char*)` in the last line is a ``cast", which converts one data type to another), `px` and `pc` both point to the same location in memory--the address of `x`--but `px + 1` and `pc + 1` point to *different* memory locations.

Consider the following simple code.

---

```
void main()
{
    float x, y;                         /* x and y are of float type         */
    float *fp, *fp2;                    /* fp and fp2 are pointers to float  */

    x = 6.5;                            /* x now contains the value 6.5      */

                                        /* print contents and address of x   */
    printf("Value of x is %f, address of x %ld\n", x, &x);

    fp = &x;                            /* fp now points to location of x    */

                                        /* print the contents of fp          */
    printf("Value in memory location fp is %f\n", *fp);

                                        /* change content of memory location */
    *fp = 9.2;
    printf("New value of x is %f = %f \n", *fp, x);

                                        /* perform arithmetic                */
    *fp = *fp + 1.5;
    printf("Final value of x is %f = %f \n", *fp, x);

                                        /* transfer values                   */
    y = *fp;
    fp2 = fp;
    printf("Transfered value into y = %f and fp2 = %f \n", y, *fp2);
}
```

---

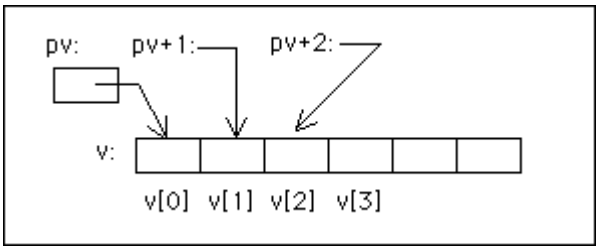
Run this code to see the results of these different operations. Note that, while the value of a pointer (if you print it out with `printf`) is typically a large integer, denoting some particular memory location in the computer, pointers are *not* integers--they are a completely different data type.

# 7. Arrays

Arrays of any type can be formed in C. The syntax is simple:

```
type name[dim];
```

In C, arrays starts at position 0. The elements of the array occupy adjacent locations in memory. C treats the name of the array as if it were a pointer to the first element--this is important in understanding how to do arithmetic with arrays. Thus, if `v` is an array, `*v` is the same thing as `v[0]`, `*(v+1)` is the same thing as `v[1]`, and so on:




Pointer use for an array

Consider the following code, which illustrates the use of pointers:

---

```
#define SIZE 3

void main()
{
    float x[SIZE];
    float *fp;
    int   i;
                                        /* initialize the array x         */
                                        /* use a "cast" to force i        */
                                        /* into the equivalent float      */
    for (i = 0; i < SIZE; i++)
        x[i] = 0.5*(float)i;
                                        /* print x                        */
    for (i = 0; i < SIZE; i++)
        printf("  %d  %f \n", i, x[i]);
                                        /* make fp point to array x       */
    fp = x;
                                        /* print via pointer arithmetic   */
                                        /* members of x are adjacent to   */
                                        /* each other in memory           */
                                        /* *(fp+i) refers to content of   */
                                        /* memory location (fp+i) or x[i] */
    for (i = 0; i < SIZE; i++)
        printf("  %d  %f \n", i, *(fp+i));
}
```

---

(The expression ``i++" is C shorthand for ```i = i + 1`".) Since `x[i]` means the `i`-th element of the array `x`, and `fp = x` points to the start of the `x` array, then `*(fp+i)` is the content of the memory address `i` locations beyond `fp`, that is, `x[i]`.

# 8. Character Arrays

A *string constant* , such as

```
"I am a string"
```

is an array of characters. It is represented internally in C by the ASCII characters in the string, i.e., ``I", blank, ``a", ``m",... for the above string, and terminated by the special null character ``\0" so programs can find the end of the string.

String constants are often used in making the output of code intelligible using `printf` ;

```
printf("Hello, world\n");
printf("The value of a is: %f\n", a);
```

String constants can be associated with variables. C provides the `char` type variable, which can contain one character--1 byte--at a time. A character string is stored in an array of character type, one ASCII character per location. Never forget that, since strings are conventionally terminated by the null character ``\0", we require *one extra storage location* in the array!

