Functions

Function is a subprogram to carry out a specific task.

C functions can be classified into two categories,

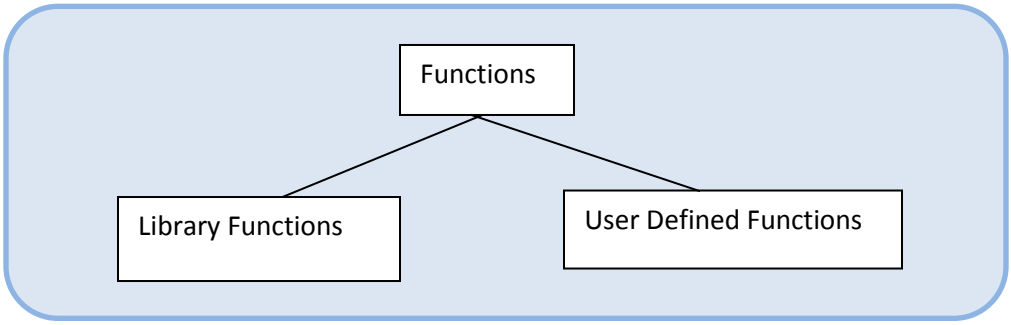


Fig 1: Function and Subcategories

Library functions are those functions which are defined by C library, example printf() , scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

User-defined functions are those functions which are defined by the user at the time of writing program.

Need for User Defined Functions.

The program may become too large and complex as a result,the task of debugging, testing and maintaining becomes difficult. In industry scenario writing monolithic program by a single programmer is not possible as the tasks are divided among team members.

If the program is divided into functional parts then each part may be independently coded and later combined into a single unit. These independently coded programs are subprograms that are much easier to understand, debug and test. In 'C' such subprograms are referred to as **functions.**

Benefits of Using Functions

1. Modularity.
2. Code Reusability.
3. Easy Debugging.
4. Maintainability.

Time and Space Complexity in multi-function program.

If a certain block of code is required in mainstream program for several times and the code is in lined with main()(then the amount of memory required for the program is more. E.g. if the subprogram need 10 k of memory, and if it is in lined with the mainstream program it will consume total 40 k memory. If we write the function for the same block of code the memory required will be 10k once and 2 bytes for each function call. So the memory requirement will be drastically reduced.

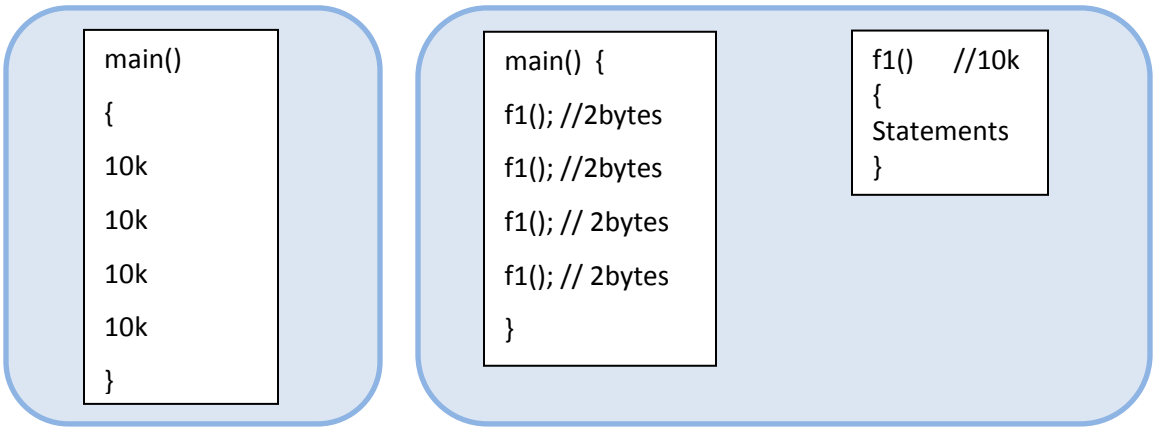


Fig 2: Time and Space Complexity

In case of time complexity the context switch will happen several times when the function is called. When the function is called the current status of the program is saved and the control is transferred to the called function. Once the function execution gets over the control is transferred back to the point from where the function was called. The previous status of the program is restored and the program execution further continues. During this process several times the context gets switched by operating system which causes the overhead on execution time. But with today's processor speed this overhead is negligible.

A Multi-Function Program

A function is a self-contained block of code that performs a particular task. Once the function is designed it can be treated as a black box that takes some data from the main program and returns some value. The inner details of operation are invisible to rest of the program.

