Logging in From a Linux System or Localhost
4 days 36 min ago
Auto response
4 days 6 hours ago
Re: final issue.
4 days 22 hours ago
Re: Rocky, thank you for this
4 days 23 hours ago

Learning C/C++ Step-By-Step - Page 06

Submitted by ganesh35 (Contact Author) (Forums) on Wed, 2009-01-07 18:24. ::

06. Step-by-Step C/C++ --- C Programming - Functions

Functions

I. Introduction

Here is a program to print the address of a person twice, which is written in both methods using functions and without using functions. It will demonstrate the advantage of functions.

```
#include <stdio.h>
int main()
{
     printf("\nName of the Person");
     printf("\nStreet, Apartment//House No. ");
     printf("\nzip, City");
     printf("\nCountry");

     printf("\nName of the Person");
     printf("\nStreet, Apartment//House No. ");
```

```
#include <stdio.h>
void address()
{
     printf("\nName of the Person");
     printf("\nStreet, Apartment//House No. ");
     printf("\nzip, City");
     printf("\nCountry");
}

int main()
```

```
    printf("\nzip, City");
    printf("\nCountry");

    return 0;
}
```

```
{
    address();
    address();

    return 0;
}
```

II. Function Definition

A statement block, which has ability to accept values as arguments and return results to the calling program. So, A function is a self-contained block of statements that perform a specific task.

III. Types of functions

- Built-in functions/ Library Functions/ Pre-Defined functions
- User defined functions

IV. Library Functions

Library functions are designed by the manufacturer of the software, They were loaded in to the disk whenever the software is loaded.

The following functions are the example of the library functions.

1. Numeric Functions

Function	Syntax	Eg.	Result
Abs	Abs(n)	abs(-35)	35
ceil	ceil(n)	ceil(45.232)	46
floor	floor(n)	floor(45.232)	45
fmod	fmod(n,m)	fmod(5,2)	1
cos	cos(n)	cos(60)	0.5
sin	sin(n)	sin(60)	0.866
tan	tan(n)	tan(60)	1.732
sqrt	sqrt(n)	sqrt(25)	5
pow	pow(n,m)	pow(2,3)	8

2. String Functions

Functions	Syntax	Eg.
strlen	strlen(str)	strlen("Computer")
strcpy	strcpy(target,source)	strcpy(res,"Pass")
strcat	strcat(target,source)	strcat("mag","gic")
strcmp	strcmp(str1,str2)	strcmp("abc","Abc")
strrev	strrev(target,scr)	fstrrev(res,"LIRIL")

3. Character Test Functions

Function	Description
isalnum	is a letter or digit
isalpha	is a letter
isdigit	is a digit
iscntrl	is an ordinary control character
isascii	is a valid ASCII character
islower	is a lower character
isupper	is a upper character
isspace	is a space character
isxdigit	is hexa decimal character

There is a huge library of functions available, I have given you a tiny portion of it. For more Library Fuctions refer the Help Manual.

V. User-Defined Functions

The programs you have already seen perform divisions of labor. When you call **gets, puts**, or **strcmp**, you don't have to worry about how the innards of these functions work.

These and about 400 other functions are already defined and compiled for you in the Turbo C library. To use them, you need only include the appropriate header file in your program, "The run-time library," in the **library reference** to make sure you understand how to call the functions, and what value (if any) it returns.

But you'll need to write your own functions. To do so, you need to break your code into discrete sections (functions) that each perform a single, understandable task for your functions, you can call them throughout your program in the same way that you call **C** library functions.

Steps to implement a function

```
1. Declaration
2. Function Call
3. Definition
```

- Every function must be declared at the beginning of the program.
- Function definition contains the actual code of execution task.

- If a function is defined at the beginning of the program, there is no need of function declaration.

An example function to demonstrate the implementation

```
/* 32_egfun.c */
#include <stdio.h>

void address();      /* Declaration */

int main()
{
     address();      /* Function Call */
     address();      /* Function Call */

     return 0;
}

void address()      /* Definition */
{
     printf("\nName of the Person");
     printf("\nStreet, Apartment//House No. ");
     printf("\nzip, City");
     printf("\nCountry");
}
```

User defined functions can be divided in to 4 types based on how we are calling them.

1. **Simple Functions**
2. **Function with Arguments**
3. **Function with Returns**
4. **Function with Recursion**

1. Simple Functions

Performs a specific task only, no need of arguments as well as return values

Example of Simple Function

```
/* 33_line.c */
#include <stdio.h>
void line();      /* Declaration */
int main()
{
     line();      /* Function call */
     return 0;
}
void line()      /* Definition */
{
     int i;
     for(i =1;i<80; i++)
          putch('*');
}
```

2. Function with Arguments

A function, which accepts arguments, is known as function with arguments.

