Chapter: 5

Functions in C

Details about: Prototypes, Recursion & Parameters, const, volatile, class-specifire, function-pointer.

5.1 The PROTOTYPE

The general form of a function prototype is shown here:

```
type function_name( type parameter_name1,
                    type parameter_name2,
                    . . . . .
                    type parameter_nameN);
```

A prototype declares three attributes associated with a function:

1. Its ***return type***.
2. The ***number of*** its ***parameters***.
3. The ***type of*** its ***parameters***.

Arguments and parameter of a function: Let ***`average(int a, int b, int c)`*** is a function. In the ***declaration*** of this function i.e. ***declaring*** its ***prototype*** , eg: ***`int average(int a, int b, int c) ;`*** : the variable list with their data types ***"`int a, int b, int c`"*** is the parameter list. Parameter lists is also used in function definition (we discuss function definition later, definition and declaration of a function are not same, but ***declaration/prototype declaration*** can be replaced by ***definition*** in small program cases).
Arguments of a function are the variable list (or direct data list) which is used in calling the function . Eg: ***`average(x, y, z)`*** or ***`average(2, 4, 5)`*** in function arguments no data type declaration is required.

5.1.1 what a prototype does ?

[1] Prototypes inform the compiler about the return type of a function. When a function is called, the compiler needs to know the type of data returned by that function so that it can generate the proper code to handle that data. Because : different data types have different sizes.

(a) If you use a function that is ***not prototyped***, then the compiler will simply assume that it is ***returning an integer***.

(b) However, If it is ***returning some other type***, an ***error will occur***. If the function is in the ***same file*** as the rest of your program, then the compiler will catch this error.

(c) But if the function is in another file or a library, then the error will go uncaught-and this will lead to trouble when the program is executed.

[2] Prototypes allow the compiler to find and report illegal type conversions between the type of arguments used to call a function and the type definition of its parameters. It is important to understand, however, that not all kinds of type conversions are illegal in a function call. In fact, C automatically converts most types of arguments into the type of data specified by the parameter. But a few type conversions are inherently wrong. For example, you cannot convert an integer into a pointer. A ***function prototype allows the compiler to catch and return this type of conversion error.***

[3] Prototypes also enable the compiler to report when the number of arguments passed to a function is not the same as the number of parameters declared by the function. In the absence of a function prototype, it is not syntactically wrong to call a function with incompatible arguments or with more or less arguments than the function has parameters. However these are wrong even though the compiler may accept. The use of a function ***prototype prevents these errors by enabling the compiler to find them***.

5.1.2 Problems when prototype is not present/ OLD-Fashioned FORWARD DECLARATION:

Although the old-style forward declaration is no longer used in new code, you will still find it quite frequently in older programs. In early versions of C, before prototypes were invented, it was still necessary to tell the compiler about the return type of a function. This was done using a forward declaration. A forward declaration is essentially truncated form of a prototype that declares only the return type of a function-not the type and number of its parameters. Compilers still allow this type of declarations.
The following program demonstrates an old-style forward declaration, It uses it to inform the compiler of ***`volume()`***'s return type.

```
#include <stdio.h>
double volume();     /* old-style forward declaration for volume() */

        int main(void){     double vol;
                            vol = volume(12.2, 5.67. 9.03) ; printf(“Volume: %f" vol);
                            return 0;}

                        /* Compute the volume of a cube. */
double volume(double s1, double s2, double s3) {return s1*s2*s3;}
```

Since the old-style declaration does not inform the compiler about any of volume( )'s parameters it is not a function prototype. Instead, it simply states vo1tifne( )'§ return type.

The problem/trouble is that the lack of a full prototype will allow volume( ) to be called using an incorrect type and/or number of arguments For example, the following ***will not generate a compiler error message even though it is wrong***.

```
        volume(120.2, 99.3);        /* missing last argument */
```

Since the compiler has not been given information about volume( ) s parameters (no prototype declaration) it won't catch the fact that this call is wrong.

5.1.3 Two Resolved Issues when function prototypes were added to C:

[1] The ***first issue*** was how to handle the old-style forward declaration, which does not use a parameter list. To do so,

(a) The ANSI C standard specifies that when a function declaration occurs without a parameter list, nothing whatsoever is being said about the parameters to the function. It might have parameters, it might not.

(b) How to prototype a function that takes no arguments? For example, this function simply outputs a line of periods:

```
void line (){ int i; for(i=0; i<80; i++) printf("."); }
```

If you try to use the ***void line ()*** as a prototype, it won't work because the compiler will think that as a old-style declaration method.

