RV College of Engineering®

Go, change the world

Unit –2

Introduction to C

Contents

1. Introduction, structure of a C program, Writing the first program, Files used in a C program. Compiling and executing C Programs using comments, C Tokens, Character set in C, Keywords, Identifiers, Basic Data Types in C, Variables, Constants, I/O statements in C.

2. Operators in C, Type conversion and type casting, scope of variables.

Introduction
COrientation

- Created in 1972 to write operating systems (Unix in particular) By Dennis Ritchie at Bell Labs
- Evolved from B
- It can be portable to other hardware (with careful design – use Plauger's The Standard C Library book)
- Built for performance and memory management – operating systems, embedded systems, real-time systems, communication systems

- C is a High level, general – purpose structured programming language. Instructions of C consists of terms that are very closely same to algebraic expressions, consisting of certain English keywords such as if, else, for, do and while
- C contains certain additional features that allows it to be used at a lower level, acting as bridge between machine language and the high level languages. Hence, it may be called as Middle level language
- This allows C to be used for system programming as well as for applications programming

Later Languages

- 1979 C++ by Bjarn Stroustrup also at Bell
  - Object orientation
- 1991 Java by Sun
  - Partial compile to java bytecode: virtual machine code
  - Write once, run anywhere
  - Memory manager – garbage collection
  - Many JVMs written in C / C++

Structure of a C program

- A C program contains one or more functions, where a function is defined as a group of statements that perform a well-defined task.
- The statements in a function are written in a logical sequence to perform a specific task. The *main*() function is the most important function and is a part of every C program.
- A C program can have any number of functions depending on the tasks that have to be performed, and each function can have any number of statements arranged according to specific meaningful sequence.

```
main()
{
Statement 1;
Statement 2;
.............
Statement N;
}
Function1()
{
Statement 1;
Statement 2;
Statement N;
}
Function2()
{
Statement 1;
Statement 2;
Statement N;
}
```

First Program

```
#include <stdio.h>
int main()
{
        /* My first program */
        printf("Hello World! \n");

        return 0;
}
```

```
Output :

 Hello World!
```

- C is case sensitive.
- End of each statement must be marked with a semicolon (;).
- Multiple statements can be on the same line.
- ***White space*** *(e.g. space, tab, enter, …)* is ignored.

First Program

```
#include <stdio.h>
int main()
{
        /* My first program */
        printf("Hello World! \n");

        return 0;
}
```

```
Output :

 Hello World!
```

- The C program starting point : **`main().`**
- **`main()  {}`** indicates where the program actually starts and ends.
- In general, braces {} are used throughout C to enclose a block of statements to be treated as a unit.
- ***COMMON ERROR: unbalanced number of open and close curly brackets!***

First Program

```
#include <stdio.h>
int main()
{
        /* My first program */
        printf("Hello World! \n");

        return 0;
}
```

```
Output :

 Hello World!
```

- **#include <stdio.h>**
  - Including a header file `stdio.h`
  - Allows the use of `printf` function
  - For each function built into the language, an associated ***header file must be included.***
- **printf() is actually a function (procedure) in C that is used for printing variables and text**

First Program

```
#include <stdio.h>
int main()
{
        /* My first program */
        printf("Hello World! \n");

        return 0;
}
```

```
Output :

 Hello World!
```

- Comments
  - /* My first program */
  - Comments are inserted between “/*” and “*/”
  - Or, you can use “//”
  - Primarily they serve as *internal documentation for program structure and function*.

First Program

- first.c. If you are a Windows user, then open the command prompt by clicking Start->Run and typing “command” and clicking Ok. Using the command prompt, change to the directory in which you saved your file and then type:

  C:\>tc first.c

- In case you are working on UNIX/Linux operating system, then exit the text editor and type

  $cc first.c –ofirst

- If everything is right, then no error(s) will be reported and the compiler will create an .exe file for your program. This .exe file can be directly run by typing

  "***first.exe***" for Windows and "***./first***" for UNIX/Linux operating system

- When you run the .exe file, the output of the program will be displayed on screen. That is,

  Hello World!

First Program

Using Comments

Comments are a way of explaining what a program does. C supports two types of comments.

- // is used to comment a single statement.
- /* is used to comment multiple statements. A /* is ended with */ and all statements that lie between these characters are commented.

Note that comment statements are not executed by the compiler. Rather, they are ignored by the compiler as they are simply added in programs to make the code understandable by programmers as well as other users.

