User Defined Function

Unit 8

Introduction

- A function is a block of code that performs a specific task.
- There are two types of function
    - Library/ Built in Function
    - User Defined Function

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

- The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file. Hence, to use the printf() function, we need to include the stdio.h header file using #include <stdio.h>.
- The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.
- scanf()
- strcpy()
- strcmp()
- strlen()

User Defined Function

- C allows programmer to define their own function according to their requirement.
- Such functions created by the user are known as user-defined functions.

Function Prototype (Declaration)

- Every function in C programming should be declared before they are used.
- Function prototype gives compiler information about function name, types of argument to be passed, and a return type.

  Syntax: return-type function_name(parameters list)

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

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

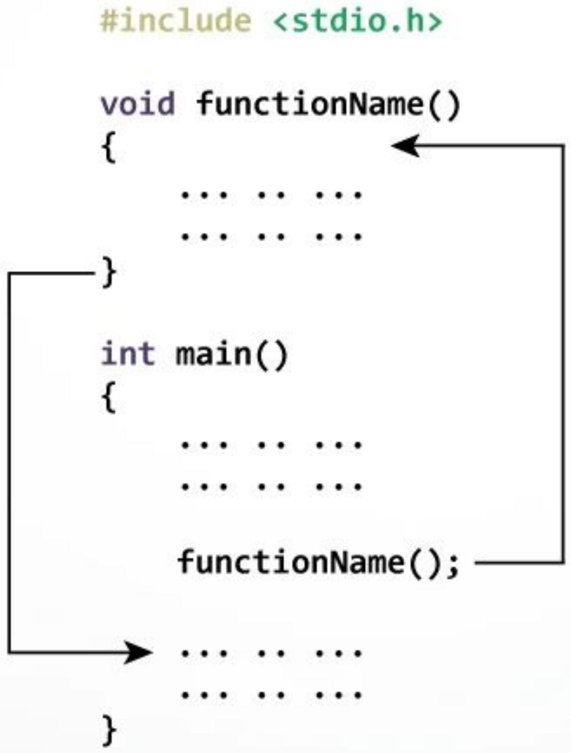
The execution of a C program begins from the main() function.

When the compiler encounters functionName();, control of the program jumps to “**functionName()**”

And, the compiler starts executing the codes inside functionName().

The control of the program jumps back to the main() function once code inside the function definition is executed.

How function works in C programming?



```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

Note, function names are identifiers and should be unique.

This is just an overview of user-defined functions

Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

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

}
```

Return type: A function may return a value. The return type is the data type of the value of the function returns. Some functions perform the desired operation 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 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. 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.

Calling a function

Control of the program is transferred to the user-defined function by calling it.

