1.6 Functions

Objectives

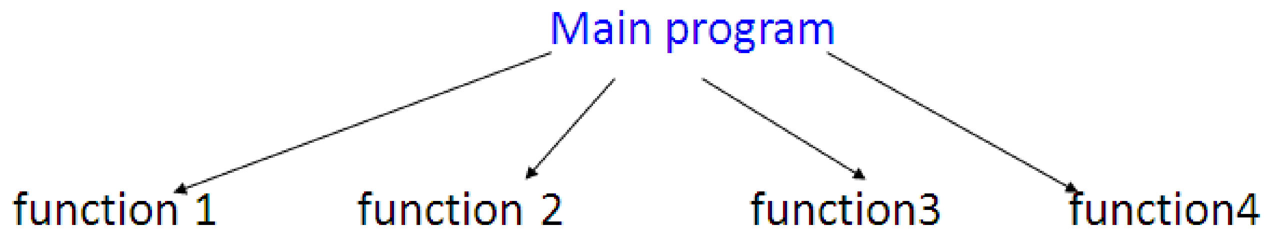
- To understand the concept of modularization.
- To know about the types of functions.
- To study about formal arguments and actual arguments.
- To understand the need of passing arguments to function.

Agenda

- Introduction to Function
- Types of C Functions
- Function naming rule in C
- General Form of a Function
- Parameters / Arguments
- Scope of a Function
- Returning Value – Control from a Function
- Three Main Parts of a Function
- Categorization based on Arguments and Return value
- Calling Functions – Two Methods
- Creating user defined header files
- Storage classes
- Summary

Introduction to Functions

Functions are the building blocks of C and the place where all program activity occurs.



<u>Benefits of Using Functions:</u>

- It provides modularity to the program.
- Easy code reusability. You just have to call the function by its name to use it.
- In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.

Credits: http://www.studytonight.com/c/types-of-function-calls.php

Contd..

A function is independent:

- ➢ It is "completely" **self-contained**
- ➢ It can be **called at any place** of your code and can be **ported** to another program
  - ✓ **reusable -** Use existing functions as building blocks for new programs
  - ✓ **Readable -** more meaningful
  - ✓ **procedural abstraction -** hide internal details
  - ✓ **factoring of code-** divide and conquer

Contd..

A function:

receives zero or more parameters,

performs a specific task, and

returns zero or one value

A function is invoked / called by name and parameters

Communication between function and invoker code is through the parameters and the return value

- In C, no two functions can have the same name

Types of C Functions

- Library function
- User defined function

<u>Library function</u>

- Library functions are the in-built function in C programming system

For example:

❖main() - - The execution of every C program

❖printf() - prinf() is used for displaying output in C.

❖scanf() - scanf() is used for taking input in C.

Some of the math.h Library Functions

- sin() → returns the sine of a radian angle.
- cos() → returns the cosine of an angle in radians.
- tan() → returns the tangent of a radian angle.
- floor() → returns the largest integral value less than or equal to x.
- ceil() → returns the smallest integer value greater than or equal to x.
- pow() → returns base raised to the power of exponent(x^y).

Some of the conio.h Library Functions

- clrscr() → used to clear the output screen.
- getch() → reads character from keyboard.
- textbackground() → used to change text background

Contd..

<u>User defined function</u>

- Allows programmer to define their own function according to their requirement.