The solution to this problem is through the use of the ***void*** keyword, when a function has no parameters, its prototype uses ***void*** inside the ***parentheses***. For example, here is ***line( )***'s proper prototype: ***void line(void);***

This explicitly ***tells the compiler that the function has no parameters, and any call to that function that has parameters is an error***. You must make sure to ***also use void when the function is defined (definition of function discussed later)***. For example, ***line( )*** must look like this:

```
void line (void){   int i; for(i=0; i<80; i++) printf("."); }
```

[2] The second issue related to prototyping is the way it affects C's **automatic type promotions**. Because when a non-prototyped function is called, all integral promotions take place (for example, ***characters*** are converted to ***integers*** and all ***floats*** are converted to ***doubles***). However, these type promotions seem to violate the purpose of the prototype. The resolution to this problem is that ***when a prototype exists, the types specified in the prototype are maintained, and no type promotions will occur***.

5.1.4 Variable Length Argument in C

Variable length argument is a feature that allows a function to receive any number of arguments. There are situations where we want a function to handle variable number of arguments according to requirement. Variable number of arguments are represented by ***three dotes*** (**. . .**). It requires the header file ***STDARG.H*** .

Variable length argument lists relates to prototypes. **We won't be creating any functions in this book that use a variable number of arguments because they require the use of some advanced techniques**. It is sometimes quite useful. For example, both ***printf( )*** and ***seanf( )*** accept a variable number of arguments. To specify a variable number of arguments, use **...** in the prototype. For example:

```
int myfunc(int a, ...);
```

specifies a function that has one integer parameter and a variable number of other parameters.

5.1.5 DEFINITION & DECLARATION of a Function

In C programming there has been a long-standing confusion about the usage of two terms: ***declaration*** and ***definition***. A declaration specifies the type of an object. A definition causes storage for an object to be created. As these terms relate to functions, **a prototype is a declaration**. The **function, itself, which contains the body of the function is a definition**.

- In C, it is also legal to fully **define a function prior to its first use**, thus eliminating the need for a separate prototype. That is if a function is defined before it is called, it **does not require a separate prototype**. However, this works only in very small programs. For example :

```
include<stdio.h>
void line (void){   int i; for(i=0; i<80; i++) printf("."); } /*No PROTOTYPE is required*/
int main(void){ line(); return 0;}
```

  That's why some people define a function before ***main()*** to get rid of prototype declaration. Also its easy to renew the old fashion program by simply copying the defined functions and paste them before ***main()***.

- Remember that if a function does not return a value, then its return type should be specified as ***void***-both in its definition and in its prototype .

5.1.6 why prototypes are important ?

[1] Function prototypes enable you to write better, more reliable programs because they help ensure that the functions in your programs are being called with correct types and numbers of arguments.
[2] Fully prototyped programs are the norm and represent the current state of the art of C programming. Frankly, no professional C programmer today would write programs without them.
[3] Also, future versions of the ANSI C standard may mandate function prototypes and c++ requires them now.

NOTES

[1] As you know, the standard library function ***sqrt( )*** returns a double value. You might be wondering how the compiler knows this. The answer is that ***sqrt( )*** is prototyped in its header file ***MATH.H***. To see the importance of using the header file, try to program consisting ***sqrt()*** without its header file.

[2] In general, each of C's *standard library functions has its prototype specified in a header file*. For example, ***printf()*** and ***scanf()*** have their prototypes in ***STDIO.H***. This is one of the reasons that ***it is important to include the appropriate header file for each library function you use.***

[3] Some ***'character-based'***-functions have **a return type of *int*** rather than ***char***. For example, the ***getchar( )*** function's return type is ***int***, not ***char***. The reason for this is found in the fact that C very cleanly handles the **conversion of characters to integers and integers back to characters**. There is no loss of information. For example, the following program is perfectly valid:

```
#include <stdio.h>
int char_as_int(void);     /*int type return value instead of char type*/
```

```
        int main (void){    char ch;
                            ch = char_as_int(); printf("%c", ch);
                            return 0;    }
        int char_as_int(void){     return 'a';   }
```

When ***char_as_int()*** returns, it elevates the ***character 'a'*** to an ***integer*** by adding a high-order byte (or bytes) containing zeros. When this value is assigned to ***ch*** in ***main( )***, the high-order byte (or bytes) is removed. **One reason to declare functions like *char_as_int()* as returning an integer instead of a character is to allow various error values to be returned that are intentionally outside the range of a char.**

[4] When a function returns a pointer, both the function and its prototype must declare the same pointer return type.

```
#include <stdio.h>
int *in_it(int x);  /*prototype of pointer returning function */
int count;          /*count as global variable*/
int main(void){     int *p; /*global variable count is int type so p is int type pointer*/
                    p = in_it(110);     /* return pointer equal : p=&count*/
                    printf("count (through p) is %d", *p);
                    return 0;}
int * in_it(int x){ count = x;
                    return &count;      /* return a pointer */     }
```

Prints = count (through p) is 110

the function ***in_it( )*** returns a pointer to the global variable count. Notice the way that the return type for ***in_it( )*** is specified. This same ***general form is used for any sort of pointer return type***.