First Program

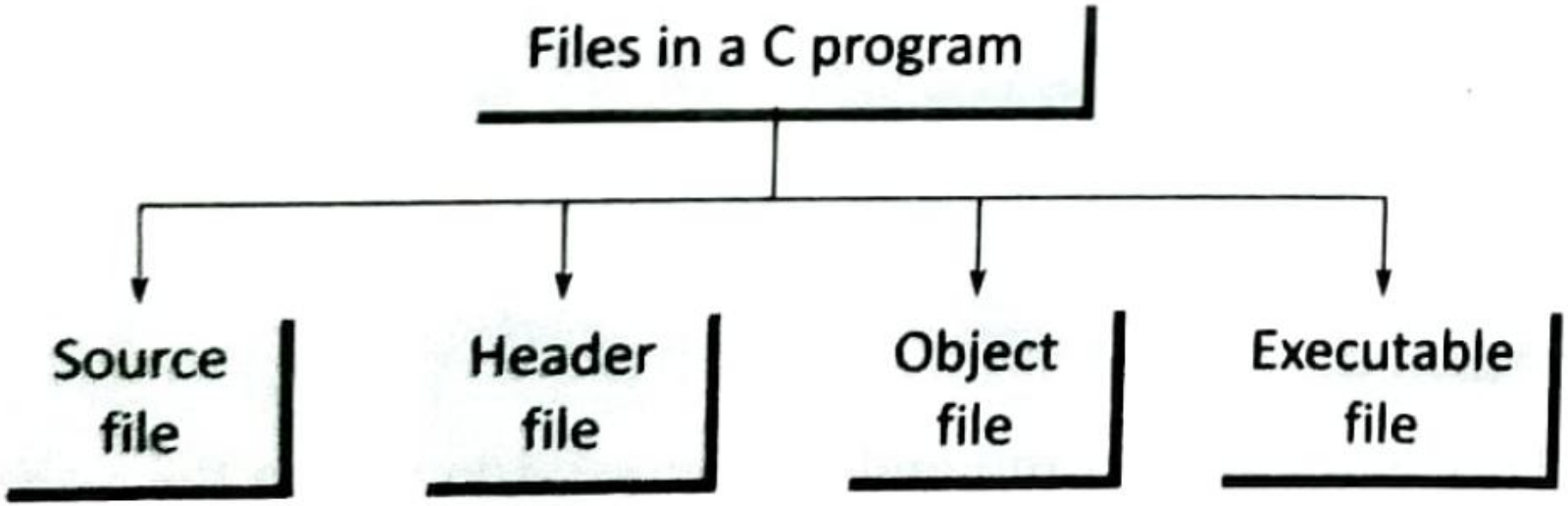
Standard Header Files

Standard header files include:

- **string.h** : for string handling functions
- **stdlib.h** : for some miscellaneous functions
- **stdio.h** : for standardized input and output functions
- **math.h** : for mathematical functions
- **alloc.h** : for dynamic memory allocation
- **conio.h** : for clearing the screen

All the header files are referenced at the start of the source code file that uses one or more functions from these files.

Files used in a C program



Files used in a C program

- Editor - code by programmer
- Compiling using gcc:
  - Preprocess – expand the programmer's code
  - Compiler – create machine code for each file
  - Linker – links with libraries and all compiled objects to make executable
- Running the executable:
  - Loader – puts the program in memory to run it
  - CPU – runs the program instructions

