**Function Basics**

**Writer:** Sayak Haldar

**Theory:**A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

**Defining a Function**

The general form of a function definition in C programming language is as follows −

**return\_type function\_name( parameter list ) {**

**body of the function**

**}**

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function −

**Return Type** − A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword void.

**Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

**Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

**Function Body** − The function body contains a collection of statements that define what the function does.

**Example**

Given below is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum value between the two −

**/\* function returning the max between two numbers \*/**

**int max(int num1, int num2) {**

**/\* local variable declaration \*/**

**int result;**

**if (num1 > num2)**

**result = num1;**

**else**

**result = num2;**

**return result;**

**}**

**Function Declarations**

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

**return\_type function\_name( parameter list );**

For the above defined function max(), the function declaration is as follows −

**int max(int num1, int num2);**

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

**int max(int, int);**

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.  
  
**Calling a Function**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

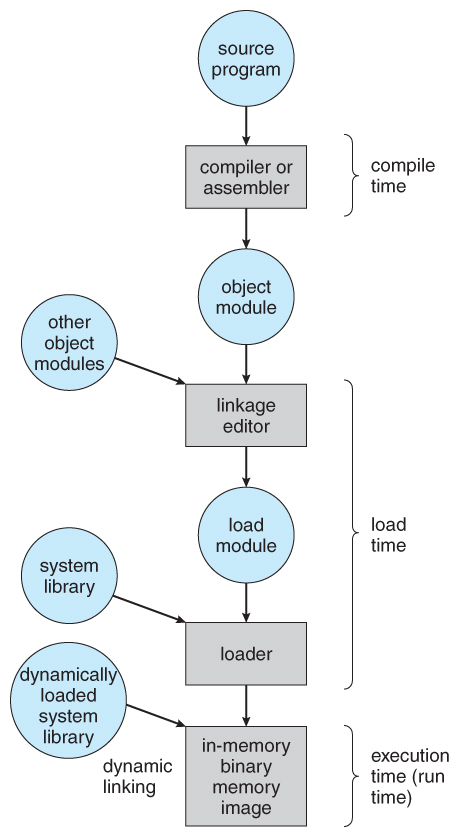
else

result = num2;

return result;

}

**Function Prototype:**In computer programming, a function prototype or function interface is a declaration of a function that specifies the function's name and type signature (arity, data types of parameters, and return type), but omits the function body. While a function definition specifies how the function does what it does (the "implementation"), a function prototype merely specifies its interface, i.e. what data types go in and come out of it. The term function prototype is particularly used in the context of the programming languages C and C++ where placing forward declarations of functions in header files allows for splitting a program into translation units, i.e. into parts that a compiler can separately translate into object files, to be combined by a linker into an executable or a library.



**(forward declaration:**

[In computer programming, a](https://en.wikipedia.org/wiki/Forward_declaration)**[forward declaration](https://en.wikipedia.org/wiki/Forward_declaration)**[is a declaration of an identifier for which the programmer has not yet given a complete definition.)](https://en.wikipedia.org/wiki/Forward_declaration)

However, the linking provided by function prototype is **static. (for user defined function)**

**Can A Function Be Static In c?**In C, functions are global by default. The “static” keyword before a function name makes it static. For example, below function fun() is static.

**static int fun(void)**

**{**

**printf("I am a static function ");**

**}**

Unlike global functions in C, access to static functions is restricted to the file where they are declared. **( to mark the function as having internal linkage so it cannot be referenced in other translation units)** Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.  
  
(**Note that:** static function usage is deprecated in C++. Unnamed namespaces are preferred for this usage.)

**Difference Between Normal Function Vs Static Function:**All functions are implicitly declared as extern, which means they're visible across translation units. But when we use static it restricts visibility of the function to the translation unit in which it's defined. So we can say Functions that are visible only to other functions in the same file are known as static functions.

**1. What is the output of this C code?**

#include <stdio.h>

int main()

{

void foo();

printf("1 ");

foo();

}

void foo()

{

printf("2 ");

}

a) 1 2

b) Compile time error

c) 1 2 1 2

d) Depends on the compiler

**Answer: a**

**2. What is the output of this C code?**

#include <stdio.h>

int main()

{

void foo(), f();

f();

}

void foo()

{

printf("2 ");

}

void f()

{

printf("1 ");

foo();

}

a) Compile time error as foo is local to main

b) 1 2

c) 2 1

d) Compile time error due to declaration of functions inside main

**Answer: b**

**3. What is the output of this C code?**

#include <stdio.h>

int main()

{

void foo();

void f()

{

foo();

}

f();

}

void foo()

{

printf("2 ");

}

a) 2 2

b) 2

c) Compile time error

d) Depends on the compiler

**Answer: d**

Explanation: Even though the answer is 2, this code will compile fine only with gcc. GNU C supports nesting of functions in C as a language extension where as standard C compiler doesn’t.