- The main function calls user defined function.
- Any function can call any other function.
- A function can be called more than once.
- A function can call itself.
- A called function can also call another function.

- The functions can be placed in any order.

The following fig illustrates the flow of control in multifunction program.

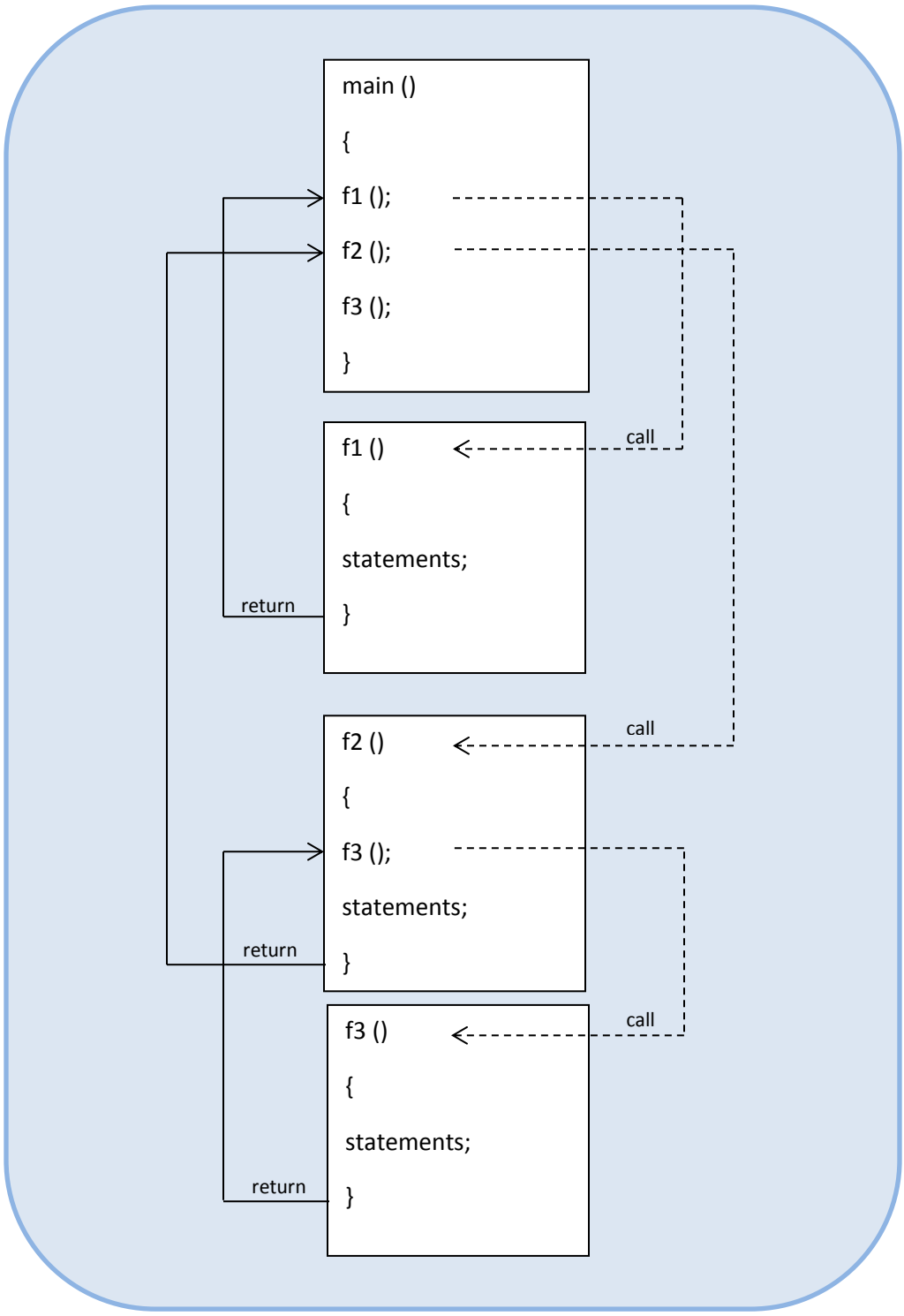


Fig 3: Flow of control in multi -function Program

In order to make use of user defined functions we need to establish three elements that are related to functions.