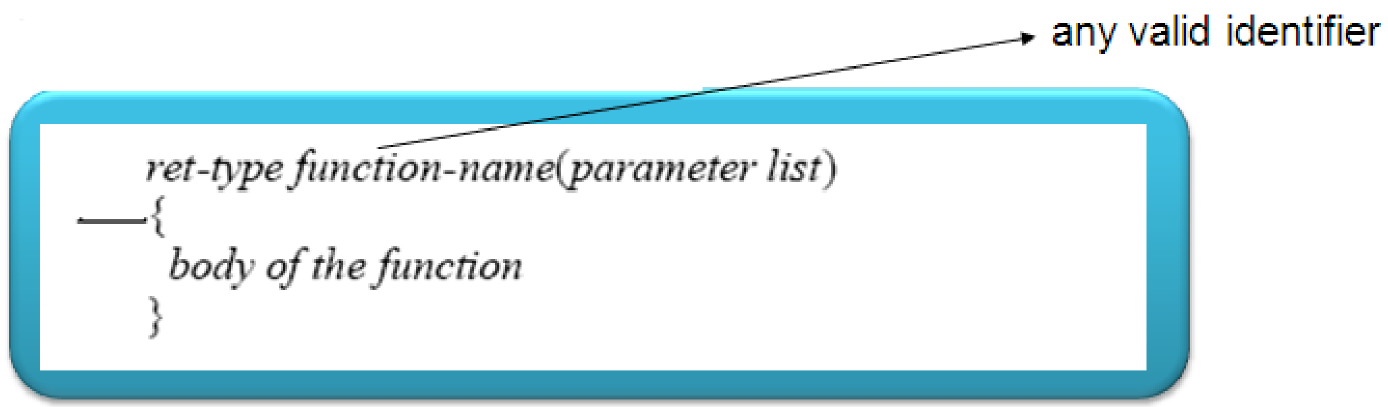
Advantages of user defined functions

- It helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmer working on large project can divide the workload by making different functions.

Function naming rule in C

- Name of function includes only alphabets, digit and underscore.
- First character of name of any function must be an alphabet or underscore.
- Name of function cannot be any keyword of c program.
- Name of function cannot be global identifier.
- Name of function cannot be exactly same as of name of function in the same scope.
- Name of function is case sensitive
- Name of function cannot be register Pseudo variable

The General Form a Function



```
ret-type function-name(parameter list)
{
  body of the function
}
```

The ret-type specifies the type of data that the function returns.
A function may return any type (default: int)of data except an array

The parameter (formal arguments) list is a comma-separated list of variable names and their associated types
The parameters receive the values of the arguments when the function is called

A function can be without parameters:
An empty parameter list can be explicitly specified as such by placing the keyword void inside the parentheses

More about formal argument/parameter list ……….

(type varname1, type varname2, . . . , type varnameN)

You can declare **several variables to be of the same type**

By using a comma separated list of variable names.

In contrast, all function parameters **must be declared individually**, each including both the type and name

- f(int i, int k, int j)
- f(int i, k, float j) Is it correct?
- /* wrong, k must have its own type specifier */

Scope of a function

Each function is a **discrete block of code.** Thus, a function **defines a block scope**:

A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function.

- **Variables that are defined within a function are local variables**
- A local variable comes into existence when the function is entered and is destroyed upon exit
- A local variable **cannot hold its value between function calls**

Contd..

The **formal arguments /parameters** to a function also fall **within the function's scope:**

is known throughout the entire function

comes into existence when the function is called and is destroyed when the function is exited.

Even though they perform the special task of receiving the value of the arguments passed to the function, they behave like any other local variable

Returning value, control from a function

If nothing returned

- return;
- or, until reaches right curly brace

If something returned

- return expression;

Only one value can be returned from a C function

A function can **return only one value**, though it can return **one of several values** based on the evaluation of certain conditions.

Multiple return statements can be used within a single function
(eg: inside an "if-then-else" statement…)

The return statement not only returns a value back to the calling function,
it **also returns**
control back to the calling function

Three Main Parts of a Function

- Function Declaration (Function prototype)
- Function Definition
- Function Call

Structure of a C program with a Function

Function prototype giving the name, return type and the type of formal arguments

main( )

{

……….

