

Programming in C

Pradip Dey & Manas Ghosh

CHAPTER 2

Basics of C

OBJECTIVE

- Understand the basic structure of a program in C .
- Learn the commands used in UNIX/LINUX and MS-DOS for compiling and running a program in C.
- Obtain a preliminary idea of the keywords in C.
- Learn the data types, variables, constants, operators, and expressions in C.
- Understand and grasp the precedence and associativity rules of operators in C.
- Get acquainted with the rules of type conversions in C.

INTRODUCTION

- Ken Thompson at Bell Labs, USA, wrote his own variant over Martin Richards's Basic Combined Programming Language and called it B .
- Dennis Ritchie, at Bell Labs, is credited for designing C in the early 1970s.
- Today C is a high-level language which also provides the capabilities that enable the programmers to 'get in close' with the hardware and allows them to interact with the computer on a much lower level.



Year

1960

ALGOL



1967

BCPL



1970

B



1973

Traditional C



1989

C or C89



1990

ISO C

Developed by

International Group

Martin Richard

Ken Thompson

Dennis Ritchie

ANSI Committee

ISO Committee

KEY WORDS

- **ASCII** : It is a standard code for representing characters as numbers that is used on most microcomputers, computer terminals, and printers. In addition to printable characters, the ASCII code includes control characters to indicate carriage return, backspace, etc.
- **Assembler** :The assembler creates the object code.
- **Associativity** :The associativity of operators determines the order in which operators of equal precedence are evaluated when they occur in the same expression. Most operators have a left-to-right associativity, but some have right-to-left associativity.

KEY WORDS

- **Compiler:** A system software that translates the source code to assembly code.
- **Constant :**A constant is an entity that doesn't change.
- **Data type:** The type, or data type, of a variable determines a set of values that the variable might take and a set of operations that can be applied to those values.
- **Debugger:** A debugger is a program that enables you to run another program step-by-step and examine the value of that program's variables.
- **Identifier:** An identifier is a symbolic name used in a program and defined by the programmer.

KEY WORDS

- **IDE**: An Integrated Development Environment or IDE is an editor which offers a complete environment for writing, developing, modifying, deploying, testing, and debugging the programs.
- **Identifier**: An identifier or name is a sequence of characters invented by the programmer to identify or name a specific object.
- **Keyword**: Keywords are explicitly reserved words that have a strict meaning as individual tokens to the compiler. They cannot be redefined or used in other contexts.

KEY WORDS

- **Linker:** If a source file references library functions or functions defined in other source files, the linker combines these functions to create an executable file.
- **Precedence :** The precedence of operators determines the order in which different operators are evaluated when they occur in the same expression. Operators of higher precedence are applied before operators of lower precedence.
- **Pre processor :** The C pre processor is used to modify the source program before compilation according to the pre processor directives specified.

KEY WORDS

- **Lvalue:** An lvalue is an expression to which a value can be assigned.
- **Rvalue :**An rvalue can be defined as an expression that can be assigned to an lvalue.
- **Token:** A token is one or more symbols understood by the compiler that help it interpret your code.
- **Word :**A word is the natural unit of memory for a given computer design. The word size is the computer's preferred size for moving units of information around; technically it's the width of the processor's registers.

KEY WORDS

- **Whitespace** Space, newline, tab character and comment are collectively known as whitespace.
- **Variable:** A variable is a named memory location. Every variable has a type, which defines the possible values that the variable can take, and an identifier, which is the name by which the variable is referred.
- **Bug:** Any type of error in a program is known as bug. There are three types of errors that may occur:
 - Compile errors,
 - Linking errors,
 - Runtime errors

WHY LEARN C?

- There are a large number of programming languages in the world today even so, there are several reasons to learn C, some of which are stated as follows:

a . C is quick.

b . C is a core language : In computing, C is a general purpose, cross-platform, block structured procedural, imperative computer programming language.

c . C is a small language: C has only thirty-two keywords. This makes it relatively easy to learn compared to bulkier languages.

d . C is portable.

DEVELOPING PROGRAMS IN C

- There are mainly three steps:

