TOPICS
FUNCTIONS AND STORAGE CLASSES

Anusuya.R.P(1831004)
Harini.S(1831020)

FUNCTIONS

- 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.

FUNCTION DECLARATION

- 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.

Syntax of a function

```
return_type function_name (argument list)
{
    Set of statements – Block of code
}
```

DEFINING A 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.

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
}
```

BUILT-IN FUNCTION

Built-in Functions

C has many built-in functions that you can use in your programs. So far we have learned 15 built-in functions:

main( )	gets( )	strlen( )	strcat( )	isdigit( )
printf( )	puts( )	strcmp( )	strstr( )	isupper( )
scanf( )	strcpy( )	stricmp( )	isalpha( )	islower( )

C – String functions

strlen - Finds out the length of a string
strlwr - It converts a string to lowercase
strupr - It converts a string to uppercase
strcat - It appends one string at the end of another
strncat - It appends first n characters of a string at the end of another.
strcpy - Use it for Copying a string into another
strncpy - It copies first n characters of one string into another
strcmp - It compares two strings
strncmp - It compares first n characters of two strings
strcmpi - It compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp - It compares two strings without regard to case (identical to strcmpi)
strnicmp - It compares first n characters of two strings, Its not case sensitive
strdup - Used for Duplicating a string
strchr - Finds out first occurrence of a given character in a string
strrchr - Finds out last occurrence of a given character in a string
strstr - Finds first occurrence of a given string in another string
strset - It sets all characters of string to a given character
strnset - It sets first n characters of a string to a given character
strrev - It Reverses a string

USER DEFINED FUNCTION

User-Defined Functions

If you have a special set of instructions that aren't in a built-in function, you can create a user-defined function. Here are the steps:

1. give your function a name that isn't already used in C (by built-in functions, types of variables, keywords, etc.)
2. create a **function header**, which contains three things:

FUNCTION PROTOTYPE

- A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.
- A function prototype gives information to the compiler that the function may later be used in the program.

FUNCTION PROTOTYPE SYNTAX

Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

Example

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

CALL BY VALUE

- This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

FUNCTION:

```
/* function definition to swap the values */
void swap(int x, int y) {
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
     return;
}
```

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
```

OUTPUT:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

CALL BY REFERENCE

- This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

FUNCTION:

```
/* function definition to swap the values */
void swap(int *x, int *y) {
int temp;
temp = *x;    /* save the value at address x */
*x = *y;      /* put y into x */
*y = temp;    /* put temp into y */
 return;
}
```