C does not provide any operator which manipulate entire strings at once. Strings are manipulated either via pointers or via special routines available from the standard *string* library `string.h`. Using character pointers is relatively easy since the name of an array is a just a pointer to its first element. Consider the following code:

---

```
void main()
{
     char text_1[100], text_2[100], text_3[100];
     char *ta, *tb;
     int i;
                                     /* set message to be an arrray  */
                                     /* of characters; initialize it */
                                     /* to the constant string "..." */
                                     /* let the compiler decide on   */
                                     /* its size by using []         */
     char message[] = "Hello, I am a string; what are you?";

     printf("Original message: %s\n", message);

                                      /* copy the message to text_1   */
                                      /* the hard way                 */
     i=0;
     while ( (text_1[i] = message[i]) != '\0' )
                i++;
     printf("Text_1: %s\n", text_1);

                                      /* use explicit pointer arithmetic */
     ta=message;
     tb=text_2;
     while ( ( *tb++ = *ta++ ) != '\0' )
                               ;
     printf("Text_2: %s\n", text_2);

}
```

---

The standard ``string" library contains many useful functions to manipulate strings; a description of this library can be found in an appendix of the K & R textbook. Some of the most useful functions are:

```
char *strcpy(s,ct)     -> copy  ct into s, including ``\0''; return s
char *strncpy(s,ct,n)  -> copy ncharcater of ct into s, return s
char *strncat(s,ct)    -> concatenate ct to end of s; return s
char *strncat(s,ct,n)  -> concatenate n character of ct to end
                          of s, terminate with ``\0''; return s
int strcmp(cs,ct)      -> compare cs and ct; return 0 if cs=ct,
                             <0 if cs0 if cs>ct
char *strchr(cs,c)     -> return pointer to first occurence of c
                            in cs or NULL if not encountered
size_t strlen(cs)      -> return length of cs
```

(`s` and `t` are `char*`, `cs` and `ct` are `const char*`, `c` is an `char` converted to type `int`, and `n` is an `int`.)

Consider the following code which uses some of these functions:

---

```
#include < string.h>

void main()
{
     char line[100], *sub_text;
                                       /* initialize string */
     strcpy(line,"hello, I am a string;");
     printf("Line: %s\n", line);
                                       /* add to end of string */
     strcat(line," what are you?");
     printf("Line: %s\n", line);
                                       /* find length of string */
                                       /* strlen brings back    */
                                       /* length as type size_t */

     printf("Length of line: %d\n", (int)strlen(line));

                                       /* find occurence of substrings */
     if ( (sub_text = strchr ( line, 'W' ) )!= NULL )
            printf("String starting with \"W\" ->%s\n", sub_text);

     if ( ( sub_text = strchr ( line, 'w' ) )!= NULL )
            printf("String starting with \"w\" ->%s\n", sub_text);

     if ( ( sub_text = strchr ( sub_text, 'u' ) )!= NULL )
            printf("String starting with \"w\" ->%s\n", sub_text);

}
```

---

# 9. I/O Capabilities

## Character level I/O

C provides (through its libraries) a variety of I/O routines. At the character level, `getchar()` reads one character at a time from `stdin`, while `putchar()` writes one character at a time to `stdout`. For example, consider

---

```
#include < stdio.h>

void main()
{
    int i, nc;

    nc = 0;
    i = getchar();
    while (i != EOF) {
        nc = nc + 1;
        i = getchar();
    }
    printf("Number of characters in file = %d\n", nc);
}
```

---

This program counts the number of characters in the input stream (e.g. in a file piped into it at execution time). The code reads characters (whatever they may be) from `stdin` (the keyboard), uses `stdout` (the X-terminal you run from) for output, and writes error messages to `stderr` (usually also your X-terminal). These streams are always defined at run time. EOF is a special return value, defined in `stdio.h`, returned by `getchar()` when it encounters an *end-of-file* marker when reading. Its value is computer dependent, but the C compiler hides this fact from the user by defining the variable EOF. Thus the program reads characters from `stdin` and keeps adding to the counter `nc`, until it encounters the ``end of file".