Compiling and executing C Programs

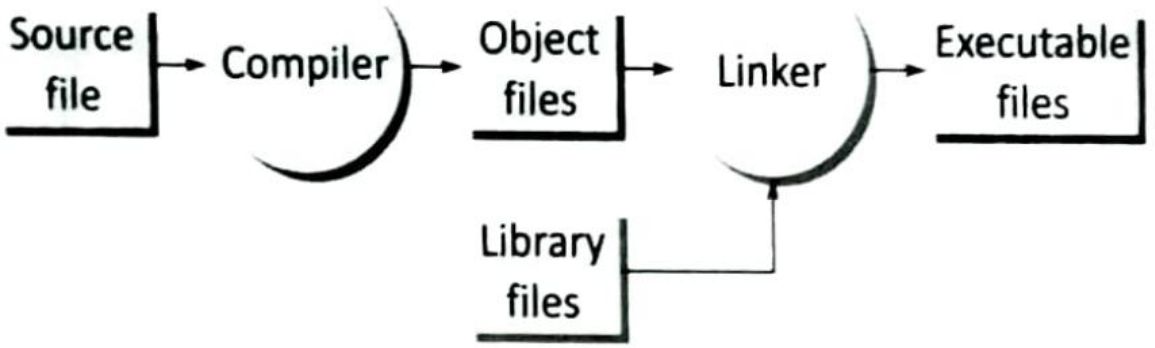


Figure 2.4 Overview of compilation and execution process

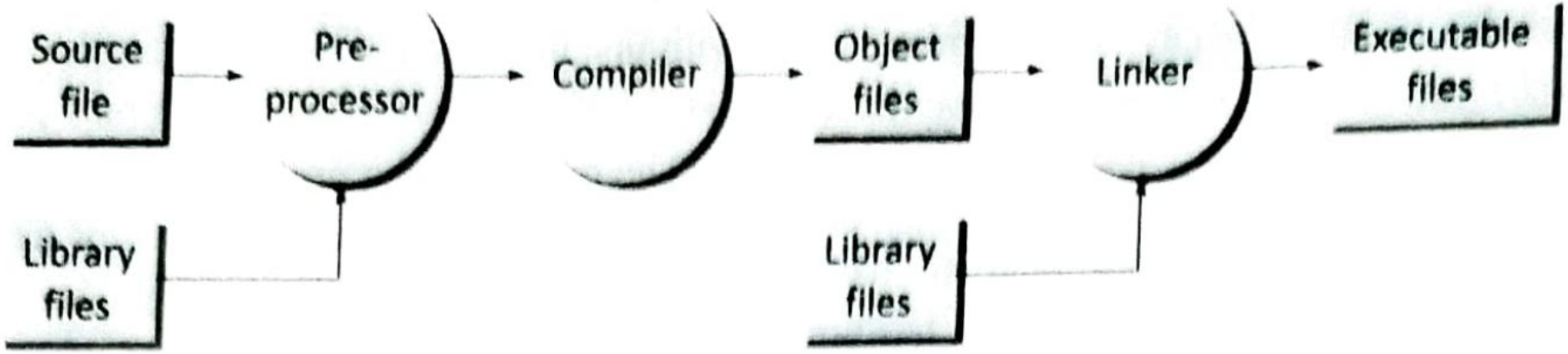


