Learnem.com

Programming Courses

Learn'em

Programming in C in 7 days!

By: Siamak Sarmady

LEARN'EM PROGRAMMING COURSES

Programming in C

Ver. 2.08.01

By: Siamak Sarmady

- **"Programming in C in 7 days!"** includes only the first 7 lessons of the more complete e-book **"Quickly Learn Programming in C"**. You can obtain the more complete e-book on Learnem.com website.
- Support for the free e-book **"Programming in C in 7 days!"** is provided on Learnem.com discussion boards.

Table of Contents

Lesson 1

Quick Start with C

C programming language is perhaps the most popular programming language. C was created in 1972 by Dennis Ritchie at the Bell Labs in USA as a part of UNIX operating system. C was also used to develop some parts of this operating system. From that time C programming language has been the de facto programming language when fast programs are needed or the software needs to interact with the hardware in some way. Most of the operating systems like Linux, Windows™, and Mac™ are either developed in C language or use this language for most parts of the operating system and the tools coming with it.

This course is a quick course on C Programming language. In our first lesson we will first write our first C program. We will then learn about printing to screen, variables and functions. We assume that you are familiar with at least one of the popular operating systems.

For this course you can use the following compilers or Programming Environments.

- Gcc and cc in Unix and Linux operating systems
- Borland C or Turbo C in DOS operating system or in Command line environment of windows operating system
- "Bloodshed Dev-Cpp" integrated development environment (IDE) gives you a complete and compact programming environment. It comes with "MinGW" and "GCC" C Compilers and you should not need anything else for this course.

We use "Bloodshed Dev-Cpp" for this course and we suggest you also use it. "Bloodshed Dev-Cpp" is free and it can be downloaded from the website http://www.bloodshed.net (currently under the URL http://www.bloodshed.net/dev/devcpp.html).

Your first C program

Let's write our first C program.

Example 1-1: example1-1.c

```
#include <stdio.h>
main()
{
 printf("Hello World!\n");
```

```
 system("pause"); //this line is only needed under windows
}
```

First step for running this program is to make a text file containing above code. Be sure that the file is a pure text file. You must save the text file with .c extension.

Then you must compile the source code. The result of compile step is an executable file that you can run it.

If you are running your example on Unix/Linux type operating systems you can remove the line which is commented to be necessary just under Windows.

Compiling and Running on Unix/Linux

To compile program under Unix operating system use the command:

```
$ cc test.c
```

and under linux type

```
$ gcc test.c
```

The resulting executable file is a.out file. To run this executable you must type:

```
$./a.out
```

Program output must appear on your screen.

```
Hello World!
```

Compiling and Running under Windows with Dev-CPP

To develop, compile and run the program in Bloodshed environment follow below steps. If you have problem working with your compiler you may ask your problem in our support forums.

1- Run bloodshed and select "New -> Project" from File menu. In the appeared window enter a name for your project (Figure 1.1). Also select "Console Application", "C Project" and "Make default Language" as the settings of your application.

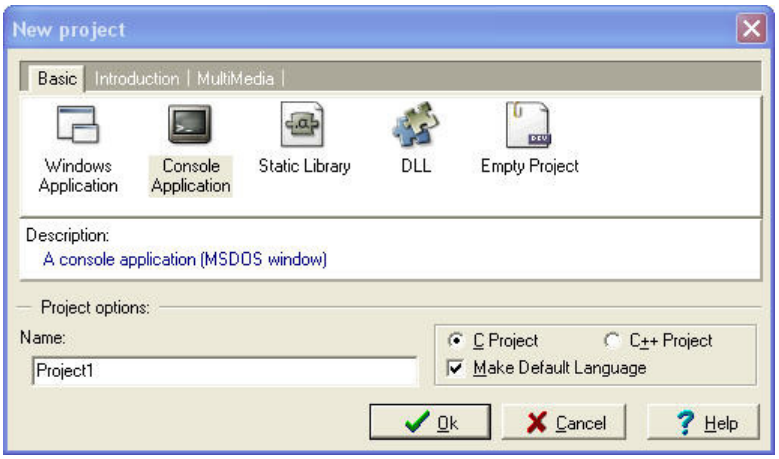

FIGURE 1.1: Creating a new project in Bloodshed Dev-CPP.

2- A window will open and ask for a place to save the project. We suggest that you create a separate directory for each project to avoid their files being mixed with each other (or worse, overwrite each other). A project is a set of related C programming language files. In this case we just have a single C file in our project but big projects may have even hundreds of C files.

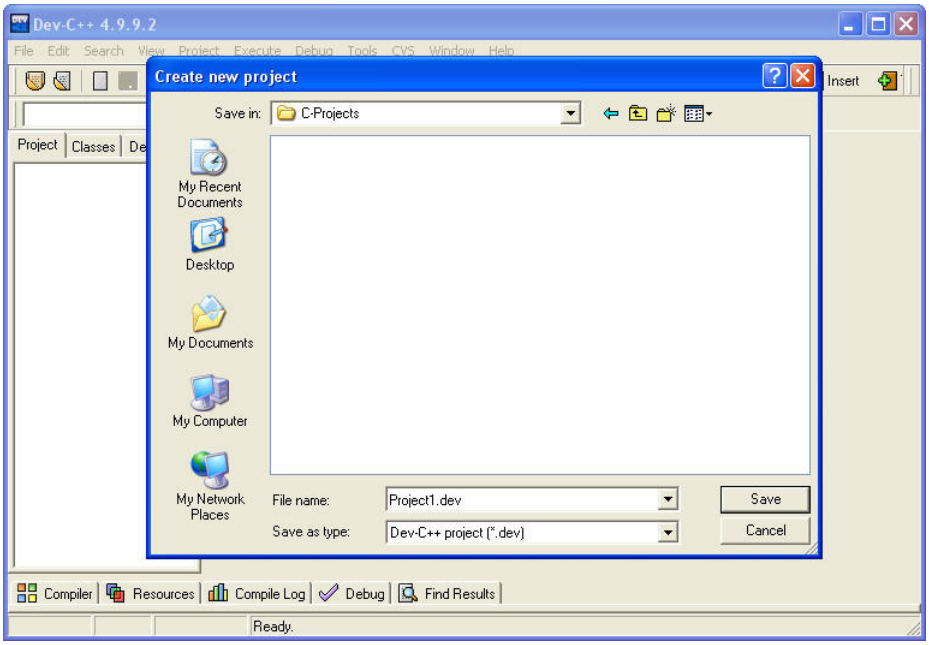

FIGURE 1.2: Saving a project in Bloodshed Dev-CPP.

3- Dev-CPP creates the project and generates a sample C language file for you. Dev-CPP creates this first sample C program with the file name "main.c". You should change the source code to the source code of our Example 1-1 (Figure 1.3 and 1.4). After changing the code, press "Save File" button. This will give you the opportunity to change the default "main.c" file name to whatever file name you prefer. You might want to save the file with the related example number (**example1-1.c** in this case) or you can leave it as it is.