An experienced C programmer would probably code this example as:

---

```
#include < stdio.h>

void main()
{
    int c, nc = 0;

    while ( (c = getchar()) != EOF ) nc++;

    printf("Number of characters in file = %d\n", nc);
}
```

---

C allows great brevity of expression, usually at the expense of readability!

The `()` in the statement `(c = getchar())` says to execute the call to `getchar()` and assign the result to `c` before comparing it to EOF; the brackets are necessary here. Recall that `nc++` (and, in fact, also `++nc`) is another way of writing `nc = nc + 1`. (The difference between the prefix and postfix notation is that in `++nc`, `nc` is incremented before it is used, while in `nc++`, `nc` is used before it is incremented. In this particular example, either would do.) This notation is more compact (not always an advantage, mind you), and it is often more efficiently coded by the compiler.

The UNIX command `wc` counts the characters, words and lines in a file. The program above can be considered as your own `wc`. Let's□ add a counter for the lines.

---

```
#include < stdio.h>

void main()
{
    int c, nc = 0, nl = 0;
```

```
    while ( (c = getchar()) != EOF )
    {
        nc++;
        if (c == '\n') nl++;
    }

    printf("Number of characters = %d, number of lines = %d\n",
           nc, nl);
}
```

---

Can you think of a way to count the number of words in the file?

## Higher-Level I/O capabilities

We have already seen that `printf` handles formatted output to `stdout`. The counterpart statement for reading from `stdin` is `scanf`. The syntax

```
        scanf("format string", variables);
```

resembles that of `printf`. The format string may contain blanks or tabs (ignored), ordinary ASCII characters, which must match those in `stdin`, and conversion specifications as in `printf`.

Equivalent statements exist to read from or write to character strings. They are:

```
    sprintf(string, "format string", variables);
    scanf(string, "format string", variables);
```

The ``string" argument is the name of (i.e. a pointer to) the character array into which you want to write the information.

## I/O to and from files

Similar statements also exist for handling I/O to and from files. The statements are

```
#include < stdio.h>

    FILE *fp;

    fp = fopen(name, mode);

    fscanf(fp, "format string", variable list);
    fprintf(fp, "format string", variable list);

    fclose(fp );
```

The logic here is that the code must

- define a local ``pointer" of type `FILE` (note that the uppercase is necessary here), which is defined in < stdio.h>
- ``open" the file and associate it with the local pointer via `fopen`
- perform the I/O operations using `fscanf` and `fprintf`
- disconnect the file from the task with `fclose`

The ``mode" argument in the `fopen` specifies the purpose/positioning in opening the file: ``r" for reading, ``w" for writing, and ``a" for appending to the file. Try the following:

---

```
#include < stdio.h>

void main()
{
    FILE *fp;
    int i;

    fp = fopen("foo.dat", "w");        /* open foo.dat for writing */

    fprintf(fp, "\nSample Code\n\n");  /* write some info */
    for (i = 1; i <= 10 ; i++)
        fprintf(fp, "i = %d\n", i);

    fclose(fp);                        /* close the file  */
}
```

Compile and run this code; then use any editor to read the file `foo.dat`.

# 10. Functions

Functions are easy to use; they allow complicated programs to be parcelled up into small blocks, each of which is easier to write, read, and maintain. We have already encountered the function `main` and made use of I/O and mathematical routines from the standard libraries. Now let's look at some other library functions, and how to write and use our own.

## Calling a Function

The call to a function in C simply entails referencing its name with the appropriate arguments. The C compiler checks for compatibility between the arguments in the calling sequence and the definition of the function.