Figure 2.5 Preprocessing before compilation

Compiling and executing C Programs

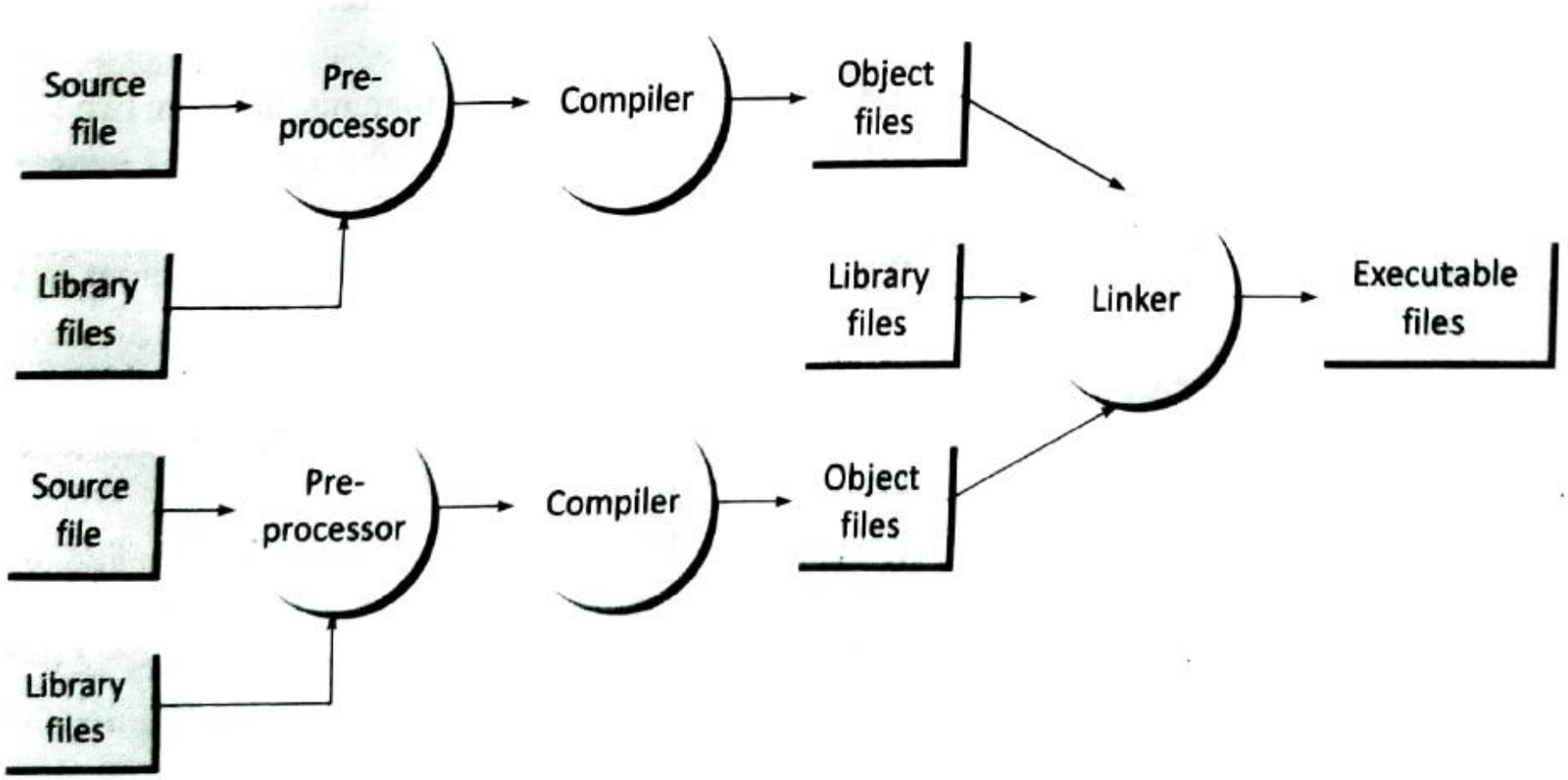
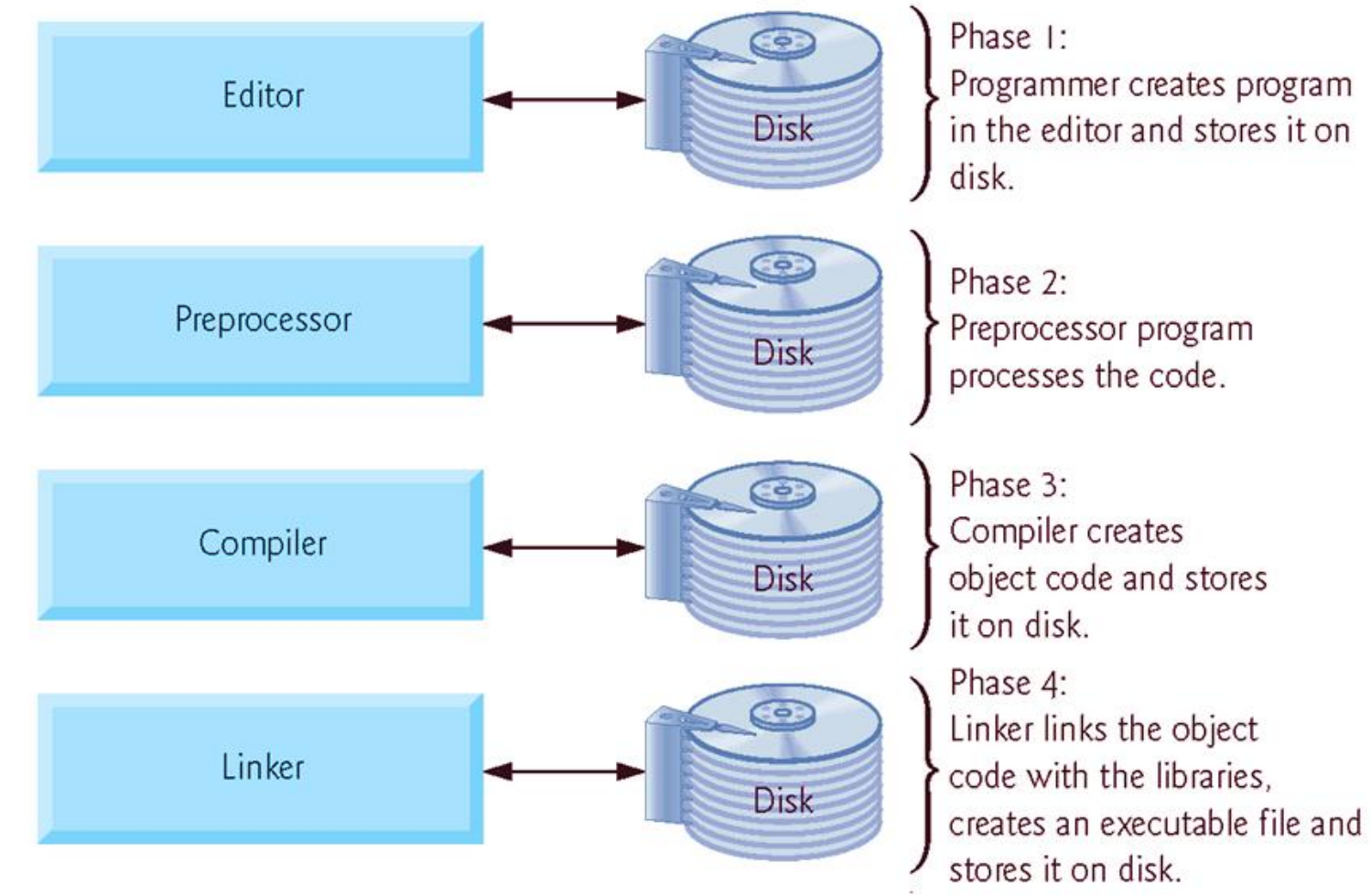
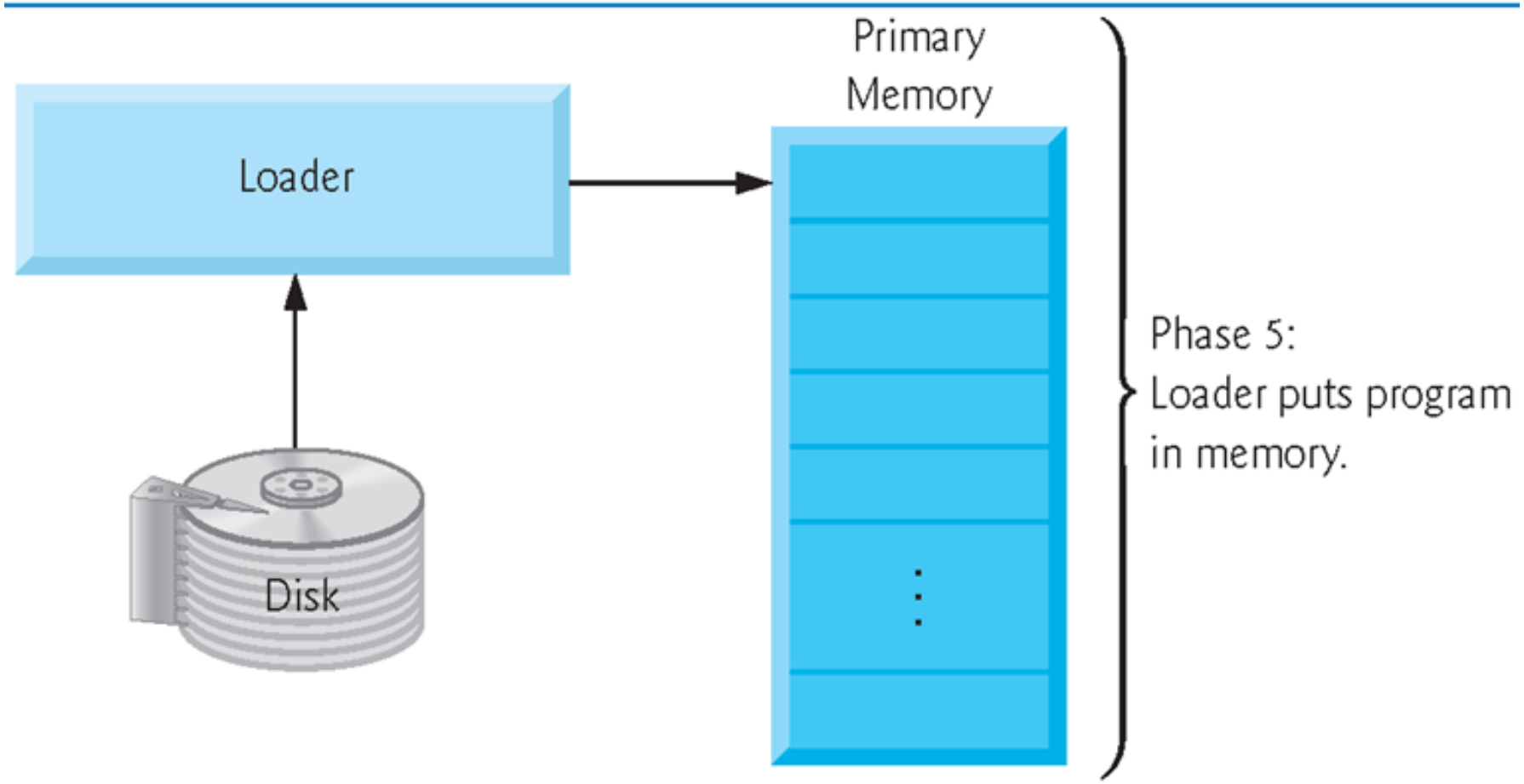