[5] If a function returns a pointer, then it must make sure that the object being pointed to does not go out-of-scope when the function returns. This means that you must not return pointers to local variables.

[6] The ***main( )*** function does not have (nor does it require) a ***prototype***. This allows you to define ***main( )*** any way that is supported by your ***compiler***. Here we use ***int main(void) { ... }*** because it is one of the most common forms. Another frequently used form of ***main( )*** is

void main(void) { ... }

This form is used when no value is returned by main( ).

- The reason ***main( )*** does not have a ***prototype*** is to allow C to be used in the widest ***variety of environments***. Since the precise conditions present at program start-up and what actions must occur at program termination may **differ widely from one operating system to the next**, C allows the acceptable forms of ***main( )*** to be determined by the ***compiler***. However, **nearly all compilers will accept *int main(void)*** and ***void main(void)***.

5.2 Recursion

Recursion is the process by which *something is defined in terms of itself*. When applied to computer languages, ***recursion means that a function can call itself***. Not all computer languages support recursive functions, but C does.

A very simple example of recursion is shown in this program:

```
#include <stdio.h>
void recurse(int i);
int main(void){ recurse(0) ;    return 0;}

void recurse(int i){
    if(i<10){               /* Condition to control recursion */
        recurse(i+1);       /* recursive call */
        printf(" %d ", i);
        }
}
```

This program prints ***9 8 7 6 5 4 3 2 1 0*** on the screen.
Let's see why.

[1] The ***recurse()*** function is first called with ***0***. This is ***recurse()***'s first activation. Since ***0 < 10***, ***recurse()*** ,then calls itself with the value of ***(i = 0) + 1***,

[2] This is the second activation of ***recurse()*** and ***i = 1***. Since ***1<10***, this causes ***recurse()*** to be called again using the value ***(i=1) + 1***.

[3] This process repeats until ***recurse()*** is called with the value ***10***. Since 10 ≮ 10 This causes ***recurse()*** to return (and stops recursion). Since it ***returns*** to the point of its call, it will execute the ***printf)*** statement in its previous activation (i.e. 9th activation), print ***9***, and return.

[4] This, then, ***returns*** to the point of its ***call*** in the previous activation (8th activation), which causes print ***8***.

[5] The process continues until all the calls ***return***, and the program ***terminates***. We can assemble this in following stages :
The recursion starts at i=0, from call recurse(0); The fuction is then activated

- ⇨ ***Stage 0*** : condition check : i=0<10. Call **recurse()** with (i=0)+1 i.e. **recurse(1)** is called.
- ⇨ ***Stage 1*** : condition check : i=1<10. Call **recurse()** with (i=1)+1 i.e. **recurse(2)** is called.
- ⇨ ***Stage 3*** : condition check : i=2<10. Call **recurse()** with (i=2)+1 i.e. **recurse(3)** is called.

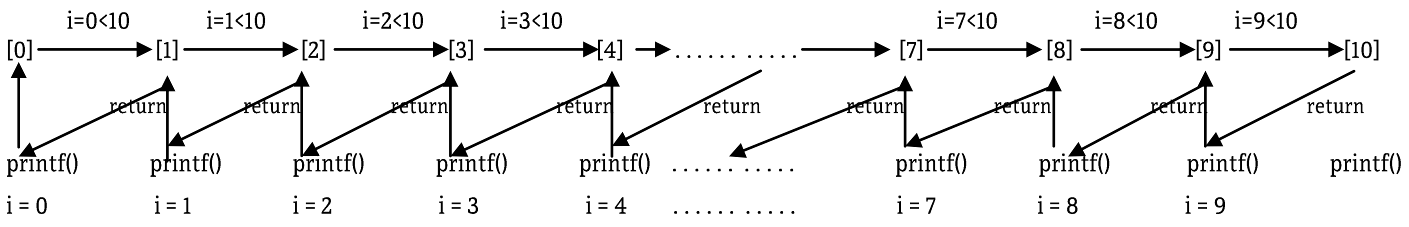
....

. . . .

⇨ ***Stage 9*** : condition check : `i=9<10`. Call **`recurse()`** with `(i=9)+1` i.e. **`recurse(10)`** is called.

⇨ ***Stage 10*** : condition check : `i=10<10` is false and condition fails. **`recurse()`** stopped. Function returns to ***stage 9***.