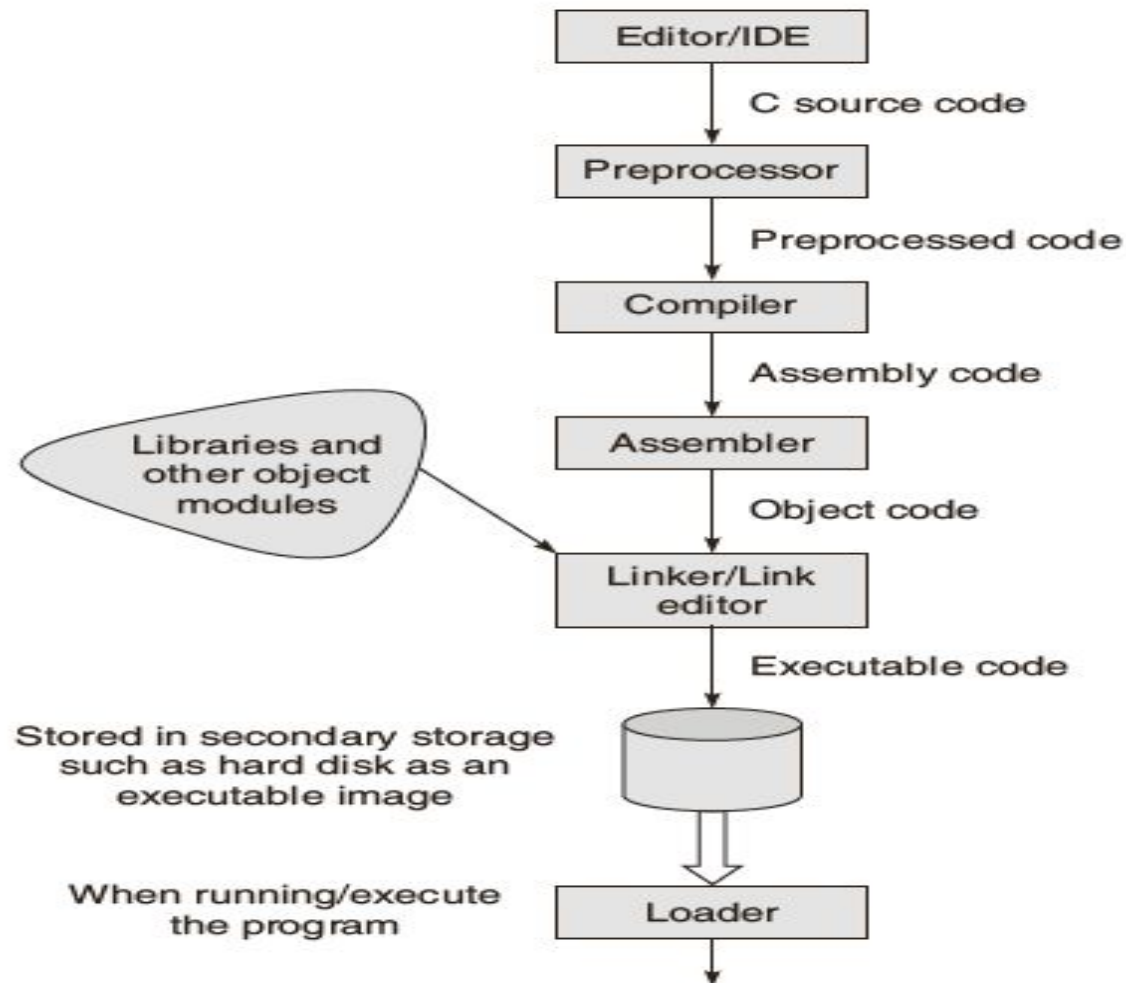
1. Writing the C program

2. Compiling the program and

3. Executing it.

- For these steps, some software components are required, namely an operating system, a text editor(*integrated development environment*), the C compiler, assembler, and linker.
- C uses a semicolon as a statement terminator; the semicolon is required as a signal to the compiler to indicate that a statement is complete.
- All program instructions, which are also called statements, have to be written in lower case characters.

DEVELOPING PROGRAMS IN C



PREPROCESSING

- Preprocessing is the first phase of the C compilation.
- It processes include-files, conditional compilation instructions and macros.
- The C preprocessor is used to modify your program according to the preprocessor directives in your source code.
- A preprocessor directive is a statement (such as `#define`) that gives the preprocessor specific instructions on how to modify your source code.
- The preprocessor is invoked as the first part of your compiler program's compilation step.
- It is usually hidden from the programmer because it is run automatically by the compiler.

COMPILEATION

- Compilation is the second pass.
- It takes the output of the preprocessor, and the source code, and generates assembler source code.
- The compiler examines each program statement contained in the source program and checks it to ensure that it conforms to the syntax and semantics of the language.
- If any mistakes are discovered by the compiler during this phase, they are reported to the user.
- The errors then have to be corrected in the source program (with the use of an editor), and the program has to be recompiled.

ASSEMBLER

- Assembly is the third stage of compilation.
- It takes the assembly source code and produces an assembly listing with offsets.
- The assembler output is stored in an object file.
- After the program has been translated into an equivalent assembly language program, the next step in the compilation process is to translate the assembly language statements into actual machine instructions.
- On most systems, the assembler is executed automatically as part of the compilation process. The assembler takes each assembly language statement and converts it into a binary format known as object code, which is then written into another file on the system.
- This file typically has the same name as the source file under UNIX, with the last letter an 'o' (for object) instead of a 'c'. Under Windows, the suffix letters "obj" typically replace the "c" in the filename.

LINKER

- Linking is the final stage of compilation.
- After the program has been translated into object code, it is ready to be linked.
- The purpose of the linking phase is to get the program into a final form for execution on the computer.
- The functions are the part of the standard C library, provided by every C compiler. The program may use other source programs that were previously processed by the compiler.
- These functions are stored as separate object files which must be linked to our object file.
- Linker handles this linking.

- The process of compiling and linking a program is often called building.
- The final linked file, which is in an executable object code format, is stored in another file on the system, ready to be run or executed.
- Under UNIX, this file is called a.out by default.
- Under Windows, the executable file usually has the same name as the source file, with the .c extension replaced by an exe extension.

ERROR TYPES

- . There are three types of errors that may occur:
- **Compile errors**
 - These are given by the compiler and prevent the program from not running.
- **Linking errors**
 - These are given by the linker or at runtime and ends the program. The linker can also detect and report errors, for example, if part of the program is missing or a non-existent library component is referenced.
- **Runtime errors**
 - These are given by the operating system.

COMPILATION : BORLAND

- 1. Open MS-DOS prompt.
- 2. At the prompt `c:\windows>` give the following command:
`c:\windows> cd c:\borland\bcc55\bin` Press **<Enter>**.
This changes the directory to `c:\borland\bcc55\bin` and the following prompt appears: `c:\borland\bcc55\bin>`
Now, enter `bcc32`
`-If:\borland\bcc55\include-Lf:\borland\bcc55\Lib c:\cprg\first.c`
- 3. Press **<Enter>**.

COMPIRATION : LINUX

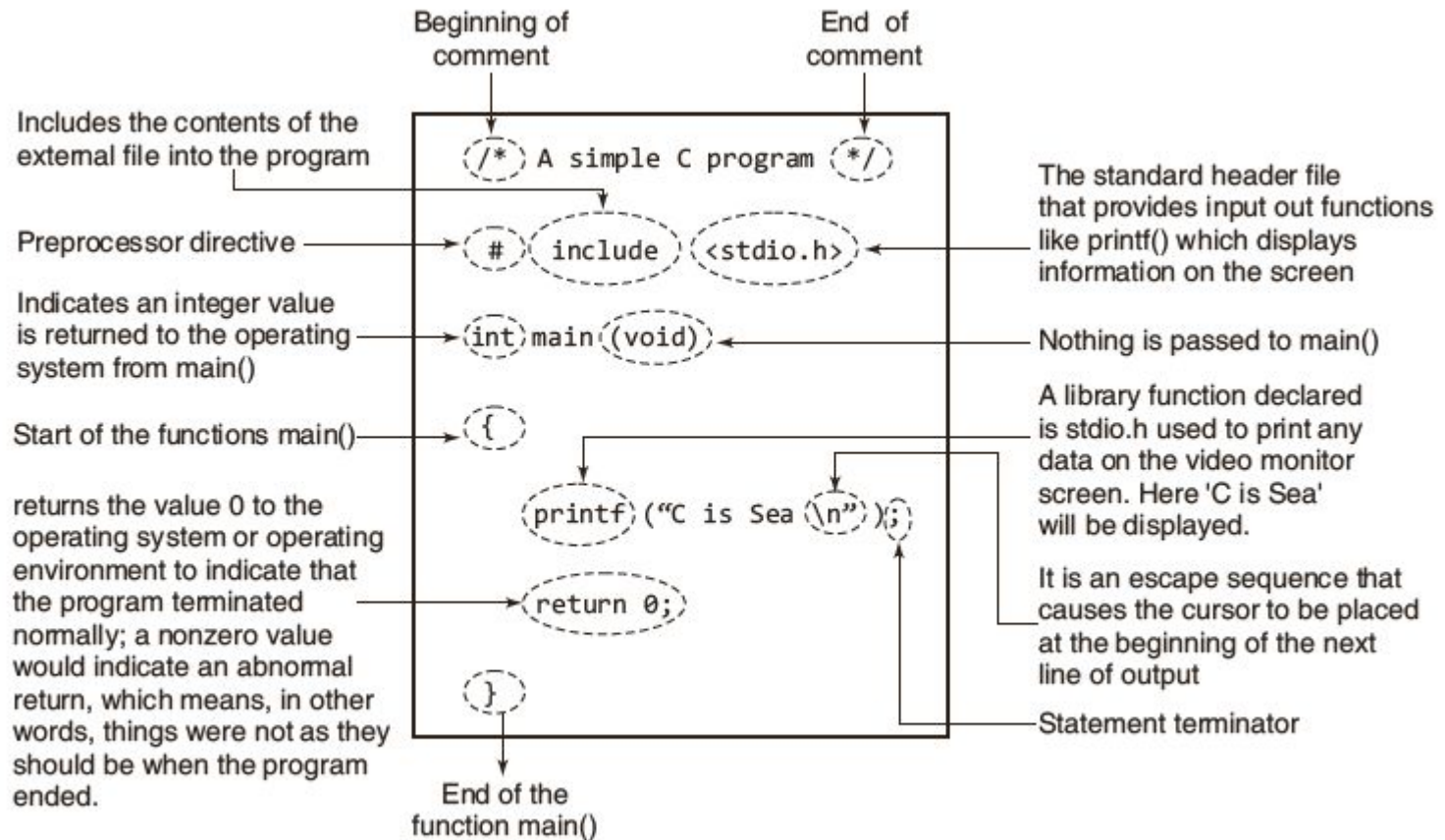
- In the LINUX operating system, a C source program, where first.c is the name of the file, may be compiled by the command