Call to the function:

Variable to hold the value returned by the function = Function name with actual arguments

………

}

Function definition:

Header of function with name, return type and the type of formal arguments as given in the prototype

Function body within { } with local variables declared , statements and return statement

Function Prototype

- Functions should be declared before they are used
- Prototype only needed if function definition comes after use in
- program
- Function prototypes are always declared at the beginning of
- the program indicating :
- name of the function, data type of its arguments &
- data type of the returned value

```
return_type   function_name ( type1  name1, type2  name2,
                         ..., typen  namen );
```

Function Definition

Function header

```
return_type function_name ( type1 name1, type2 name2,
                                                ...,typen namen)
{
    local variable declarations
   .... otherstatements...
   return statement
}
```

Function Body

Function call

A function is called from the main( )

A function can in turn call a another function

Function call statements invokes the function which means the **program control passes to that function**

Once the function completes its task, the **program control is passed back to the calling environment**

Contd..

Variable = function_name (actual argument list);

Or

Function_name (actual argument list);

Function **name and the type and number of arguments must match** with that of the function declaration stmt and the header of the function definition

Examples:

result = sum(5, 8);   display( );   calculate(s, r, t);

Return statement

To return a value from a C function you must explicitly return it with a return statement

```
                    return <expression>;
```

The expression can be **any valid C expression** that resolves to the type defined in the function header

```
add( int a, int b)                      add( int a, int b)

{                                              {
                                                  int c = a+ b;;
 return (a + b);                                  return ( c );
}                                              }
```

Ex: Function call: **int value = add(5,8)**

Here, add( ) sends back the value of the expression (a + b) or value of c to main( )

Examples

Function Prototype Examples

```
double squared (double number);
void print_report (int);
int get _menu_choice (void);
```

Function Definition Examples

```
double squared (double number)
{
   return (number * number);
}

void print_report (int report_number)
{
   if (report_nmber == 1)
         printf(“Printer Report 1”);
      else
         printf(“Not printing Report 1”);
 }
```

Example C program..

```
#include<stdio.h>

float average(float, float, float);

int main( )

{
   float a, b, c;
   printf("Enter three numbers please\n");
   scanf("%f, %f, %f",&a, &b, &c);
   printf("Avg of 3 numbers = %.3f\n",  average(a, b, c) );
   return 0;
}
```

Function prototype

Function call

Contd..

The definition of function average:

```
float average(float x, float y, float z) //local variables x, y, z
  {
        float r; // local variable
  r = (x+y+z)/3;
  return r;
  }
```

Function header

Function Body

Categorization based on arguments and return value

- Function with no arguments and no return value
- Function with no arguments and return value
- Function with arguments but no return value
- Function with arguments and return value.

Credits : http://www.programiz.com/c-programming/types-user-defined-functions

Calling Functions – Two methods

Call by value

- **Copy of argument passed**
- Changes in function do not effect original
- Use when function does not need to modify argument
  - Avoids accidental changes

Call by reference

- **Passes original argument**
- Changes in function effect original
- Only used with trusted functions

Call by Value

When a function is called by an argument/parameter which **is not a pointer** the **copy of the argument is passed to the function.**

Therefore a possible change on the copy does not change the original value of the argument.

Example:

Function call **func1 (a, b, c);**

Function header **int func1 (int x, int y, int z)**

Here, the parameters **x , y and z** are initialized by the values of **a, b and c**

int x = a
int y = b
int z = c

Example C Program

```
void swap(int, int );

 main( )
     {
       int a=10, b=20;
       swap(a, b);
       printf(" %d %d \n", a, b);
     }

 void swap (int x, int y)
    {
      int temp = x;
      x=  y;
      y=temp;
    }
```

Intricacies of the preceding example

- In the preceding example, the function main() declared and initialized two integers a and b, and then invoked the function swap( ) by **passing a and b as arguments** to the **function swap( )**.

- The **function swap( ) receives the arguments a and b into its parameters x and y**. In fact, the functionswap( ) receives a **copy of the values** of a and b into its parameters.