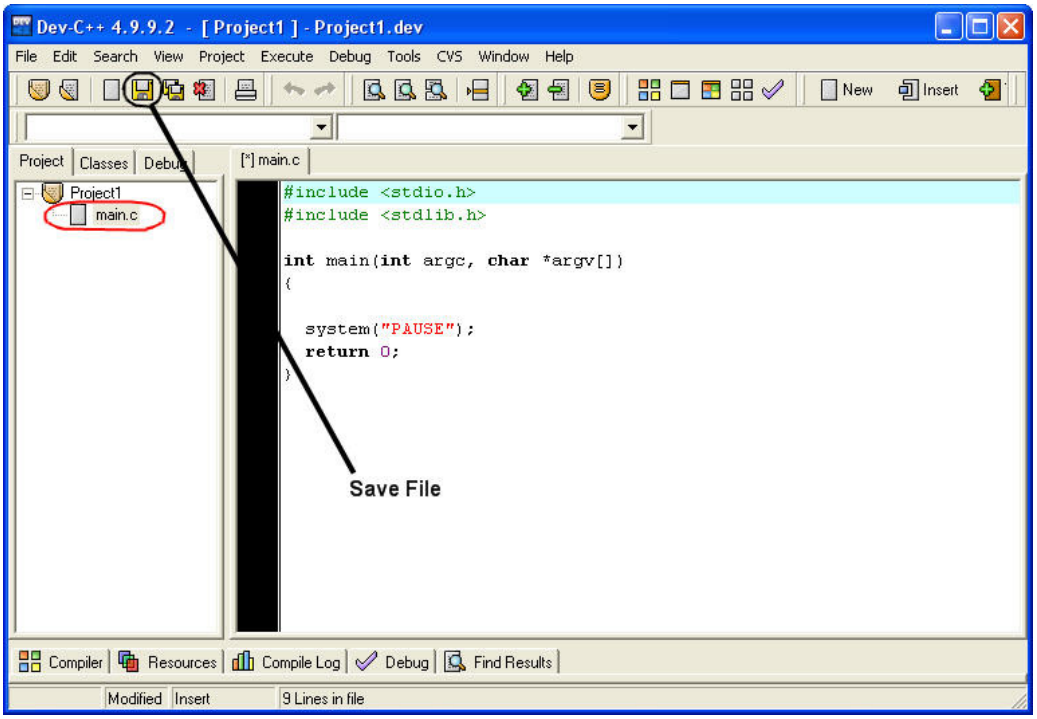

FIGURE 1.3: New project is created in Bloodshed Dev-CPP and a simple code is generated.

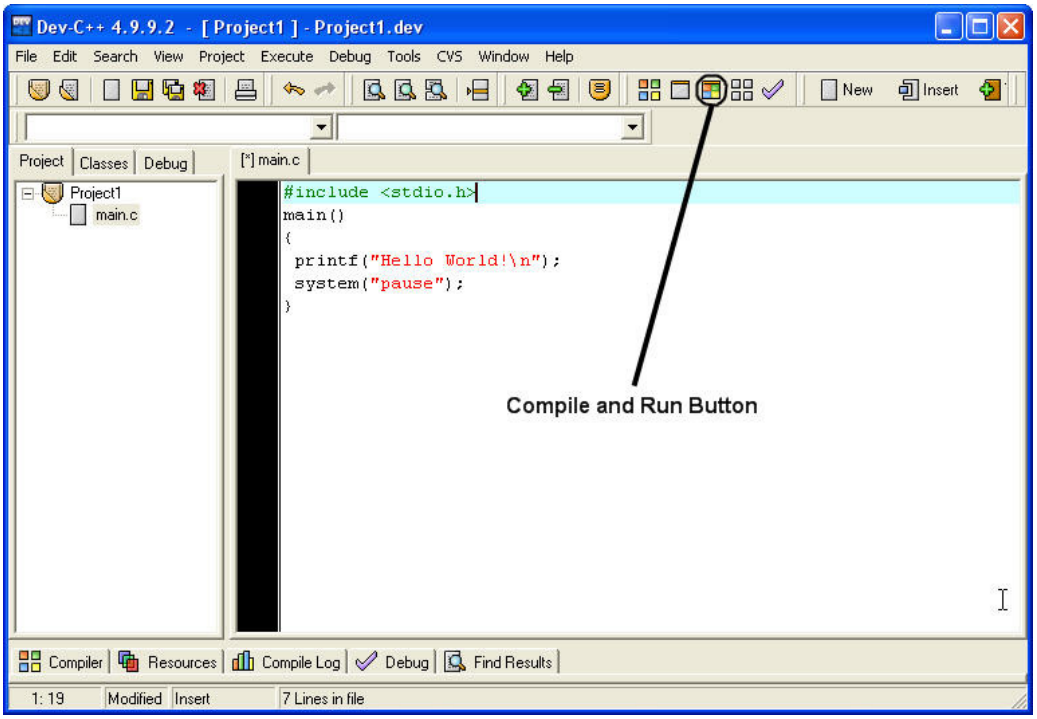

FIGURE 1.4: Change the code to our Example 1-1.

4- Click on "Compile and Run" button (or press F9 key). This will compile the code to machine executable format and run the program (Figure 1.5). To close the output window press any key in output window.

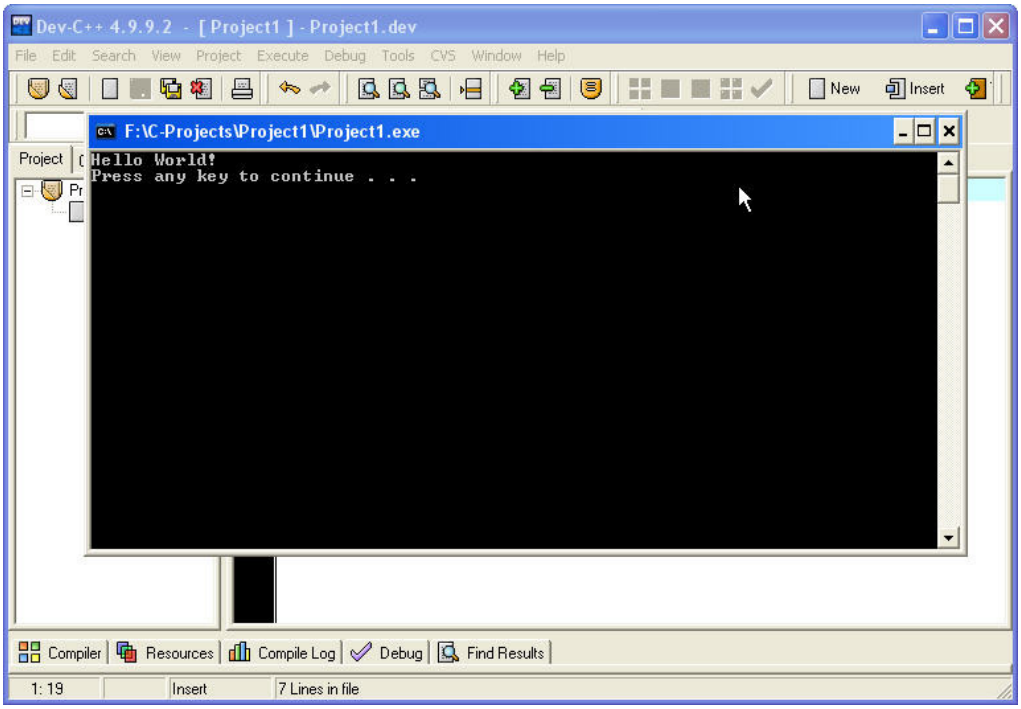

FIGURE 1.5: Output window shows the result of the program.

If you look into the directory which you saved your project, you will find 5 different files:

- **main.c** (main C program file)
- **main.o** (intermediate compile file called "object file")
- **Makefile.win** (Make file is used by Dev-CPP compile settings of your project)
- **Project1.dev** (Project File contains Dev-CPP settings of your project)
- **Project1.exe** (Executable file which can be run independent from Dev-CPP or C Compiler)

The final product of your C program is the windows executable (.exe) file. You will normally distribute the executables of your software to users and keep the source code (*.c) for your own.

Details of Test program

- **#include <stdio.h>**
  Tells C compiler to include the file "stdio.h" in this point of your C program before starting compile step. This "include file" contains several definitions, declarations etc.

- **main()**
  C program consist of one or more functions. Functions are building blocks of C programs. main() function is different from other functions by that it is the start point of program

execution. Our program contains only function while complicated programs may contain thousands.