gcc first.c

- When the compiler has successfully translated the program, the compiled version, or the executable program code is stored in a file called a.out
- if the compiler option -o is used, the executable program code is put in the file listed after the -o option specified in the compilation command. It is more convenient to use -o and file name in the compilation as shown.

gcc -o program first.c

ILLUSTRATED VIRSION OF A PROGRAM



BACKSLASH CODE

Code	Meaning
\a	Ring terminal bell (a is for alert) [ANSI] extension]
\?	Question mark [ANSI extension]
\b	Backspace
\r	Carriage return
\f	Form feed
\t	Horizontal tab
\v	Vertical tab
\0	ASCII null character
\\	Backslash
\"	Double quote
\'	Single quote
\n	New line
\o	Octal constant
\x	Hexadecimal constant

PARTS OF C PROGRAMS

Header File

- The header files, usually incorporate data types, function declarations and macros, resolves this issue. The file with .h extension is called header file, because it's usually included at the head of a program.
- Every C compiler that conforms to the international standard (ISO/IEC 9899) for the language will have a set of standard header files supplied with it.
- The header files primarily contain declarations relating to standard library functions and macros that are available with C.

STANDARD HEADER FILES

- During compilation, the compilers perform type checking to ensure that the calls to the library and other user-defined functions are correct. This form of checking helps to ensure the semantic correctness of the program.

assert.h	inttypes.h	signal.h	stdlib.h
complex.h	iso646.h	stdarg.h	string.h
ctype.h	limits.h	stdbool.h	tgmath.h
errno.h	locale.h	stddef.h	time.h
fenv.h	math.h	stdint.h	wchar.h
float.h	setjmp.h	stdio.h	wctype.h

INCLUDE

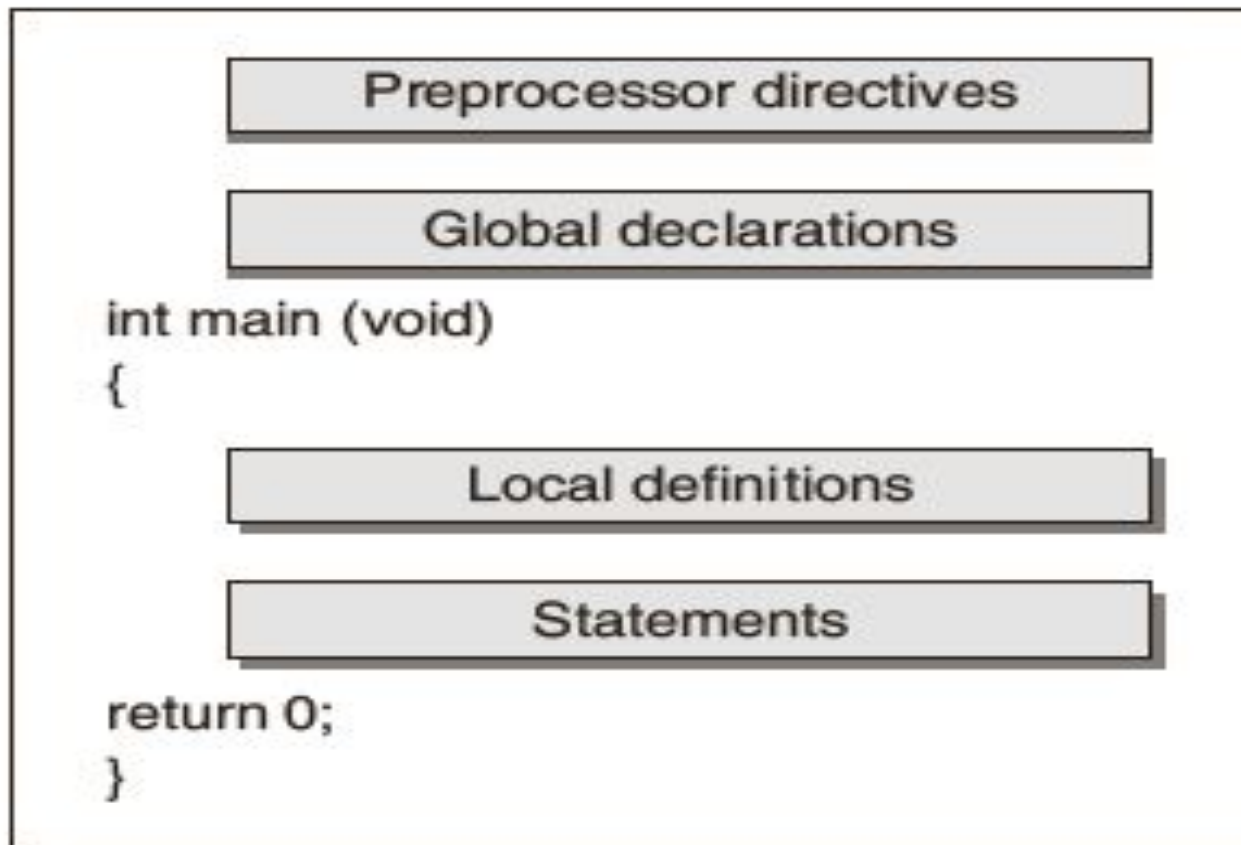
- `#include <stdio.h>`
- `#include "users.h"`