- The **parameters of a function are local to that function**, and hence, any **changes made by the called function to its parameters affect only the copy received by the called function**, and do not affect the value of the variables in the called function. This is the **call by value mechanism**.

Call by Reference

When a function is called by an argument/parameter which **is a pointer** (address of the argument) the **copy of the address of the argument is passed to the function**

Therefore, a possible change on the data at the referenced address changes the original value of the argument.

Will be dealt later in detail…

Illustration for creating user defined header files using user created functions

Make Your Own Header File ?

Step1 : Type this Code

```
int add(int a,int b)
{
        return(a+b);
}
```

- In this Code write only function definition as you write in General C Program

Step 2 : Save Code

- **Save** Above Code with **[.h] Extension** .
- Let name of our header file be **myhead** [myhead.h]
- Compile Code if required.

Step 3 :Write Main Program

```
#include<stdio.h>
#include"myhead.h“
  main() {
   int num1 = 10, num2 = 10, num3;
   num3 = add(num1, num2);
   printf("Addition of Two numbers : %d", num3);
                }
```

Here,

- Instead of writing **< myhead.h>** use this terminology **“myhead.h”**
- All the Functions defined in the **myhead.h header** file are now ready for use .
- **Directly call function add()**; [Provide proper parameter and take care of return type]

