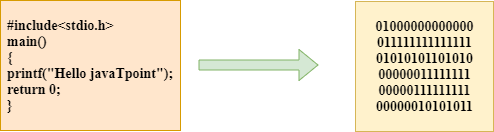
Compilation process in c

What is a compilation?

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

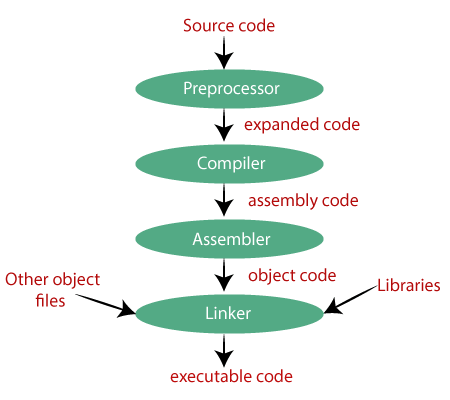


The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

The preprocessor takes the source code as an input, and it removes all the comments from the source code. The preprocessor takes the preprocessor directive and interprets it. For example, if **<stdio.h>,** the directive is available in the program, then the preprocessor interprets the directive and replace this directive with the content of the **'stdio.h'** file.

The following are the phases through which our program passes before being transformed into an executable form:

* **Preprocessor**
* **Compiler**
* **Assembler**
* **Linker**



Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is **'hello.c',** then the name of the object file would be 'hello.obj'.

Linker

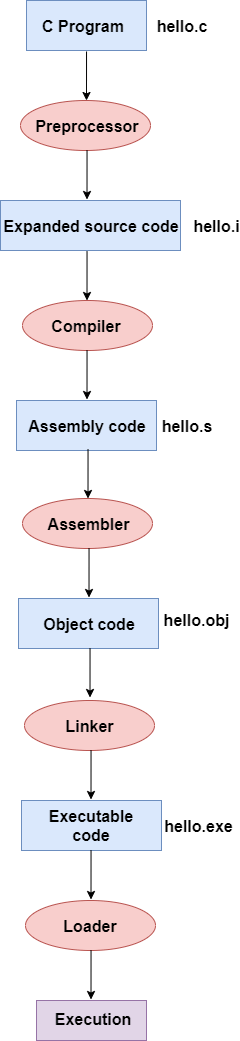
Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.

**Let's understand through an example.**

**hello.c**

1. #include <stdio.h>
2. **int** main()
3. {
4. printf("Hello javaTpoint");
5. **return** 0;
6. }

**Now, we will create a flow diagram of the above program:**



**In the above flow diagram, the following steps are taken to execute a program:**

* Firstly, the input file, i.e., **hello.c,** is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i.**
* The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s.**
* This assembly code is then sent to the assembler, which converts the assembly code into object code.
* After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

AD

printf() and scanf() in C

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:

1. printf("format string",argument\_list);

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The **scanf() function** is used for input. It reads the input data from the console.

1. scanf("format string",argument\_list);

Program to print cube of given number

Let's see a simple example of c language that gets input from the user and prints the cube of the given number.

1. #include<stdio.h>
2. **int** main(){
3. **int** number;
4. printf("enter a number:");
5. scanf("%d",&number);
6. printf("cube of number is:%d ",number\*number\*number);
7. **return** 0;
8. }

**Output**

enter a number:5

cube of number is:125

The **scanf("%d",&number)** statement reads integer number from the console and stores the given value in number variable.

The **printf("cube of number is:%d ",number\*number\*number)** statement prints the cube of number on the console.

Program to print sum of 2 numbers

Let's see a simple example of input and output in C language that prints addition of 2 numbers.

1. #include<stdio.h>
2. **int** main(){
3. **int** x=0,y=0,result=0;
5. printf("enter first number:");
6. scanf("%d",&x);
7. printf("enter second number:");
8. scanf("%d",&y);
10. result=x+y;
11. printf("sum of 2 numbers:%d ",result);
13. **return** 0;
14. }

**Output**

enter first number:9

enter second number:9

sum of 2 numbers:18

Variables in C

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

1. type variable\_list;

The example of declaring the variable is given below:

1. **int** a;
2. **float** b;
3. **char** c;

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

1. **int** a=10,b=20;//declaring 2 variable of integer type
2. **float** f=20.8;
3. **char** c='A';

Rules for defining variables

* A variable can have alphabets, digits, and underscore.
* A variable name can start with the alphabet, and underscore only. It can't start with a digit.
* No whitespace is allowed within the variable name.
* A variable name must not be any reserved word or keyword, e.g. int, float, etc.