Figure 2.6 Modular programming—the complete compilation and execution process

Compiling and executing C Programs



Typical C development environment



Typical C development environment

Character Set in C

Just like we use a set of various words, numbers, statements, etc., in any language for communication, the C programming language also consists of a set of various different types of characters. These are known as the characters in C. They include digits, alphabets, special symbols, etc. The C language provides support for about 256 characters.

Every program that we draft for the C program consists of various statements. We use words for constructing these statements. Meanwhile, we use characters for constructing these statements. These characters must be from the C language character set. Let us look at the set of characters offered by the C language.

Types of Characters in C

The C programming language provides support for the following types of characters. In other words, these are the valid characters that we can use in the C language:

Type of character		Description
Digits		0 to 9
Alphabets	Lowercase Alphabets	a to z
	Uppercase Alphabets	A to Z
Main characters		` ~ @ ! $ # ^ * % & ( ) [ ] { } < > + = _ – | / \ ; : ‘ “ , . ?

All of these serve a different set of purposes, and we use them in different contexts in the C language.

White space character	Meaning
\b	Blank space
\t	Horizontal tab
\v	Vertical return
\r	Carriage return
\f	Form feed
\n	New line

Figure 2.8 White space characters in C

CTokens

The words formed from the character set are building blocks of C and are sometimes known as tokens. These tokens represent the individual entity of language. The following different types of token are used in C

1) Keywords
2) Identifiers , Variables
3) Constants
4) Strings
5) Special symbols
6) Operators

Keywords

- C has a set of reserved words often known as keywords that cannot be used as an identifier.
- All keywords are basically a sequence of characters that have a fixed meaning. By convention, all keywords must be written in lower case letters.
- There are 32 keywords in C program

Keywords in C language

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

Identifiers

Identifiers are basically names given to program elements such as variables, arrays, and functions. They are formed by using a sequence of letters (both uppercase and lowercase), numerals, and underscores

- Identifiers cannot include any special characters or punctuation marks (like #, $, ^, ?, ., etc.) except the underscore "_".
- There cannot be two successive underscores.
- Keywords cannot be used as identifiers.
- The case of alphabetic characters that form the identifier name is significant. For example, 'FIRST' is different from 'first' and 'First'.
- Identifiers must begin with a letter or an underscore. However, use of underscore as the first character must be avoided because several complier-defined identifiers in the standard C library have underscore as their first character. So, inadvertently duplicated names may cause definition conflicts.
- Identifiers can be of any reasonable length. They should not contain more than 31 characters. (They can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.)

BASIC DATA TYPES

Data type determines the set of values that a data item can take and the operations that can be performed on the item. C language provides four basic data types.

Type	Description
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.

Data Type	Size in Bytes	Range	Use
char	1	–128 to 127	To store characters
int	2	–32768 to 32767	To store integer numbers
float	4	3.4E–38 to 3.4E+38	To store floating point numbers
double	8	1.7E–308 to 1.7E+308	To store big floating point numbers

BASIC DATA TYPES

Data Type	Size in Bytes	Range
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	–128 to 127
int	2	–32768 to 32767
unsigned int	2	0 to 65535
signed int	2	–32768 to 32767
short int	2	–32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	–2147483648 to 2147483647
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

BASIC DATA TYPES

Floating point numbers use the IEEE (Institute o Electrical and Electronics Engineering) format to represent mantissa and exponents

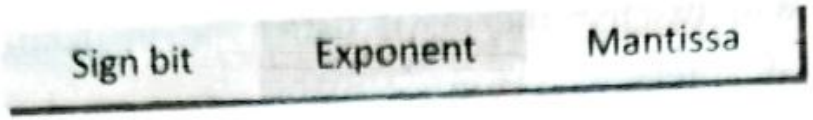


Figure 2.9 IEEE format for storing floating point numbers

Convert the floating point number 5.32 into an IEEE normalized form.

2 | 5 | R		
2 | 2 | 1		
2 | 1 | 0		
0 | 1		
Write the remainders in the reverse order of generation		
	$0.32 \times 2 = 0.64$	0
	$0.64 \times 2 = 1.28$	1
	$0.28 \times 2 = 0.56$	0
	$0.56 \times 2 = 1.12$	1
	Write the whole numbers in the same order of generation	

Thus, the binary equivalent of 5.32 = 101.0101.
The normalized form of this binary number is obtained by adjusting the exponent until the decimal point is to the right of the most significant 1.
Therefore, the normalized binary equivalent = 1.010101×2^2.

Variables

- A variable is nothing but 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. There are following basic variable types –

Type	Description
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.

Variables

Numeric Variables

Numeric variables can be used to store either integer values or floating point values. Modifiers like short, long, signed, and unsigned can also be used with numeric variables.

Character Variables

Character variables are just single characters enclosed within single quotes. These characters could be any character from the ASCII character set—letters (‘a’, ‘A’), numerals (‘2’), or special characters (‘&’).

Variables

Declaring Variables

To declare a variable, specify the data type of the variable followed by its name. The data type indicates the kind of values that the variable can store. In C, variable declaration always ends with a semi-colon. For example,

```
int emp_num;
float salary;
char grade;
double balance_amount;
unsigned short int acc_no;
```

Initializing Variables

While declaring the variables, we can also initialize them with some value. For example,

```
int emp_num = 7;
float salary = 9800.99
char grade = 'A';
double balance_amount = 100000000;
```

Constants

- A constant is a value or an identifier whose value cannot be altered in a program. For example: 1, 2.5,
- As mentioned, an identifier also can be defined as a constant. eg. const double PI = 3.14
- Here, PI is a constant. Basically what it means is that, PI and 3.14 is same for this program.

Integer constants

- A integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:
- decimal constant (base 10) ex: 10,525, 22000
- octal constant (base 8) Ex: 023,045,07
- hexadecimal constant (base 16) Ex: 0x24. 0xA5, 0x72b

Declaring Constants

To declare a constant, precede the normal variable declaration with const keyword and assign it a value.

```
const float pi = 3.14;
```

Constants

Floating-point constants

A floating point constant is a numeric constant that has either a fractional form or an exponent form. For example: 2.0,0.0000234,-0.22E-5

Character constants

A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'

String constants

String constants are the constants which are enclosed in a pair of double-quote marks. For example: "good" ,"x", "Earth is round\n"

Escape Sequences

Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc. In order to use these characters, escape sequence is used.

For example: \n is used for newline. The backslash (\ ) causes "escape" from the normal way the characters are interpreted by the compiler.escape

Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

Operators in C

C language supports different types of operators, which can be used with variables and constants to form expressions. These operators can be categorized into the following major groups:

• Arithmetic operators • Equality operators • Unary operators • Bitwise operators • Comma operator	• Relational operators • Logical operators • Conditional operator • Assignment operators • Size of operator

Arithmetic Operators

Consider three variables declared as,

int a=9, b=3, result;

Table shows the arithmetic operators, their syntax, and usage in C language.

Table Arithmetic operators

Operation	Operator	Syntax	Comment	Result
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a / b	3
Addition	+	a + b	result = a + b	12
Subtraction	-	a - b	result = a - b	6
Modulus	%	a % b	result = a % b	0

a and b (on which the operator is applied) are called operands.

Relational Operators

A relational operator, also known as a comparison operator, is an operator that compares two values or expressions. Relational operators return true or false value, depending on whether the conditional relationship between the two operands holds or not.

Table Relational operators

Operator	Meaning	Example
<	Less than	3 < 5 gives 1
>	Greater than	7 > 9 gives 0
<=	Less than or equal to	100 <= 100 gives 1
>=	Greater than equal to	50 >=100 gives 0

Equality Operators

C language also supports two equality operators to compare operands for strict equality or inequality. They are equal to (==) and not equal to (!=) operators. The equality operators have lower precedence than the relational operators.