Note : While running your program precaution to be taken :

Both files [**myhead.h and sample.c**] should be in same folder.

Storage Classes

- A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.
  - Automatic variables → auto
  - External variables → extern
  - Static variables → static
  - Register variables → register

auto - Storage Class

auto is the default storage class for all local variables.

```
    {
  int Count;
  auto int Month;
    }
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

extern - Storage Class

- These variables are declared outside any function.
- These variables are active and alive throughout the entire program.
- Also known as global variables and default value is zero.

static - Storage Class

- The value of static variables persists until the end of the program.
- It is declared using the keyword static like

```
static int x;
static float y;
```

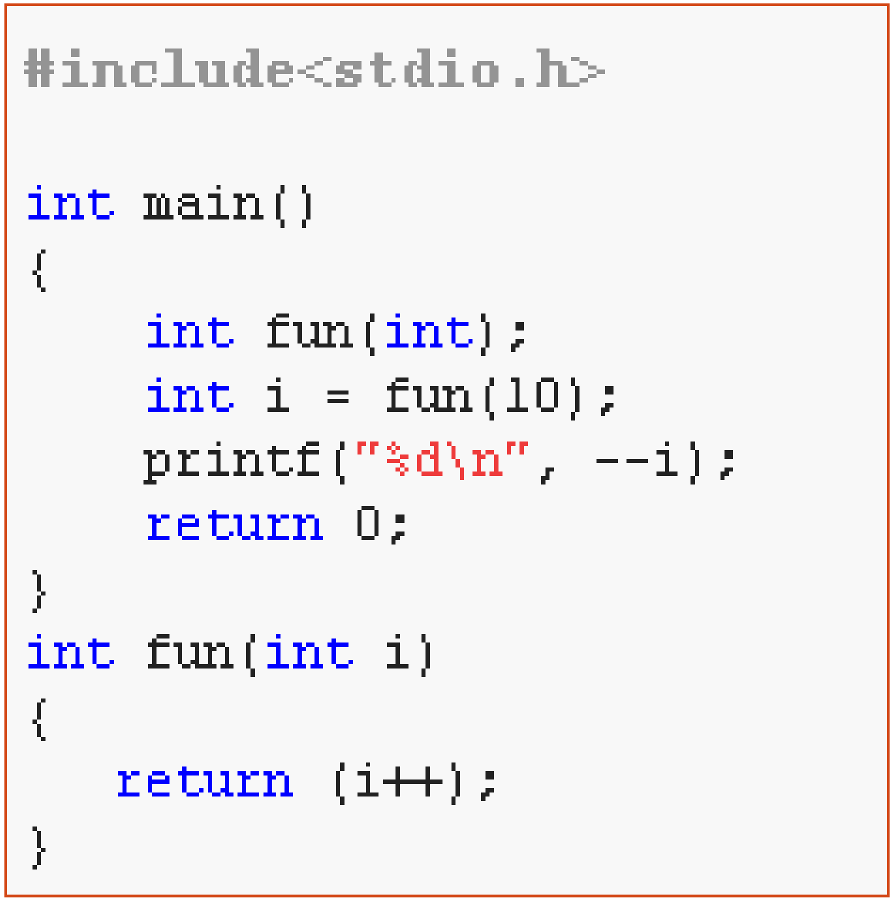
- It may be of external or internal type depending on the place of there declaration.
- Static variables are initialized only once, when the program is compiled.

register - Storage Class

- These variables are stored in one of the machine's register and are declared using register keyword.

  eg. register int count;
- Since register access are much faster than a memory access keeping frequently accessed variables in the register lead to faster execution of program.
- Don't try to declare a global variable as register. Because the register will be occupied during the lifetime of the program.

1. Predict the output of the program.