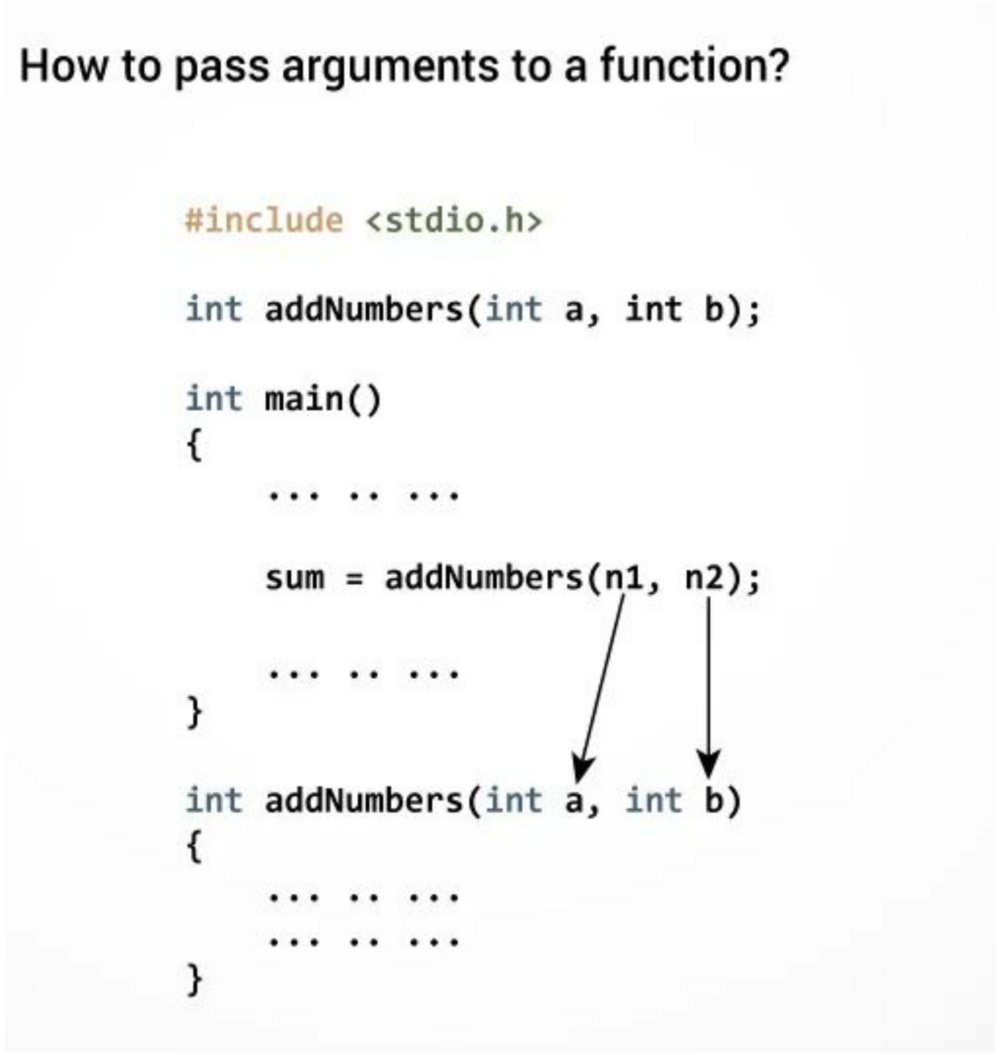
Syntax of function call

```
functionName(argument1, argument2, ...);
```

```
#include <stdio.h>

int addNumbers(int a, int b);         // function prototype

int main()

{

    int n1,n2,sum;

    printf("Enters two numbers: ");

    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);        // function call

    printf("sum = %d",sum);

    return 0;          }

int addNumbers(int a, int b) {        // function definition

    int result;

    result = a+b;

    return result;            }      // return statement
```

In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.