1. Function Definition.

2. Function call.

3. Function Declaration.

1. Function Definition (Syntax)

```
Return type  Function_name (parameters) // Parameters are optional.
{  // Beginning of the function.
Local Variables Declaration;
Statements within the function body;
Return statement;
```

- Function names and variable names are considered as identifiers and therefore they must follow the rules of identifiers.
- Like variables functions have types associated with them.
- Like variables function names and there types must be declared and defined before they are used in a program.
- The function type specifies the type of value that the function is expected to return to the calling function (e.g. int, float)
- If the function is not returning anything then the type is mentioned as void.
- The parameter list declares the variables that will receive the data sent by the calling program. As they represent actual input values they are referred to as '**formal Parameters'.**
- The parameters are also known as arguments.
- The parameter list contains declaration of variables separated by commas and surrounded by parentheses.

  E.g. int sum (int a, int b)

  int sum (int a, b)// This is invalid.

2. Function Call

```
Function_name (parameters);
```

- A function can be called by using the function names followed by list of parameters (if any) enclosed in parentheses. These parameters are called as actual parameters.
- The actual parameters must match the functions formal parameters in type, order and number. Multiple actual parameters must be separated by comma.

3. Function Declaration.

All functions must be declared before they are invoked. The function declaration is as follows.

```
Return type  Function_name (parameters);
```

- The parameter list must be separated by commas.
- The parameter names do not need to be the same in prototype declaration and the function definition.
- The types must match the type of parameters in the function definition in number and order.
- Use of parameter names in the declaration is optional.
- When the declared types do not match with the types in the function definition, compiler will produce an error.

Example 1: /* Compiler will understand that f1(); is a function call but it will not find the function definition. So It will produce output but by issuing the warning.*/

```
include<stdio.h>
int main()
{
f1();
}
void f1(void)
{
printf("Hello");
}/*Output:  Hello --- With warning*/
```

Example 2: /* If we write function definition before calling function it will give the correct output without any error or warning*/

```
#include<stdio.h>
void f1(void)
{
printf("Hello");
}
int main()
{
f1();
}            /*Output :  Hello*/
```

But it is not possible to write the function definitions before main() or calling function when large scale applications are written. So we need to write the function declaration which is information to compiler about the existence of function definition in program.

Example 3:

```
#include<stdio.h>
void f1(void);
int main()
{
f1();
}
void f1(void)
{
printf("Hello");
}              /* Hello */
```

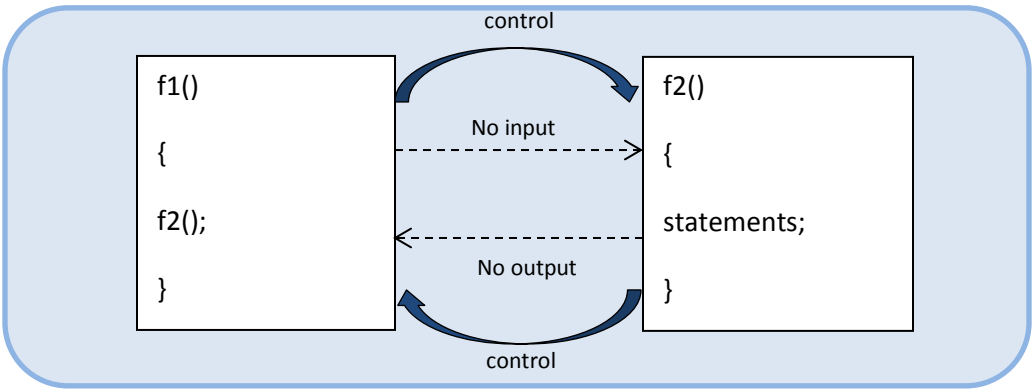
Categories of Functions

Depending on whether the arguments are present or not and whether a value is returned or not functions are classified into 4 categories.

1. Function with no arguments and no return values.
2. Function with arguments but no return values.
3. Function with arguments and return values.
4. Function with no arguments but return values.

These are described in below sections.

1. Function with no arguments and no return values.

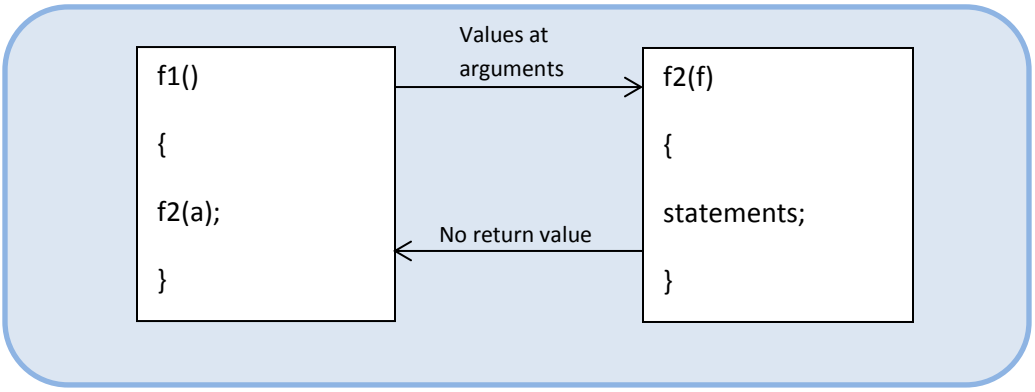


Example: Function with no arguments and no return values

```
#include<stdio.h>
void f1(void);
int main()
{
f1();
}

void f1(void)
{
printf("Hello");
}
```