- **{**
  Opening brace marks the start of a block. Closing brace will mark its end. This one marks main () function start

- **printf("Hello world!");**
  This line of code prints the statement between quotation marks on your output screen. \n tells program to start a new line in output screen.

- Each command line in C ends with ";" character. Control statements are exceptions. You will soon be able to determine when you must use ; to end a line of code.

- **system("pause");**
  The output window will close in Windows™, immediately after program execution has been finished. In this way you will not be able to see results of the execution (as it happens very fast). We have put this command to pause the window and wait for a keystroke before closing the window. **You can remove this line from our examples if you do not use Windows operating system.** This command actually sends the "pause" command to windows operating system and windows runs the its "pause" command at this point. We will learn more about this command in later lessons.

- **}**
  closes main() function.

This program contains only one function while complicated programs may contain several functions.

Data Types and Variables

C uses several data types of data. These include characters, integer numbers and float numbers. In C language you must declare a variable before you can use it. By declaring a variable to be an integer or a character for example will let computer to allocate memory space for storing and interpreting data properly.

Naming a variable

It is better that you use meaningful names for your variables even if this causes them to become long names. Also take this in mind that C is case sensitive. A variable named "COUNTER" is different from a variable named "counter".

Functions and commands are all case sensitive in C Programming language. You can use letters, digits and underscore _ character to make your variable names. Variable names can be up to 31 characters in ANSI C language.

The declaration of variables must take place just after the opening brace of a block. For example we can declare variables for main() function as below code:

```
main()
{
int count;
float sum,area;
.
.
.
}
```

First character in a variable name must be a letter or an underscore character. It cannot be a C programming language-reserved word (i.e. Commands and pre defined function names etc). An example for using variables comes below:

Example 1-2: example1-2.c

```
#include<stdio.h>
main()
{
 int sum;
 sum=12;
 sum=sum+5;
 printf("Sum is %d",sum);
 system("pause");
}
```

General form for declaring a variable is:

Type name;

The line sum=sum+5; means: Increase value of sum by 5. We can also write this as sum+=5; in C programming language. printf function will print the following:

```
Sum is 17
```

In fact %d is the placeholder for integer variable value that its name comes after double quotes.

Common data types are:

```
int      integer
long     long integer
float    float number
double   long float
char     character
```

Other placeholders are:

```
%d     decimal integer
%ld    decimal long integer
%s     string or character array
%f     float number
%e     double (long float)
```

printf () function used in this example contains two sections. First section is a string enclosed in double quotes. It is called a format string. It determines output format for printf function. Second section is "variable list" section.

We include placeholders for each variable listed in variable list to determine its output place in final output text of printf function.

Control characters

As you saw in previous examples \n control character makes a new line in output. Other control characters are:

```
\n New line
\t tab
\r carriage return
\f form feed
\v vertical tab
```

Multiple functions

Look at this example:

Example 1-3: example1-3.c

```
#include<stdio.h>
main()
{
  printf("I am going inside test function now\n");
  test();
  printf("\nNow I am back from test function\n");
  system("pause");
}

test()
{
  int a,b;
  a=1;
  b=a+100;
  printf("a is %d and b is %d",a,b);
}
```

In this example we have written an additional function. We have called this function from inside main function. When we call the function, program continues inside test () function and after it reached end of it, control returns to the point just after test() function call in main(). You see declaring a function and calling it, is an easy task. Just pay attention that we used ";" when we called the function but not when we were declaring it.

We finish this lesson here. Now try to do lesson exercises and know the point that you will not learn anything if you do not do programming exercises.

Exercises

- Write, compile and test your programs under "Bloodshed Dev-CPP" under windows or Gcc under Linux/UNIX.
- Paid students need to submit their exercises inside e-learning virtual campus. Corrected exercises will be available inside virtual campus.
- If you have obtained the e-Book only, you can discuss your homework questions in Learnem.com support forums (in registered e-book users section).

---

1. **What is the exact output result of this code?**

```
#include <stdio.h>
main()
{
printf("Hi\nthere\nWhat is the output\n?");
}
```

2. **Write a program that declares two floating numbers. Initialize them with float values. Then print their sum and multiplication in two separate lines.**

3. **Write the output result of "multiple function example" in this lesson (run the program and see it).**

4. **Why these variable names are not valid?**

```
test$var
my counter
9count
Float
```

Lesson 2

Getting input, Arrays, Character Strings and Preprocessors

In previous lesson you learned about variables and printing output results on computer console. This lesson discusses more about variables and adds important concepts on array of variables, strings of characters and preprocessor commands. We will also learn more on getting input values from console and printing output results after performing required calculations and processes. After this lesson you should be able to develop programs which get input data from user, do simple calculations and print the results on the output screen.

Receiving input values from keyboard

Let's have an example that receives data values from keyboard.

Example 2-1: example2-1.c

```
#include<stdio.h>
main()
{
 int a,b,c;
 printf("Enter value for a :");
 scanf("%d",&a);
 printf("Enter value for b :");
 scanf("%d",&b);
 c=a+b;
 printf("a+b=%d\n",c);
 system("pause");
}
```

Output results:

```
Enter value for a : 10
Enter value for b : 20
a+b=30
```

As scanf itself enters a new line character after receiving input, we will not need to insert another new line before asking for next value in our second and third printf functions.

General form of scanf function is :

```
  scanf("Format string",&variable,&variable,...);
```

Format string contains placeholders for variables that we intend to receive from keyboard. A '&' sign comes before each variable name that comes in variable listing. Character strings are exceptions from this rule. They will not come with this sign before them. We will study about character strings in this lesson.

Important: You are not allowed to insert any additional characters in format string other than placeholders and some special characters. Entering even a space or other undesired character will cause your program to work incorrectly and the results will be unexpected. So make sure you just insert placeholder characters in scanf format string.

The following example receives multiple variables from keyboard.

```
float a;
int   n;
scanf("%d%f",&n,&a);
```

Pay attention that scanf function has no error checking capabilities built in it. Programmer is responsible for validating input data (type, range etc.) and preventing errors.

Variable Arrays

Arrays are structures that hold multiple variables of the same data type. An array from integer type holds integer values.

```
int scores[10];
```

The array "scores" contains an array of 10 integer values. We can use each member of array by specifying its index value. Members of above array are scores[0],...,scores[9] and we can work with these variables like other variables:

```
scores[0]=124;
scores[8]=1190;
```

Example 2-2: example2-2.c

Receive 3 scores of a student in an array and finally calculate his average.

```
#include<stdio.h>
main()
{
 int scores[3],sum;
 float avg;
```

```
 printf("Enter Score 1 : ");
 scanf("%d",&scores[0]);
 printf("Enter Score 2 : ");
 scanf("%d",&scores[1]);
 printf("Enter Score 3 : ");
 scanf("%d",&scores[2]);
 sum=scores[0]+scores[1]+scores[2];
 avg=sum/3;

 printf("Sum is = %d\nAverage = %f\n",sum,avg);
 system("pause");
}
```

Output results:

```
Enter Score 1 : 12
Enter Score 2 : 14
Enter Score 3 : 15
Sum is = 41
Average = 13.000000
```

Character Strings

In C language we hold names, phrases etc in character strings. Character strings are arrays of characters. Each member of array contains one of characters in the string. Look at this example:

Example 2-3: example2-3.c

```
#include<stdio.h>
main()
{
 char name[20];

 printf("Enter your name : ");
 scanf("%s",name);

 printf("Hello, %s , how are you ?\n",name);
 system("pause");
}
```

Output Results:

```
Enter your name : Brian
Hello, Brian, how are you ?
```

If user enters "Brian" then the first member of array will contain 'B' , second cell will contain 'r' and so on. C determines end of a string by a zero value character. We call this character as "NULL" character and show it with '\0' character. (It's only one character and its value is 0, however we show it with two characters to remember it is a character type, not an integer)

Equally we can make that string by assigning character values to each member.

```
  name[0]='B';
  name[1]='r';
  name[2]='i';
  name[3]='a';
  name[4]='n';
  name[5]=0;  //or name[5]='\0';
```

As we saw in above example placeholder for string variables is %s. Also **we will not use a '&' sign for receiving string values**. For now be sure to remember this fact and we will understand the reason in future lessons.