The parameters a and b accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

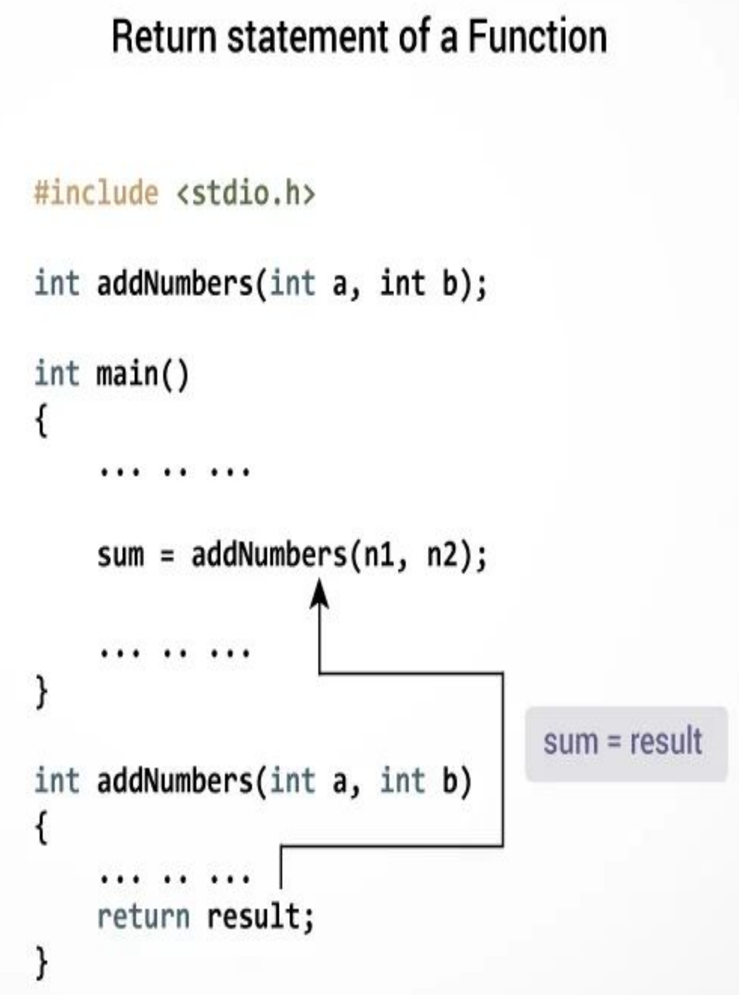


How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the result variable is returned to the main function. The sum variable in the main() function is assigned this value.



Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

Types of User Defined Function

- Functions with no argument and no return value
- Functions with no argument and return value
- Functions with arguments and no return value
- Functions with argument and return value

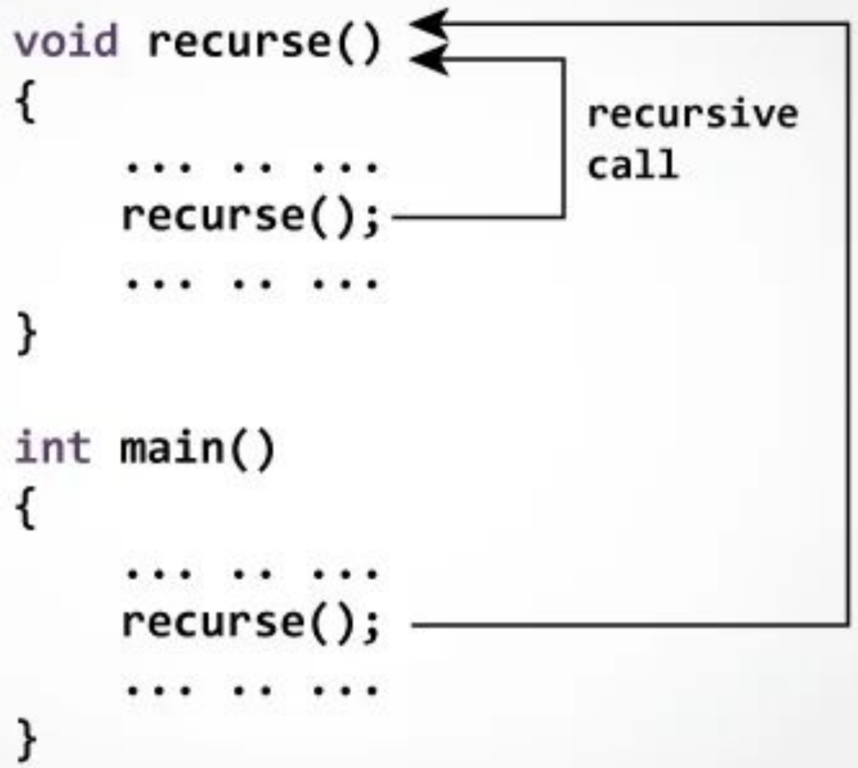
Recursive Function

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

How recursion works?