Library functions are generally not available to us in source form. Argument type checking is accomplished through the use of header files (like `stdio.h`) which contain all the necessary information. For example, as we saw earlier, in order to use the standard mathematical library you must include `math.h` via the statement

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

at the top of the file containing your code. The most commonly used header files are

```
< stdio.h>  -> defining I/O routines
< ctype.h>  -> defining character manipulation routines
< string.h> -> defining string manipulation routines
< math.h>   -> defining mathematical routines
< stdlib.h> -> defining number conversion, storage allocation
                and similar tasks
< stdarg.h> -> defining libraries to handle routines with variable
                numbers of arguments
< time.h>   -> defining time-manipulation routines
```

In addition, the following header files exist:

```
< assert.h> -> defining diagnostic routines
< setjmp.h> -> defining non-local function calls
< signal.h> -> defining signal handlers
< limits.h> -> defining constants of the int type
< float.h>  -> defining constants of the float type
```

Appendix B in the K & R book describes these libraries in great detail.

## Writing Your Own Functions

A function has the following layout:

```
return-type function-name ( argument-list-if-necessary )
{
    ...local-declarations...

    ...statements...

    return return-value;
}
```

If `return-type` is omitted, C defaults to `int`. The `return-value` must be of the declared type.

A function may simply perform a task without returning any value, in which case it has the following layout:

```
void function-name ( argument-list-if-necessary )
{
    ...local-declarations...

    ...statements...
}
```

As an example of function calls, consider the following code:

---

```
                        /* include headers of library */
                        /* defined for all routines   */
                        /* in the file                */
#include < stdio.h>
#include < string.h>
                        /* prototyping of functions   */
                        /* to allow type checks by    */
                        /* the compiler               */

void main()
{
    int  n;
    char string[50];
                        /* strcpy(a,b) copies string b into a */
                        /* defined via the stdio.h header     */
    strcpy(string, "Hello World");

                        /* call own function */
    n = n_char(string);
    printf("Length of string = %d\n", n);
}

                        /* definition of local function n_char */
int n_char(char string[])
{
                        /* local variable in this function   */
    int n;
```

```
                        /* strlen(a) returns the length of   */
                        /* string a                          */
                        /* defined via the string.h header   */
    n = strlen(string);
    if (n > 50)
        printf("String is longer than 50 characters\n");

                        /* return the value of integer n  */
    return n;
}
```

---

Arguments are always passed *by value* in C function calls. This means that local ``copies" of the values of the arguments are passed to the routines. Any change made to the arguments internally in the function are made only to the local copies of the arguments. In order to change (or define) an argument in the argument list, this argument must be passed as an address, thereby forcing C to change the ``real" argument in the calling routine.

As an example, consider exchanging two numbers between variables. First let's illustrate what happen if the variables are passed by value:

---

```
#include < stdio.h>

void exchange(int a, int b);

void main()
{                       /* WRONG CODE */
    int a, b;

    a = 5;
    b = 7;
    printf("From main: a = %d, b = %d\n", a, b);

    exchange(a, b);
    printf("Back in main: ");
    printf("a = %d, b = %d\n", a, b);
}

void exchange(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    printf(" From function exchange: ");
    printf("a = %d, b = %d\n", a, b);
}
```

---

Run this code and observe that `a` and `b` are NOT exchanged! Only the copies of the arguments are exchanged. The RIGHT way to do this is of course to use pointers:

---

```
#include < stdio.h>

void exchange ( int *a, int *b );
```

```
void main()
{                       /* RIGHT CODE */
    int a, b;

    a = 5;
    b = 7;
    printf("From main: a = %d, b = %d\n", a, b);

    exchange(&a, &b);
    printf("Back in main: ");
    printf("a = %d, b = %d\n", a, b);
}

void exchange ( int *a, int *b )
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
    printf(" From function exchange: ");
    printf("a = %d, b = %d\n", *a, *b);
}
```