After returning to ***stage 9*** ,the next statement after the function calls executed i.e. prints ***i=9*** using ***`printf().`*** The ***stage 9*** is now complete. It then returns to ***stage 8*** and similarly prints ***i=8***. This process repeats until ***stage 0*** is complete. The function then returns to ***`main()`*** and reach to the end of the program.



- We need to remember one point that, the ***value of i*** in each stage is stored before the ***recursive call***. And the numbers displayed 9 8 7 6 5 .... 0.
- The reason for that is : the ***`printf()`*** is ***appeared after*** the ***`recursive-call-point`***, in this case all recursive calls need to be done first and then proceed to ***`printf()`*** for each corresponding stage.
- However, if ***`printf()`*** appears before the ***`recursive-call-point`*** it could display : 0 1 2 3 .... 9. As given below:

```
void recurse(int i){
    if(i<10){              /* Condition to control recursion */
        printf(" %d ", i); /* printf() appears before the recursive-call-point */
        recurse(i+1);      /* recursive call */

        }
}
```

<u>Recursion and control statement :</u> Recursion is essentially a new type of ***program control mechanism*** (i.e. it can be used as a ***control statement***). This is why every recursive function you write will have a ***conditional statement*** that controls whether the function will call itself again or return. ***Without*** such a (***conditional***) statement, a recursive function will simply ***run wild***, using up all the memory allocated to the stack and then crashing the program.

<u>Usage of recursion :</u> Recursion can be quite useful in simplifying certain ***algorithms***. For example, the ***Quick-sort*** sorting algorithm is difficult to implement without the use of recursion. If you are new to programming in general, you might find yourself uncomfortable with recursion.

NOTE

[1] It is important to understand that there are not multiple copies of a recursive function. Instead, only one copy exists. When a function is called, ***storage for its parameters and local data are allocated on the stack***. Thus, when a function is called ***recursively***, the function begins executing with a new set of parameters and local variables, ***but the code that constitutes the function remains the same.***

[2] ***<u>Stack :</u>*** A stack is a data structure which is used to store data in a particular way. Two operations can be performed on a stack : ***push*** operation which ***inserts*** an element into the stack. ***Pop*** operation which ***removes*** the last element that was added into the stack. It follows ***last in first out*** (***LIFO***) order

5.3 Parameters Advanced

<u>Parameter and pointer, relation to arguments :</u> For computer languages in general, a subroutine (function) can be passed arguments in one of two ways.

- The first is called ***call by value*** and there is no relation between ***parameters*** and ***arguments***.
    - This method ***copies*** the value of an argument into the formal parameter of the subroutine.
    - Therefore, changes (Eg: interchange for swap) made to a ***parameter*** of the ***subroutine*** have no effect on the ***argument*** used to call it.
    - By ***default***, C uses call by value to pass arguments. This means that you ***cannot alter*** the ***arguments*** used in a call to a function (Eg: cannot ***swap***). What occurs to a *parameter inside the function* will have ***no effect*** on the *argument outside the function.*
- The second way a subroutine can have ***arguments passed*** to ***it*** is through can by ***reference*** (i.e. using ***pointers***), in this method ***parameters*** and ***arguments*** are ***related*** to each others and ***data-types*** of arguments and parameter had to be ***same***.
    - In this method, the ***address*** of an ***argument*** is ***copied*** into the ***parameter***. Inside the ***subroutine***, the address is used to access the ***actual argument***. This means that ***changes*** made ***to*** the ***parameter*** will ***affect*** the, ***argument***.
    - It is possible to manually construct a call by reference by passing a pointer to an argument. Since this causes the address of the argument to be passed, it then is possible to change the value of the argument outside the function.

The classic ***example*** of a ***call-by-reference*** function is ***swap( )***. It exchanges the value of its two ***integer arguments***.

```
#include <stdio.h>
void swap(int *i, int *j);
int main (void){int num1 , num2;
                num1 = 100;
                num2 = 800;
                printf("Before swap, num1: %d , num2: %d \n ", num1, num2 );
                swap(&num1, &num2);    /* Function call by location */
                printf("After swap, num1: %d , num2: %d \n ", num1, num2 );
        return 0;}

/* Exchange the values pointed to by two integer pointers . */
void swap(int *i, int *j){  int temp;
                            temp = *i;
                            *i = *j;
                            *j = temp;}
```

Since pointers to the two integers are passed to the function, the actual values pointed to by the ***arguments*** are ***exchanged***.

Parameter and array: We know that when an ***array*** is used as an ***argument*** to a function, only the *address of the array* is passed, not a copy of the *entire array*, which implies ***call-by-reference***. This means that the *parameter declaration* must be of a ***compatible*** *pointer type*. There are three ways to declare a parameter that is to receive a pointer to an array.