**4. What is the output of this C code?**

#include <stdio.h>

void foo();

int main()

{

void foo();

foo();

return 0;

}

void foo()

{

printf("2 ");

}

a) Compile time error

b) 2

c) Depends on the compiler

d) Depends on the standard

**Answer: b**

**5. What is the output of this C code?**

#include <stdio.h>

void foo();

int main()

{

void foo(int);

foo(1);

return 0;

}

void foo(int i)

{

printf("2 ");

}

a) 2

b) Compile time error

c) Depends on the compiler

d) Depends on the standard

**Answer: a**

**6. What is the output of this C code?**

#include <stdio.h>

void foo();

int main()

{

void foo(int);

foo();

return 0;

}

void foo()

{

printf("2 ");

}

a) 2

b) Compile time error

c) Depends on the compiler

d) Depends on the standard

**Answer: b**

Compiler will say the number of arguments don’t match with the prototype.

**7. What is the output of this C code?**

include <stdio.h>

void m()

{

printf("hi");

}

void main()

{

m();

}

a) hi

b) Run time error

c) Nothing

d) Varies

**Answer: a**

**8. What is the output of this C code?**

#include <stdio.h>

void m();

void n()

{

m();

}

void main()

{

void m()

{

printf("hi");

}

}

a) hi

b) Compile time error

c) Nothing

d) Varies

**Answer: b**

**Undefined reference to m.**

1. **What is the output of this C code?**

#include <stdio.h>

void main()

{

m();

void m()

{

printf("hi");

}

}

a) hi

b) Compile time error

c) Nothing

d) Varies

**Answer: b**

**(my first guess is:**

**Undefined reference to m**

The prototype must be present/ the declaration must be present before the function call.

)

The compiler will say:

ctrial52.c:5: warning: conflicting types for ‘m’

ctrial52.c:5: error: static declaration of ‘m’ follows non-static declaration

ctrial52.c:4: note: previous implicit declaration of ‘m’ was here

So, basically compiler is misinterpreting things.

**#include <stdio.h>**

**void m();**

**void main()**

**{**

**m();**

**void m()**

**{**

**printf("hi");**

**}**

**}**

Even it wont fix things. It will make compiler say, Undefined reference to m.

So, the definition also must be present before the function call.  
 **#include <stdio.h>**

**void m();**

**void main()**

**{**

**void m()**

**{**

**printf("hi");**

**}**

**m();**

**}**

It would fix the error.

**10. What is the output of this C code?**

#include <stdio.h>

void main()

{

m();

}

void m()

{

printf("hi");

m();

}

a) Compile time error

b) hi

c) Infinite hi

d) Nothing

View Answer

**Answer: c**

**11. What is the output of this C code?**

#include <stdio.h>

void main()

{

static int x = 3;

x++;

if (x <= 5)

{

printf("hi");

main();

}

}

a) Run time error

b) hi

c) Infinite hi

d) hi hi

**Answer: d**

**12. Which of the following is a correct format for declaration of function?**

a) return-type function-name(argument type);

b) return-type function-name(argument type)

{}

c) return-type (argument type)function-name;

d) all of the mentioned

**Answer: a**

**13. Which of the following function declaration is illegal?**

a) int 1bhk(int);

b) int 1bhk(int a);

c) int 2bhk(int\*, int []);

d) all of the mentioned

**Answer: d**

Identifier names cannot start with a digit.

