Computer Programming & Problem Solving
CS100

Mrs Sanga G. Chaki
Department of Computer Science and Engineering
National Institute of Technology, Goa
April, 2023

Functions

1. A function is a self-contained block of statements that perform a specific, well defined task.
    a) a task that is always performed exactly in the same way
2. Example
    1. A function to add two numbers
    2. A function to find the largest of n numbers
3. A function will carry out its intended task whenever it is **called or invoked**
    1. Can be called multiple times

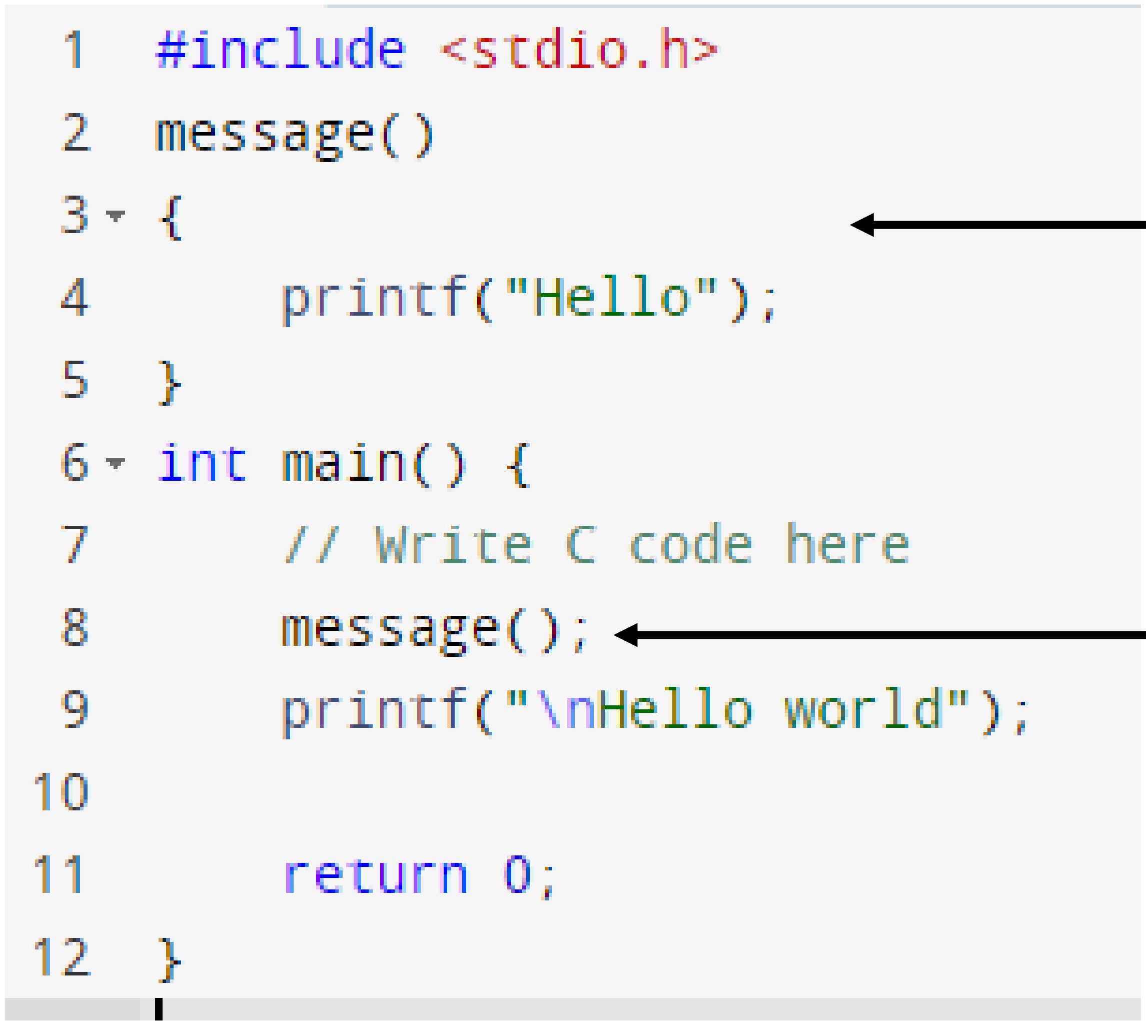
Function – Why Use Functions

1. Re-use: avoids rewriting the same code over and over.
2. Divide-and-conquer approach: easier to write programs and keep track of what they are doing

Functions

1. Every C program consists of one or more functions.
2. One of these functions must be called “main”.
3. Execution of the program always begins by carrying out the instructions in “main”.
4. Functions may call other functions (or itself) as instructions

Function – first example



```c
#include <stdio.h>
message()
{
      printf("Hello");
}
int main() {
      // Write C code here
      message();
      printf("\nHello world");

      return 0;
}
```