```c
#include<stdio.h>

int main()
{
     int fun(int);
     int i = fun(10);
     printf("%d\n", --i);
     return 0;
}
int fun(int i)
{
   return (i++);
}
```

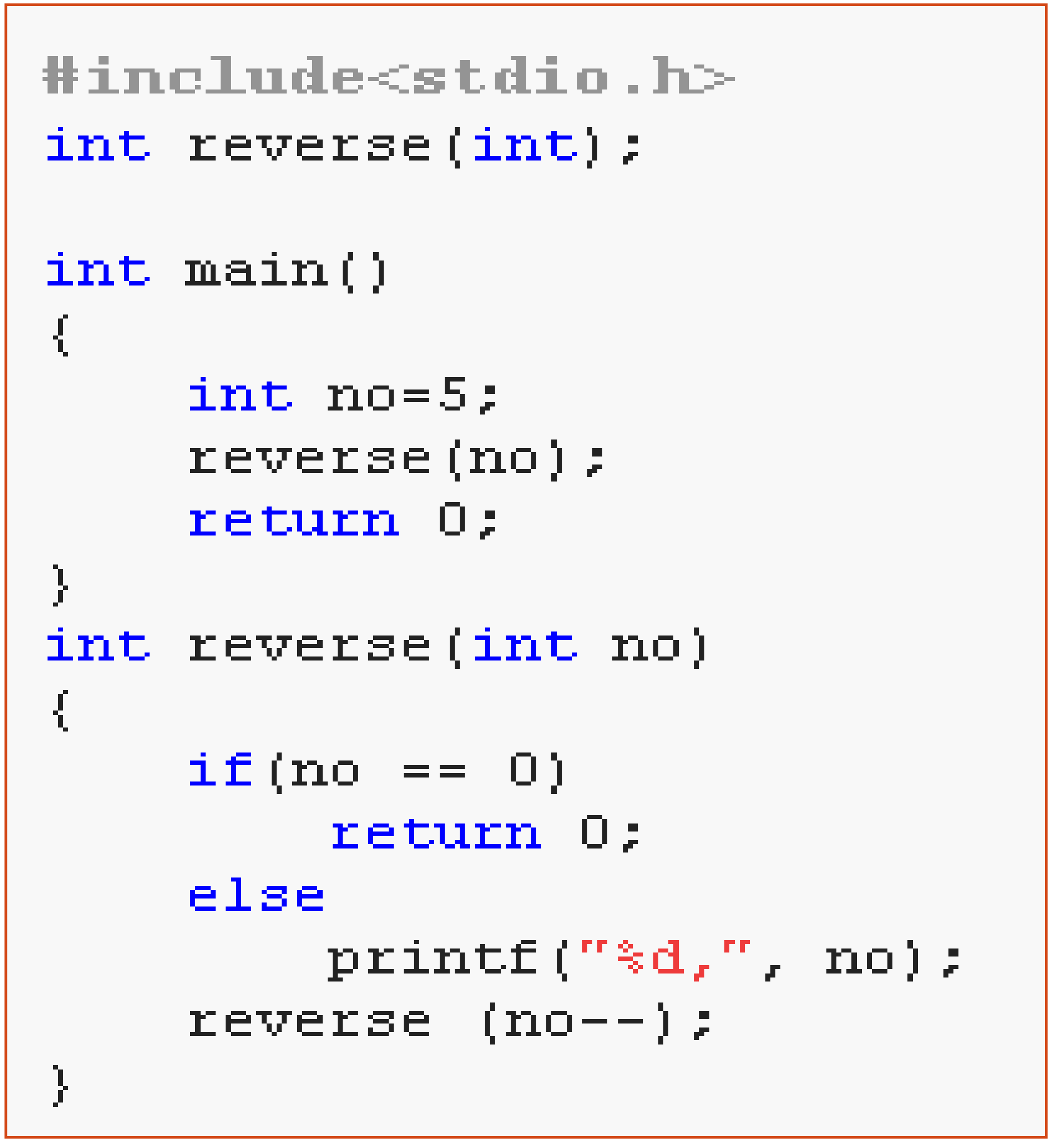
1. Solution

Answer: 9

Explanation:

- **Step 1**: *int fun(int);* Here we declare the prototype of the function *fun()*.
- **Step 2**: *int i = fun(10);* The variable *i* is declared as an integer type and the result of the *fun(10)* will be stored in the variable *i*.
- **Step 3**: *int fun(int i){ return (i++); }* Inside the *fun()* we are returning a value return(i++). It returns 10. because *i++* is the post-increement operator.
- **Step 4**: Then the control back to the *main* function and the value 10 is assigned to variable *i*.
- **Step 5**: *printf("%d\n", --i);* Here *--i* denoted pre-increement. Hence it prints the value 9.

2. What will be the output of the program?