1. First, the ***parameter*** may be ***declared*** as an *array of the same* ***type*** *and* ***size*** as that used to call the function.
2. Second, it may be specified as an ***unsized array***.
3. Finally, and most commonly, it may be specified as a ***pointer*** to the ***base type*** of the ***array***.

The following program demonstrates all three methods:

```
#include <stdio.h>
void f1(int num[5]), f2(int num[]), f3(int *num);
int main (void){    int count[5] = {1, 2, 3, 4, 5};
                    f1(count);  f2(count);  f3(count);      /* All three function call */
                    return 0;}

/* parameter specified as array */
void f1(int num[5]){    int i;
                        for(i=0; i<5; i++) printf("%d ", num[i]); }

/* parameter specified as unsized array */
void f2(int num[]){ int i;
                    for(i=0; i<5; i++) printf("%d ", num[i]); }

/* parameter specified as pointer */
void f3(int *num){  int i;
                    for(i=0; i<5; i++) printf ("%d ", num[i]); }
```

Even though the three methods of declaring a parameter that will receive a pointer to an array look different, they all result in a pointer parameter being created.

Example : Following uses the function **prompt()** to display a prompting message and then to read a number entered by the user.

```
#include <stdio.h>
void prompt(char *msg, int *num); /* header function like declaration */
int main(void){ int i;
                prompt("Enter a num: ", &i);
                printf("\n Your number is: %d", i);
                return 0;}

void prompt(char *msg, int *num){   printf (msg);
                                    scanf("%d", num);}
```

Because the parameter num is already a pointer, you do not need to precede it with an ***&*** in the call to ***scanf( )*** (we used "***&***" in ***scanf()*** in all previous example and compare with the example of ***swap()***). In fact, it would be an error to do so. And in ***printf()*** we used the third way described in "parameter ad array" .

NOTE : ***pointer-address*** Mathematical formulation :

$$\begin{aligned} &pointer\ p\ =\ address\ of\ q \\ &\Rightarrow p\ =\ \&q \\ &\Rightarrow *p\ = *(\&q) \\ &\Rightarrow *p = q,\quad \therefore *p = value\ of\ q \end{aligned}$$

5.4 Pass Arguments To Main()

Command-line argument : Many programs allow ***command-line arguments*** to be specified when they are run. A ***command-line argument*** is the information that follows the *program's name* on the *command line* of the *operating system*. ***Command-line arguments*** are used to pass information into a program.

For example, when you use a text editor, you probably specify the name of the file you want to edit after the name of the text editor. Assuming you use a text editor called ***EDTEXT***, then this line causes the file ***TEST*** to be edited.

```
EDTEXT TEST
```

Here, ***TEST*** is a *command-line argument*.

C *command-line arguments* are passed to a C program through two arguments (or parameters) to the ***main()*** function. The parameters are called ***argc*** and ***argv***.

argc : The ***argc*** parameter *holds the number of arguments on the command-line* and is an ***integer***. It will always be at least ***1*** because the *name of the program qualifies as the first argument*.

argv : The argv parameter is an array of string pointers. The most common method for declaring argv is shown here:

```
char *argv [] ;
```

- ⇨ The empty brackets indicate that it is an array of ***undetermined*** length.
- ⇨ All *command-line arguments* are passed to ***main()*** as strings . To access an individual string, *index **argv***. For example, ***argv[0]*** points to the *program's name* and ***argv[1]*** points to the *first argument*.

This program displays all the command-line arguments that are present when it is executed.

```
#include <stdio.h>
int main(int argc, char *argv[]){ int i;
                                  for(i=1; i<argc; i++) printf("%s", argv[i]);
                                  return 0;}
```

NOTE

[1] You can use ***windows powershell*** instead of ***windows command line***.

[2] C does not specify what constitutes a *command-line argument*, because operating systems vary considerably on this point. However, the most common convention is as follows: Each *command-line argument* must be ***separated*** by a ***space*** or a ***tab*** character. "*Commas*", "*semicolons*", and the like are *not considered separators*. For example,

This is a test is made up of ***four strings***, but ***this,that,and,another*** is ***one string***.

[3] If you need to pass a *command-line argument* that does ***contain spaces***, you must place it ***between quotes***, as shown in this example: ***"This is a test"***

[4] The names of ***argv*** and ***argc*** are ***arbitrary***-you can use any names you like. It is a good idea to use these names so that anyone reading your program can quickly identify them as the *command-line parameters*.