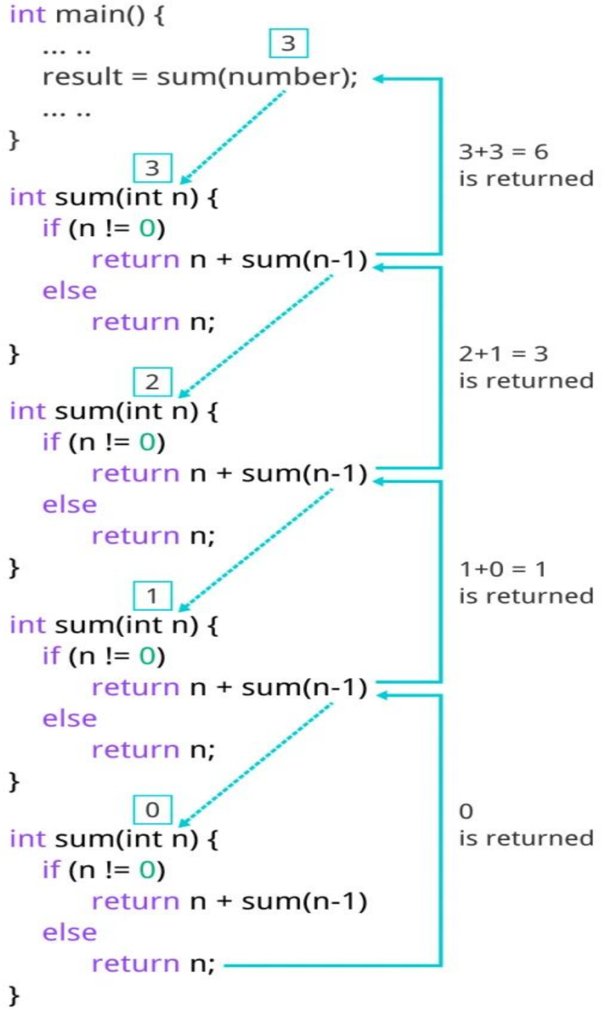
```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}
int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

How does recursion work?



```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

```
#include <stdio.h>

int sum(int n);

int main() {

    int number, result;

    printf("Enter a positive integer: ");

    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);

    return 0;

}

int sum(int n) {

    if (n != 0)

        // sum() function calls itself

        return n + sum(n-1);

    else

        return n;          }
```



```
int main() {
   ... ..
   result = sum(number);
   ... ..
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}

int sum(int n) {
   if (n != 0)
       return n + sum(n-1)
   else
       return n;
}
```

Factorial of a number using recursion

```
#include<stdio.h>

long int multiplyNumbers(int n);

int main() {

        int n;

        printf("Enter a positive integer: ");

        scanf("%d",&n);

        printf("Factorial of %d = %ld", n,
multiplyNumbers(n));

        return 0;

}
```

```
long int multiplyNumbers(int n) {

        if (n>=1)

        return n*multiplyNumbers(n-1);

        else

        return 1;

}
```

C Storage Class

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

Storage class

Storage specifier	**Storage**	**Initial Value**	**Scope**	**Life**
auto	stack	Garbage	Within block	End of block
extern	Data segment	0	Global multiple files	Till end of program
static	Data segment	0	Within block	Till end of program
register	CPU register	garbage	Within block	End of block

Auto Storage class

The variables defined using auto storage class are called as local variables.
Auto stands for automatic storage class.
A variable is in auto storage class by default if it is not explicitly specified.
The scope of an auto variable is limited with the particular block only.
Once the control goes out of the block, the access is destroyed.
This means only the block in which the auto variable is declared can access it.
A keyword auto is used to define an auto storage class.
By default, an auto variable contains a garbage value.

```
#include <stdio.h>
int main( )
{
  auto int j = 1;
  {
    auto int j= 2;
    {
      auto int j = 3;
      printf ( " %d ", j);
    }
    printf ( "\t %d ",j);
  }
  printf( "%d\n", j);

    return 0;
}
```

Extern storage class

Extern stands for external storage class.
Extern storage class is used when we have global functions or variables which are shared between two or more files.
Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.
The variables defined using an extern keyword are called as global variables.
These variables are accessible throughout the program.
Notice that the extern variable cannot be initialized it has already been defined in the original file.