```c
#include<stdio.h>
int reverse(int);

int main()
{
    int no=5;
    reverse(no);
    return 0;
}
int reverse(int no)
{
    if(no == 0)
        return 0;
    else
        printf("%d,", no);
    reverse (no--);
}
```

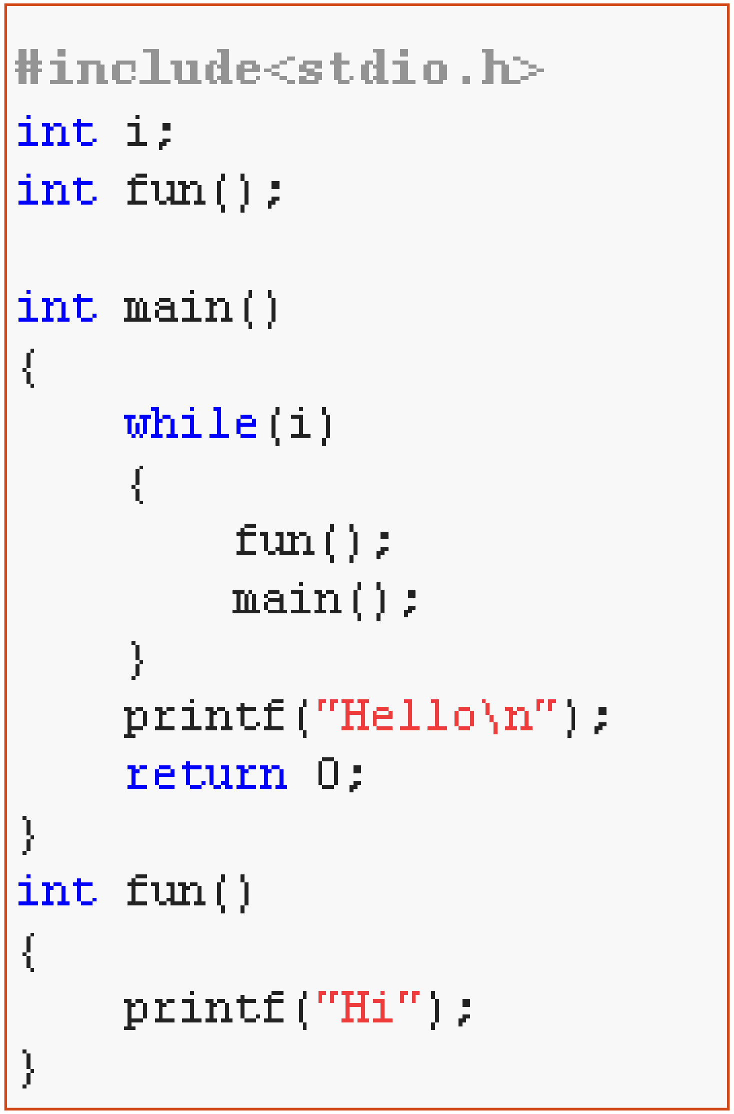
2. Solution

Answer: Infinite loop

Explanation:

- **Step 1**: *int no=5;* The variable *no* is declared as integer type and initialized to 5.
- **Step 2**: *reverse(no);* becomes *reverse(5);* It calls the function *reverse()* with '5' as parameter.
- The function reverse accept an integer number 5 and it returns '0'(zero) if(5 == 0) if the given number is '0'(zero) or else *printf("%d,", no);* it prints that number 5 and calls the function *reverse(5);*.
- The function runs infinetely because the there is a post-decrement operator is used. It will not decrease the value of 'n' before calling the reverse() function. So, it calls *reverse(5)* infinitely.
- Note: If we use pre-decrement operator like *reverse(--n)*, then the output will be 5, 4, 3, 2, 1. Because before calling the function, it decrements the value of 'n'.

3. Predict the output of the program.



```
#include<stdio.h>
int i;
int fun();

int main()
{
    while(i)
    {
        fun();
        main();
    }
    printf("Hello\n");
    return 0;
}
int fun()
{
    printf("Hi");
}
```

3.Solution

Answer: Hello

Explanation:

- Step 1: *int i;* The variable *i* is declared as an integer type.
- Step 1: *int fun();* This prototype tells the compiler that the function *fun()* does not accept any arguments and it returns an integer value.
- Step 1: *while(i)* The value of *i* is not initialized so this while condition is failed. So, it does not execute the *while* block.
- Step 1: *printf("Hello\n");* It prints "Hello".
- Hence the output of the program is "Hello".

1. Point out the error in the program.

```
#include<stdio.h>

int main()
{
    int a=10;
    void f();
    a = f();
    printf("%d\n", a);
    return 0;
}
void f()
{
    printf("Hi");
}
```

1. Solution

Answer: Error: Not allowed assignment

Explanation:

The function *void f()* is not visible to the compiler while going through main() function. So we have to declare this prototype *void f();* before to main() function. This kind of error will not occur in modern compilers.

2. Point out the error in the program.