Table Equality operators

Operator	Meaning
==	Returns 1 if both operands are equal, 0 otherwise
!=	Returns 1 if operands do not have the same value, 0 otherwise

Logical Operators

C language supports three logical operators. They are logical AND (&&), logical OR (||), and logical NOT (!). Logical AND and LogicaL OR combines two relational quantities(Ex A>B && B<C). Logical operators in C are used to combine multiple conditions/constraints. Logical Operators returns either 0 or 1,. As in case of arithmetic expressions, logical expressions are evaluated from left to right.

Truth table of logical AND

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Truth table of logical OR

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

Truth table of logical NOT

A	! A
0	1
1	0

Explanation of logical operator program

a= 5, b=5, c=10

(a == b) && (c > 5) evaluates to 1

because both operands (a == b) and (c > b) is 1 (true).

(a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).

(a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).

(a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).

!(a != b) evaluates to 1 because operand (a != b) is 0 (false).

Hence, !(a != b) is 1 (true).

!(a == b) evaluates to 0 because (a == b) is 1 (true).

Hence, !(a == b) is 0 (false).

Explanation of logical operator program

a= 5, b=5, c=10

- (a == b) && (c > 5) evaluates to 1
- because both operands (a == b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false).
- Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true).
- Hence, !(a == b) is 0 (false).

Unary Operators

Unary operators act on single operands. C language supports three unary operators. They are unary minus, increment, and decrement operators.

Unary Minus (–)

Unary minus operator negates the value of its operand. For example,

```
int a, b = 10;
a = -(b);
```

The result of this expression is a = –10

Increment Operator (++) and Decrement Operator (– –)

The increment operator is a unary operator that increases the value of its operand by 1. Similarly, the decrement operator decreases the value of its operand by 1.

The increment/decrement operators have two variants: prefix and postfix. In a prefix expression (++x or – –x), the operator is applied before the operand while in a postfix expression (x++ or x– –), the operator is applied after the operand.

Example

```
int x = 10, y;
y = x++; is equivalent to writing
y = x;     /*y = 10*/        (First Assign the value and then Increment)
x = x + 1;    /*x = 11*/
Whereas y = ++x; is equivalent to writing
x = x + 1;    /*x = 11*/       (First Increment and than Assign the value)
y = x;    /*y = 11*/

int x = 10, y;
y = x- -; is equivalent to writing
y = x;     /*y = 10*/ (First Assign the value and then Decrement)
x = x - 1;    /*x = 9*/
Whereas y = - -x; is equivalent to writing
x = x - 1;    /*x = 9*/     (First Decrement  and than Assign the value)
y = x;    /*y = 9*/
```

Conditional Operator

The syntax of the conditional operator is

exp1 ? exp2 : exp3

exp1 is evaluated first. If it is true, then exp2 is evaluated and becomes the result of the expression, otherwise exp3 is evaluated and becomes the result of the expression. For example,

```
large = (a > b) ? a : b
```

The conditional operator is used to find the larger of two given numbers. First exp1, that is `a > b`, is evaluated. If `a` is greater than `b`, then large = `a`, else large = `b`. Hence, large is equal to either `a` or `b`, but not both.

Exercises: Find greatest of 2 and 3 numbers using ternary operator.

Bitwise Operators

As the name suggests, bitwise operators perform operations at the bit level. These operators include: bitwise AND, bitwise OR, bitwise XOR, and shift operators.

Bitwise AND

Like boolean AND (&&), bitwise AND operator (&) performs operation on bits instead of bytes, chars, integers, etc.

For example,

10101010 & 01010101 = 00000000

Bitwise OR

example,

10101010 | 01010101 = 11111111

Bitwise XOR

example,

10101010 ^ 01010101 = 11111111

Table Truth table of bitwise XOR

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise NOT (~)

The bitwise NOT or complement is a unary operator that performs logical negation on each bit of the operand.

example,

~10101011 = 01010100

Shift Operators

C supports two bitwise shift operators. They are shift left (<<) and shift right (>>). The syntax for a shift operation can be given as

operand op num

For example, if we have

x = 0001 1101

then x << 1 produces 0011 1010

example, if we have x = 0001 1101, then

x >> 1 gives result = 0000 1110

Assignment Operators

In C language, the assignment operator is responsible for assigning values to the variables. While the equal sign (=) is the fundamental assignment operator, C also supports other assignment operators that provide shorthand ways to represent common variable assignments.

For example,

```
int x;
x = 10;
```

assigns the value 10 to variable x. The assignment operator has right-to-left associativity, so the expression

```
a = b = c = 10;
```

is evaluated as

```
(a = (b = (c = 10)));
```

Assignment Operators

Table Assignment operators

Operator	Example	Operator	Example

/=	float a=9.0;
	float b=3.0;
	a /= b;
		&=	int a = 10;
			int b = 20;
			a &= b;

\=	int a= 9;
	int b = 3;
	a \= b;
		^=	int a = 10;
			int b = 20;
			a ^= b;

*=	int a= 9;
	int b = 3;
	a *= b;
		<<=	int a= 9;
			int b = 3;
			a <<= b;

+=	int a= 9;
	int b = 3;
	a += b;
		>>=	int a= 9;
			int b = 3;
			a >>= b;

-=	int a= 9;
	int b = 3;
	a -= b;		

Comma Operator

The comma operator, which is also called the sequential-evaluation operator, takes two operands. It works by evaluating the first expression and discarding its value, and then evaluates the second expression and returns the value as the result of the expression.

For example, the following statement first increments *a*, then increments *b*, and then assigns the value of *b* to *x*.

```
int a=2, b=3, x=0;
x = (++a, b+=a);
```

Now, the value of *x* = 6.

sizeof Operator

sizeof is a unary operator used to calculate the size of data types. This operator can be applied to all data types. When using this operator, the keyword sizeof is followed by a type name, variable, or expression. The operator returns the size of the data type, variable, or expression in bytes. That is, the sizeof operator is used to determine the amount of memory space that the data type/variable/expression will take.

For example, sizeof(char) returns 1, that is the size of a character data type.

