FUNCTIONS

INTRODUCTION TO FUNCTIONS

- Functions are the essence of modular/structured programming languages.
- The basic idea behind modular programming is to take a large problem and divide into smaller sub-problems.
- Then we design a small piece of code exclusively to handle each sub-problem. These are called functions.
- A function is a set of statements that take inputs, do some specific computation and produces output.
- Functions are also called as modules or sub-routines or procedures

WHY DO WE NEED FUNCTIONS?

- Functions are important because:
  - They take large complicated programs and to divide them into smaller manageable pieces.
  - Functions designed to be used with several programs. These functions perform a specific task and thus are usable in many different programs.
  - Functions help us in reducing code redundancy
  - Functions make code modular and easy to read.
  - Functions provide abstraction.

HOW TO WRITE FUNCTIONS?

- There are three main components of a function:
  - Function definition: It defines the body of the function i.e the actual task or code that the function executes.
  - Function call: To use a function, you will have to call that function to perform the defined task.
  - Function declaration/prototype: It informs to the compiler specific features of the function.

FUNCTION DEFINITION

- Syntax:

```
return_type function_name ( function_arguments )
{
//body of the function
}
```

- function_name: Used to refer the function when a task needs to be performed.
- function_arguments: The input to the function
- return_type: Once the function finishes computation, it returns some result. The datatype of the returned value is specified in the return_type.

FUNCTION CALL

- Syntax:

  function_name (function arguments)

- function_name : Used to specify which function do we want to execute
- function_arguments : Provide the values to the function which goes as input to the function definition.

FUNCTION - EXAMPLE

- Define a function, that adds two integers and returns the sum.

datatype arg1, datatype arg2…. datatype argn

```
    int add ( function_arguments )
    {
Step 1: Add function arguments
Step 2: Store result in a third variable
Step 3: Return the result
    }
```

FUNCTION - EXAMPLE

- Define a function, that adds two integers and returns the sum.

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

FUNCTION- CALL

- Now, that you have defined the function “add”, can you write the call statement?

value1, value2… valuen

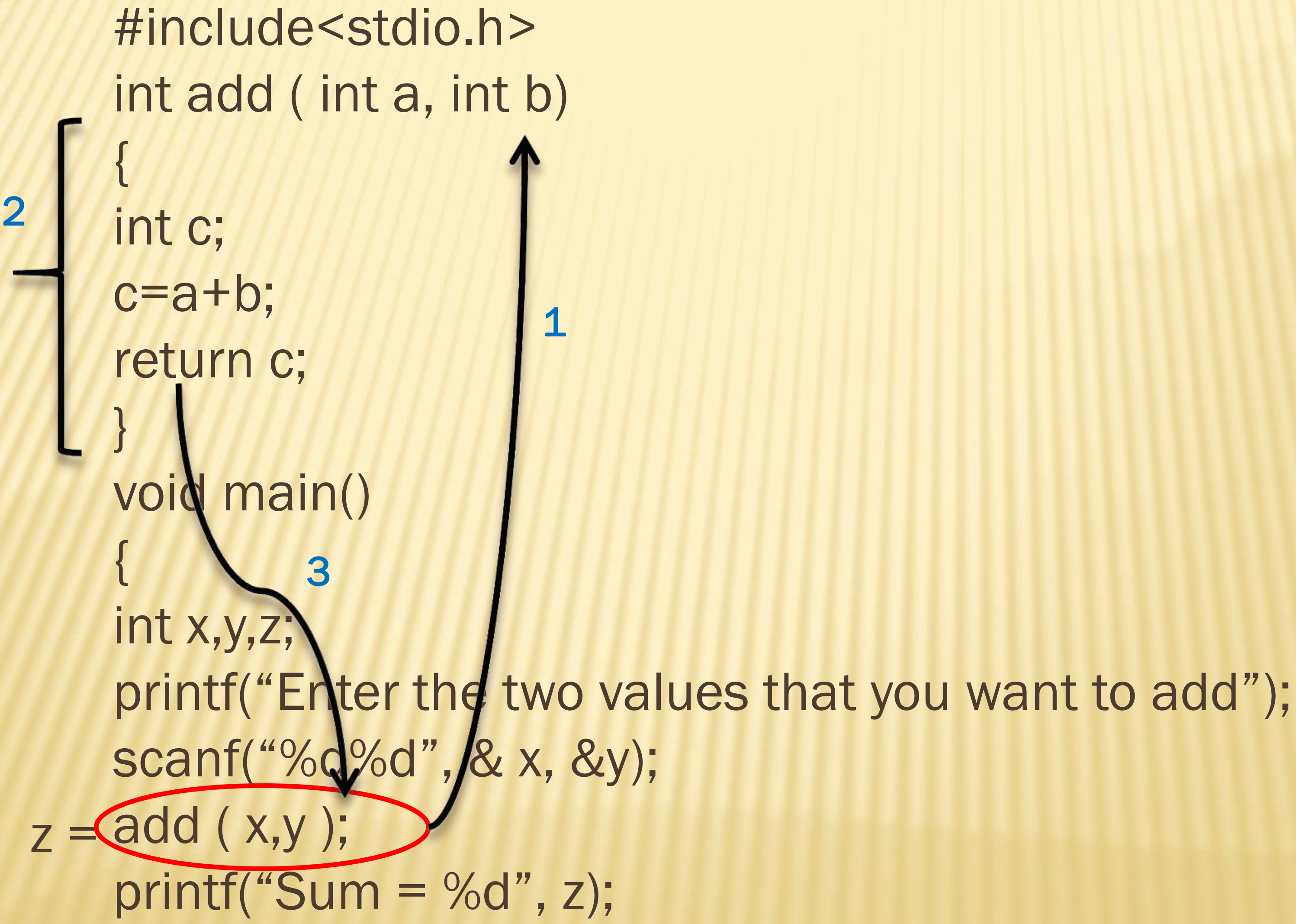
add (function_arguments)