Preprocessor

Preprocessor statements are those lines starting with '#' sign. An example is #include<stdio.h> statement that we used to include stdio.h header file into our programs.

Preprocessor statements are processed by a program called preprocessor before compilation step takes place. After preprocessor has finished its job, compiler starts its work.

#define preprocessor command

#define is used to define constants and aliases. Look at this example:

Example 2-4: example2-4.c

```
#include<stdio.h>

#define PI 3.14
#define ERROR_1 "File not found."
#define QUOTE "Hello World!"

main()
{
 printf("Area of circle = %f * diameter", PI );
 printf("\nError : %s",ERROR_1);
 printf("\nQuote : %s\n",QUOTE);
 system("pause");
}
```

Output results:

```
Area of circle = 3.140000 * diameter
Error : File not found.
Quote : Hello World!
```

Preprocessor step is performed before compilation and it will change the actual source code to below code. Compiler will see the program as below one:

```
#include<stdio.h>
main()
{
 printf("Area of circle = %f * diameter", 3.14 );
 printf("\error : %s","File not found.");
 printf("\nQuote : %s","Hello World!\n");
 system("pause");
}
```

In brief #define allows us to define symbolic constants. We usually use uppercase names for #define variables. Pay attention that we do not use ';' after preprocessor statements.

Variable limitations

Variable limit for holding values is related to the amount of memory space it uses in system memory. In different operating systems and compilers different amount of memory is allocated for specific variable types. For example int type uses 2 bytes in DOS but 4 bytes in windows environment. Limitations of variable types are mentioned in your compiler documentation. If oue program is sensitive to the size of a variable (we will see examples in next lessons), we should not assume a fixed size for them. We should instead use sizeof() function to determine size of a variable or variable type (and we should do t).

Example 2-5: example2-5.c

```
#include<stdio.h>
main()
{
 int i;
 float f;

 printf("Integer type uses %d bytes of memory.\n", sizeof(i));
 printf("float type uses %d bytes of memory.\n", sizeof(float));
 system("pause");
}
```

You see we can use both a variable and a variable type as a parameter to sizeof() function. Below table shows variable limitations of Turbo C and Microsoft C in DOS operating system as an example.

```
         Bytes used         Range
char         1                256
int          2                65536
short        2                65536
long         2                4 billion
float        4                6 digits * 10e38
double       8                10 digits * 10e308
```

We have two kinds of variables from each of the above types in C programming language: signed and unsigned. Signed variables can have negative values too while unsigned values only support positive numbers.

If a signed variable is used, high boundary will be divided by two. This is because C will divide the available range to negative and positive numbers. For example signed int range is (-32768,+32767).

You can declare a variable as "signed" or "unsigned" by adding "signed" or "unsigned" keywords before type name.

Example:

```
signed int a;
unsigned int b;

a=32700;
b=65000;
```

We are allowed to assign values larger than 32767 to variable "b" but not to variable "a". C programming language may not complain if we do so but program will not work as expected. Alternatively we can assign negative numbers to "a" but not to "b".

Default kind for all types is signed so we can omit signed keyword if we want a variable to be signed.

Exercises

- Write, compile and test your programs under "Bloodshed Dev-CPP" under windows or Gcc under Linux/UNIX.
- Paid students need to submit their exercises inside e-learning virtual campus. Corrected exercises will be available inside virtual campus.
- If you have obtained the e-Book only, you can discuss your homework questions in Learnem.com support forums (in registered e-book users section).

---

1. **Write a program that asks for work hours, wage per hour and tax rate and then prints payable money for a person.**
2. **Using arrays write a program that receives tax rate, work hours and wage per hour for two persons, saves work hours of those persons in an array and then calculates and prints payable money for each of them.**
3. **Write a program that asks for your name and outputs 3 first characters of it, each in a separate line. Use %c as placeholder of a single character in printf format string.**

Lesson 3

Operators, Loops, Type Conversion

In previous lesson you learned about arrays, strings and preprocessors. In this lesson we will learn about using mathematical and logical operators. Mathematical operators are equivalent to the operators being used in mathematics (+-*/…). There are differences which we will mention in this lesson however. Loops are also important part of programming languages. In this lesson we will also learn how to convert variables from a specific type into variables with other types.

Operators

There are many kinds of operators in each programming language. We mention some of the operators being used in C language here:

```
()  Parentheses
+  Add
-  Subtract
*  Multiply
/  Divide
```

There are also some other operators which work differently:

```
%  Modulus
++  Increase by one
--  Decrease by one
=  Assignment
```

sizeof( ) return value is the size of a variable or type inside parentheses in bytes. It is actually the size that variable takes in system memory.

Examples:

```
c=4%3;     c will be equal to 1 after execution of this command.
i=3;
i=i*3;     i will be equal to 9

f=5/2;     if f is integer then it will be equal to 2. If it
           is a float type variable its value will be 2.5

j++;       Increases the value of j by one.
```

```
j--;          Decreases value of j by one

sizeof(int)   returned value is 2 in dos and 4 in windows

int a=10;
c=sizeof(a);  c will be 2 in dos and 4 in windows as the size of integer
              is different in different Os.
```

Loops

Sometimes we want some part of our code to be executed more than once. We can either repeat the code in our program or use loops instead. It is obvious that if for example we need to execute some part of code for a hundred times it is not practical to repeat the code. Alternatively we can use our repeating code inside a loop.

```
while(not a hundred times)
 {
  code
 }
```

There are a few kinds of loop commands in C programming language. We will see these commands in next sections.

While loop

while loop is constructed of a condition and a single command or a block of commands that must run in a loop. As we have told earlier a block of commands is a series of commands enclosed in two opening and closing braces.

```
while( condition )
    command;

while( condition )
 {
   block of commands
 }
```

Loop condition is a boolean expression. A boolean expression is a logical statement which is either correct or incorrect. It has a value of 1 if the logical statement is valid and its value is 0 if it is not. For example the Boolean statement (3>4) is invalid and therefore has a value of 0. While the statement (10==10) is a valid logical statement and therefore its value is 1.

Example 3-1: example3-1.c

```
#include<stdio.h>
main()
{
int i=0;
```

```
while( i<100 )
  {
  printf("\ni=%d",i);
  i=i+1;
  }

 system("pause");
}
```

In above example i=i+1 means: add 1 to i and then assign it to i or simply increase its value. As we saw earlier, there is a special operator in C programming language that does the same thing. We can use the expression i++ instead of i=i+1.

We will learn more about logical operators in next lessons.

Type Conversion

From time to time you will need to convert type of a value or variable to assign it to a variable from another type. This type of conversions may be useful in different situations, for example when you want to convert type of a variable to become compatible with a function with different type of arguments.

Some rules are implemented in C programming language for this purpose.

- Automatic type conversion takes place in some cases. Char is automatically converted to int. Unsigned int will be automatically converted to int.
- If there are two different types in an expression then both will convert to better type.
- In an assignment statement, final result of calculation will be converted to the type of the variable which will hold the result of the calculation (ex. the variable "count" in the assignment count=i+1;)

For example if you add two values from int and float type and assign it to a double type variable, result will be double.

Using loops in an example

Write a program to accept scores of a person and calculate sum of them and their average and print them.