**Valid variable names:**

1. **int** a;
2. **int** \_ab;
3. **int** a30;

**Invalid variable names:**

1. **int** 2;
2. **int** a b;
3. **int** **long**;

Types of Variables in C

There are many types of variables in c:

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

Local Variable

A variable that is declared inside the function or block is called a local variable.

It must be declared at the start of the block.

1. **void** function1(){
2. **int** x=10;//local variable
3. }

You must have to initialize the local variable before it is used.

Global Variable

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.

It must be declared at the start of the block.

1. **int** value=20;//global variable
2. **void** function1(){
3. **int** x=10;//local variable
4. }

Static Variable

A variable that is declared with the static keyword is called static variable.

It retains its value between multiple function calls.

1. **void** function1(){
2. **int** x=10;//local variable
3. **static** **int** y=10;//static variable
4. x=x+1;
5. y=y+1;
6. printf("%d,%d",x,y);
7. }

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

Automatic Variable

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

1. **void** main(){
2. **int** x=10;//local variable (also automatic)
3. auto **int** y=20;//automatic variable
4. }

External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

*myfile.h*

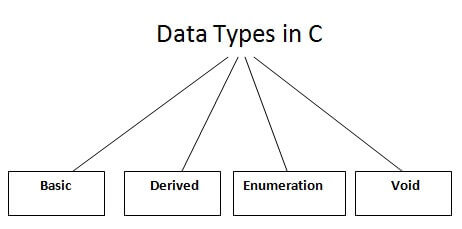
1. **extern** **int** x=10;//external variable (also global)

*program1.c*

1. #include "myfile.h"
2. #include <stdio.h>
3. **void** printValue(){
4. printf("Global variable: %d", global\_variable);
5. }

# Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

|  |  |
| --- | --- |
| **Types** | **Data Types** |
| Basic Data Type | int, char, float, double |
| Derived Data Type | array, pointer, structure, union |
| Enumeration Data Type | enum |
| Void Data Type | void |

## Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

|  |  |  |
| --- | --- | --- |
| **Data Types** | **Memory Size** | **Range** |
| **char** | 1 byte | −128 to 127 |
| signed char | 1 byte | −128 to 127 |
| unsigned char | 1 byte | 0 to 255 |
| **short** | 2 byte | −32,768 to 32,767 |
| signed short | 2 byte | −32,768 to 32,767 |
| unsigned short | 2 byte | 0 to 65,535 |
| **int** | 2 byte | −32,768 to 32,767 |
| signed int | 2 byte | −32,768 to 32,767 |
| unsigned int | 2 byte | 0 to 65,535 |
| **short int** | 2 byte | −32,768 to 32,767 |
| signed short int | 2 byte | −32,768 to 32,767 |
| unsigned short int | 2 byte | 0 to 65,535 |
| **long int** | 4 byte | -2,147,483,648 to 2,147,483,647 |
| signed long int | 4 byte | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 byte | 0 to 4,294,967,295 |
| **float** | 4 byte |  |
| **double** | 8 byte |  |
| **long double** | 10 byte |  |

# Keywords in C

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| auto | break | case | char | const | continue | default | do |
| double | else | enum | extern | float | for | goto | if |
| int | long | register | return | short | signed | sizeof | static |
| struct | switch | typedef | union | unsigned | void | volatile | while |

We will learn about all the C language keywords later.

C Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

There are following types of operators to perform different types of operations in C language.

* Arithmetic Operators
* Relational Operators
* Shift Operators
* Logical Operators
* Bitwise Operators
* Ternary or Conditional Operators
* Assignment Operator
* Misc Operator

Precedence of Operators in C

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

Let's understand the precedence by the example given below:

1. **int** value=10+20\*10;

The value variable will contain **210** because \* (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C operators is given below:

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

Comments in C

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single Line Comments

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

1. #include<stdio.h>
2. **int** main(){
3. //printing information
4. printf("Hello C");
5. **return** 0;
6. }

Output:

Hello C

Even you can place the comment after the statement. For example:

1. printf("Hello C");//printing information

Mult Line Comments

Multi-Line comments are represented by slash asterisk \\* ... \*\. It can occupy many lines of code, but it can't be nested. Syntax:

1. /\*
2. code
3. to be commented
4. \*/

Let's see an example of a multi-Line comment in C.

1. #include<stdio.h>
2. **int** main(){
3. /\*printing information
4. Multi-Line Comment\*/
5. printf("Hello C");
6. **return** 0;
7. }

Output:

Hello C

C Format Specifier

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character.