PHILOSOPHY : MAIN

- `main()` is a user defined function. `main()` is the first function in the program which gets called when the program executes. The start up code calls `main()` function. We can't change the name of the `main()` function.
- `main()` is must.
- According to ANSI/ISO/IEC 9899:1990 International Standard for C, the function called at program start up is named `main`. The implementation declares no prototype for this function. It can be defined with no parameters:

```
int main(void) { /* ... */ }
```
- or with two parameters (referred to here as `argc` and `argv`) :
- ```
int main(int argc, char *argv[]) { /* ... */ }
```

# STRUCTURE : C PROGRAM



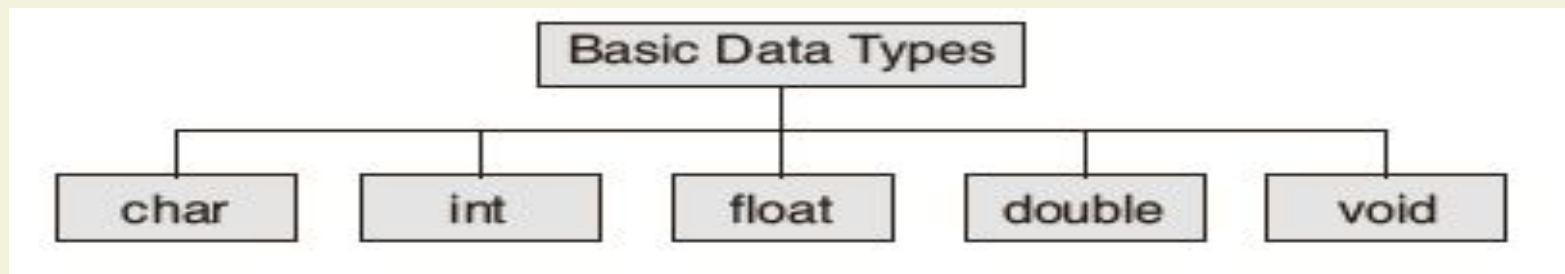
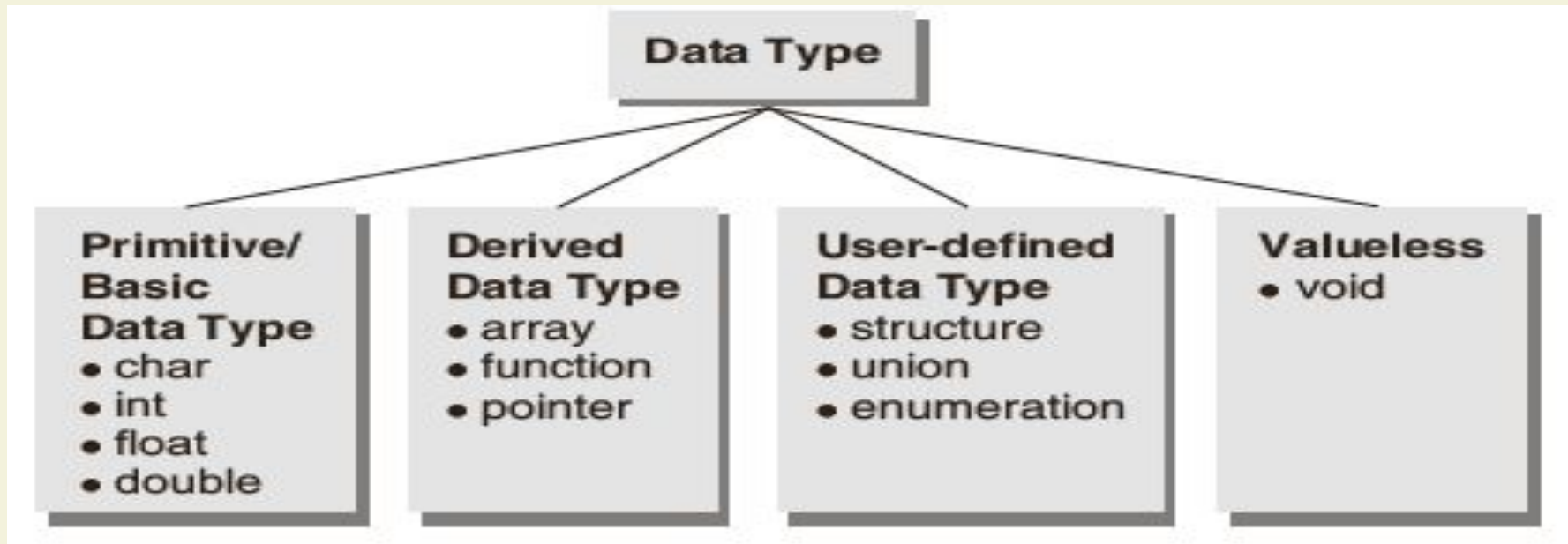
# DECLARATION & DEFINITION

- Declaration means describing the type of a data object to the compiler but not allocating any space for it.
  - *A declaration announces the properties of a data object or a function. If a variable or function is declared and then later make reference to it with data objects that do not match the types in the declaration, the compiler will complain.*
  - `data_type variable_name_1,`
- Definition means declaration of a data object and also allocating space to hold the data object.
  - *A definition, on the other hand, actually sets aside storage space (in the case of a data object) or indicates the sequence of statements to be carried out (in the case of a function).*

# VARIABLES:ATTRIBUTES

- **All variables have three important attributes:**
  - A **data type** that is established when the variable is defined, e.g., integer, real, character. Once defined, the type of a C variable cannot be changed.
  - A **name** of the variable.
  - A **value** that can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values, e.g., 2, 100, -12.