```
#include<stdio.h>
int f(int a)
{
  a > 20? return(10): return(20);
}
int main()
{
    int f(int);
    int b;
    b = f(20);
    printf("%d\n", b);
    return 0;
}
```

2. Solution

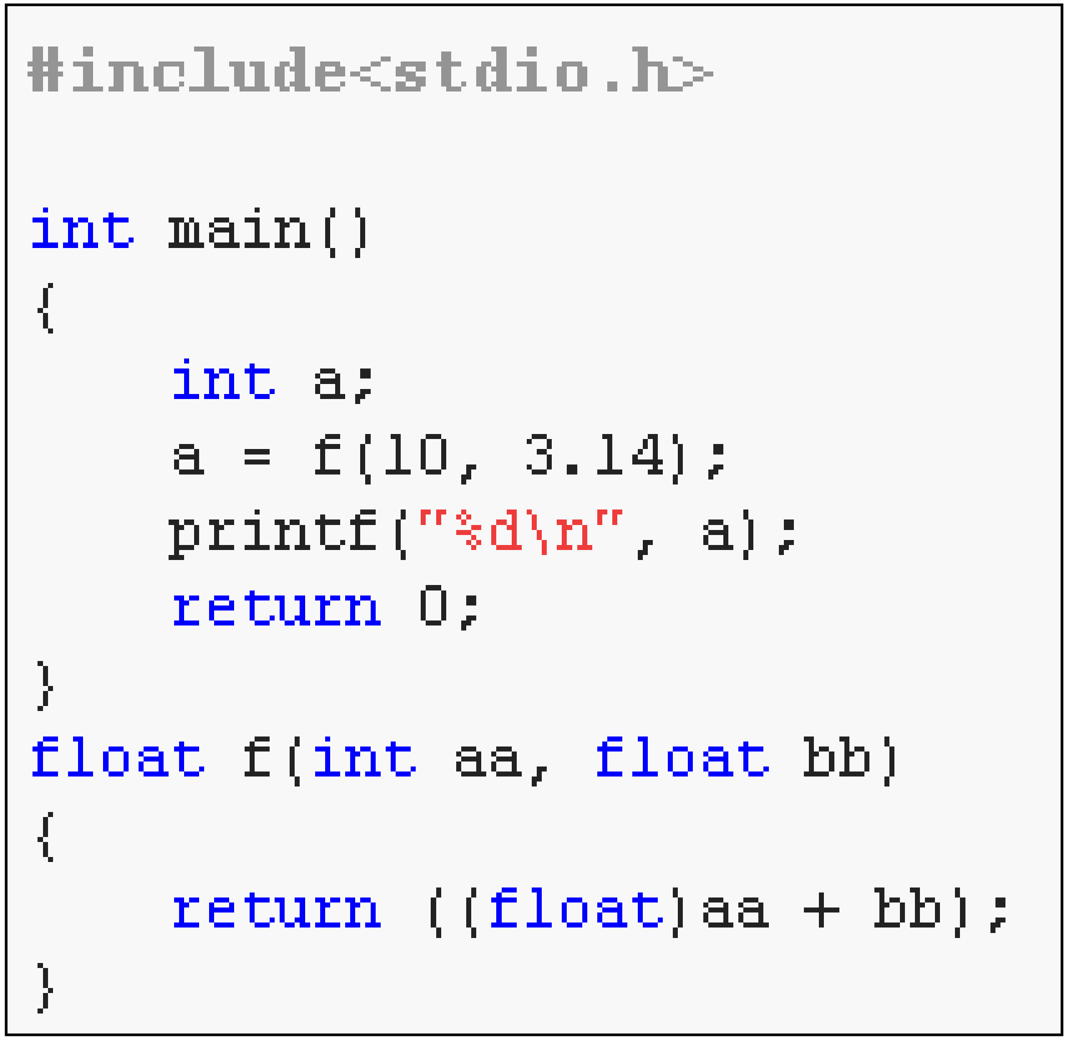
Answer:

Error: return statement cannot be used with conditional operators

Explanation:

In a ternary operator, we cannot use the return statement. The ternary operator requires expressions but not code.

3. There is an error in the below program. Which statement will you add to remove it?



```c
#include<stdio.h>

int main()
{
    int a;
    a = f(10, 3.14);
    printf("%d\n", a);
    return 0;
}
float f(int aa, float bb)
{
    return ((float)aa + bb);
}
```

3. Solution

Answer: Add prototype: *float f(int, float);*

Explanation:

- The correct form of function *f* prototype is

  float f(int, float);

Summary

- Discussed the modularization techniques in C.
- Illustration of functions with different parts – Prototype, Call and Definition.
- Discussed formal and actual parameters and passing mechanism.