**14. Which function definition will run correctly?**

a) int sum(int a, int b)

return (a + b);

b) int sum(int a, int b)

{return (a + b);}

c) int sum(a, b)

return (a + b);

d) none of the mentioned

View Answer

**Answer: b**

**15. Can we use a function as a parameter of another function? [ Eg: void wow(int func()) ].**

a) Yes, and we can use the function value conveniently

b) Yes, but we call the function again to get the value, not as convenient as in using variable

c) No, C does not support it

d) This case is compiler dependent

**Answer: c**

**16. The value obtained in the function is given back to main by using \_\_\_\_\_\_\_\_ keyword?**

a) return

b) static

c) new

d) volatile

**Answer: a**

**17. What is the return-type of the function sqrt()**

a) int

b) float

c) double

d) depends on the data type of the parameter

**Answer: c**

**18. Which of the following function declaration is illegal?**

a) double func();

int main(){}

double func(){}

b) double func(){};

int main(){}

c) int main()

{

double func();

}

double func(){//statements}

d) None of the mentioned

View Answer

**Answer: d**

**19. What is the output of this code having void return-type function?**

#include <stdio.h>

void foo()

{

return 1;

}

void main()

{

int x = 0;

x = foo();

printf("%d", x);

}

a) 1

b) 0

c) Runtime error

d) Compile time error

**Answer) d) Compile time error.**

**20. What will be the data type returned for the following function?**

#include <stdio.h>

int func()

{

return (double)(char)5.0;

}

a) char

b) int

c) double

d) multiple type-casting in return is illegal

**Answer: b**

**21. What is the problem in the following declarations?**

int func(int);

double func(int);

int func(float);

a) A function with same name cannot have different signatures

b) A function with same name cannot have different return types

c) A function with same name cannot have different number of parameters

d) All of the mentioned

**Answer: d**

**22. The output of the code below is**

#include <stdio.h>

void main()

{

int k = m();

printf("%d", k);

}

void m()

{

printf("hello");

}

a) hello 5

b) Error

c) Nothing

d) Junk value

**Answer: a**

**23. The output of the code below is**

#include <stdio.h>

int \*m()

{

int \*p = 5;

return p;

}

void main()

{

int \*k = m();

printf("%d", k);

}

a) 5

b) Junk value

c) 0

d) Error

**Answer: a  
  
(Note: during compilation of the program, it will say the following thing as warning:**

**initialization makes pointer from integer without a cast)**

Now, you cannot deference k here. Because, k’s memory location is 5.

**24. The output of the code below is**

#include <stdio.h>

int \*m();

void main()

{

int \*k = m();

printf("hello ");

printf("%d", k[0]);

}

int \*m()

{

int a[2] = {5, 8};

return a;

}

a) hello 5 8

b) hello 5

c) hello followed by garbage value

d) Compilation error

Answer: c

**This will generate the following warning during compilation:**

**ctrial55.c:12: warning: function returns address of local variable**

Now, it actually generates undefined behaviour while printing k[0] or \*k. (both are same)

For instance, I do some tests with **gcc version 4.4.7**

**#include <stdio.h>**

**int \*m();**

**void main()**

**{**

**int \*k = m();**

**printf("hello ");**

**printf("%d", k[0]);**

**}**

**int \*m()**

**{**

**int a[2] = {5, 8};**

**return a;**

**}**

This is generating hello garbage value

So, is this:

**#include <stdio.h>**

**int \*m();**

**void main()**

**{**

**int \*k = m();**

**//printf("The memory location pointed by k is %p\n",k);**

**printf("hello ");**

**//printf("%d",\*k);**

**printf("%d",\*k);**

**}**

**int \*m()**

**{**

**int a[2] = {5, 8};**

**//printf("The starting location of array a is %p\n",a);**

**return a;**

**}**

But, the followings will generate proper result:

**#include <stdio.h>**

**int \*m();**

**void main()**

**{**

**int \*k = m();**

**//printf("The memory location pointed by k is %p\n",k);**

**printf("hello ");**

**//printf("%d",\*k);**

**printf("%d",\*k);**

**}**

**int \*m()**

**{**

**int a[2] = {5, 8};**

**printf("The starting location of array a is %p\n",a);**

**return a;**

**}**

As well as:

**#include <stdio.h>**

**int \*m();**

**void main()**

**{**

**int \*k = m();**

**printf("The memory location pointed by k is %p\n",k);**

**printf("hello ");**

**//printf("%d",\*k);**

**printf("%d",\*k);**

**}**

**int \*m()**

**{**

**int a[2] = {5, 8};**

**printf("The starting location of array a is %p\n",a);**

**return a;**

**}**

As well as:

**#include <stdio.h>**

**int \*m();**

**void main()**

**{**

**int \*k = m();**

**printf("The memory location pointed by k is %p\n",k);**

**printf("hello ");**

**printf("%d",k[0]);**

**//printf("%d",\*k);**

**}**

**int \*m()**

**{**

**int a[2] = {5, 8};**

**printf("The starting location of array a is %p\n",a);**

**return a;**

**}**

So, it truly invokes some undefined behaviour.