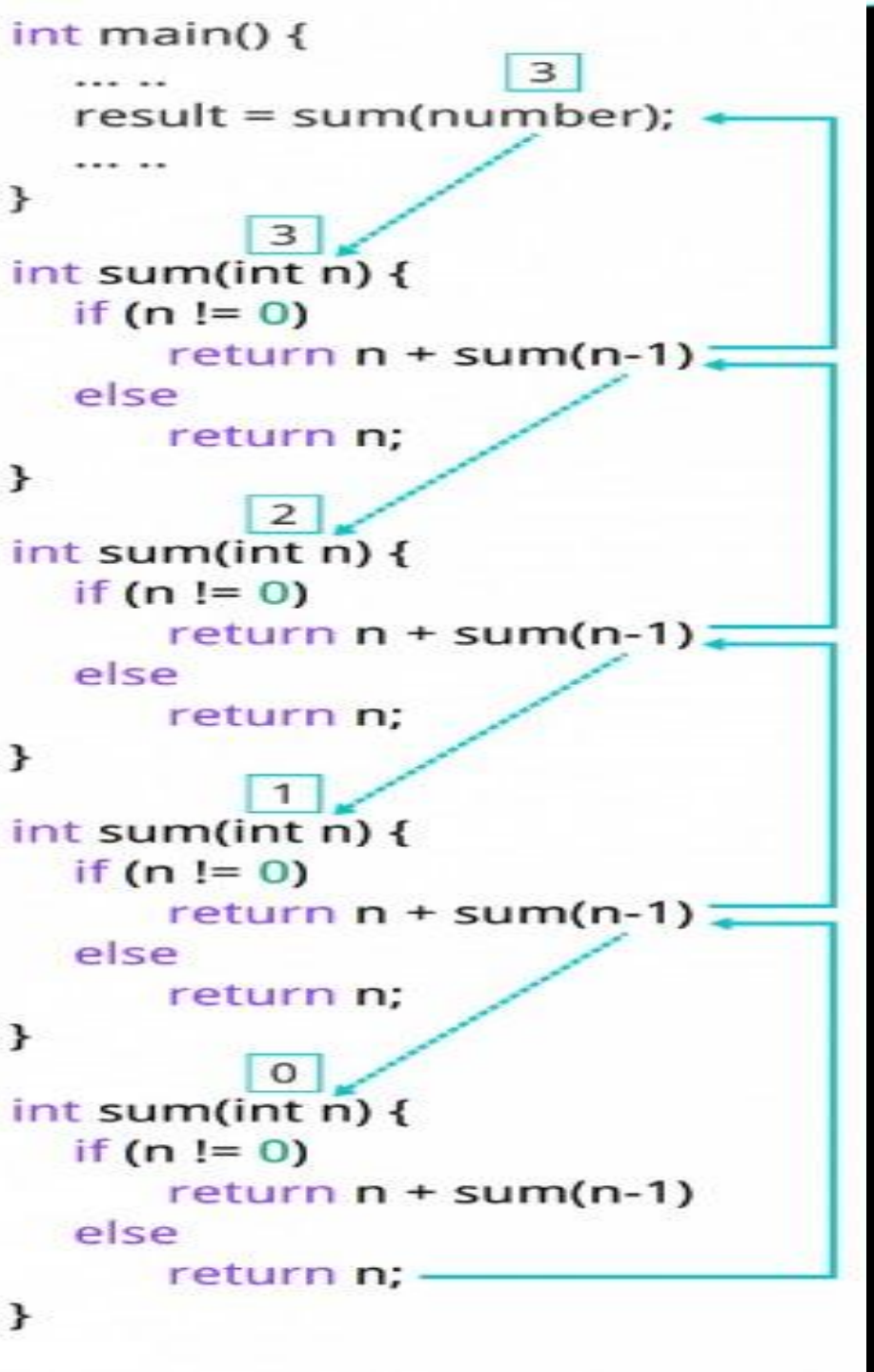
```
#include <stdio.h>

 /* function declaration */
void swap(int *x, int *y);
 int main () {

  /* local variable definition */
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable
a and
      * &b indicates pointer to b ie. address of variable
b.
   */

   swap(&a, &b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
```

OUTPUT:

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100

RECURSION

- Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.


int main() {
... ..
result = sum(number);
... ..
}
3
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
3
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
2
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
1
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
0
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}

Example: Sum of Natural Numbers Using Recursion

```c
#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;
}
```

Output

```
Enter a positive integer:3
sum = 6
```

STORAGE CLASS

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −
- auto
- register
- static
- extern

AUTO

The **auto** storage class is the default storage class for all local variables.

```
{
        Int mount;
        auto int mount;
}
```

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

EXTERN

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

REGISTER

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
        register int miles;
}
```

STATIC

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

File Edit Search Run Compile Debug Project Options Window Help

NONAME05.CPP

```
#include <stdio.h>
#include <conio.h>

int x=1; //external variable
static int count =0;

void main()
{
        int z; // auto variable  or local variable
        register int answer; // register variable
        clrscr();
        ++count;
        printf("Enter the value 1 : ");
        scanf("%d",&x);

        printf("Enter the value 2 : ");
        scanf("%d",&z);
        answer=x+z;
        printf("%d.Addition: %d",count,answer);
        getch();
}
```

19:32

THANK YOU…