# CLASSIFICATION :DATA TYPE





# BASIC DATA TYPES:SIZE & RANGE

## ■ 16 bit computer:

| Data type | Size (in bits) | Range                                                   |
|-----------|----------------|---------------------------------------------------------|
| char      | 8              | -128 to 127                                             |
| int       | 16             | -32768 to 32767                                         |
| float     | 32             | $1.17549 \times 10^{-38}$ to $3.40282 \times 10^{38}$   |
| double    | 64             | $2.22507 \times 10^{-308}$ to $1.79769 \times 10^{308}$ |
| void      | 8              | valueless                                               |

## ■ 32 bit computer:

| Data type | Size (in bits) | Range                                                   |
|-----------|----------------|---------------------------------------------------------|
| char      | 8              | -128 to 127                                             |
| int       | 32             | -2147483648 to 2147483647                               |
| float     | 32             | $1.17549 \times 10^{-38}$ to $3.40282 \times 10^{38}$   |
| double    | 64             | $2.22507 \times 10^{-308}$ to $1.79769 \times 10^{308}$ |
| void      | 8              | valueless                                               |

# SPECIFIER OR MODIFIERS

- In addition, C has four type specifiers or modifiers and three type qualifiers.
  - Each of these type modifiers can be applied to the base type int.
  - The modifiers signed and unsigned can also be applied to the base type char.
  - In addition, long can be applied to double.
  - When the base type is omitted from a declaration, int is assumed.
  - The type void does not have these modifiers.

# SPECIFIERS : DATA TYPES

- The specifiers and qualifiers for the data types can be broadly classified into three types:
  - Size specifiers— short and long
  - Sign specifiers— signed and unsigned
  - Type qualifiers— const, volatile and restrict

|           | 16-bit Machine | 32-bit Machine | 64-bit Machine |
|-----------|----------------|----------------|----------------|
| short int | 2              | 2              | 2              |
| int       | 2              | 4              | 4              |
| long int  | 4              | 4              | 8              |

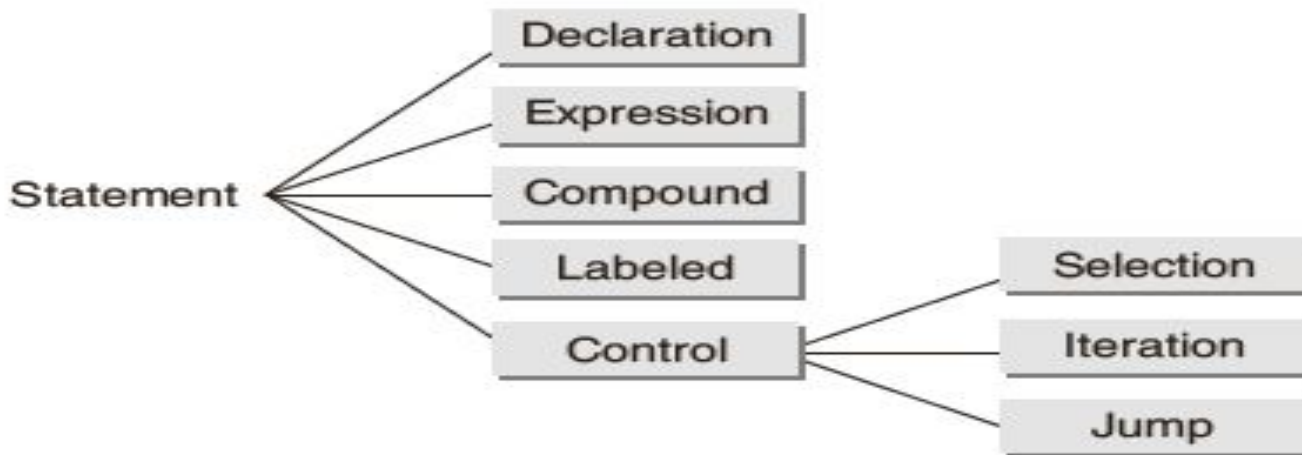
|                                                    | Size (in bytes) | Range                                                               |
|----------------------------------------------------|-----------------|---------------------------------------------------------------------|
| long long int                                      | 8               | 9, 223, 372, 036, 854, 775, 808 to +9, 223, 372, 036, 854, 775, 807 |
| unsigned long int<br>or unsigned long              | 4               | 0 to 4, 294, 967, 295                                               |
| unsigned long long int<br>or<br>unsigned long long | 8               | 0 to +18, 446, 744, 073, 709, 551, 615                              |

**Table** Allowed combinations of basic data types and modifiers in C for a 16-bit computer

| Data Type          | Size (bits) | Range                     | Default Type                          |
|--------------------|-------------|---------------------------|---------------------------------------|
| char               | 8           | −128 to 127               | signed char                           |
| unsigned char      | 8           | 0 to 255                  | None                                  |
| signed char        | 8           | −128 to 127               | char                                  |
| int                | 16          | −32768 to 32767           | signed int                            |
| unsigned int       | 16          | 0 to 65535                | unsigned                              |
| signed int         | 16          | −32768 to 32767           | int                                   |
| short int          | 16          | −32768 to 32767           | short, signed short, signed short int |
| unsigned short int | 16          | 0 to 65535                | unsigned short                        |
| signed short int   | 16          | −32768 to 32767           | short, signed short, short int        |
| long int           | 32          | −2147483648 to 2147483647 | long, signed long, signed long int    |
| unsigned long int  | 32          | 0 to 4294967295           | unsigned long                         |
| signed long int    | 32          | −2147483648 to 2147483647 | long int, signed long, long           |
| float              | 32          | 3.4E−38 to 3.4E+38        | None                                  |
| double             | 64          | 1.7E−308 to 1.7E+308      | None                                  |
| long double        | 80          | 3.4E−4932 to 1.1E+4932    | None                                  |

# PROGRAM STATEMENTS

- A statement is a syntactic constructions that performs an action when a program is executed. All C program statements are terminated with a semi-colon (;).



**Figure** Different types of program statements available in C

# CONT.

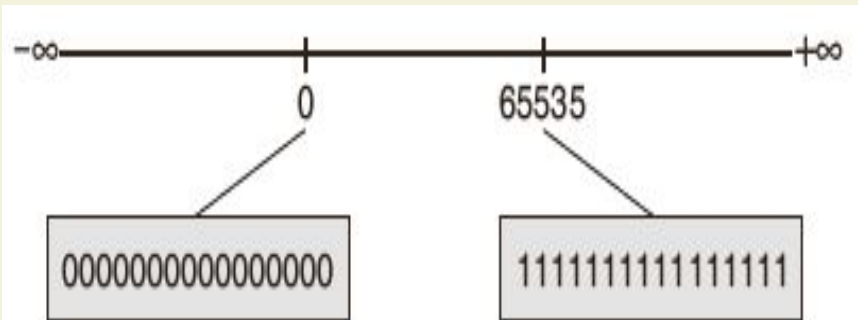
- ***Declaration*** :It is a program statement that serves to communicate to the language translator information about the name and type of the data objects needed during program execution.
- ***Expression statement***: It is the simplest kind of statement which is no more than an expression followed by a semicolon. An *expression* is a sequence of operators and operands that specifies computation of a value . Example : $x = 4$
- ***Compound statement*** is a sequence of statements that may be treated as a single statement in the construction of larger statements.
- ***Labelled statements*** can be used to mark any statement **so** that control may be transferred to the statement by *switch* statement.