Static Storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

- Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
- Static global variables are global variables visible **only to the file in which it is declared.**

Local Variable

The variables declared inside a block are automatic or local variables. The local variables exist only inside the block in which it is declared.

Let's take an example.

```
#include <stdio.h>

int main(void) {

  for (int i = 0; i < 5; ++i) {

     printf("C programming");

  }

 // Error: i is not declared at this point

  printf("%d", i);

  return 0;

}
```

```
int main() {
    int n1; // n1 is a local variable to main()
}

void func() {
   int n2;  // n2 is a local variable to func()
  }
```

Global Variable

Variables that are declared outside of all functions are known as external or global variables. They are accessible from any function inside the program.

```
#include <stdio.h>

void display();

int n = 5;  // global variable

int main()

{

    ++n;

    display();

    return 0;

}

void display()

{

    ++n;

    printf("n = %d", n);

  }
```

```
#include <stdio.h>
int main()
{
int p = 10,i;
printf("%d ",p);

{
int p = 20;
for (i=0;i<3;i++)
{
printf("%d" ,p); // 20 will be printed here 3 times because it is the given local value of p
}

}

printf("\n%d ",p); // 11 will be printed here since the scope of p = 20 has finally ended.

}
```

```
#include <stdio.h>

int main()

{

extern int x; // The compiler will start searching here if a variable x has been defined
and initialized in the program somewhere or not.

printf("%d",x);

}

int x = 20;
```

Register Variable

The `register` keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.

Unless you are working on embedded systems where you know how to optimize code for the given application, there is no use of register variables.

```c
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

```c
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Static Variable

A static variable is declared by using the `static` keyword.

The value of a static variable persists until the end of the program.

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

int main()
{
    display();
    display();
}
void display()
{
    static int c = 1;
    c += 5;
    printf("%d  ",c);
  }
```

Fibonacci Series using recursion

```
#include <stdio.h>
int fibonacci(int i){
   if(i==0) return 0;
   else if(i==1) return 1;
   else return (fibonacci(i-1)+fibonacci(i-2));
}
int main()
{
      int n,i,f;
      scanf("%d",&n);
      for(i=0;i<n;i++){
      f=fibonacci(i);
      printf("%d ",f);
      }
      return 0;
}
```

WAP to add two numbers without using + operator

```
#include<stdio.h>
int main() {
   int num1, num2, i;
    printf("please enter first number: ");
    scanf("%d",&num1);
    printf("please enter second number: ");
    scanf("%d",&num2);
    for(i=1;i<=num2;i++){
        num1++;
    }
    printf("sum = %d", num1);
}
```

Preprocessor Directives

In C programming language, preprocessor directives are commands that are processed by the preprocessor before the compilation of the code. Preprocessor directives begin with the # symbol and are used to instruct the preprocessor to perform specific tasks.

Here are some common preprocessor directives in C:

1. #include: This directive is used to include header files in the C code. The header files contain declarations of functions, macros, and other objects that are needed in the code.
2. #define: This directive is used to define constants, macros, and function-like macros.
3. #ifdef and #ifndef: These directives are used to conditionally compile code based on whether a macro is defined or not.
4. #if, #elif, and #else: These directives are used for conditional compilation. They allow parts of the code to be compiled or ignored based on certain conditions.
5. #pragma: This directive is used to give special instructions to the compiler, such as setting optimization levels, controlling alignment of data, and so on.
6. #error: This directive is used to generate a compilation error message with a specified text.

These are some of the common preprocessor directives used in C programming. They provide a powerful tool for manipulating the code before compilation.