Function Definition – First Line, function body

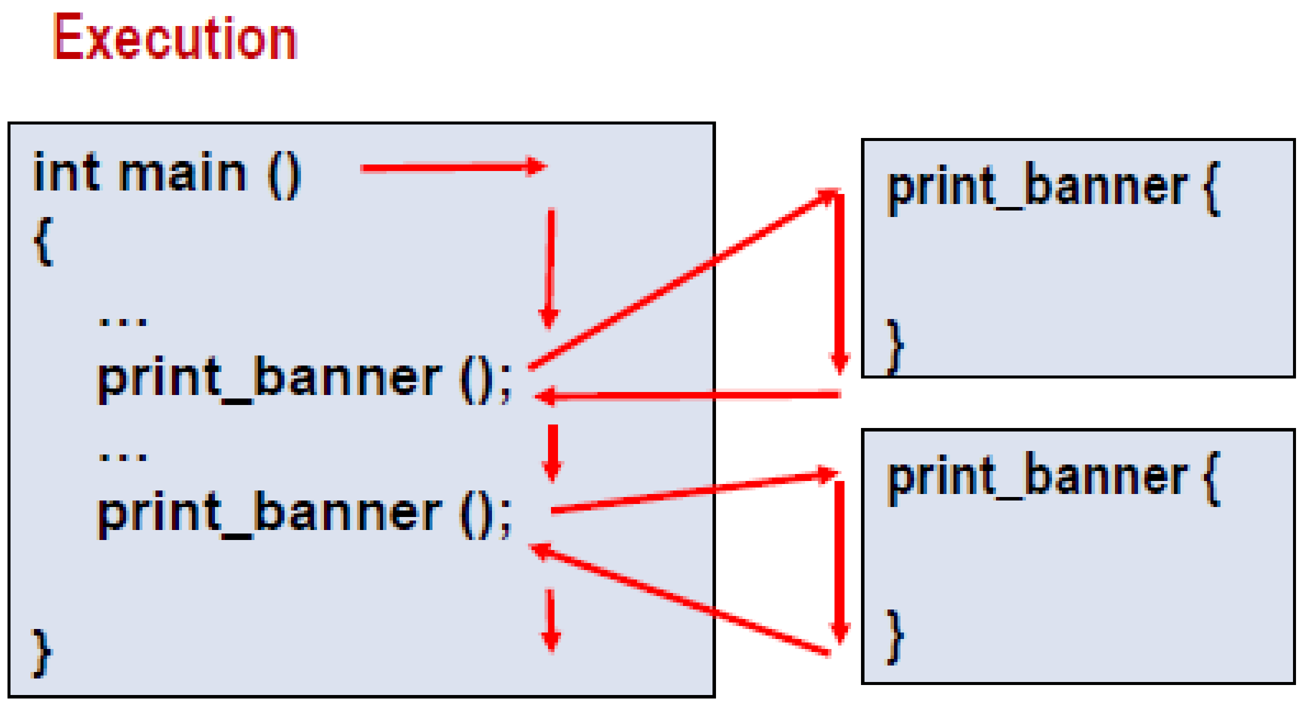
Function Call

Function – Control Flow

1. What is happening here?
2. Here, main( ) itself is a function and through it we are calling the function message( ).
3. What do we mean when we say that main( ) calls the function message( )?
4. We mean that the control passes to the function message( ).
5. The activity of main( ) is temporarily suspended; it falls asleep while the message( ) function wakes up and goes to work.
6. When the message( ) function runs out of statements to execute, the control returns to main( ), which comes to life again and begins executing its code at the exact point where it left off.
7. Thus, main( ) becomes the ‘calling’ function, whereas message( ) becomes the ‘called’ function.

Function – Control Flow

1. How/where to write function
2. How it is evoked/called?
3. void message()



```
int main ()
{
    ...
    print_banner ();
    ...
    print_banner ();

}
```

```
print_banner {

}
```

```
print_banner {

}
```

Multiple Functions – Control Flow

```
italy( )
{
        printf ( "\nI am in italy" ) ;

}
brazil( )
{
        printf ( "\nI am in brazil" ) ;
}
argentina( )
{
        printf ( "\nI am in argentina" ) ;
}
```

```
main( )
{
        printf ( "\nI am in main" ) ;
        italy( ) ;
        brazil( ) ;
        argentina( ) ;
}
```

What if we change the calling order?