2. Function with arguments but no return values.

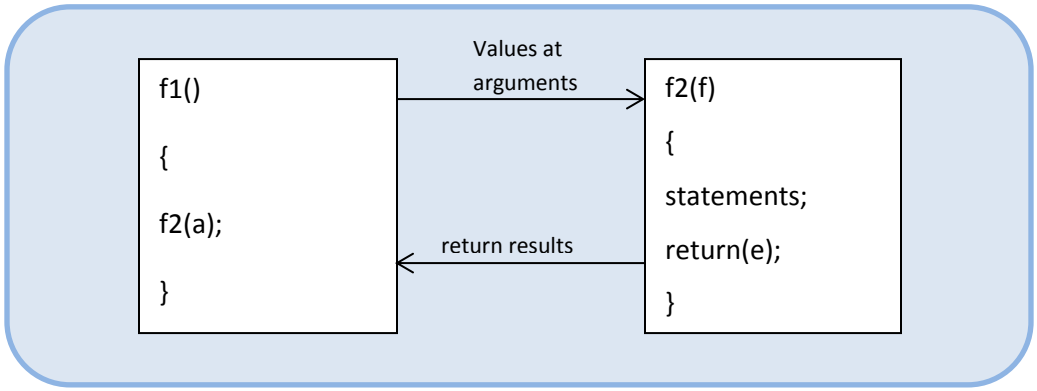


Example : Function with arguments but no return values

```
void f1(void);
int main()
{ int a=5;
f1(a);
}

void f1(int b)
{
printf("%d",b);
}          /* 5*/
```

3. Function with arguments and return values.



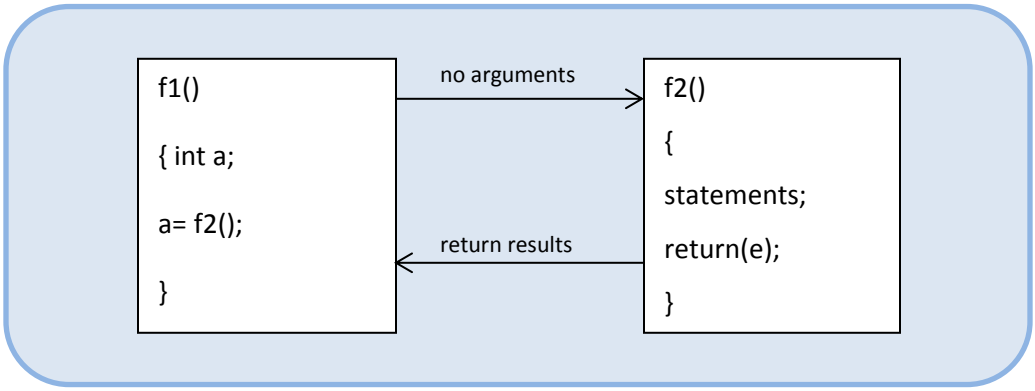
Example: Function with arguments and return value.

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

int main()
{
int k;
f1(5);

}
void f1(int a)
{
a=a+5;
return(a);
}
/* Output: 10 */
```

4. Function with no arguments but return values.



Example: Function with no arguments but return value.

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

int main()
{
int k;
k=f1();
printf("%d",k);
}

int f1(void)
{
printf("Hello\n");
return(5);
}       /* 5*/
```

Local and Global Variables.

Local Variable: The variable which is defined within the function body.