# CONT.

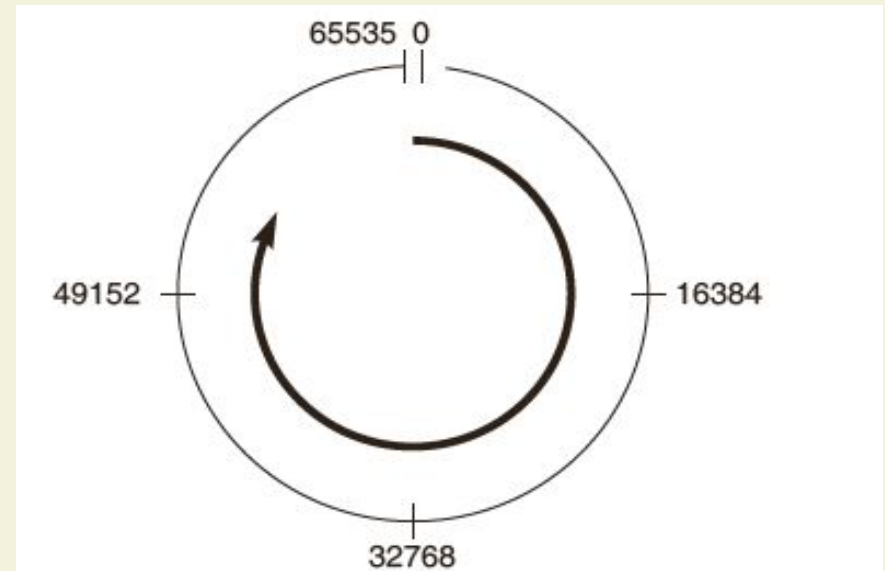
- ***Control statement is a statement whose execution results*** in a choice being made as to which of two or more paths should be followed. In other words, the control statements determine the ‘flow of control’ in a program.
  - **Selection statements allow a program to select a particular execution path from a set of one or more alternatives.** Various forms of the if..else statement belong to this category.
  - **Iteration statements are used to execute a group of one or more statements repeatedly.** “while, for, and do..while” statements falls under this group.
  - **Jump statements cause an unconditional jump to some other place in the program.** Goto statement falls in this group

# HOW THE INTEGERS ARE STORED IN MEMORY

- Storing *unsigned integers* is a straightforward process . The number is changed to the corresponding binary form & the binary representation is stored.



**Figure (a)** Range of an unsigned integer stored in a 16-bit word



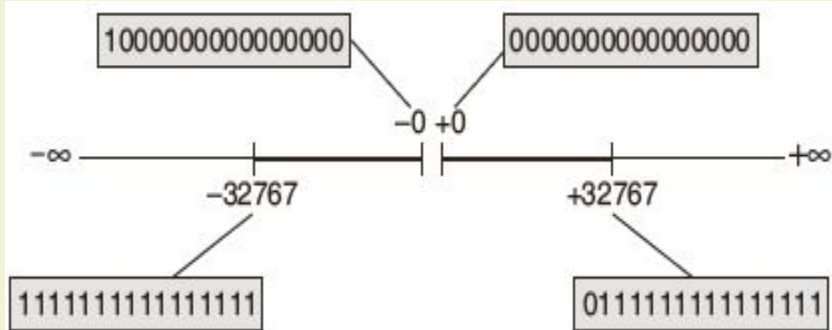
**Figure (b)** Cyclic view of the range of an unsigned integer stored in a 16-bit word



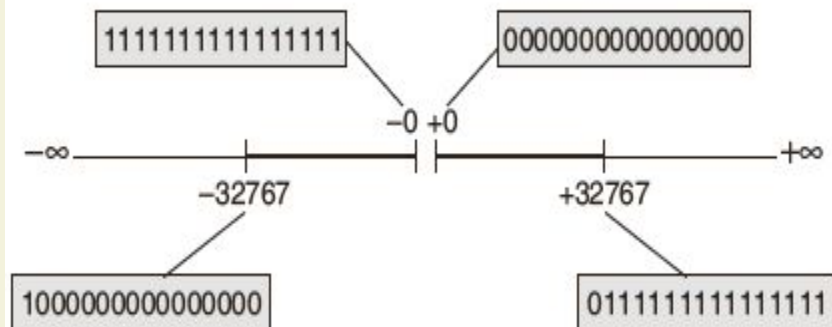
# HOW THE INTEGERS ARE STORED IN MEMORY

- For **signed integer** types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; there shall be exactly one sign bit (if there are  $M$  value bits in the signed type and  $N$  in the unsigned type, then  $M \leq N$ ). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:
  - the corresponding value with sign bit 0 is negated(sign and magnitude);
  - the sign bit has the value  $-(2^N)$  (2's complement);
  - the sign bit has the value  $-(2^N - 1)$  (1's complement).

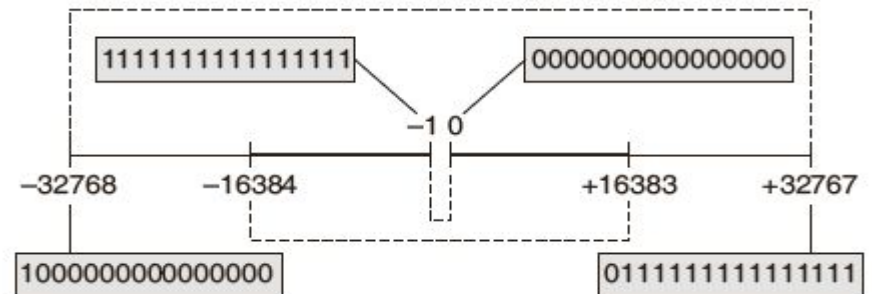
# INTEGER STORING:PROCESS



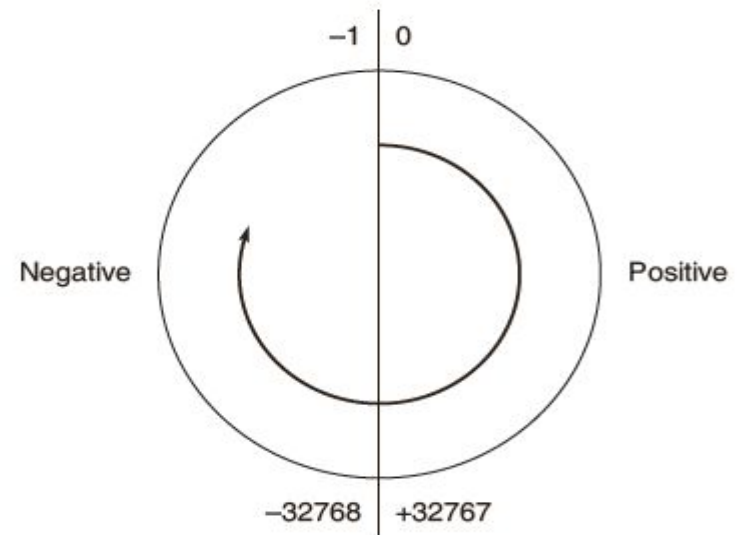
Range of a signed integer stored in a 16-bit word in sign and magnitude form