OUTPUT

```
I am in main
I am in italy
I am in brazil
I am in argentina
```

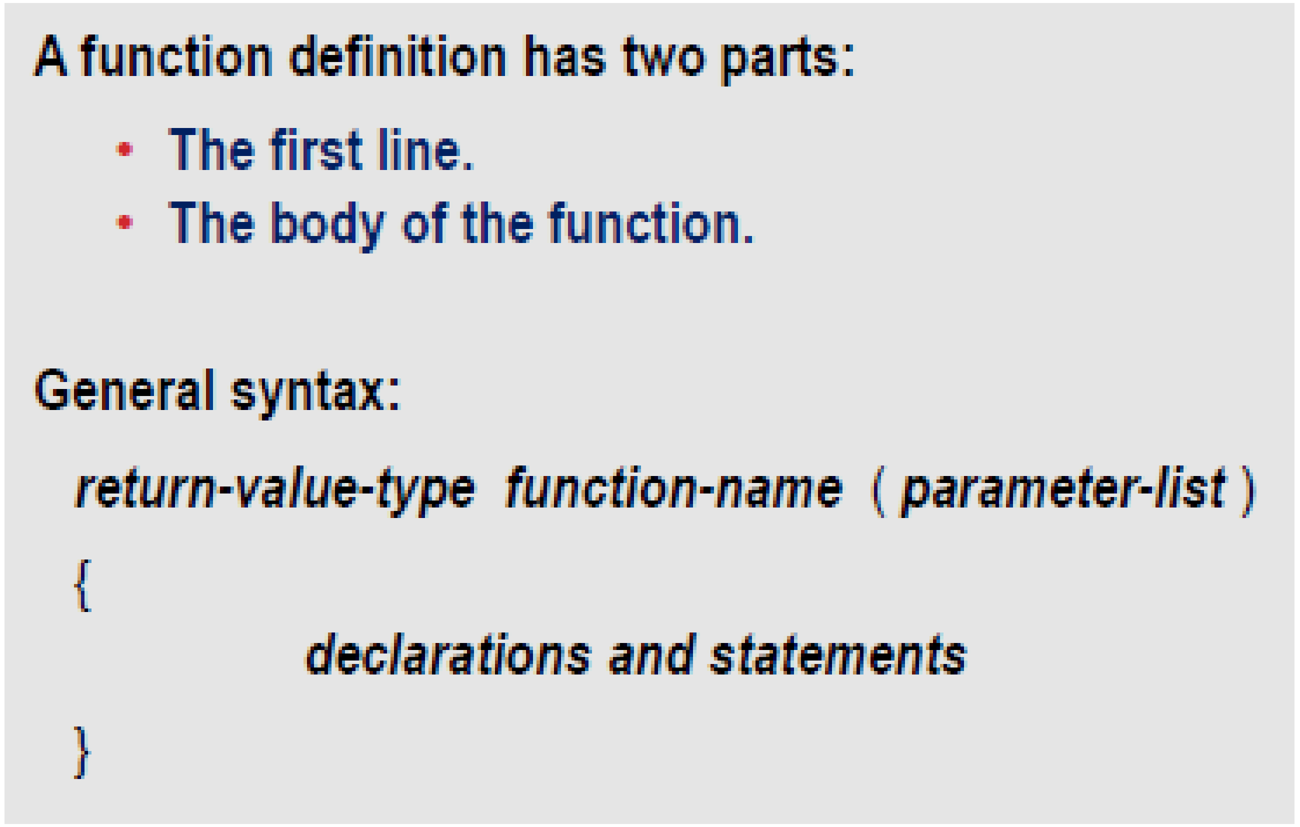
Function – Basic Rules

1. Any C program contains at least one function.
2. If a program contains only one function, it must be main( ).
3. If a C program contains more than one function, then one (and only one) of these functions must be main( ), because program execution always begins with main( ).
4. There is no limit on the number of functions that might be present in a C program.
5. Each function in a program is called in the sequence specified by the function calls in main( ).
6. After each function has performed its task, control returns to main().
7. When main( ) runs out of function calls, the program ends.

Function – Types

1. Library functions Ex. printf( ), scanf( ) etc.
2. User-defined functions Ex. argentina( ), brazil( ) etc.

Defining a Function



A function definition has two parts:

- The first line.
- The body of the function.

General syntax:

```
return-value-type  function-name  ( parameter-list )
{
        declarations and statements
}
```

Defining a Function

The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses.

- Each argument has an associated type declaration.
- The arguments are called *formal arguments* or *formal parameters*.

Example:

```
int  gcd  (int  A,  int  B)
```

int sum(int A, int B)