[5] The ***ANSI C*** standard only defines the ***argc*** and ***argv*** parameters. However, your *compiler may allow additional parameters* to ***main().***

[6] ***New functions :*** When you pass ***numeric*** data to a program, that data will be received in its ***string*** form. Your program will need to convert it into the *proper internal format* using one or another of *C's standard library functions*. The most *common conversion functions are shown here*, using their prototypes:

```
int atoi(char *str);
double atof(char *str);
long atol(char *str);
```

These functions use the ***STDLIB.H*** header file.

- ⇨ The ***atoi()*** function returns the *int equivalent* of its *string argument*.
- ⇨ The ***atof()*** returns the *double equivalent* of its *string argument*,
- ⇨ The ***atol()*** returns the *long equivalent* of its *string argument*.

If you call one of these functions with a string that is not a valid number, *zero will be returned*.

The following program demonstrates these functions. To use it, enter an ***integer***, a ***long integer*** and a ***floating-point*** number on the command line It will then redisplay them on the screen.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){   int i; double d; long l;
                                    i = atoi(argv[1]);
                                    l = atol(argv[2]);
                                    d = atof(argv[3]);
                                    printf("%d   %ld   %f", i, l, d);
                               return 0;}
```

New function : *When some condition necessary for a program's execution has not been met*, most C programmers call the standard library function ***exit()*** to ***terminate*** the program. The ***exit()*** function has following prototype and uses the ***STDLIB.H*** header file.:

```
void exit(int return-code);
```

When ***exit()*** terminates the program, it returns the value of ***return-code*** to the operating system . By convention, most operating systems use *a return code of zero to mean that a program has terminated normally. Nonzero values indicate abnormal termination.*

5.5 Old-Style Parameter Declarations

The original parameter declaration method is now called the ***old-style*** or ***classic form***. The general form of the ***old-style*** parameter declaration is shown here:

```
type function-name(parameter_1, parameter_2,. . . . . ., parameter_N)
type parameter_1;
type parameter_2;
. . . .
. . . .
type parameter_N;
              {
              FUNCTION-CODE
              }
```

Notice that the declaration is divided into ***two parts***:

- ▶ Within the parentheses, only the names of the parameters are specified.
- ▶ Outside the parentheses, the types and names are specified.

For example, given the following modern declaration and then old-style

```
float f(char ch, long size, double max){ . . function code . . }
```

the equivalent old-style declaration is

```
float f(ch, size, max)
char ch;
long size;
double max;
       { . . function code . . }
```

NOTE

☑ One advantage of the ***old-style*** declaration is that you can specify more than one parameter after the type name (sometimes you find it quite useful). For example, this is perfectly valid:

```
myfunc (i. j. k)
int i, j, k;
      { . . function code . . }
```

☑ The ***ANSI C*** standard specifies that either the old-style or the modern declaration form may be used. The reason for this is to maintain compatibility with older C programs.

5.6 variable storage class specifire (Advanced topic)

Variables in C not only have ***data types***, they also have a ***storage class***. There are four types of ***variable storage class*** :

[1] ***Automatic*** variables, [2] ***External*** variables, [3] ***Register*** variables, [4] ***Static*** variables.

For the four different ***variable storage class***, C defines four type ***modifiers*** that affect how a variable is stored. They are

[1] auto ***[2] extern*** ***[3] register*** ***[4] static***

These specifiers ***precede*** the ***type*** name. That is

`storage_class data_type variable_name ;` Example : `extern int count;`

[1] ***auto :*** The specifier ***auto*** is completely unnecessary. Automatic variables are simply ***local*** variables, which are ***auto*** by ***default***.

[2] ***extern :*** As the size of a program grows, it takes longer to compile. For this reason, C allows us to break a program into two or more files. We can separately compile these files and then link them together (the actual method of separate compilation and linking will be explained in the instructions that accompany your compiler).

- ◘ Global data may need to be accessed by two or more files that form a program. But global data can only be defined once.
- ◘ In this case, each source file must inform the compiler about the global data it uses. To accomplish this you will need to use the keyword extern.

  To understand why, consider the following program, which is split between two files:

```
/* FILE #1 */
#include <stdio.h>
int count;
void f1 (void) ;
int main(void){
  int i;
  f1(); /* set count's value */
  for(i=0; i<count; i++) printf("%d", i);
return 0;}
```