If we have,

```
int a = 10;
unsigned int result;
result = sizeof(a);
```

then result = 2, that is, space required to store the variable *a* in memory. Since a is an integer, it requires 2 bytes of storage space.

Operator precedence Chart

Table lists the operators that C language supports in the order of their precedence (highest to lowest). The associativity indicates the order in which the operators of equal precedence in an expression are evaluated.

Table Operators precedence chart

Operator	Associativity
() [] . ->	left-to-right
++(postfix) --(postfix)	right-to-left
++(prefix) --(prefix) +(unary) - (unary) ! ~ (type) *(indirection) &(address) sizeof	right-to-left
* / %	left-to-right
+ -	left-to-right
<< >>	left-to-right

Operator	Associativity
< <= > >=	left-to-right
== !=	left-to-right
&	left-to-right
^	left-to-right
|	left-to-right
&&	left-to-right
||	left-to-right
?:	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	right-to-left
,(comma)	left-to-right

Examples

If we have the following variable declarations:

int a = 0, b = 1, c = –1;

float x = 2.5, y = 0.0;

then,

(a) a && b = 0

(b) a < b && c < b = 1

(c) b + c || ! a

= (b + c) || (!a)

= 0 ||1

= 1

*(d) x * 5 && 5 || (b / c)*

*= ((x * 5) && 5) || (b / c)*

= (12.5 && 5) || (1/–1)

= 1

(e) a <= 10 && x >= 1 && b

= ((a <= 10) && (x >= 1)) && b

= (1 && 1) && 1

= 1

(f) !x || !c || b + c

= ((!x) || (!c)) || (b + c)

= (0 || 0) || 0

= 0

*(g) x * y < a + b || c*

*= ((x * y) < (a + b)) || c*

= (0 < 1) || –1

= 1

(h) (x > y) + !a || c++

= ((x > y) + (!a)) || (c++)

= (1 + 1) || 0

= 1

Practice Expressions

int a = 1, b = 0, c = 7;

Expression	Numeric Value	True/False
a		
b		
c		
a + b		
a && b		
a || b		
!c		
!!c		
a && !b		
a < b && b < c		
a > b && b < c		
a >= b || b > c		

BASIC DATA TYPES

Data type determines the set of values that a data item can take and the operations that can be performed on the item. C language provides four basic data types. Table 1.2 lists the data types, their size, range, and usage for a C programmer.

Table Basic data types in C

Data Type	Size in Bytes	Range	Use
char	1	-128 to 127	To store characters
int	2	-32768 to 32767	To store integer numbers
float	4	3.4E-38 to 3.4E+38	To store floating point numbers
double	8	1.7E-308 to 1.7E+308	To store big floating point numbers

BASIC DATA TYPES

- In addition, C also supports four modifiers—two sign specifiers (signed and unsigned) and two size specifiers (short and long). Table 1.3 shows the variants of basic data types.
- Using modifirs we can extend or limit the range of data type

Table Basic data types and their variants

Data Type	Size in Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

I/O statements in C

- The most fundamental operation in a C program is to accept input values from a standard input device and output the data produced by the program to a standard output device.
- Input and Output statement are used to read and write the data in C programming. These are embedded in *stdio.h* (standard Input/Output header file).
- Input means to provide the program with some data to be used in the program and Output means to display data on screen or write the data to a printer or a file.C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.
- There are mainly two of Input/Output functions are used for this purpose. These are discussed as:
- Formatted I/O functions
- Unformatted I/O functions

Formatted I/O functions

- If we want to assign value to variable that is inputted from the user at run-time? this is done by using the ***scanf*** function that reads data from the keyboard. Similarly, for outputting results of the program, ***printf*** function is used that sends results to a terminal.
- A program that uses standard input/output functions must contain the following statement at the beginning of the program:
  #***include*** <***stdio.h***>

scanf()

The *scanf()* function is used to read formatted data from the keyboard. The syntax of the *scanf()* function can be given as,

scanf ("control string", arg1, arg2, arg3...argn);

The prototype of the control string can be given as,

%[*][width][modifier]type

* is an optional argument that suppresses assignment of the input field.

width is an optional argument that specifies the maximum number of characters to be read.

modifier is an optional argument (**h, l,** or **L**) , which modifies the type specifier.

type specifies the type of data that has to be read.

- The *scanf* function ignores any blank spaces, tabs, and newlines entered by the user.

the following code that shows how we can input value in a variable of ***int*** data type:

```
int num;
scanf(" %4d ", &num);
```

The ***scanf*** function reads first four digits into the address or the memory location pointed by num.

Table Type specifiers

Type	**Qualifying Input**
%c	For single characters
%d, %i	For integer values
%e,%E,%f,%g,%G	For floating point numbers
%o	For octal numbers
%s	For a sequence of (string of) characters
%u	For unsigned integer values
%x,%X	For hexadecimal values

printf()

The *printf* function is used to display information required by the user and also prints the values of the variables. Its syntax can be given as:

printf ("control string", arg1,arg2,arg3,...,argn);

The prototype of the control string can
be given as below:

%[flags][width][.precision][modifier]type

Each control string must begin with a % sign.

flags is an optional argument, which specifies output justification like decimal point, numerical sign, trailing zeros or octadecimal or hexadecimal prefixes.

width is an optional argument which specifies the minimum number of positions that the output characters will occupy.

precision is an optional argument which specifies the number of digits to print after the decimal point or the number of characters to print from a string.

modifier field is same as given for *scanf()* function.

type is used to define the type and the interpretation of the value of the corresponding argument.

The most simple *printf* statement is

printf ("Welcome to the world of C language");

For float x = 8900.768, the following examples show output under different format specifications:

printf("%f", x);

8	9	0	0	.	7	6	8

printf("%10f", x);

		8	9	0	0	.	7	6	8

printf("%9.2f", x);

		8	9	0	0	.	7	7

Table Flags in *printf()*

Flags	Description
-	Left-justify within the given field width
+	Displays the data with its numeric sign (either + or -)
#	Used to provide additional specifiers like o, x, X, 0, 0x, or 0X for octal and hexadecimal values respectively for values different than zero
0	The number is left-padded with zeroes (0) instead of spaces

Unformatted I/O functions

There are mainly six unformatted I/O functions discussed as follows:

- getchar()
- putchar()
- gets()
- puts()
- getch()
- getche()
- getchar()

getchar()

This function is an Input function. It is used for reading a single character from the keyboard. It is a buffered function. Buffered functions get the input from the keyboard and store it in the memory buffer temporally until you press the Enter key.