1. The mechanism used to convey information to the function is the ‘argument’

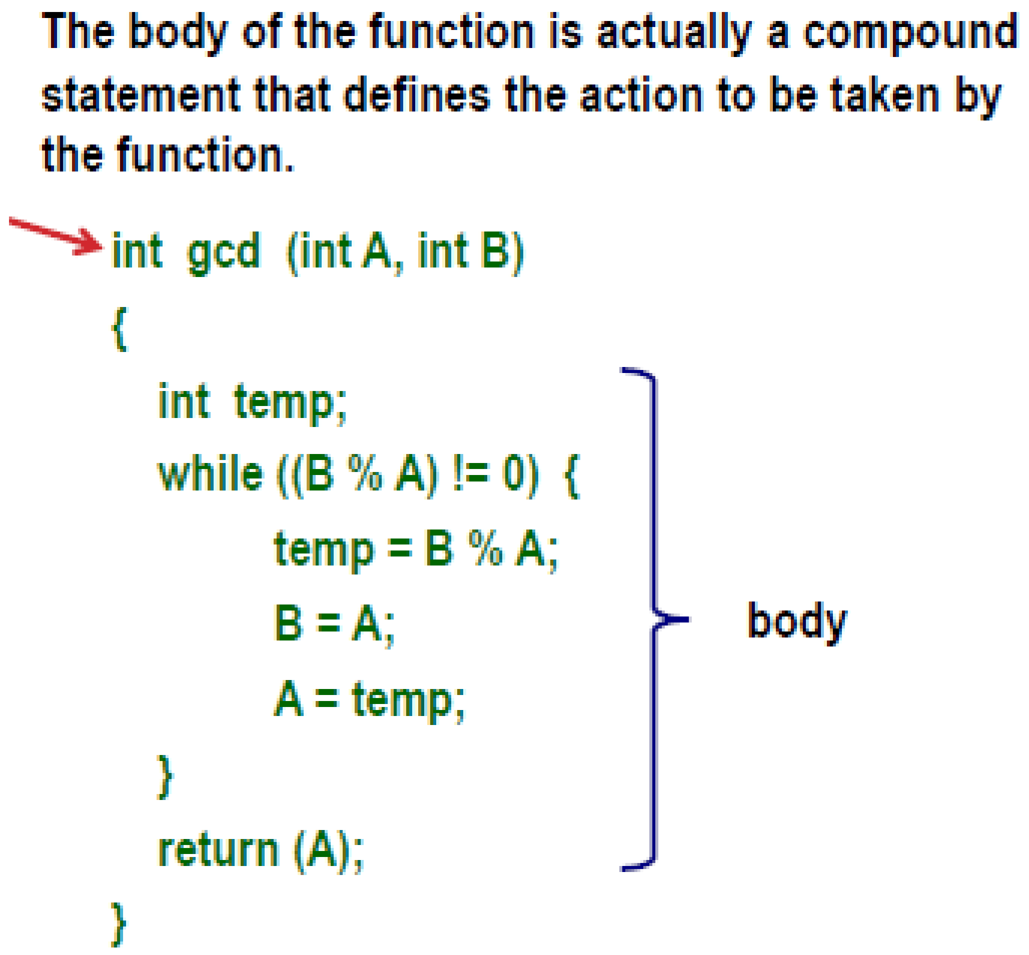
Calling a function - Rules

- Called by specifying the function name and parameters in an instruction in the calling function

- When a function is called from some other function, the corresponding arguments in the function call are called actual arguments or actual parameters
  - The function call must include a matching actual parameter for each formal parameter
  - Position of an actual parameter in the parameter list in the call must match the position of the corresponding formal parameter in the function definition
  - The formal and actual arguments must match in their data types

Defining a Function - Example

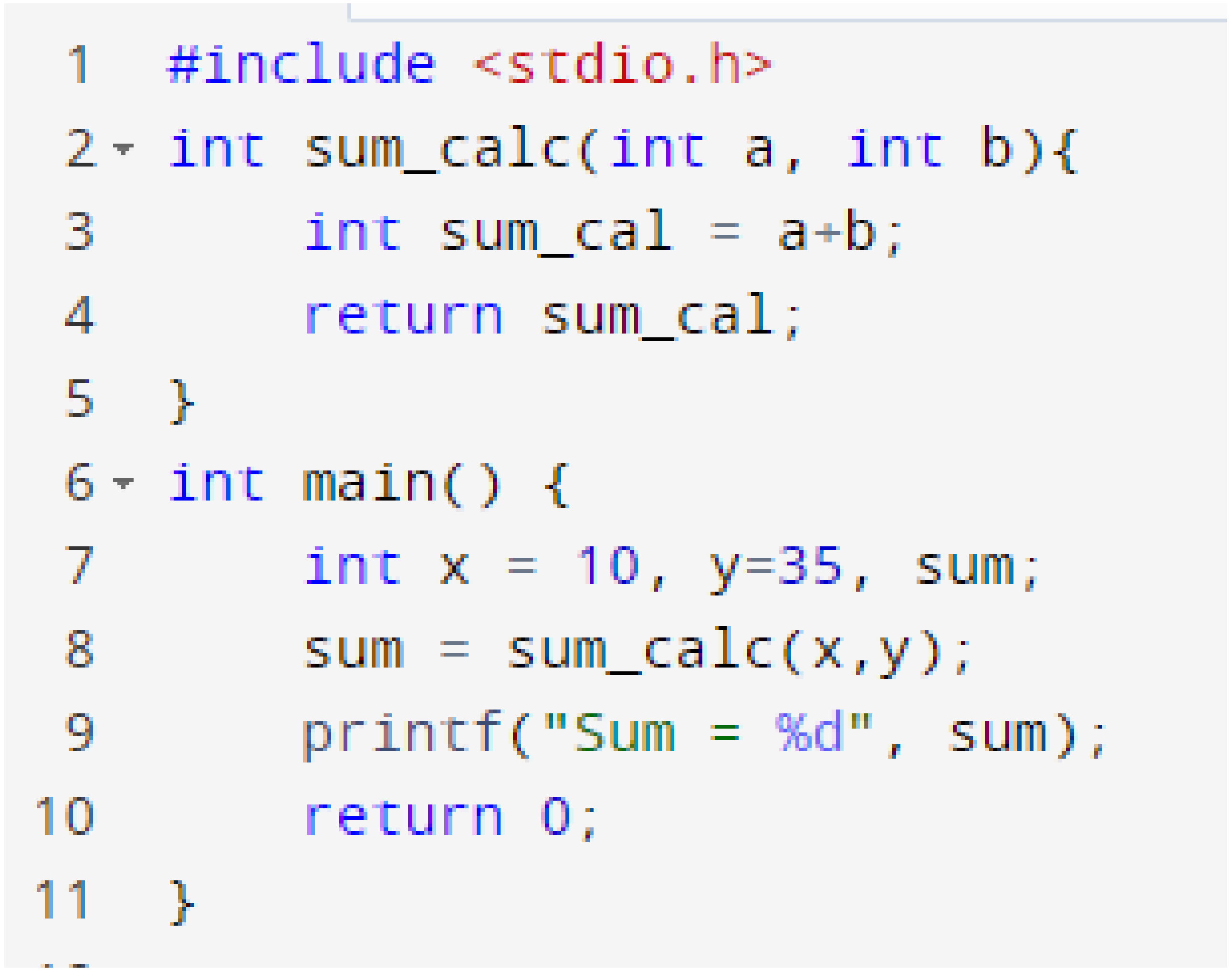
```
int sum(int A, int B)
{
        int sum;
        sum = A+B;
        return (sum);
}
```



