

Every C program consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

```
#include <stdio.h>
int main(void)
{
    /* my first program in C */
    printf("Hello, World! \n");
    return 0;
}
```

Various parts of the above program:

- The first line of the program `#include <stdio.h>` is a preprocessor command, which tells a C compiler to include `stdio.h` file before going into actual compilation.
- The next line `int main(void)` is the main function where the program execution begins.
- The next line `/*...*/` will be ignored by the compiler and it has been put to add additional comments in the program. So, such lines are called comments in the program.
- The next line `printf(...)` is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
- The next line `return 0;` terminates the `main()` function and returns the value 0 to the calling function or operating system.

The C program consists of various tokens - a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens:

```
printf("Hello, World! \n");
```

The individual tokens are:

```
printf
(
"Hello, World! \n"
)
;
```

The semicolon is a statement terminator - each individual statement must end with a semicolon, it indicates the end of one logical entity.

Below are two different C statements:

```
printf("Hello, World! \n");
return 0;
```

Comments are text areas within your C program that are ignored by the compiler. They start with `//` for a single line or `/* ... */` for multi-line as shown below:

```
/* my first program in C */
// my first program in C
```

The C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore ( `_` ) followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, \$, and % within identifiers. C is a case-sensitive programming language. Thus, *Manpower* and *manpower* are two different identifiers in C.

The following list shows the reserved, or keywords in C. These reserved words may not be used as constant, variable, or any other identifier names.

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

A line containing only whitespace is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as `int`, ends and the next element begins. Therefore, in the following statement:

```
int age;
```

There must be at least one whitespace character (usually a space) between `int` and `age` for the compiler to be able to distinguish them. On the other hand, in the following statement:

```
fruit=apples+oranges;    // get the total fruit
```

No whitespace characters are necessary between `fruit` and `=`, or between `=` and `apples`, or between `apples` and `+`, or between `+` and `oranges`, although you are free to include some if you wish to increase readability.

Data types in C refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows:

#### Basic Types

They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.

#### Enumerated types

They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.

#### The type void

The type specifier `void` indicates that no value is available.

#### Derived types

They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see the basic types in the following section, where as other types will be covered in the upcoming handouts.

The following table provides the details of standard integer types with their storage sizes and value ranges:

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

To get the exact size of a type or a variable on a particular platform, you can use the `sizeof()` operator. The expressions `sizeof(type)` yields the storage size of the object or type in bytes.

```
#include <stdio.h>
#include <limits.h>
void main (void)
{
    printf("Storage size for int : %d \n", sizeof(int));
}
```

When you compile and execute the above program, it produces the following result:

Storage size for int: 4

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision:

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file `float.h` defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values:

```
#include <stdio.h>
#include <float.h>
void main (void)
{
    printf("Storage size for float : %d \n", sizeof(float));
    printf("Minimum float positive value: %E\n", FLT_MIN );
    printf("Maximum float positive value: %E\n", FLT_MAX );
    printf("Precision value: %d\n", FLT_DIG );
}
```

When you compile and execute the above program, it produces the following result:

Storage size for float : 4  
Minimum float positive value: 1.175494E-38  
Maximum float positive value: 3.402823E+38  
Precision value: 6

The void type specifies that no value is available. It is used in three kinds of situations:

Function returns as void

There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status);

#### Function arguments as void

There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, int rand(void);

#### Pointers to void

A pointer of type void \* represents the address of an object, but not its type. For example, a memory allocation function void \*malloc( size\_t size ); returns a pointer to void which can be casted to any data type.

A variable is a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types:

char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, type must be a valid C data type including char, w\_char, int, float, double, bool, or any user-defined object; and variable\_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line int i, j, k; declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;         // definition and initializing d and f.
byte z = 22;              // definition and initializes z.
char x = 'x';             // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use the keyword `extern` to declare a variable at any place. Though you can declare a variable multiple times in your C program, it can be defined only once in a file, a function, or a block of code.

#### Example

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function

```
#include <stdio.h>
// Variable declaration:
extern int a, b;
extern int c;
extern float f;
void main (void)
{
    /* variable definition: */
    int a, b;
    int c;
    float f;
    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;
    printf("value of c : %d \n", c);
    f = 70.0/3.0;
    printf("value of f : %f \n", f);
}
```

When the above code is compiled and executed, it produces the following result:

```
value of c : 30
value of f : 23.333334
```

The same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example:

```
// function declaration
int func();
int main()
{
    // function call
    int i = func();
}
// function definition
int func()
{
```

```
    return 0;
}
```

## Lvalues and Rvalues in C

There are two kinds of expressions in C:

- **lvalue:** Expressions that refer to a memory location are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue:** The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.

Variables are lvalues and so they may appear on the left-hand side of an assignment. Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side. Take a look at the following valid and invalid statements:

```
int g = 20; // valid statement
10 = 20; // invalid statement; would generate compile-time error
```

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an *integer constant*, a *floating constant*, a *character constant*, or a *string literal*. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212          /* Legal */
215u         /* Legal */
0xFFeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of integer literals:

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
```

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159      /* Legal */
314159E-5L   /* Legal */
510E         /* Illegal: incomplete exponent */
210f         /* Illegal: no decimal or exponent */
.e55         /* Illegal: missing integer or fraction */
```

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of char type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C that represent special meaning when preceded by a backslash for example, newline (\n) or tab (\t).

Here, you have a list of such escape sequence codes:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh...	Hexadecimal number of one or more digits

Following is the example to show a few escape sequence characters:

```
#include <stdio.h>
int main()
{
    printf("Hello\tWorld\n\n");
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello      World
```

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using white spaces.