The general syntax is as:

v = getchar();

where v is the variable of character type. For example:

A simple C-program to read a single character from the keyboard is as:

```
/*To read a single character from the keyboard using the getchar() function
#include <stdio.h>
main()
{
char n;
n = getchar();
}
```

putchar()

This function is an output function. It is used to display a single character on the screen. The general syntax is as:

putchar(v);

where v is the variable of character type. For example:

A simple program is written as below, which will read a single character using getchar() function and display inputted data using putchar() function:

```
/*Program illustrate the use of getchar() and putchar() functions*/
#include <stdio.h>
main()
{
char n;
n = getchar();
putchar(n);
}
```

gets()

This function is an input function. It is used to read a string from the keyboard. It is also a buffered function. It will read a string when you type the string from the keyboard and press the Enter key from the keyboard. It will mark null character ('\0') in the memory at the end of the string when you press the enter key. The general syntax is as:

gets(v);

where v is the variable of character type. For example:

A simple C program to illustrate the use of gets() function:

```
/*Program to explain the use of gets() function*/
#include <stdio.h>
main()
{
char n[20];
gets(n);
}
```

puts()

This is an output function. It is used to display a string inputted by gets() function. It is also used to display a text (message) on the screen for program simplicity. This function appends a newline (“\n”) character to the output.

The general syntax is as:

puts(v);

or

puts("text line");

where v is the variable of character type.

Cont.

A simple C program to illustrate the use of puts() function:

```
/*Program to illustrate the concept of puts() with gets() functions*/
#include <stdio.h>
main()
{
char name[20];
puts("Enter the Name");
gets(name);
puts("Name is :");
puts(name);
}
```

The Output is as follows:

```
Enter the Name
Geek
Name is:
Geek
```

getch()

This is also an input function. This is used to read a single character from the keyboard like getchar() function. But getchar() function is a buffered is function, getchar() function is a non-buffered function. The character data read by this function is directly assigned to a variable rather it goes to the memory buffer, the character data is directly assigned to a variable without the need to press the Enter key.

Another use of this function is to maintain the output on the screen till you have not press the Enter Key. The general syntax is as:

v = getch();

where v is the variable of character type.

Cont.

A simple C program to illustrate the use of getch() function:

```
/*Program to explain the use of getch() function*/
#include <stdio.h>
main()
{
char n;
puts("Enter the Char");
n = getch();
puts("Char is :");
putchar(n);
getch();
}
```

The output is as follows:

```
Enter the Char
Char is L
```

getche()

All are same as getch() function except it is an echoed function. It means when you type the character data from the keyboard it will visible on the screen. The general syntax is as:

v = getche();

where v is the variable of character type.

Cont.

A simple C program to illustrate the use of getch() function:

```
/*Program to explain the use of getch() function*/
#include <stdio.h>
main()
{
char n;
puts("Enter the Char");
n = getche();
puts("Char is :");
putchar(n);
getche();
}
```

The output is as follows:

```
Enter the Char L
Char is L
```

TYPE CONVERSION AND TYPECASTING

Type conversion or typecasting of variables refers to changing a variable of one data type into another. While type conversion is done implicitly, casting has to be done explicitly by the programmer.

Type Conversion

Type conversion is done when the expression has variables of different data types. So to evaluate the expression, the data type is promoted from lower to higher level where the hierarchy of data types can be given as: ***double, float, long, int, short, and char***. For example, type conversion is automatically done when we assign an integer value to a floating point variable. Consider the following code:

```
float x;
int y = 3;
x = y;
```

Now, x = 3.0, as integer value is automatically converted into its equivalent

Typecasting

Typecasting is also known as forced conversion. It is done when the value of one data type has to be converted into the value of another data type. The code to perform typecasting can be given as:

```
float salary = 10000.00;
int sal;
sal = (int) salary;
```

When floating point numbers are converted to integers, the digits after the decimal are truncated. Therefore, data is lost when floating point representations are converted to integral representations.

As we can see in the code, typecasting can be done by placing the destination data type in parentheses followed by the variable name that has to be converted. Hence, we conclude that typecasting is done to make a variable of one data type to act like a variable of another type.

Write a program to calculate the area of a circle.

```
#include <stdio.h>
#include <conio.h>
int main()
{
float radius;
double area;
clrscr();
printf("\n Enter the radius of the circle : ");
scanf("%f", &radius);
area = 3.14 * radius * radius;
printf(" \n Area = %.2lf", area);
return 0;
}
```

Output

```
Enter the radius of the circle : 7
Area = 153.86
```

Write a program to convert an integer into the corresponding floating point number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
float f_num;
int i_num;
clrscr();
printf("\n Enter any integer: ");
scanf("%d", &i_num);
f_num = (float)i_num;
printf("\n The floating point variant of %d is = %f", i_num, f_num);
return 0;
}
```

Output

Enter any integer: 56

The floating point variant of 56 is = 56.000000

Scope of Variables

A scope is a region of the program and broadly speaking there are three places, where variables can be declared:

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

Here let us explain what local and global variables are.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

Example : local Variable

```
#include <stdio.h>
int main ()
{
  // Local variable declaration:
  int a, b;
  int c;   // actual initialization
  a = 10;
  b = 20;
  c = a + b;
Printf(“%d”,c);
  return 0;
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their type throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

Example for global variable

```
#include <stdio.h>
// Global variable declaration:
int g;
 int main ()
{
  // Local variable declaration:
  int a, b;
  // actual initialization
  a = 10;
  b = 20;
  g = a + b;
  printf(”%d”,g);
  return 0;
}
```

```
#include <stdio.h>
// Global variable declaration:
int g = 20;

int main ()
{
  // Local variable declaration:
  int g = 10;
Printf("%d:,g);
   return 0;
}
```

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

```
10
```

END of UNIT-II