The body of the function is actually a compound statement that defines the action to be taken by the function.

```
int gcd (int A, int B)
{
    int temp;
    while ((B % A) != 0) {
          temp = B % A;
          B = A;
          A = temp;
    }
    return (A);
}
```

Calling a function – Example – Actual and Formal Parameters

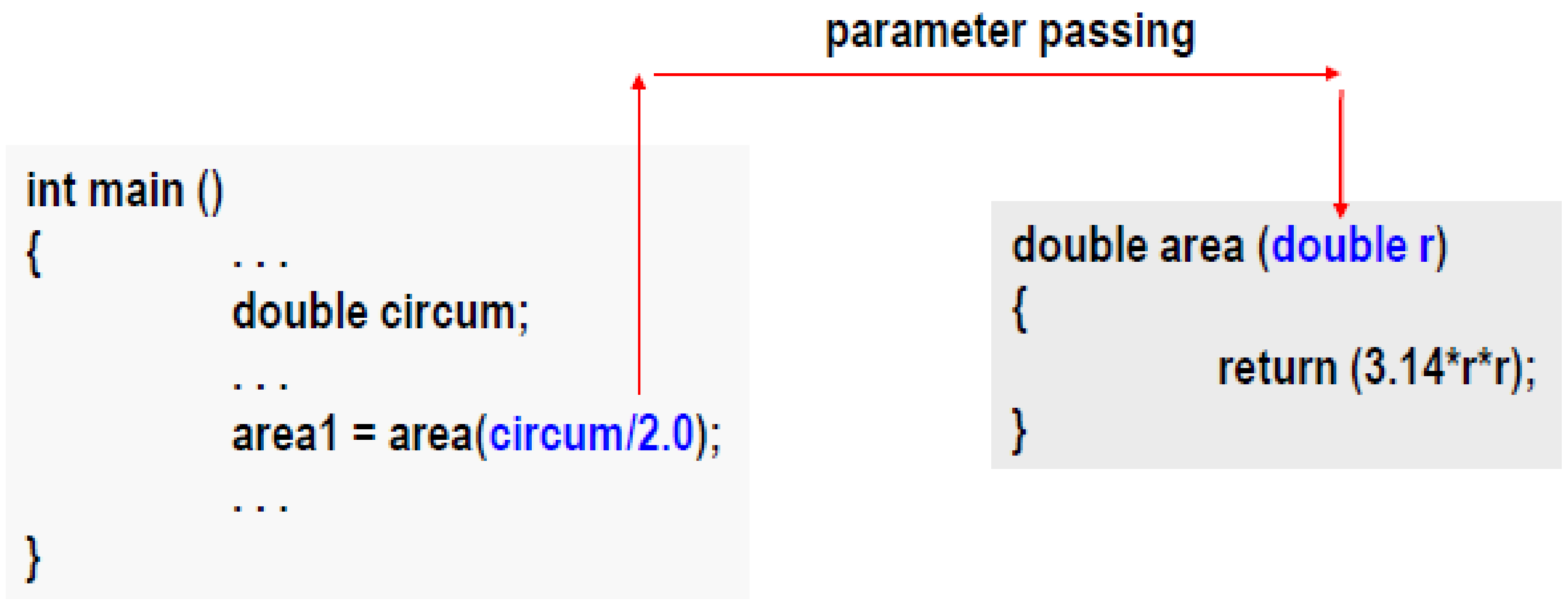


```c
#include <stdio.h>
int sum_calc(int a, int b){
      int sum_cal = a+b;
      return sum_cal;
}
int main() {
      int x = 10, y=35, sum;
      sum = sum_calc(x,y);
      printf("Sum = %d", sum);
      return 0;
}
```