```
/* FILE #2 */
void f1(void){
      extern int count; /* Global count */
      count = 17 ;}
```

- ◘ By placing ***extern*** in front of **count's** declaration in ***FILE #2***, you are telling the compiler that count is an integer defined ***elsewhere***. In other words, ***using extern informs the compiler about the existence and the type of the variable it precedes, but it does not cause storage for that variable.***

☞ To link multiple files in ***code::block***, use project, create new ***project*** (console), put the ***source code*** files in the project name folder. Compile, run the whole ***.cbp*** file.

If we didn't use extern two ***errors*** would occur :
Error 1 : If we directly use : `/* FILE #2 */ void f1(void){count = 17 ;}` an error will be reported because count is ***not defined***.
Error 2 : If we even define count as `/* FILE #2 */ void f1(void){ int count; count = 17 ;}` many ***linkers*** will report a ***duplicate-symbol*** error, which means that count is defined twice, and the linker doesn't know which to use.

The solution to these problem is C's ***extern*** specifier.

- ***Use extern in same file to mention the global variable :*** We can use ***extern*** inside a function to declare a ***global*** variable defined elsewhere in the same file (i.e. to mention that it is *global* variable so that it won't be treated as a *local* variable). ***But it is rarely used.*** *Because, whenever the compiler encounters a variable name not defined by the function as a local variable, it assumes that it is* ***global.*** For example, the following is valid:

```
#include <stdio.h>
int count;
int main(void){     extern int count; /* this refers to global count *1
                    count = 10;
                    printf("%d", count);
                    return 0;}
```

[3] ***register :*** When you specify a ***register*** variable you are telling the compiler that you want access to that variable to be as fast as possible. In early version of C it causes the variables to be held in a ***register*** of the ***CPU***. (This is how the name ***register*** came about.) By using a ***register*** of the ***CPU***, extremely ***fast access times*** are achieved.

- In modern versions of C, the requirement that register variables must be held in a ***CPU register*** was ***removed***. Instead, the ANSI C standard stipulates that a register variable will be stored in such a way as ***to minimize access time***. This means that,
  - ***register*** variables of type ***int*** and ***char*** continue to be held in a ***CPU register.***
  - No matter what storage method is used, only so many variables can be granted the fastest possible access time.

For example, the CPU has a limited number of ***registers***. When fast-access locations are exhausted, the compiler is free to make register variables into regular variables.
You must choose carefully which variables you modify with register. One good choice is to make a *frequently used variable*, such as the *variable that controls a loop*, into a ***register*** variable. The more times a variable is accessed, the greater the increase in performance when its access time is decreased. Generally, you can assume that at least two variables per function can be truly optimized for access speed.

- ***Restriction of using pointes :*** Because a register variable may be stored in a register of the CPU, it may not have a memory address. This means that you **cannot use the & to find the address** of a ***register variable*** and cannot use ***pointers***.

[4] ***static :*** Recall 3.2 how a local variable act : a ***local*** variable is created upon entry into its function and ***destroyed*** upon ***exit***. And a ***global*** variable is ***active*** in whole program.

- ***Use of static modifier on local variables :*** When you use the ***static*** modifier, you cause the contents of a ***local*** variable to be **preserved between function calls** (That is it will not destroy when function exit and it act like a global variable inside its own function).

Unlike normal ***local*** variables, which are ***initialized*** each time a function is entered, a ***static local*** variable is ***initialized*** only ***once***.
For example, take a look at this program,

```
#include <stdio.h>
void f(void);
int main(void){ int i;
                for(i=0; i<10; i++) f();
                return 0;}

void f(void){   static int count = 0;
                count++;
                printf("count is : %d  ", count);}
```

which displays the following output:
`count is : 1 count is : 2 . . . .  count is : 10`. Here count retains its value between function calls.
but without static modifier int count = 0; gives the result : ***`count is : 1 count is : 1 . . . .  count is : 1`***.
Which is obvious and we explained in 3.2 in chapter 3.

So why didn't we use a global variable in the previous example? The answer is : The advantage to using a ***static local variable*** over a ***global*** one is that the ***static local*** variable is still known to and accessible by ***only the function*** (*global* variables active outside any function) in which it is declared.

- ***Use of static modifier on global variables :*** The ***static*** modifier may also be used on ***global*** variables (more like a ***local*** variable for its own file). It causes the ***global*** variable to be known to and accessible by only the ***functions in the same file*** in which it is declared. This static global variable has **no effect on other file** (when we work with different files) which use the same named global variable.

This means that ***there are no name conflicts if a static global variable in one file has the same name as another global variable in a different file of the same program.***
For example, consider these two fragments, which are parts of the same program:

```
FilE #1                                       FilE #2
int count;                                    static int count;
. . .                                         . . .
. . .                                         . . .
count = 10;                                   count = 5:
printf("%d", count);                          printf("%d", count);
```

Because count is declared as ***static*** in ***FILE #2***, no name conflicts arise. The ***printf()*** statement in ***FILE #1*** displays ***10*** and the ***printf()*** statement in ***FILE #2*** displays ***5*** because the two counts are ***different variables***.

5.7 Access modifiers : const and volatile (Advanced topic)

C includes two ***type modifiers*** that affect the way variables are accessed by both your program and the compiler. These ***modifiers*** are ***const*** and ***volatile***.

▶ ***const :*** If you precede a variable's type with const, you prevent that variable from being modified by your program. The variable may be given an initial value, however, through the use of an initialization when it is declared. The compiler is free to locate const variables in ROM (read-only memory) in environments that support it. (A const variable may also have its value changed by hardware dependent means.)

Example : The following short program shows how a const variable can be given an initial value and be used in the program, as long as it is not on the left side of an assignment statement .