---

The rule of thumb here is that

- You use regular variables if the function does not change the values of those arguments
- You MUST use pointers if the function changes the values of those arguments

# 11. Command-line arguments

It is standard practice in UNIX for information to be passed from the command line directly into a program through the use of one or more command-line arguments, or *switches*. Switches are typically used to modify the behavior of a program, or to set the values of some internal parameters. You have already encountered several of these--for example, the `"ls"` command lists the files in your current directory, but when the switch `-l` is added, `"ls -l"` produces a so-called ``long" listing instead. Similarly, `"ls -l -a"` produces a long listing, including ``hidden" files, the command `"tail -20"` prints out the last 20 lines of a file (instead of the default 10), and so on.

Conceptually, switches behave very much like arguments to functions within C, and they are passed to a C program from the operating system in precisely the same way as arguments are passed between functions. Up to now, the `main()` statements in our programs have had nothing between the parentheses. However, UNIX actually makes available to the program (whether the programmer chooses to use the information or not) two arguments to `main`: an *array of character strings*, conventionally called `argv`, and an *integer*, usually called `argc`, which specifies the number of strings in that array. The full statement of the first line of the program is

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

(The syntax `char** argv` declares argv to be a pointer to a pointer to a character, that is, a pointer to a character array (a character string)--in other words, an array of character strings. You could also write this as `char*`

`argv[]`. Don't worry too much about the details of the syntax, however--the use of the array will be made clearer below.)

When you run a program, the array `argv` contains, in order, *all* the information on the command line when you entered the command (strings are delineated by whitespace), *including the command itself*. The integer `argc` gives the total number of strings, and is therefore equal to equal to the number of arguments *plus one*. For example, if you typed

```
a.out -i 2 -g -x 3 4
```

the program would receive

```
argc = 7
argv[0] = "a.out"
argv[1] = "-i"
argv[2] = "2"
argv[3] = "-g"
argv[4] = "-x"
argv[5] = "3"
argv[6] = "4"
```

Note that the arguments, even the numeric ones, are all *strings* at this point. It is the programmer's job to decode them and decide what to do with them.

The following program simply prints out its own name and arguments:

```
#include < stdio.h>

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

    printf("argc = %d\n", argc);

    for (i = 0; i < argc; i++)
        printf("argv[%d] = \"%s\"\n", i, argv[i]);
}
```

UNIX programmers have certain conventions about how to interpret the argument list. They are by no means mandatory, but it will make your program easier for others to use and understand if you stick to them. First, switches and key terms are always preceded by a ``-" character. This makes them easy to recognize as you loop through the argument list. Then, depending on the switch, the next arguments may contain information to be interpreted as integers, floats, or just kept as character strings. With these conventions, the most common way to ``parse" the argument list is with a `for` loop and a `switch` statement, as follows:

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

main(int argc, char** argv)
{
    /* Set defaults for all parameters: */

    int a_value = 0;
    float b_value = 0.0;
    char* c_value = NULL;
    int d1_value = 0, d2_value = 0;

    int i;

    /* Start at i = 1 to skip the command name. */

    for (i = 1; i < argc; i++) {
```

```
        /* Check for a switch (leading "-"). */

        if (argv[i][0] == '-') {

            /* Use the next character to decide what to do. */

            switch (argv[i][1]) {

                case 'a':       a_value = atoi(argv[++i]);
                                break;

                case 'b':       b_value = atof(argv[++i]);
                                break;

                case 'c':       c_value = argv[++i];
                                break;

                case 'd':       d1_value = atoi(argv[++i]);
                                d2_value = atoi(argv[++i]);
                                break;

            }
        }
    }

    printf("a = %d\n", a_value);
    printf("b = %f\n", b_value);
    if (c_value != NULL) printf("c = \"%s\"\n", c_value);
    printf("d1 = %d, d2 = %d\n", d1_value, d2_value);
}
```

Note that `argv[i][j]` means the `j`-th character of the `i`-th character string. The `if` statement checks for a leading ``-'' (character 0), then the `switch` statement allows various courses of action to be taken depending on the next character in the string (character 1 here). Note the use of `argv[++i]` to increase `i` before use, allowing us to access the next string in a single compact statement. The functions `atoi` and `atof` are defined in `stdlib.h`. They convert from character strings to `int`s and `double`s, respectively.

A typical command line might be:

```
        a.out -a 3 -b 5.6 -c "I am a string" -d 222 111
```

(The use of double quotes with `-c` here makes sure that the shell treats the entire string, including the spaces, as a single object.)

Arbitrarily complex command lines can be handled in this way. Finally, here's a simple program showing how to place parsing statements in a separate function whose purpose is to interpret the command line and set the values of its arguments:

```
                    /*******************************/
                    /*                             */
                    /*   Getting arguments from    */
                    /*                             */
                    /*      the Command Line       */
                    /*                             */
                    /*******************************/


                                /* Steve McMillan       */
                                /* Written: Winter 1995 */
```

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

void get_args(int argc, char** argv, int* a_value, float* b_value)
{
    int i;

    /* Start at i = 1 to skip the command name. */

    for (i = 1; i < argc; i++) {

        /* Check for a switch (leading "-"). */

        if (argv[i][0] == '-') {

            /* Use the next character to decide what to do. */

            switch (argv[i][1]) {

                case 'a':       *a_value = atoi(argv[++i]);
                                break;

                case 'b':       *b_value = atof(argv[++i]);
                                break;

                default:        fprintf(stderr,
                                "Unknown switch %s\n", argv[i]);
            }
        }
    }
}

main(int argc, char** argv)
{
    /* Set defaults for all parameters: */

    int a = 0;
    float b = 0.0;

    get_args(argc, argv, &a, &b);

    printf("a = %d\n", a);
    printf("b = %f\n", b);
}
```

# 12. Graphical Interfaces: Dialog Boxes

Suppose you don't want to deal with command line interpretation, but you still want your program to be able to change the values of certain variables in an interactive way. You could simply program in a series `printf/scanf` lines to quiz the user about their preferences:

```
.
.
.
```

```
        printf("Please enter the value of n: ");
        scanf("%d", &n);

        printf("Please enter the value of x: ");
        scanf("%f", &x);

        .
        .
        .
```

and so on, but this won't work well if your program is to be used as part of a pipeline (see the UNIX primer), for example using ther graphics program `plot_data`, since the questions and answers will get mixed up with the data stream.

A convenient alternative is to use a simple graphical interface which generates a *dialog box*, offering you the option of varying key parameters in your program. Our graphics package provides a number of easy-to-use tools for constructing and using such boxes. The simplest way to set the integer variable `n` and the float variable `x` (i.e. to perform the same effect as the above lines of code) using a dialog box is as follows:

```
/* Simple program to illustrate use of a dialog box */

main()
{
    /* Define default values: */

    int n = 0;
    float x = 0.0;

    /* Define contents of dialog window */

    create_int_dialog_entry("n", &n);
    create_float_dialog_entry("x", &x);

    /* Create window with name "Setup" and top-left corner at (0,0) */

    set_up_dialog("Setup", 0, 0);

    /* Display the window and read the results */

    read_dialog_window();

    /* Print out the new values */

    printf("n = %d, x = %f\n", n, x);
}
```

Compile this program using the alias `Cgfx` (see the page on compilation) to link in all necessary libraries.

The two `create` lines define the entries in the box and the variables to be associated with them, `set_up_dialog` names the box and defines its location. Finally, `read_dialog_window` pops up a window and allows you to change the values of the variables. When the program runs, you will see a box that looks something like this:

Modify the numbers shown, click "OK" (or just hit carriage return), and the changes are made. That's all there is to it! The great advantage of this approach is that it operates *independently* of the flow of data through `stdin/stdout`. In principle, you could even control the operation of every stage in a pipeline of many chained commands, using a separate dialog box for each.