Parameter /Value Passing

1. The mechanism used to convey information to the function is the 'argument'

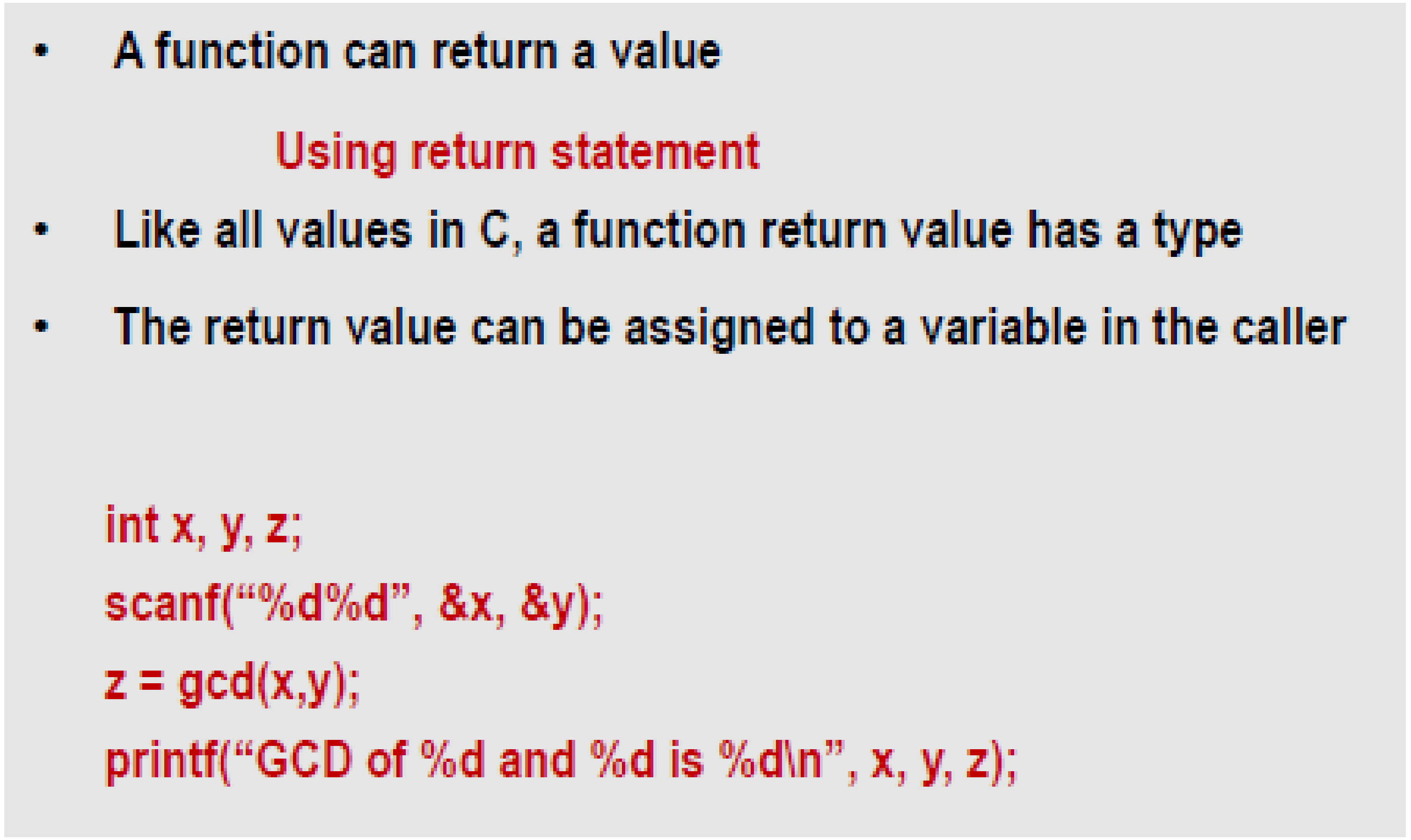
When the function is executed, the value of the actual parameter is copied to the formal parameter



```
int main ()
{            . . .
             double circum;
             . . .
             area1 = area(circum/2.0);
             . . .
}
```

```
double area (double r)
{
             return (3.14*r*r);
}
```