Example 3-2 : example3-2.c

```
#include<stdio.h>
main()
{
int count=0;
```

```
float num=0,sum=0,avg=0;

printf("Enter score (-1 to stop): ");
scanf("%f",&num);
while(num>=0)
  {
  sum=sum+num;
  count++;
  printf("Enter score (-1 to stop): ");
  scanf("%f",&num);
  }

avg=sum/count;
printf("\nAverage=%f",avg);
printf("\nSum=%f\n",sum);

system("pause");
}
```

In this example we get first number and then enter the loop. We will stay inside loop until user enters a value smaller than 0. If user enters a value lower than 0 we will interpret it as STOP receiving scores. Here are the output results of a sample run:

```
Enter score (-1 to stop): 12
Enter score (-1 to stop): 14
Enter score (-1 to stop): -1

Average=13.000000
Sum=26.000000
```

When user enters -1 as the value of num, logical expression inside loop condition becomes false (invalid) as num>=0 is not a valid statement.

Just remember that "while loop" will continue running until the logical condition inside its parentheses becomes false (and in that case it terminates).

For loop

As we told earlier, there are many kinds of loops in C programming language. We will learn about for loop in this section.

"For loop" is something similar to while loop but it is more complex. "For loop" is constructed from a control statement that determines how many times the loop will run and a command section. Command section is either a single command or a block of commands.

```
for( control statement )
    command;
```

```
for( control statement )
    {
    block of commands
    }
```

Control statement itself has three parts:

```
  for ( initialization;    test condition;   run every time command )
```

Initialization part is performed only once at "for loop" start. We can initialize a loop variable here. Test condition is the most important part of the loop. Loop will continue to run if this condition is valid (True). If the condition becomes invalid (false) then the loop will terminate.

'Run every time command' section will be performed in every loop cycle. We use this part to reach the final condition for terminating the loop. For example we can increase or decrease loop variable's value in a way that after specified number of cycles the loop condition becomes invalid and "for loop" can terminate.

At this step we rewrite example 3-1 with for loop. Just pay attention that we no more need I=I+1 for increasing loop variable. It is now included inside "for loop" condition phrase (i++).

Example 3-3: example3-3.c

```
#include<stdio.h>
main()
{
int i=0;

for(i=0;i<100;i++ )
  printf("\ni=%d",i);

 system("pause");
}
```

Example 3-4: example3-4.c

Write a program that gets temperatures of week days and calculate average temperature for that week.

```
#include<stdio.h>
main()
{
int count=0;
float num=0,sum=0,avg=0;

for(count=0;count<7;count ++)
  {
  printf("Enter temperature : ");
```

```
  scanf("%f",&num);
  sum=sum+num;
  }

 avg=sum/7;
 printf("\nAverage=%f\n",avg);

 system("pause");
}
```

In next lesson we will learn more examples about using loops and also other control structures.

Exercises

- Write, compile and test your programs under "Bloodshed Dev-CPP" under windows or Gcc under Linux/UNIX.
- Paid students need to submit their exercises inside e-learning virtual campus. Corrected exercises will be available inside virtual campus.
- If you have obtained the e-Book only, you can discuss your homework questions in Learnem.com support forums (in registered e-book users section).

---

1. **Using while loop; write a program that prints alphabets 'a' through 'z' in separate lines. Use %c in your format string. You can increment a character type variable with ++ operand.**

2. **Write a program that accepts time in "seconds" format and prints it in minutes and seconds. For example if you enter 89 it must print : 1:29 (you can use % modulus operator to compute the remaining of a division)**

3. **Using for loop, write a program that accepts net sales of a shop for each day. Then calculate sales for a month then deduct %5 tax and finally calculate "Average daily sales" after tax deduction. Print results on the screen.**

Lesson 4

Conditional Command Execution

As we saw in previous lesson we can use "for" statement to create a loop for executing a command or a block of commands. In this lesson we will learn more about "for" loops. We will also learn how to run a command conditionally.

More about "for" loops

There are three parts inside condition phrase. In initialization statement you can initialize variables including loop counter or any other variable. In condition statement you can use any kind of logical statement. This logical statement will function as the condition of loop execution. As we mentioned earlier if this condition becomes invalid (false) loop execution will terminate.

```
for( initialization;  test condition;  run every time command)
   command;
```

Last section in the "for loop" parentheses is a C language statement which is executed in each loop cycle. In previous examples we used a statement like i++ and count++. These will increase the value of a variable each time loop is executed. Increase in this variable can change the loop condition logical value to invalid (false), if the loop condition is also based on this variable. Below example will print a multiplication chart (from 1*1 to 9*9). Run the program and see the results.

Example 4-1: example4-1.c

```
#include<stdio.h>
main()
{
int i,j;

for(i=1;i<10;i++)
 {
 for(j=1;j<10;j++)
   printf("%3d",i*j);
 printf("\n");
 }

system("pause");
```

```
}
```

"if" conditional statements

Sometimes you need a command or a block of commands to be executed when a condition exists or when a condition does not exist. The condition is a logical statement similar to the condition used in while loops. A logical statement is either valid (has a true value) or it is invalid (false).

```
if(condition)
    command;

if(condition)
  {
  block of commands;
  }
```

"If statement" is a branching statement because it provides a way to select a path from several execution paths in a program. If condition is true the command or block of commands will be executed.

Example 4-2: example4-2.c

What does this program do?

```
#include<stdio.h>
main()
{
int n;
printf("Enter a number: ");
scanf("%d",&n);
if(n>=0)
  printf("Number is positive !\n");
if(n<0)
  printf("Number is negative !\n");

system("pause");
}
```

Now let's see a more complex example.

Example 4-3: example4-3.c

Write a program to solve a second-degree equation: $ax^2+bx+c=0$.

```
#include<stdio.h>
#include<math.h>
main()
{
float delta,a,b,c,x1,x2;
```

```
printf("Enter a : ");
scanf("%f",&a);
printf("Enter b : ");
scanf("%f",&b);
printf("Enter c : ");
scanf("%f",&c);
delta=b*b-(4*a*c);
if(delta<0)
  {
  printf("Equation has no answer !\n");
  system("pause");
  exit(0);
  }
if(delta==0)
  {
  x1=-b/(2*a);
  printf("Equation has two equal answers !\n");
  printf("x1=x2=%f\n",x1);
  system("pause");
  exit(0);
  }

x1=(-b+sqrt(delta))/(2*a);
x2=(-b-sqrt(delta))/(2*a);
printf("\nX1=%f",x1);
printf("\nX2=%f\n",x2);

system("pause");
}
```

Our program gets a,b and c, computes delta (the value under the square root which is delta= $b^2 - 4ac$) and 2 possible answers if they exist based on below equations:

$$x1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad \text{And} \quad x2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

We have used the sqrt() function from the "math" library and to be able to use the library the only thing we have done, is to add the "math.h" file to the included header file list.

More complex "if" statements

Simple form of "if statement" gives you the choice of executing or skipping a command or block of commands. If in a program it is needed to execute a specific command when a condition is true and execute another command when it is false, with your current knowledge you may use two simple "if" statements after each other.

```
  If(condition)
    Command;
  If(!condition)
```

```
    command
```

! Sign reverses the logical value of a Boolean expression. If it is true the result will become false with '!' sign and vice versa.

You can use an alternative form of "if statement" to avoid using two statements. "if statement" has more complete forms. Below you see one of the other forms of "if" statement.

```
  If(condition)
    Command;
  else
    command
```

In this form, an additional "else" section has been added. When condition is true first command (or block of commands) is executed otherwise "else" section will be run. Below example is the revised form of example 4-2. This time it uses "if…else…" instead of 2 normal "if… statements".

Example 4-4: example4-4.c

```
#include<stdio.h>
main()
{
int n;
printf("Enter a number: ");
scanf("%d",&n);
if(n>=0)
  printf("Number is positive !\n");
else
  printf("Number is negative !\n");

system("pause");
}
```

A useful example and more complex "if" statement

Next example uses an even more complex form of "if statement". This one has several conditional sections and a single else section.