**The commonly used format specifiers in printf() function are:**

|  |  |
| --- | --- |
| **Format specifier** | **Description** |
| %d or %i | It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values. |
| %u | It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value. |
| %o | It is used to print the octal unsigned integer where octal integer value always starts with a 0 value. |
| %x | It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc. |
| %X | It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc. |
| %f | It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'. |
| %e/%E | It is used for scientific notation. It is also known as Mantissa or Exponent. |
| %g | It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value after the decimal in input would be exactly the same as the value in the output. |
| %p | It is used to print the address in a hexadecimal form. |
| %c | It is used to print the unsigned character. |
| %s | It is used to print the strings. |
| %ld | It is used to print the long-signed integer value. |

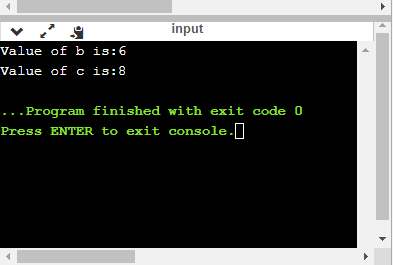
**Let's understand the format specifiers in detail through an example.**

* **%d**

1. **int** main()
2. {
3. **int** b=6;
4. **int** c=8;
5. printf("Value of b is:%d", b);
6. printf("\nValue of c is:%d",c);
8. **return** 0;
9. }

In the above code, we are printing the integer value of b and c by using the %d specifier.

**Output**



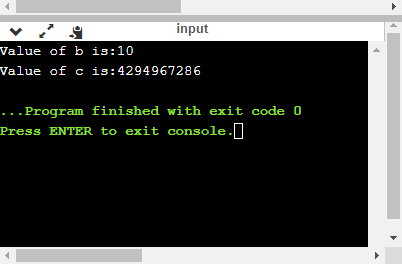
* **%u**

AD

1. **int** main()
2. {
3. **int** b=10;
4. **int** c= -10;
5. printf("Value of b is:%u", b);
6. printf("\nValue of c is:%u",c);
8. **return** 0;
9. }

In the above program, we are displaying the value of b and c by using an unsigned format specifier, i.e., %u. The value of b is positive, so %u specifier prints the exact value of b, but it does not print the value of c as c contains the negative value.

**Output**

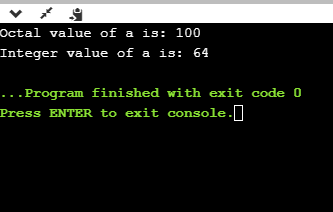


* **%o**

1. **int** main()
2. {
3. **int** a=0100;
4. printf("Octal value of a is: %o", a);
5. printf("\nInteger value of a is: %d",a);
6. **return** 0;
7. }

In the above code, we are displaying the octal value and integer value of a.

**Output**

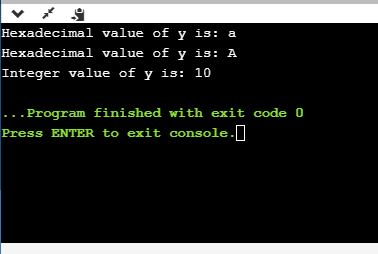


* **%x and %X**

1. **int** main()
2. {
3. **int** y=0xA;
4. printf("Hexadecimal value of y is: %x", y);
5. printf("\nHexadecimal value of y is: %X",y);
6. printf("\nInteger value of y is: %d",y);
7. **return** 0;
8. }

In the above code, y contains the hexadecimal value 'A'. We display the hexadecimal value of y in two formats. We use %x and %X to print the hexadecimal value where %x displays the value in small letters, i.e., 'a' and %X displays the value in a capital letter, i.e., 'A'.

**Output**

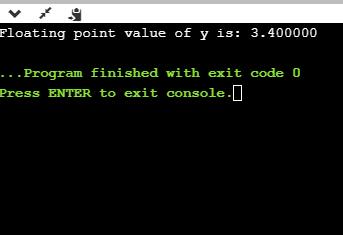


* **%f**

1. **int** main()
2. {
3. **float** y=3.4;
4. printf("Floating point value of y is: %f", y);
5. **return** 0;
6. }

The above code prints the floating value of y.