Return Value



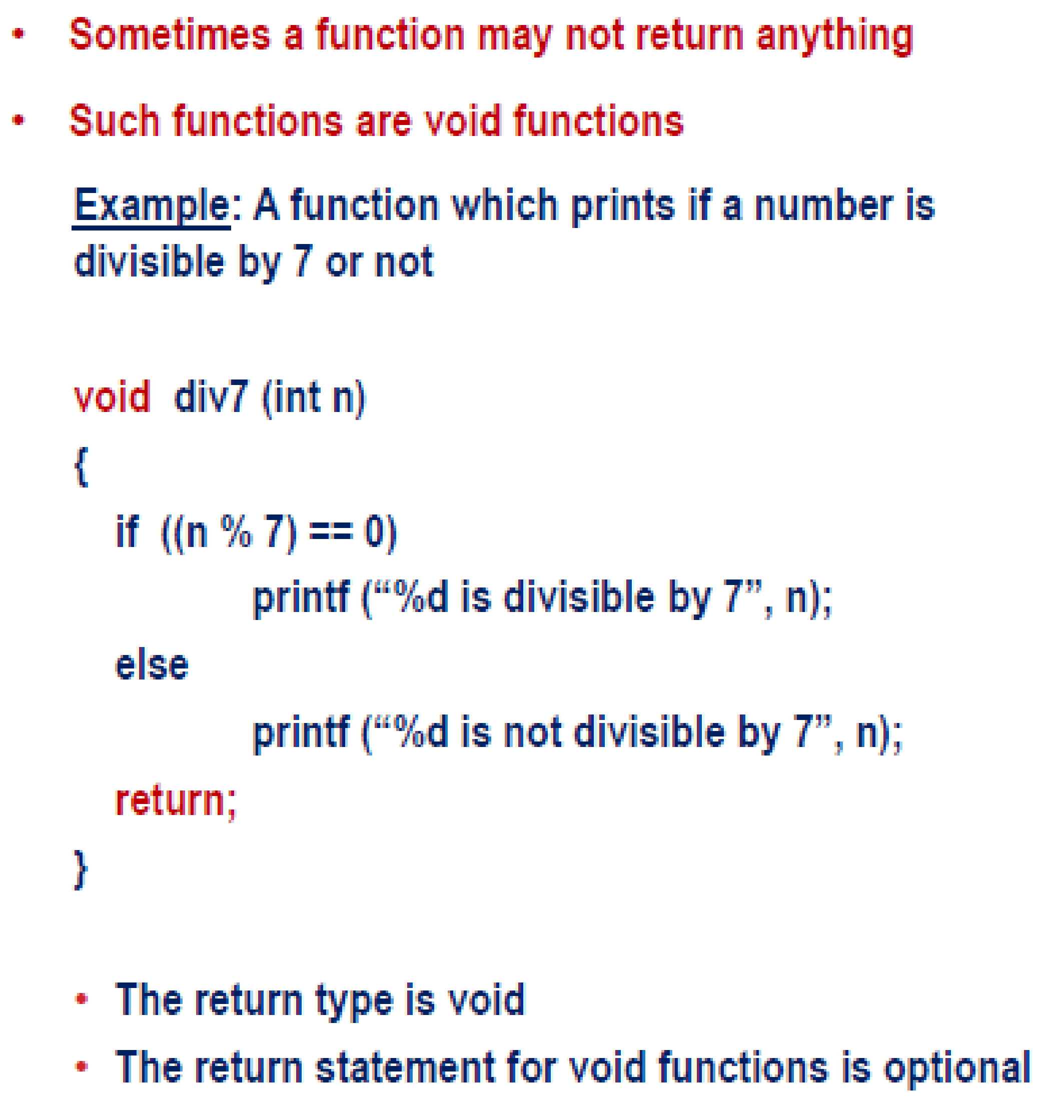
- A function can return a value

  Using return statement

- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```
int x, y, z;
scanf(“%d%d”, &x, &y);
z = gcd(x,y);
printf(“GCD of %d and %d is %d\n”, x, y, z);
```

Return Value



- Sometimes a function may not return anything
- Such functions are void functions

Example: A function which prints if a number is divisible by 7 or not

```
void  div7 (int n)
{
   if  ((n % 7) == 0)
            printf ("%d is divisible by 7", n);
   else
            printf ("%d is not divisible by 7", n);
   return;
}
```

- The return type is void
- The return statement for void functions is optional

```
void sum(int A, int B)
{
        int sum;
        sum = A+B;
        printf("sum = %d",sum);
        //return (sum);
}
```

We return sum if there are further calculations in main() using sum.

The Return statement - rules

In a value-returning function (result type is not void), return does two distinct things

- specify the value returned by the execution of the function
- terminate that execution of the callee and transfer control back to the caller

A function can only return one value

- The value can be any expression matching the return type
- but it might contain more than one return statement.

In a void function

- return is optional at the end of the function body.
- return may also be used to terminate execution of the function explicitly before reaching the end.
- No return value should appear following return.

The Return statement