Range of a signed integer stored in 16-bit word in one's complement form



Range of a signed integer stored in 16-bit word in Two's complement form



Cyclic view of the range of a signed integer stored in a 16-bit word in 2's complement form

# KEY WORDS

- Compiler vendors (like Microsoft, Borland ,etc.) provide their own keywords apart from the ones mentioned below. These include extended keywords like **near, far, asm, etc.**

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

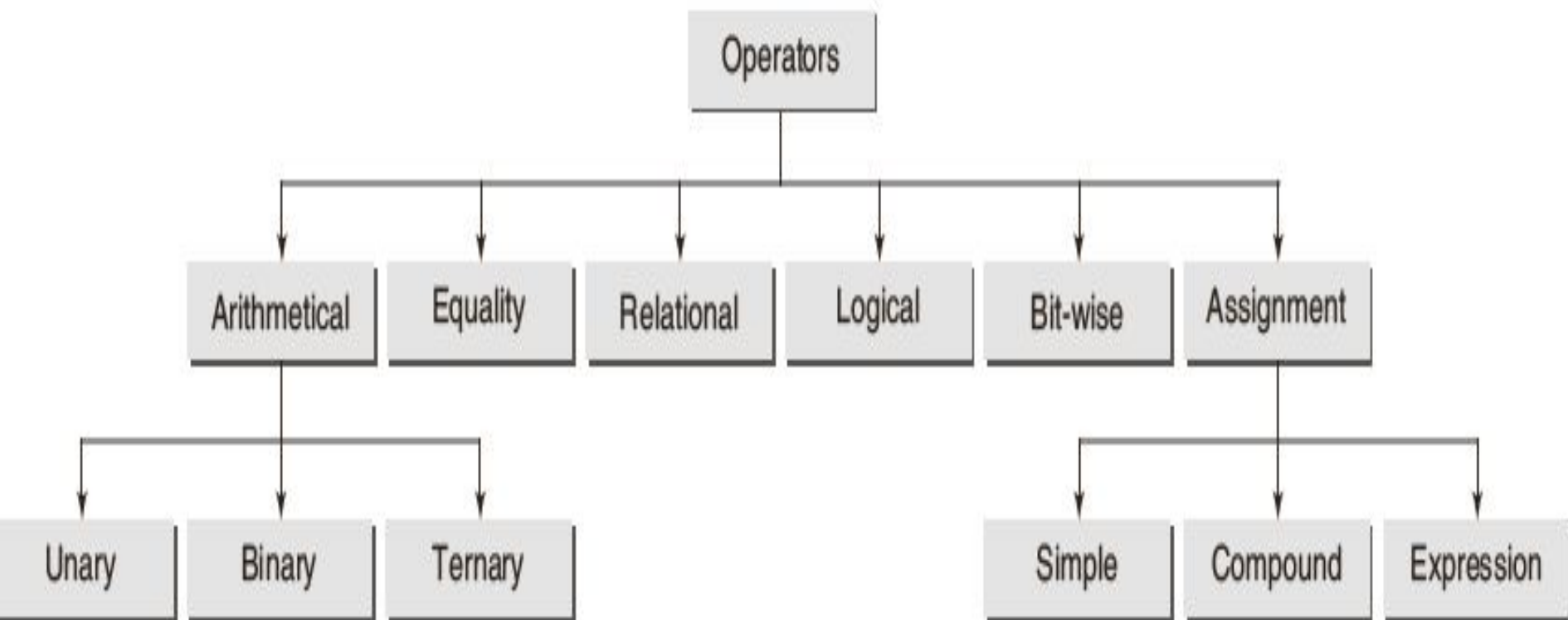
# CONSTANT

- A constant is an explicit data value written by the programmer. Thus, it is a value known to the compiler at compiling time.
- In ANSI C, a decimal integer constant is treated as an unsigned long if its magnitude exceeds that of the signed long. An octal or hexadecimal integer that exceeds the limit of int is taken to be unsigned; if it exceeds this limit, it is taken to be long; and if it exceeds this limit, it is treated as an unsigned long.
- An integer constant is regarded as unsigned if its value is followed by the letter 'u' or 'U', e.g., 0x9999u; it is regarded as unsigned long if its value is followed by 'u' or 'U' and 'l' or 'L', e.g., 0xFFFFFFFFul.

# SPECIFICATIONS OF DIFFERENT CONSTANTS

| Type              | Specification                | Example   |
|-------------------|------------------------------|-----------|
| Decimal           | nil                          | 50        |
| Hexadecimal       | Preceded by 0x or 0X         | 0x10      |
| Octal             | Begins with 0                | 010       |
| Floating constant | Ends with f/F                | 123.0f    |
| Character         | Enclosed within single quote | 'A' 'o'   |
| String            | Enclosed within double quote | "welcome" |
| Unsigned integer  | Ends with U/u                | 37 u      |
| Long              | Ends with L/l                | 37 L      |
| Unsigned long     | Ends with UL/w               | 37 UL     |

# CLASSIFICATION: OPERATORS IN C



# DIFFERENT OPERATORS

| Type of operator | Operator symbols with meanings                                                                                 |
|------------------|----------------------------------------------------------------------------------------------------------------|
| Arithmetical     | <b>Unary</b><br>+ (Unary)<br>- (Unary)<br>++ Increment<br>-- Decrement                                         |
|                  | <b>Binary</b><br>+ Addition<br>- Subtraction<br>* Multiplication<br>/ Division<br>% Modulus                    |
|                  | <b>Ternary</b><br>?: Discussed later on                                                                        |
| Assignment       | <b>Simple Assignment</b><br>=                                                                                  |
|                  | <b>Compound Assignment</b><br>+=, -=, *=, /=, %=, &=, ^=,  =                                                   |
|                  | <b>Expression Assignment</b><br>A = 5 + (b = 8 + (c = 2)) - 4                                                  |
| Relational       | >, <, >=, <=                                                                                                   |
| Equality         | == (Equal to)<br>!= (Not equal to)                                                                             |
| Logical          | && (Logical AND)<br>   (Logical OR)<br>! (Logical NOT)                                                         |
| Bitwise          | & (Bitwise AND)<br>  (Bitwise OR)<br>~ (Complement)<br>^ (Exclusive OR)<br>>> (Right Shift)<br><< (Left Shift) |
| Others           | , (Comma)<br>* (indirection),<br>. (membership operator)<br>-> (membership operator)                           |