Example 4-5: example4-5.c

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int choice;

while(1)
  {
```

```
  printf("\n\nMenu:\n");
  printf("1- Math Program\n2- Accounting Program\n");
  printf("3- Entertainment Program\n4- Exit");
  printf("\n\nYour choice -> ");
  scanf("%d",&choice);

  if(choice==1)
     printf("\nMath Program Runs. !");
  else if(choice==2)
     printf("\nAccounting Program Runs. !");
  else if(choice==3)
     printf("\nEntertainment Program Runs. !");
  else if(choice==4)
     {
     printf("\nProgram ends.\n");
     exit(0);
     }
  else
     printf("\nInvalid choice");
  }
}
```

Above example is an interesting example of a menu driven programs. A loop which continues forever, prints menu items on screen and waits for answer. Every time an answer is entered, a proper action is performed and again the menu is shown to accept another choice.

Loop continues forever unless you enter 4 as your menu selection. This selection will run the 'exit (0);' function which terminates the program.

A more complex form of "if" statement is used in above example.

```
  if(choice==1)
    command;
  else if(choice==2)
    command;
  else if(choice==3)
    command;
  else if(choice==4)
     {
      block of commands;
     }
  else
    command;
```

This kind of "if" command is used in cases that you need to select from among multiple options. At the end of the “if statement” there is an else section again.

End Note

I want to use the opportunity and give you a caution at the end of this lesson. As there are many commands and programming techniques in any programming language, you will not be able to remember all of them. The only way to remember things is to practice them. You need to start developing your own small programs. Start with lesson exercises and continue with more sophisticated ones. If you do not do this, all your efforts will become useless in a while.

I always quote this in my programming classes: **"No one becomes a programmer without programming"**

Exercises

- Write, compile and test your programs under "Bloodshed Dev-CPP" under windows or Gcc under Linux/UNIX.
- Paid students need to submit their exercises inside e-learning virtual campus. Corrected exercises will be available inside virtual campus.
- If you have obtained the e-Book only, you can discuss your homework questions in Learnem.com support forums (in registered e-book users section).

---

1. **Write a program that accepts 10 scores between 0 and 20 for each student. (use "for" loop) Then calculate average for the student. We want to determine an alphabetical average grade for each student according below table.**

A 16-20	B 12-16	C 10-12	D 7-10	E 4-7	F 0-4

2. **Rewrite example 4-5 to do the following tasks:**

   1- Add two numbers

   2- Subtract two numbers

   3- Multiply two numbers

   4- Exit

   Write necessary program inside a block to accept two numbers and perform necessary action for each menu choice. Program execution must continue until user enters 4 as menu choice.

Lesson 5

Selection using Switch Statements

In previous lesson we saw how we can use "if" statement in programs when they need to choose an option from among several alternatives. There is an alternative C programming command which in some cases can be used as an alternate way to do this. "Switch…case" command is more readable and easier language structure.

"Switch ... case" structure

We can use "if" statement yet but it is better to use "switch" statement which is created for situations that there are several choices.

```
switch(...)
 {
  case ... : command;
             command;
             break;

  case ... : command;
             break;

  default:
             command;
 }
```

In the above switch command we will be able to run different series of commands with each different case.

Example 5-1: example5-1.c

Rewrite example 4-5 of previous lesson and use switch command instead of "if" statement.

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int choice;

while(1)
```

```
  {
  printf("\n\nMenu:\n");
  printf("1- Math Program\n2- Accounting Program\n");
  printf("3- Entertainment Program\n4- Exit");
  printf("\n\nYour choice -> ");
  scanf("%d",&choice);


  switch(choice)
    {
     case 1 : printf("\nMath Program Runs. !");
              break;

     case 2 : printf("\nAccounting Program Runs. !");
              break;

     case 3 : printf("\nEntertainment Program Runs. !");
              break;

     case 4 : printf("\nProgram Ends. !");
              exit(0);

     default:
              printf("\nInvalid choice");

    }
  }
}
```

In "switch…case" command, each “case” acts like a simple label. A label determines a point in program which execution must continue from there. Switch statement will choose one of “case” sections or labels from where the execution of the program will continue. The program will continue execution until it reaches “break” command.

"break" statements have vital rule in switch structure. If you remove these statements, program execution will continue to next case sections and all remaining case sections until the end of "switch" block will be executed (while most of the time we just want one “case” section to be run).

As we told, this is because each “case” acts just as a label. The only way to end execution in switch block is using break statements at the end of each “case” section.

In one of case sections we have not used "break". This is because we have used a termination command "exit(0)" (which terminates program execution) and a break statement will not make any difference.

"default" section will be executed if none of the case sections match switch comparison.

Parameter inside "switch" statement must be of type "int or char" while using a variable in "case sections" is not allowed at all. This means that you are not allowed to use a statement like below one in your switch block.

```
  case i:   something;
            break;
```

Break statement

We used "break" statement in "switch...case" structures in previous part of this lesson. We can also use "break" statement inside loops to terminate a loop and exit it (with a specific condition).

Example 5-2: example5-2.c

```
while (num<20)
 {
  printf("Enter score : ");
  scanf("%d",&scores[num]);
  if(scores[num]<0)
     break;
  }
```

In above example loop execution continues until either num>=20 or entered score is negative. Now see another example.

Example 5-3: example5-3.c

```
#include<stdio.h>
#include<stdlib.h>
main()
{
int choice;


while(1)
  {
  printf("\n\nMenu:\n");
  printf("1- Math Program\n2- Accounting Program\n");
  printf("3- Entertainment Program\n4- Exit");
  printf("\n\nYour choice -> ");
  scanf("%d",&choice);

  switch(choice)
    {
     case 1 : printf("\nMath Program Runs. !");
              break;

     case 2 : printf("\nAccounting Program Runs. !");
              break;
```

```
    case 3 : printf("\nEntertainment Program Runs. !");
             break;

    case 4 : printf("\nProgram Ends. !");
             break;

    default:
             printf("\nInvalid choice");

   }

  if(choice==4) break;

  }
}
```

In above example we have used a break statement instead of exit command used in previous example. Because of this change, we needed a second break statement inside while loop and outside switch block.

If the choice is 4 then this second "break command" will break while loop and we reach the end of main function and when there is no more statements left in main function program terminates automatically.

getchar() and getch()

getchar() function is an alternative choice when you want to read characters from input. This function will get single characters from input and return it back to a variable or expression in our program.

```
  ch=getchar();
```

There is a function for sending characters to output too.

```
  putchar(ch);
```

Example 5-4: example5-4.c

```
#include<stdio.h>
#include<conio.h>
main()
{
char ch;

while(ch!='.')
 {
  ch=getchar();
  putchar(ch);
 }

system("pause");
```

```
}
```

First look at output results and try to guess the reason for results. Console Screen:

```
test            <--This is typed by us
test            <--output of putchar after we pressed enter
again.testing   <--Typed string includes a '.' character
again.          <--Loop terminates when it reaches '.' char
```

Above program reads any character that you have typed on keyboard until it finds a '.' character in input characters. Each key press will be both shown on console and added to a buffer. This buffer will be delivered to 'ch' variable after pressing enter key (not before this). So input characters are buffered until we press enter key and at that moment program execution continues running statements that follow getchar() function (These are loop statements).

If there is a '.' character in buffered characters, loop execution continues sending characters to console with putchar() function until it reaches '.' and after that stops the loop and comes out of while loop.
If it does not encounter '.' in characters it returns to the start of loop, where it starts getchar() function again and waits for user input.

First 'test' string appeared in output is the result of our key presses and second 'test' is printed by putchar statement after pressing enter key.

In some operating systems and some C Programming language compilers, there is another character input function "getch". This one does not buffer input characters and delivers them as soon as it receives them. (in Borland C you need to include another header file <conio.h> to make this new function work).