```
#include <stdio.h>
int main(void) {    const int i = 10;
                    printf ( "%d", i);  /* this is OK */
                  return 0; }
```

The ***following*** program tries to assign ***i*** another value. This program will ***not compile***.

```
#include <stdio.h>
int main(void) {    const int i = 10;
                    i=20;               /* this is wrong */
                    printf ( "%d", i);
                  return 0; }
```

☞ The const modifier can prevent a function from modifying the object that a parameter points to. That is, when a ***pointer parameter*** is preceded by ***const***, **no statement in the function can modify the variable pointed to by that parameter**.

☞ The most important feature of const pointer parameters is that they guarantee that many standard library functions will not modify the variables pointed to by their parameters.

Example : The following program shows how a pointer parameter can be declared as const to prevent the object it points to from being modified.

```
#include<stdio.h>

void pr_str(const char *p);

int main(void){  char str[80];
                    printf("Enter a string: "); gets(str); /*inputs a sting to str[80]*/
                    pr_str(str) ;          /*function call*/
                    return 0;}

void pr_str(const char *p){
                    while(*p) putchar(*p++);      /* this is ok */
                    }
```

This program is ok and compile with no error, but following will not compile. Because this version attempts to alter (change to upper-case) the string pointed to by p.

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

void pr_str(const char *p);

int main(void){     char str[80];
                    printf("Enter a string: "); gets(str); /*inputs a sting to str[80]*/
                    pr_str(str) ;               /*function call*/
                    return 0;}

void pr_str(const char *p){
                    while(*p){ *p=toupper(*p)  /* this wil not compile */
                               putchar(*p++); }
                    }
```

▶ ***volatile :*** When you precede a variable's type with volatile, you are telling the compiler that the value of the variable may be changed in ways not explicitly defined in the program. For example, a variable's address might be given to an interrupt service routine, and its value changed each time an interrupt occurs.

```
volatile unsigned u;
give_address_to_some_interrupt(&u);
      for(;;) {           /* watch value of u */
                  printf ("%d", u);. . . . . . .
```

In this example, if ***u*** had not been declared as ***volatile***, the compiler could have optimized the repeated calls to ***printf()*** in such a way that ***u*** was not ***reexamined*** each time. The use of ***volatile*** **forces the compiler to actually obtain the value** of ***u*** whenever it is used.

5.8 Function Pointers (Advanced topic)

A ***function pointer*** is a ***variable*** that contains the ***address*** of the ***entry point*** to a ***function***.

- ⇨ When the ***compiler*** compiles your program, it creates an ***entry point*** for each ***function*** in the program.
- ⇨ The ***entry point*** is the ***address*** to which ***execution*** transfers when a ***function is called***.
- ⇨ Since the ***entry point*** has an ***address***, it is possible to have a ***pointer*** variable ***point*** to it.
- ⇨ Once you have a ***pointer*** to a ***function***, it is possible to actually call that ***function*** using the ***pointer***.

To create a ***variable*** that can ***point*** to a ***function***, declare the ***pointer*** as having the ***same type*** as the ***return type of the function***, followed by ***any parameters***. For example, the following declares ***p*** as a ***pointer*** to a function that ***returns*** an ***integer*** and has two integer parameters, ***x*** and ***y;***

```
int (*p) (int x, int y);
```

The ***parentheses*** surrounding ****p*** are necessary because of C's ***precedence rules***.

- To assign the address of a function to a function pointer, simply use its name without any parentheses. For example, assuming that ***sum()*** has the prototype

  ```
  int sum(int a, int b);
  ```

  the assignment statement ***p = sum;*** is correct. Once this has been done, you can call ***sum() indirectly*** through ***p*** using a statement like

  ```
  result = (*p) (10, 20);
  ```

  [Again because of C's ***precedence rules*** the ***Parentheses*** are necessary around p.]

  Actually, you can also just use p directly, like this:

  ```
  result = p(10, 20);
  ```

NOTE

(*p) form inform anyone that reading your codes that a function pointer is being used to call a function indirectly, instead of calling a function named ***p***.