Eg.

```
/* 34_argu.c */
void line(char ch, int n)
int main()
{
     line("-", 50);
     line("*", 8);
     return 0;
}
void line(char ch, int n)
{
     int i;
     for( i = 1; i<=n; i++ )
          putch(ch);
}
```

3. Function with Return values

A function which can return values to the calling program is known as function with return values.

Eg.

```
/* 35_retu.c */
int abs(int n);
int main()
{
     int res;
     printf("%d", abs(-35))

     res = abs(-34);     /* Function Call*/

     printf("%d", res);
     return 0;
}
void abs(int n)
{
     if( n < 0 )
     n = n * -1;
     return n;
}
```

4. Function with Recursion

If a statement within the body of a function call the same function is called 'recursion' . Sometimes called 'circular definition', recursion is thus the process of defining something in terms of itself.

Examples of Recursive of functions

/* The following program demonstrates function call of itself */

```
int main( )
{
     printf("\nHello");
     main( ); /* A function, which can call it self */
     return 0;
}
```

Don't run this program, it is still an explanation thus program is not valid logically.

The same output can be reached using another function:

```
void disp( );

int main( )
{
     disp( );
     return 0;
}

void disp( )
{
     printf("\nHello");
     disp( );
}
```

The program must end at a certain point so the key of the recursion lies on soft interrupt, which can be defined using a conditional statement. Check the following example:

```
/* 36_recursion.c */
int i = 1;      /* Declaring a global variable */
void disp( );
int main( )
{
     disp( );
     return 0;
}

void disp( )
{
     printf("\nHello %d ", i);
     i ++;
     if( i < 10 ) /* if i value is less than 10 then call the function again */
          disp( );
}
```

Program to find the factorial of the given number:

```
/* 37_fact.c */
int factorial(int x);
void main
{
     int a, fact;
     printf("\nEnter any number "); scanf("%d", &a);
     fact = factorial(a);
     printf("\nFactorial is = %d", fact);
}
int factorial(int x)
{
     int f = 1, i;
     for( i = x; i>=1; i--)
     f = f * i;
     return f;
}
```

To find the factorial of a given number using recursion

```
/* 38_fact.c */
int rec_fact(int x);
int main( )
{
     int a, fact;
     printf("\nEnter any number "); canf("%d", &a);
     fact = rec_fact(a);
     printf("\nFactorial value is = %d", fact);
     return 0;
}
int f = 1;
int rec_fact(int x)
{
     if( x > 1)
          f = x * rec_fact(x-1);
     return f;
}
```

VI. Pointers and Functions

Parameter Passing by Reference
Call by value
Call by Reference

1. Parameter Passing by Reference

The pointer ca n be used in function declaration and this makes a complex function to be easily represented as well as accessed. The function definition makes use of pointers in it, in two ways

– Call by value
- Call by reference

The call by reference mechanism is fast compared to call by value mechanism because in call by reference, the address is passed and the manipulation with the addresses is faster than the ordinary variables. More ever, only one memory location is created for each of the actual parameter.

When a portion of the program, the actual arguments, calls a function and the values altered within the function will be returned to the calling portion of the program in the altered form. This is termed as **call by reference** or **call by address**. The use of pointer as a function argument in this mechanism enables the data objects to be altered globally ie within the function as well as within the calling portion of the program. When a pointer is passed to the function, the address of the argument is passed to the functions and the contents of this address are accessed globally. The changes made to the formal parameters (parameters used in function) affect the original value of the actual parameters (parameters used in function call in the calling program).

Eg.

```
/* 39_func.c */

void func_c( int *x );

int main()
{
     int i = 100;
     int *a;
     a = &i;
     printf("\nThe value is %d", i);
     func_c(a);
     printf("\nThe value is %d", i);
     return 0;
}

void func_c( int *x )
{
     (*x) ++;
     printf("\nThe value in function is %d ", *x);
}
```

In the above program, these are totally three 'printf' statements, tow in the main() function and one in the function subprogram. Due the effect of first printf statement the value of i is printed as 100. Later function call is made and inside function, the value is altered in and is 1001 due to increment. The altered value is again returned to main() and is printed as 1001.

Hence the output is:

The value is 100
The value in function is 101
The value is 101

More about Function Calls

Having had the first tryst with pointers let us now get back to what we had originally set out to learn – the two types of functions calls: call by value and call by reference. Arguments can generally be passed to function in one of the two ways:

a. Sending the values of the arguments
b. Sending the addresses of the arguments

2. Call by Value

In the first method the 'value' of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following programming illustrated the **Call by Value**.

```
/* 40_callbyvalue.c */
void swap( int x, int y )
int main( )
{
     int a = 10, b = 20;
     swap( a ,b );
     printf("\n a = %d, b = %d ", a, b);
     return 0;
}
void swap( int x, int y )
{
     int t;
     t = x;
     x = y;
     y = t;
     printf("\nx = %d, y = %d", x, y);
}
```


The output of the above program would be:

X = 20 y = 10
A = 10 b = 20

Note that value of **a** and **b** remain unchanged after exchanging the value of **x** and **y**.