Example 5-5: example5-5.c

```
#include<stdio.h>
main()
{
char ch;

while(ch!='.')
 {
  ch=getch();
  putch(ch);
 }

system("pause");
}
```

```
Testing.        <--program terminates immediately after we use the
                character '.', also characters are sent to output once
```

With this function we can also check validity of each key press before accepting and using it. If it is not valid we can omit it. Just pay attention that some compilers do not support getch(). (again Borland c needs the header file conio.h to be included)

Look at below example.

Example 5-6: example5-6.c

```
#include<stdio.h>
#include<stdlib.h>
main()
{
char choice;


while(1)
  {
  printf("\n\nMenu:\n");
  printf("1- Math Program\n2- Accounting Program\n");
  printf("3- Entertainment Program\n4- Exit");
  printf("\n\nYour choice -> ");

  choice=getch();

  switch(choice)
    {
     case '1' : printf("\nMath Program Runs. !");
                break;

     case '2' : printf("\nAccounting Program Runs. !");
                break;

     case '3' : printf("\nEntertainment Program Runs. !");
                break;

     case '4' : printf("\nProgram Ends. !");
                exit(0);

    }
  }

}
```

In above example we have rewritten example 5-1. This time we have used getch() function instead of scanf() function. If you test scanf based example you will see that it does not have any control on entered answer string. If user inserts an invalid choice or string (a long junk string for example), it can corrupt the screen. In getch function, user can insert one character at a time. Program immediately gets the character and tests it to see if it matches one of the choices.

In this example we have omitted optional "default" section in “switch...case”. Pay careful attention that in scanf based example we used to get an integer number as the user choice while here we get a character input. As a result previously we were using integer variable in “switch section” and integer numbers in “case sections”. Here we need to change the “switch … case” to match the character type data. So our switch variable is of char type and values being compared are used like '1', '2', '3', '4' because they are character type data ('1' character is different from the integer value 1).

If user presses an invalid key while loop will continue without entering any of "case" sections. Invalid key press will be omitted and only valid key presses will be accepted.

"continue" statement

Continue statement can be used in loops. Like break command "continue" changes flow of a program. It does not terminate the loop however. It just skips the rest of current iteration of the loop and returns to starting point of the loop.

Example 5-7: example5-7.c

```
#include<stdio.h>
main()
{
while((ch=getchar())!='\n')
  {
   if(ch=='.')
     continue;
   putchar(ch);
  }

system("pause");
}
```

In above example, program accepts all input but omits the '.' character from it. The text will be echoed as you enter it but the main output will be printed after you press the enter key (which is equal to inserting a "\n" character) is pressed. As we told earlier this is because getchar() function is a buffered input function.

Exercises

- Write, compile and test your programs under “Bloodshed Dev-CPP” under windows or Gcc under Linux/UNIX.
- Paid students need to submit their exercises inside e-learning virtual campus. Corrected exercises will be available inside virtual campus.
- If you have obtained the e-Book only, you can discuss your homework questions in Learnem.com support forums (in registered e-book users section).

1. **Write a program that reads input until enter key is pressed ('\n' is found in input) and prints number of alphabets and number of spaces in the input string. Use getchar() for reading input characters from console. (example output:** alphabet: 34 spaces: 10 **)**

2. **Write a program that reads input and replaces below characters with the character that comes in front of them. Then writes the output on your screen.**

   a -> c
   f -> g
   n -> l
   k -> r

   Use getchar() and switch...case to write the program.

Lesson 6

More on Functions

IN lesson 1 we that each C program consists of one or more functions. main() is a special function because program execution starts from it.

A function is combined of a block of code that can be called or used anywhere in the program by calling the name. Body of a function starts with '{' and ends with '}' . This is similar to the main function in our previous programs. Example below shows how we can write a simple function.

Example 6-1: example6-1.c

```
#include<stdio.h>

/*Function prototypes*/
myfunc();

main()
{
  myfunc();

  system("pause");
}

myfunc()
{
  printf("Hello, this is a test\n");
}
```

In above example we have put a section of program in a separate function. Function body can be very complex though. After creating a function we can call it using its name. Functions can also call each other. A function can even call itself. This is used in recursive algorithms using a termination condition (to avoid the program to enter in an infinite loop). For the time being do not call a function from inside itself.

By the way pay attention to "function prototypes" section. In some C compilers we are needed to introduce the functions we are creating in a program above the program. Introducing a function is being called "function prototype".

Most of C programming language commands are functions. For example "printf" is a function that accepts one or more arguments and does printing. Bodies of these functions are defined in C library files which are included in your C compiler. When you use a C function, compiler adds its body (where the actual code of the function exists) to your program and then creates the final executable program. Function prototypes (introduction before use) of internal C functions is in header files (i.e stdio.h, …) which are included at the top of a program.

Reasons for using functions

There are many reasons for using functions.

- You may need to reuse a part of code many times in different parts of a program.
- Using functions, program will be divided to separate blocks. Each block will do a specific job. Designing, understanding and managing smaller blocks of code is easier.
- A block of code can be executed with different numbers of initial parameters. These parameters are passed to function with function arguments.

Assume we want to read scores of students from a file, calculate their average and print results. If we write the program just inside a single main() function block we will have a large function. But if we spread the program functionality to different functions, we will have a small main() function and several smaller functions which perform their specific task.

```
main()
{
Read_scores();
calculate()
write_to_file();
print_results()
}
```

Now let's look at this example:

Example 6-2: example6-2.c

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

add();
subtract();
multiply();

main()
{
char choice;

while(1)
  {
  printf("\nMenu:\n");
```

```
  printf("1- Add\n2- Subtract\n");
  printf("3- Multiply\n4- Exit");
  printf("\n\nYour choice -> ");
  choice=getch();

  switch(choice)
    {
     case '1' : add();
                break;

     case '2' : subtract();
                break;

     case '3' : multiply();
                break;

     case '4' : printf("\nProgram Ends. !");
                exit(0);

     default:
                printf("\nInvalid choice");
    }
  }
}

add()
{
float a,b;

printf("\nEnter a:");
scanf("%f",&a);
printf("\nEnter b:");
scanf("%f",&b);

printf("a+b=%f",a+b);
}

subtract()
{
float a,b;

printf("\nEnter a:");
scanf("%f",&a);
printf("\nEnter b:");
scanf("%f",&b);

printf("a-b=%f",a-b);
}
```

```
multiply()
{
float a,b;

printf("\nEnter a:");
scanf("%f",&a);
printf("\nEnter b:");
scanf("%f",&b);

printf("a*b=%f",a*b);
}
```

Function arguments

Functions are able to accept input parameters in the form of variables. These input parameter variables can then be used in function body.

Example 6-3: example6-3.c

```
#include<stdio.h>

/* use function prototypes */
sayhello(int count);

main()
{
sayhello(4);

system("pause");
}

sayhello(int count)
{
int c;

for(c=0;c<count;c++)
  printf("Hello\n");
}
```

In above example we have called sayhello() function with the parameter "4". This function receives an input value and assigns it to "count" variable before starting execution of function body. sayhello() function will then print a hello message 'count' times on the screen.

Again we use a function prototype before we can call a function inside main() or another function in our program. As you see these prototypes are put after pre-compiler commands (those starting with # sign), before main() function at the top of our program. You can copy function header from the function itself for the prototype section. However do not forget that function prototype needs a semicolon at its end while in function itself it does not.

If you do not put prototypes of your functions in your programs you might get an error in most of new C compilers. This error mentions that you need a prototype for each of your functions.

Function return values

In mathematics we generally expect a function to return a value. It may or may not accept arguments but it always returns a value.

```
  y=f(x)
  y=f(x)=x+1
  y=f(x)=1 (arguments are not received or not important)
```

In C programming language we expect a function to return a value too. This return value has a type as other values in C. It can be integer, float, char or anything else. Type of this return value determines type of your function.

Default type of function is int or integer. If you do not indicate type of function that you use, it will be of type int.

As we told earlier every function must return a value. We do this with return command.