- What should be the function arguments?
  - Any two values that you actually wish to add.
  - Can be constants, variables , expressions etc.
  - Example:

add (5,8)

  - Where should the call statement be?
    - For now, let us place it inside void main()

FUNCTION EXAMPLE – LET US PUT IT TOGETHER



```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
int x,y,z;
printf(“Enter the two values that you want to add”);
scanf(“%d%d”, & x, &y);
z = add ( x,y );
printf(“Sum = %d”, z);
```

HOW FUNCTIONS WORK?

1. While creating a C function, you give a definition of what the function has to do.
2. To use a function, you will have to call that function to perform the defined task.
3. When a program calls a function, the program control is transferred to the function definition.
4. The arguments given in function call (called as actual parameters) are transferred/copied into the arguments in the function definition (called as formal parameters) from left to right.
5. Then, the called function performs a defined task and when its return statement is executed, it returns the program control back to the line of call.

QUICK EXERCISE

```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
int x,y,z;
x=10; y=15;
z = add ( x,y );
printf(“Sum = %d”, z);
}
```

Output:

```
Sum = 25
```

QUICK EXERCISE

```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
int x,y,z;
x=10; y=15;
add ( x,y );
printf("Sum = %d", z);
}
```

Output:
Sum = Garbage Value

If the returned value is not stored in a variable, it will get lost after the line of call.

QUICK EXERCISE

```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
int x,y,z;
z=add ( 10,15 );
printf("Sum = %d", z);
}
```

Output:

```
Sum = 25
```

QUICK EXERCISE

```
 #include<stdio.h>
 int add ( int a, int b)
 {
 int c;
 c=a+b;
 return c;
 }
 void main()
 {
 int x,y,z;
 z=add ( 10,15 ) + add ( 4,6 );
 printf(“Sum = %d”, z);
}
```

Output:

```
Sum = 35
```

QUICK EXERCISE

```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
int x,y,z;
z=add ( add (12, 3), 8 );
printf(“Sum = %d”, z);
}
```

Output:

```
Sum = 23
```

QUICK EXERCISE

```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
printf(“Sum = %d”, add(3,4));
}
```

Output:

```
Sum = 7
```

QUICK EXERCISE

```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
int c;
add( 6,4 );
printf(“Sum = %d”, c);
}
```

Output:
Sum = Garbage Value

Variable “c” in void main() is different than variable “c” in the function definition of “add”.

QUICK EXERCISE

```
#include<stdio.h>
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
void main()
{
int c;
c = add( 6,4,1 );
printf(“Sum = %d”, c);
}
```

Output:
Error

Number of parameters do not match

FUNCTION DECLARATION/PROTOTYPE

- In all the previous cases, we see that the definition of “add()” function comes BEFORE it’s function call.
- However, a function can also be defined AFTER it its calling statement.
- In such a case, the function needs to have a declaration.
- Syntax:

  return_type function_name (function_arguments);
- Function prototype is NOT required if the definition comes before the call.

EXAMPLE OF FUNCTION DECLARATION

```
#include<stdio.h>
int add (int,int);
void main()
{
int c;
c = add( 6,4 );
printf(“Sum = %d”, c);
}
int add ( int a, int b)
{
int c;
c=a+b;
return c;
}
```

→ Function prototype

Points to note:

1. For a function prototype, it is enough to mention ONLY data type of formal arguments. Parameter names can be ignored.
2. Semi-colon(;) is very important for declaration.
3. Declaration can be made inside or outside main() function. It MUST come before the call.

QUICK EXERCISE

- Identify whether the statements given below are function call/ declaration/definition

```
1. printf(“%d”, x); → Function call
2. c= sqrt(5); → Function call
3. int *p= malloc( n *sizeof(int)); → Function call
4. int add(int a)
     {
     a=a+1;          → Function definition
     }
5. int add(int a); → Function prototype
6. int main()
     {
                     → Function definition

     }
```

DO'S AND DON'T'S FOR FUNCTIONS

- The actual parameters have NO data type. They are simply VALUES that we pass to the function definition.
- "return" keyword will ALWAYS take the control back to the line of call.
- Functions must ALWAYS accept input through arguments. You must NOT write "scanf()" inside the function definition.
- The names of actual parameters and formal parameters can be same or different. However, they represent two different entities.
- A function definition CANNOT be placed inside another function definition.
- A function call however CAN be placed within another call.
- If we have,

```
void main()
{
int c;
c=add(3,6);
}
```

→ void main() – Is the CALLING function

add () – Is the CALLED function

QUICK EXERCISE

- Define a function to calculate the factorial of a number. Use that function to calculate ${}^{n}p_{r}$ and ${}^{n}c_{r}$
- Define a function “large” that compares two numbers and returns the larger of the two. Use that function to calculate the largest of three numbers.