**25. The output of the code below is**

#include <stdio.h>

int \*m();

void main()

{

int k = m();

printf("%d", k);

}

int \*m()

{

int a[2] = {5, 8};

return a;

}

a) 5

b) 8

c) Nothing

d) Varies

**Answer: d**

**But, most importantly, what you will learn that this wont throw any compilation error.**

When this program is compiled, it will tell the following things:

**ctrial56.c: In function ‘main’:**

**ctrial56.c:5: warning: initialization makes integer from pointer without a cast**

**ctrial56.c: In function ‘m’:**

**ctrial56.c:11: warning: function returns address of local variable**

**26. The output of the code below is**

#include <stdio.h>

void m(int k)

{

printf("hi");

}

void m(double k)

{

printf("hello");

}

void main()

{

m(3);

}

a) hi

b) hello

c) Compile time error

d) Nothing

View Answer

**Answer: c**

You cannot overload a function in c.

**27. What is the default return type if it is not specified in function definition?**

a) void

b) int

c) double

d) short int

**Answer: b**

**28. What is the output of this C code?**

#include <stdio.h>

int foo();

int main()

{

int i = foo();

}

foo()

{

printf("2 ");

return 2;

}

a) 2

b) Compile time error

c) Depends on the compiler

d) Depends on the standard

**Answer: a**

Why?

Because of this:

If no return statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. If a return value is not required, declare the function to have void return type; otherwise, the default return type is int .

**29. What is the output of this C code?**

#include <stdio.h>

double foo();

int main()

{

foo();

return 0;

}

foo()

{

printf("2 ");

return 2;

}

a) 2

b) Compile time error

c) Depends on the compiler

d) Depends on the standard

View Answer

Answer: b  
  
Now, since, the functions which do not have have a return type in their definition have the default return type as int, the function declaration and definition will mismatch.

**30. functions can return structure in c?**

a) true

b) false

c) Depends on the compiler

d) Depends on the standard

Answer: a

**31. functions can return enumeration constants in c?**

a) true

b) false

c) depends on the compiler

d) depends on the standard

Answer: a

**32. What is the output of code given below?**

#include <stdio.h>

enum m{JAN, FEB, MAR};

enum m foo();

int main()

{

enum m i = foo();

printf("%d\n", i);

}

int foo()

{

return JAN;

}

a) Compile time error

b) 0

c) Depends on the compiler

d) Depends on the standard

**Answer: a**

Because, foo() function’s declaration and definition would mismatch.

**#include <stdio.h>**

**enum m{JAN, FEB, MAR};**

**int foo();**

**int main()**

**{**

**enum m i = foo();**

**printf("%d\n", i);**

**}**

**int foo()**

**{**

**return JAN;**

**}**

This will not generate any compilation error. This will return 0.

Now, what does does **enum m{JAN,FEB,MAR},** mean?

Here, we define a new datatype enum m which can have the value: JAN, FEB, MAR.

However, it can have any integral constant value.

**#include <stdio.h>**

**enum m{JAN, FEB, MAR};**

**int foo();**

**int main()**

**{**

**enum m month=99;**

**/\*jan has the value 0, feb has the value 1 and MAR has the value 2**

**\*/**

**printf("the month value is %d\n",month);**

**return 0;**

**}**

If you try to assign a floating type constant, compile is ok about it as long as it can convert the value as integral constant in some way.

**#include <stdio.h>**

**enum m{JAN, FEB, MAR};**

**int foo();**

**int main()**

**{**

**enum m month=99.123f;**

**/\*jan has the value 0, feb has the value 1 and MAR has the value 2**

**\*/**

**printf("the month value is %d\n",month);**

**return 0;**

**}**

It is also ok.

However,

**#include <stdio.h>**

**enum m{JAN, FEB, MAR};**

**int foo();**

**int main()**

**{**

**enum m month="Sayak";**

**/\*jan has the value 0, feb has the value 1 and MAR has the value 2**

**\*/**

**printf("the month value is %d\n",month);**

**return 0;**

**}**

This will generate compilation error. Compiler will tell:

**incompatible types when initializing type ‘enum m’ using type ‘char \*’**