3. Call by Reference

This time the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
* 41_callbyref.c */

void swap( int *x, int *y )

int main()
{
     int a = 10, b = 20;
     swap( &a, &b);
     printf("\na = %d, b = %d", a, b);
     return 0;
}

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

The output of the above program would be:

A = 20, b = 10

Note that this program manages to exchange the values of **a** and **b** using their addresses stored in **x** and **y**. Usually in C programming we make a call by value. I.e. in general you cannot alter the actual arguments. But if desired, it can always be achieved through a call by reference.
Using call by Reference intelligently **we can make a function, which can return more than one value at a time**, which is not possible ordinarily. This is shown in the program given below.

```
/* 42_callbyref.c */

void areaperi(int r, float *a, float *p)

int main()
{
     int radius;
     float area, perimeter;
     printf("\nEnter radius of a circle :"); scanf("%d", &radius);

     areaperi(radius, &area, &perimeter);

     printf("\nArea = %f ", area);
     printf("\nPerimeter = %f", perimeter);
     return 0;
}

void areaperi(int r, float *a, float *p)
{
     *a = 3.14 * r * r;
     *p = 2 * 3.14 * r;
}
```

And here is the output:

Enter radius of a circle 5
Are = 78.500000
Perimeter = 31.400000

Here, we are making a mixed call, in the sense, we are passing the value of **radius** but, address of **area** and **perimeter**. And since we are passing the addresses, any change that make in values stored at address contained in the variables **a** and **p**, would make the change effective in main. That is why when the control returns from the function **areaperi( )** we are able to output the values of **area** and **perimeter**.
Thus, we have been able to return two values from a called function, and hence, have overcome the limitation of the **return** statement, which can return only one value from a function at a time.

VII. Local Vs Global Variables

According to the Scope of Identifiers Variables are declared as of types.

```
/* 42_globalid.c */
int i=4000; /* Global variable declaration*/
int main()
{
     int a=10, b=20; /* Local Variable */
     int i=100; /* Local Variable */
     printf("%d %d", a, b);
     printf("\nLocal i : %d", i); /* Accessing Local variable */
     printf("\nGlobal i : %d ", ::i); /* Accessing Global variable */
     return 0;
}
```

Note: Scope Resolution (::) Operator can be available in C++ only.

VIII. Storage Class Specifiers

Until this point of view we are already familiar with the declaration of variables. To fully define a variable one needs to mention not only its 'type' but also its '**Storage Class**'.
According to this section variables are not only have a 'data type', they also have a 'Storage Class'.

Storage Classes are of 4 Types

1. Automatic Storage Class
2. Register Storage Class
3. Static Storage Class
4. External Storage Class

1. Automatic Storage Class

Keyword	auto
Storage	Memory
Default Value	Null
Scope	Local to the block in which the variable defined
Life	Until the execution of its block

Eg:

```
/* 43_auto.c */
#include <stdio.h>
int main()
{
     auto int i, j;
     printf("%d %d", i, u);
     return 0;
}
```

2. Register Storage Class

Keyword	register
Storage	CPU Registers
Default Value	Null
Scope	Local to the block in which the variable defined
Life	Until the execution of its block

Eg:

```
/* 44_register.c */
#include <stdio.h>
int main( )
{
     register int i, j;
     for(i=1;i<=10;i++)
          printf("\n%d", i);
     return 0;
}
```

3. Static Storage Class

Keyword	static
Storage	Memory
Default Value	Zero
Scope	Local to the block in which the variable defined
Life	Value of the variable persists between different function calls

Eg:

```
/* 45_static.c */
#include <stdio.h>
void add();
int main()
{
     add();
     add();
     add();
     return 0;
}
void add()
{
     static int i = 1;
     printf("%d\n",i++);
}
```

4. External Storage Class

Keyword	extern
Storage	Memory
Default Value	Zero
Scope	Global
Life	As long as the program's execution doesn't come to an end

Eg:

```
/* 46_extern.c */
#include <stdio.h>
int i;
void add(); /* Extern variable */
int main( )
{
     extern j=10; /* Extern variable */
     for(i=1;i<=10;i++)
          add();
     return 0;
}
void add( )
{
     j++;
     printf("%d %d\n", i, j);
}
```

previous up next
Learning C/C++ Step-By-Step - Page 05 Learning C/C++ Step-By-Step - Page 07

add comment | view as pdf | print: this | all page(s)

Related Tutorials

- Beginner's Guide To c++
- An Explanation of Pointers (C++)

Please do not use the comment function to ask for help! If you need help, please use our forum.
Comments will be published after administrator approval.