```
Sum()
{
 int a,b,c;
 a=1;
 b=4;
 c=a+b;
 reurn c;
}
```

Above function returns the value of variable "c" as the return value of function. We can also use expressions in return command. For example we can replace two last lines of function with 'return a+b;'

If you forget to return a value in a function you will get a warning message in most of C compilers. This message will warn you that your function must return a value. Warnings do not stop program execution but errors stop it.

In our previous examples we did not return any value in our functions. For example you must return a value in main() function.

```
main()
{
.
.
.
return 0;
}
```

Default return value for an int type function is 0. If you do not insert 'return 0' or any other value in your main() function a 0 value will be returned automatically. If you are planning to return an int value in your function, it is seriously preferred to mention the return value in your function header and make.

"void" return value

There is another "void" type of function in C language. Void type function is a function that does not return a value. You can define a function that does not need a return value as "void".

```
void test ()
{
/* fuction code comes here but no return value */
}
```

void functions cannot be assigned to a variable because it does not return value. So you cannot write:

```
  a=test();
```

Using above command will generate an error.

In next lesson we will continue our study on functions. This will include function arguments and more.

Exercises

- Write, compile and test your programs under "Bloodshed Dev-CPP" under windows or Gcc under Linux/UNIX.
- Paid students need to submit their exercises inside e-learning virtual campus. Corrected exercises will be available inside virtual campus.
- If you have obtained the e-Book only, you can discuss your homework questions in Learnem.com support forums (in registered e-book users section).

---

1. **Write a program that accepts a decimal value and prints its binary value. Program should have a function which accepts a decimal number and prints the binary value string.**

2. **Write a program that asks for radius of a circle and calculates its area using a function with return value of float type.**

Lesson 7

Function Arguments

In previous lesson you saw how we could send a single value to a function using a function argument. In this lesson we will send multiple parameters to a function. We will also learn about call by value and call by reference "call types".

Multiple Parameters

In fact you can use more than one argument in a function. Example 7-1 will show you how you can do this.

Example 7-1: example7-1.c

```
#include<stdio.h>

int min(int a,int b);
main()
{
 int m;
 m=min(3,6);
 printf("Minimum is %d",m);

 system("pause");
 return 0;
}

int min(int a,int b)
{
 if(a<b)
   return a;
 else
   return b;
}
```

As you see you can add your variables to arguments list easily. We learned earlier that function prototype is a copy of function header with the difference that prototype ends with semicolon ';' but function header does not.

Call by value

C programming language function calls, use call by value method. Let's see an example to understand this subject better.

Example 7-2: example7-2.c

```
#include<stdio.h>

void test(int a);

main()
{
 int m;
 m=2;

 printf("\nM is %d",m);
 test(m);
 printf("\nM is %d\n",m);

 system("pause");
 return 0;
}

void test(int a)
{
 a=5;
}
```

In main() function, we have declared a variable m. We have assigned value '2' to m. We want to see if function call is able to change value of 'm' as we have modified value of incoming parameter inside test() function. What do you think?

Program output result is:

```
M is 2
M is 2
```

So you see function call has not changed the value of argument. This s is because function-calling method only sends "value of variable" m to the function and it does not send variable itself. Actually it places value of variable m in a memory location called stack and then function retrieves the value without having access to main variable itself. This is why we call this method of calling "call by value".

Call by reference

There is another method of sending variables being called "Call by reference". This second method enables function to modify value of argument variables used in function call.

We will first see an example and then we will describe it.

Example 7-3: example7-3.c

```
#include<stdio.h>
void test(int *ptr);

main()
{
int m;
m=2;

printf("\nM is %d",m);
test(&m);
printf("\nM is %d\n",m);

system("pause");
return 0;
}

void test(int *ptr)
{
printf("\nModifying the value inside memory address %ld",&m);
*ptr=5;
}
```

To be able to modify the value of a variable used as an argument in a function, function needs to know memory address of the variable (where exactly the variable is situated in memory).

In C programming language '&' operator gives the address at which the variable is stored. For example if 'm' is a variable of type 'int' then '&m' will give us the starting memory address of our variable. We call this resulting address 'a pointer'.

```
ptr=&m;
```

In above command ptr variable will contain memory address of variable m. This method is used in some of standard functions in C. For example scanf function uses this method to be able to receive values from console keyboard and put it in a variable. In fact it places received value in memory location of the variable used in function. Now we understand the reason why we add '&' sign before variable names in scanf variables.

```
  Scanf("%d",&a);
```

Now that we have memory address of variable we must use an operator that enables us to assign a value or access to value stored in that address.

As we told, we can find address of variable 'a' using '&' operator.

```
   ptr=&a;
```

Now we can find value stored in variable 'a' using '*' operator:

```
   Val=*ptr;  /* finding the value ptr points to */
```

We can even modify the value inside the address:

```
   *ptr=5;
```

Let's look at our example again. We have passed pointer (memory address) to function. Now function is able to modify value stored inside variable. If you run the program, you will get these results:

```
M is 2
Modifying the value inside memory address 2293620
M is 5
```

So you see this time we have changed value of an argument variable inside the called function (by modifying the value inside the memory location of our main variable).

A useful example, Bubble Sort

In this section we will write a program which sorts an array of numbers using a famous bubble sort algorithm. This algorithm uses a "swap" function which swaps values stored in two memory locations with each other.

Bubble sort compares each two cells of the array and if they are not in correct order, it swaps them. If we need the array to be sorted in ascending order, each cell should be bigger or equal to previous cell and the first cell should be smaller than any other cell. If we perform these compares and swaps for "n-1" times on the entire array, our array will be completely sorted.

Example 7-4 (Bubble sort): example7-4.c

```
#include<stdio.h>
void swap(int *a,int *b);

main()
{
int ar[5],i,j,n;

ar[0]=7;ar[1]=3;ar[2]=9;ar[3]=2;ar[4]=11;

printf("Array before sort:\n\n");
for(i=0;i<5;i++)
  printf("ar[%d]=%d\n",i,ar[i]);

n=5; /*numberof items in sort array*/

for(i=0;i<n-1;i++)
```

```
  for(j=0;j<n-1;j++)
    {
     if(ar[j]>ar[j+1])
      swap(&ar[j],&ar[j+1]);
    }

printf("Array after sort:\n\n");
for(i=0;i<5;i++)
  printf("ar[%d]=%d\n",i,ar[i]);


system("pause");
return 0;
}

void swap(int *a,int *b)
{
int temp;

temp=*a;
*a=*b;
*b=temp;
}
```

The output is:

```
Array before sort:

ar[0]=7
ar[1]=3
ar[2]=9
ar[3]=2
ar[4]=11
Array after sort:

ar[0]=2
ar[1]=3
ar[2]=7
ar[3]=9
ar[4]=11
```

End Note

We have learned fundamentals of programming in C in these 7 first lessons but there are a lot of things we must learn.

In next lessons we will learn more about arrays, pointers, strings, files, structures, memory management etc. But it's enough for now!

Exercises

- Write, compile and test your programs under "Bloodshed Dev-CPP" under windows or Gcc under Linux/UNIX.
- Paid students need to submit their exercises inside e-learning virtual campus. Corrected exercises will be available inside virtual campus.
- If you have obtained the e-Book only, you can discuss your homework questions in Learnem.com support forums (in registered e-book users section).

---

1. **Write a function with two arguments. First argument is "ch" from character type and second variable is "repeat" from int type. When you pass a character and a repeat number (int) it prints character for "repeat" times on console.**

2. **Write a program that reverses the order of members of an array. Use addresses and pointers to displace values in array.**

3. **Describe (in terms of memory locations etc.) how the swap function in our last example works.**