```
void compute_and_print_itax ()
{
   float income;
   scanf (“%f”, &income);
   if (income < 50000)   {
               printf (“Income tax = Nil\n”);
               return;  /* Terminates function execution */
   }
   if (income < 60000)   {
               printf (“Income tax = %f\n”, 0.1*(income-50000));
               return;  /* Terminates function execution */
   }
   if (income < 150000) {
                printf (“Income tax = %f\n”, 0.2*(income-60000)+1000);
               return ; /* Terminates function execution */
   }
   printf (“Income tax = %f\n”, 0.3*(income-150000)+19000);
}
```

Function Prototypes – Declaring function

Usually, a function is defined before it is called

- main() is the last function in the program written
- Easy for the compiler to identify function definitions in a single scan through the file

However, many programmers prefer a top-down approach, where the functions are written after main()

- Must be some way to tell the compiler
- Function prototypes are used for this purpose
    - Only needed if function definition comes after use

- Function prototypes are usually written at the beginning of a program, ahead of any functions (including main())
- Prototypes can specify parameter names or just types (more common)
- Examples:

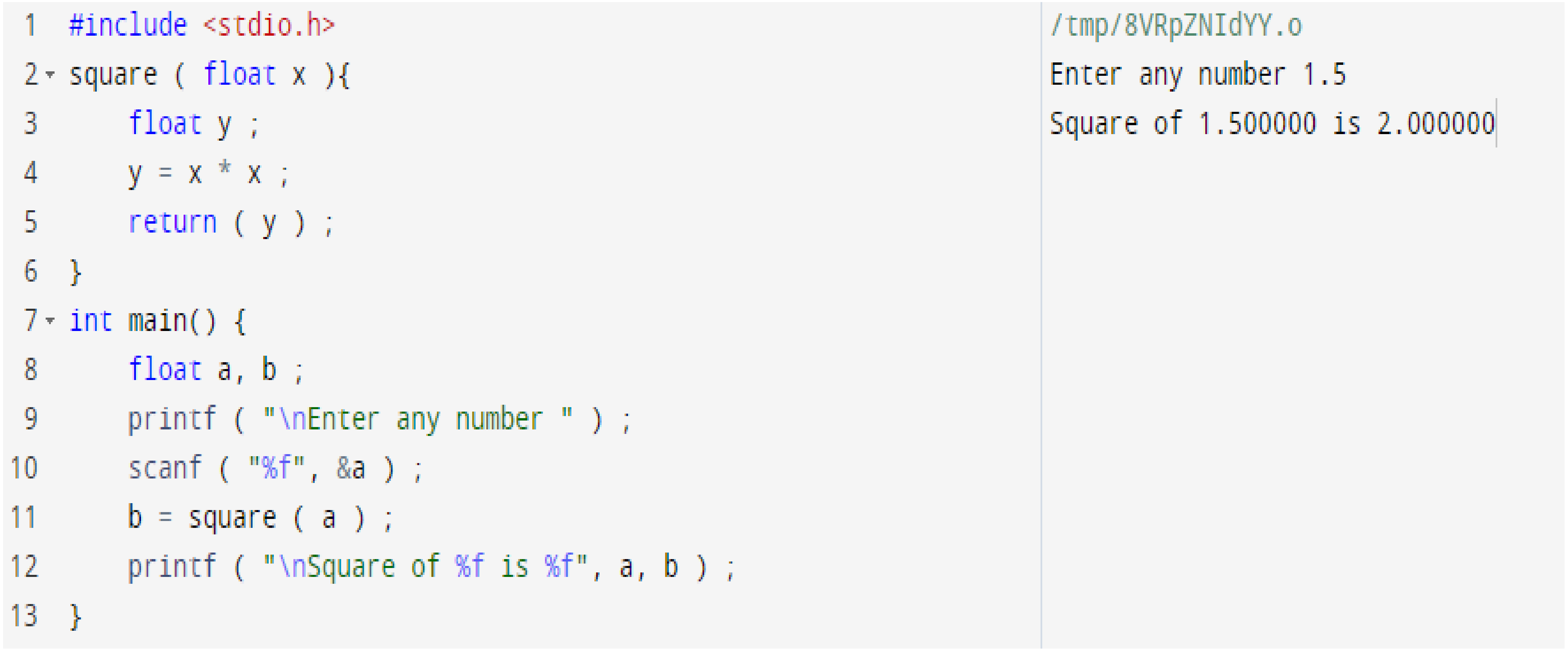
```
int  gcd (int , int );
void div7 (int number);
```

- Note the semicolon at the end of the line.
- The parameter name, if specified, can be anything; but it is a good practice to use the same names as in the function definition

Function Prototypes – Example

```
#include <stdio.h>
int sum( int, int );
int main( )
{
        int x, y;
        scanf(“%d%d”, &x, &y);
        printf(“Sum = %d\n”, sum(x, y));
}
int sum (int a, int b)
{
        return(a + b);
}
```

Function Example



```c
#include <stdio.h>
square ( float x ){
     float y ;
     y = x * x ;
     return ( y ) ;
}
int main() {
     float a, b ;
     printf ( "\nEnter any number " ) ;
     scanf ( "%f", &a ) ;
     b = square ( a ) ;
     printf ( "\nSquare of %f is %f", a, b ) ;
}
```

```
/tmp/8VRpZNIdYY.o
Enter any number 1.5
Square of 1.500000 is 2.000000
```

Function Categories

1. So according to whether arguments are present or not and any values are returned or not, functions are of following types:
    a) No arguments no return value
    b) No arguments, return value
    c) With arguments, no return value
    d) With arguments, with return value