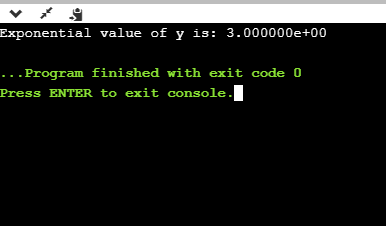
**Output**



* **%e**

1. **int** main()
2. {
3. **float** y=3;
4. printf("Exponential value of y is: %e", y);
5. **return** 0;
6. }

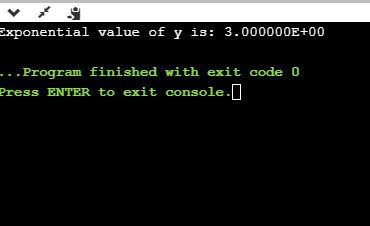
**Output**



* **%E**

1. **int** main()
2. {
3. **float** y=3;
4. printf("Exponential value of y is: %E", y);
5. **return** 0;
6. }

**Output**

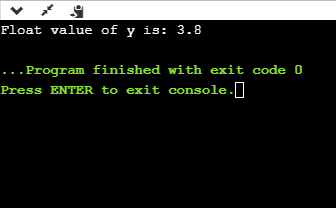


* **%g**

1. **int** main()
2. {
3. **float** y=3.8;
4. printf("Float value of y is: %g", y);
5. **return** 0;
6. }

In the above code, we are displaying the floating value of y by using %g specifier. The %g specifier displays the output same as the input with a same precision.

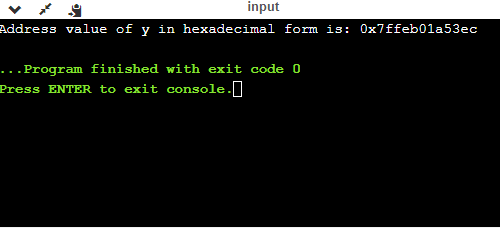
**Output**



* **%p**

1. **int** main()
2. {
3. **int** y=5;
4. printf("Address value of y in hexadecimal form is: %p", &y);
5. **return** 0;
6. }

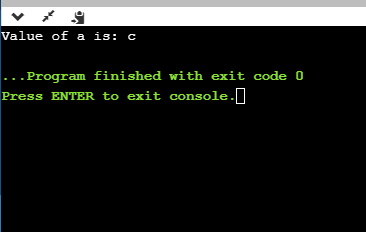
**Output**



* **%c**

1. **int** main()
2. {
3. **char** a='c';
4. printf("Value of a is: %c", a);
5. **return** 0;
6. }

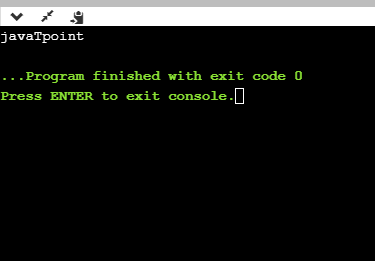
**Output**



* **%s**

1. **int** main()
2. {
3. printf("%s", "javaTpoint");
4. **return** 0;
5. }

**Output**



Minimum Field Width Specifier

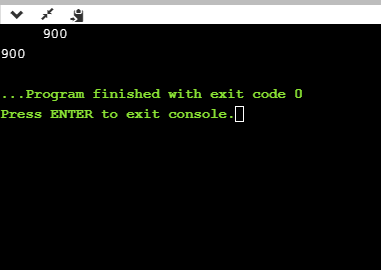
Suppose we want to display an output that occupies a minimum number of spaces on the screen. You can achieve this by displaying an integer number after the percent sign of the format specifier.

AD

1. **int** main()
2. {
3. **int** x=900;
4. printf("%8d", x);
5. printf("\n%-8d",x);
6. **return** 0;
7. }

In the above program, %8d specifier displays the value after 8 spaces while %-8d specifier will make a value left-aligned.

**Output**

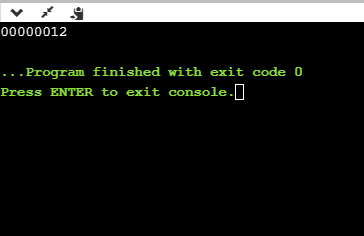


**Now we will see how to fill the empty spaces. It is shown in the below code:**

1. **int** main()
2. {
3. **int** x=12;
4. printf("%08d", x);
5. **return** 0;
6. }

In the above program, %08d means that the empty space is filled with zeroes.

**Output**



Specifying Precision

We can specify the precision by using '.' (Dot) operator which is followed by integer and format specifier.

1. **int** main()
2. {
3. **float** x=12.2;
4. printf("%.2f", x);
5. **return** 0;
6. }

**Output**