Global Variable: The variable which is defined outside the function.

```
int main() {
int a,b;
a=a+b;
printf("%d",a);
{ Int h;
Printf("%d",h);
} }
```

In the above sample code a and b are local variables whose scope is limited to the main section where as variable h is defined inside the block who has block scope only.
The variable which is defined outside the main function is global variable. The scope of the global variable is full program. Even program may span multiple lines.

```
int g=25;
int main()
{
int a=5,b=6;
a=a+b;
printf("%d",a);
printf("%d",g);
}
/* Output:11  25*/
```

Here variable g is declared outside the main function, it is global variable. If we use same variable name for global as well as local variable local variable has highest priority over Global

```
#include<stdio.h>
int g=25;
int main(){

{
int g=50;
printf("%d",g);
}
/* Output: 50*/
```

If we declare global variable after the main function section the compiler will throw an error saying that g is undeclared. For that we need to give the declaration inside main() that g is

declared somewhere else in the program with the additional qualifier extern. The example is given below.

```
#include<stdio.h>
#include<stdio.h>
{
int g=50;
printf("%d",g);
}{
int g=25;
/* Output:  error: g
```

```
#include<stdio.h>
int main()
{ extern int g;
printf("%d",g);
}
int g=25;
/* Output: 25 */
```

Storage Classes:

In C language, each variable has a storage class which decides scope, visibility and lifetime of that variable. The 4 types of storage classes are as follows.

1) Automatic Variables (auto)

Automatic variables are declared inside a function in which they are utilized. These variables takes birth as soon as the function is called and gets destroyed automatically when the function is exited. Even if we don't mention the qualifier by default it is auto.

2) External Variables (extern)

Variables that are both alive and active throughout the entire program are known as external variables. Global variables can be declared anywhere in the program outside the function. The qualifier extern is used with the declaration to inform the compiler that this variable is defined somewhere else in the program.

3) Static Variables (static)

The variable can be declared static using the keyword static. The static variables can be either local or global. The scope of the static local variable is only for the block in which it is declared but the life of these variables is throughout the file.

Global variables in C are by default extern.. (i.e) they have external linkage.. To restrict the external linkage, '**static**' storage class specifier can be used for the **global variable**.. if **static** specifier is used, then the **variable** has file scope.

The following example demonstrates the auto and static variables:

```
int main()
{f1();
f1();
f1();
}
/* Output
0  0
0  1
0  2 */
```

```
Void f1(void)
{
auto int i=0;
static int j=0;
printf("%d  %d\n", I,j);
i++; j++;
}
```

In this example function f1 has 2 local variables I and j initialized to zero. Additional qualifier is the storage class. I is an automatic variable and it takes birth as soon as function f1 is invoked. Initial value is 0 and after printing the value it gets incremented to 1. But it expires as soon as the function is exited. So when fi is called second time again it gets initialized to zero, whereas the variable j is static variable initialized to 0. It takes birth when the function is invoked for the first time and it remains alive even after the function is exited. So when text time the function f1 is invoked as variable is alive it holds the previous value. Even the life is throughout the file the scope is within the function only. So if we try to modify the value outside the function even if the variable is alive compiler throws an error. It is illustrated in following code.

```
int main()
{f1();
f1();
f1();
j++;   }
/* Output
Error: j is undeclared.*/
```

```
Void f1(void)
{
auto int i=0;
static int j=0;
printf("%d  %d\n", I,j);
i++; j++;
}
```

4) Register Variables (register)

Usually when we define any variable e.g int a=5; the memory is allocated in RAM but as the CPU register access is much faster than memory access the frequently accessed variables can be stored in one of the register by using additional qualifier register. E.g: Register int a;

The allocation of register is not gaurenteed. It depends upon the availability of the CPU register.

Additional Qualifiers:

1) typedef.

The C programming language provides a keyword called **typedef**, which we can use to give a alias name to the data type. In the following example datatype intis getting an alias name age.

e.g:

```
int main()
{typedef int age;
Age a =45;
Printf("%d",a);
}
/* Output: 45*/
```

2) Volatile.

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time.

E.g: int a=10; To store value 10 we need not have a full integer (4 bytes) so compiler implicitly does optimization and converts it as short int. But when we are reading the values form external devices w have to inform the compiler not to go for any optimization. For that the qualifier volatile is used in variable definition.

3) Constant(const)

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed

e.g: const in a=10;

tells the compiler that the value of a is read only and can't be modified.

Recursion:

The function calling itself with the termination condition is recursion. Recursion is used to solve various problems by dividing it into smaller problems. The syntax of recursive function is as follows:

```
Return type recursive_function(arguments)
{
statements;
recursive_function(arguments);
}
```

Consider the following program code.

```
int main()
{
printf("Hello!");
main();
Return 0;
}  /* output: Hello! For infinite times*/
```

In this program, we are calling *main()* from *main()* which is recursion. But we haven't defined any condition for the program to exit. Hence this code will print "**Hello!"** infinitely in the output screen. In order to prevent infinite recursive call, we need to define proper exit condition in a recursive function.