# ARITHMETIC OPERATOR

- There are three types of arithmetic operators in C: binary, unary, and ternary.
- **Binary operators:** C provides five basic arithmetic binary operators.
  - Arithmetic binary operators:

| Operator | Name           | Example                   |
|----------|----------------|---------------------------|
| +        | Addition       | 12 + 4.9 /* gives 16.9*/  |
| -        | Subtraction    | 3.98 - 4 /* gives -0.02*/ |
| *        | Multiplication | 2 * 3.4 /* gives 6.8 */   |
| /        | Division       | 9 / 2.0 /* gives 4.5 */   |
| %        | Remainder      | 13 % 3 /* gives 1 */      |



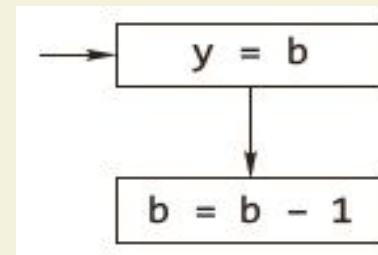
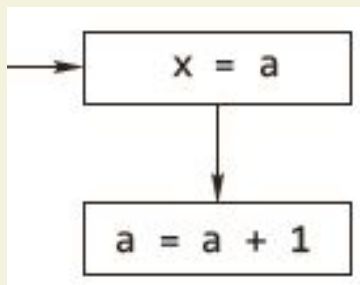
# UNARY OPERATION

- ***Unary operators:*** The unary ‘-’ operator negates the value of its operand (clearly, a signed number). A numeric constant is assumed positive unless it is preceded by the negative operator. That is, there is no unary ‘+’. It is implicit. Remember that -x does not change the value of x at the location where it permanently resides in memory.
- ***Unary increment and decrement operators ‘++’ and ‘--’*** operators increment or decrement the value in a variable by 1.
- ***Basic rules for using ++ and -- operators:***
  - The operand must be a variable but not a constant or an expression.
  - The operator ++ and -- may precede or succeed the operand.

# POSTFIX

## Postfix:

- **(a)  $x = a++$ ;**
  - First action: store value of  $a$  in memory location for variable  $x$ .
  - Second action: increment value of  $a$  by 1 and store result in memory location for variable  $a$ .
- **(b)  $y = b--$ ;**
  - First action: put value of  $b$  in memory location for variable  $y$ .
  - Second action: decrement value of  $b$  by 1 and put result in memory location for variable  $b$ .



# PREFIX

## Prefix :

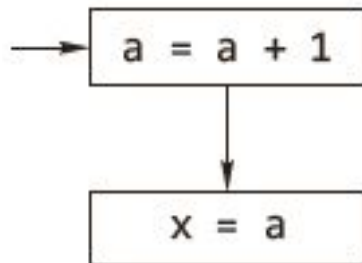
- **(a)  $x = ++a;$**

- *First action: increment value of  $a$  by 1 and store result in memory location for variable  $a$ .*
- *Second action: store value of  $a$  in memory location for variable  $x$ .*

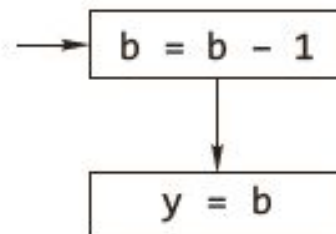
- **(b)  $y = --b;$**

- *First action: decrement value of  $b$  by 1 and put result in memory location for variable  $b$ .*
- *Second action: put value of  $b$  in memory location for variable  $y$ .*

**(a)  $x = ++a;$**



**(b)  $y = --b;$**



# RELATIONAL OPERATORS

- C provides six relational operators for comparing numeric quantities. Relational operators evaluate to 1, representing the *true outcome*, or 0, representing the *false outcome*.

| Operator | Action                | Example                |
|----------|-----------------------|------------------------|
| ==       | Equal                 | 5 == 5 /* gives 1 */   |
| !=       | Not equal             | 5 != 5 /* gives 0 */   |
| <        | Less than             | 5 < 5.5 /* gives 1 */  |
| <=       | Less than or equal    | 5 <= 5 /* gives 1 */   |
| >        | Greater than          | 5 > 5.5 /* gives 0 */  |
| >=       | Greater than or equal | 6.3 >= 5 /* gives 1 */ |

# LOGICAL OPERATORS

- C provides three logical operators for forming logical expressions. Like the relational operators, logical operators evaluate to 1 or 0.
  - Logical negation is a unary operator that negates the logical value of its single operand. If its operand is non-zero, it produces 0, and if it is 0, it produces 1.
  - Logical AND produces 0 if one or both its operands evaluate to 0. Otherwise, it produces 1.
  - Logical OR produces 0 if both its operands evaluate to 0. Otherwise, it produces 1.

| Operator | Action           | Example        | Result |
|----------|------------------|----------------|--------|
| !        | Logical Negation | !(5 == 5)      | 0      |
| &&       | Logical AND      | 5 < 6 && 6 < 6 | 0      |
|          | Logical OR       | 5 < 6    6 < 5 | 1      |

# BIT WISE OPERATORS

- C provides six bitwise operators for manipulating the individual bits in an integer quantity . Bitwise operators expect their operands to be integer quantities and treat them as bit sequences.
  - Bitwise *negation* is a unary operator that complements the bits in its operands.
  - Bitwise AND compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise.
  - Bitwise OR compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1 otherwise.
  - Bitwise *exclusive or* compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.

# BIT WISE OPERATORS

| Operator | Action               | Example                               |
|----------|----------------------|---------------------------------------|
| ~        | Bitwise Negation     | ~'\011'<br>/* gives '\066' */         |
| &        | Bitwise AND          | '\011' & '\027'<br>/* gives '\001' */ |
|          | Bitwise OR           | '\011'   '\027'<br>/* gives '\037' */ |
| ^        | Bitwise Exclusive OR | '\011' ^ '\027'<br>/* gives '\036' */ |
| <<       | Bitwise Left Shift   | '\011' << 2<br>/* gives '\044' */     |
| >>       | Bitwise Right Shift  | '\011' >> 2<br>/* gives '\002' */     |

How the bits are calculated

| Example | Octal value | Bit sequence |   |   |   |   |   |   |   |
|---------|-------------|--------------|---|---|---|---|---|---|---|
| x       | 011         | 0            | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| y       | 027         | 0            | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| ~x      | 366         | 1            | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| x & y   | 001         | 0            | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| x   y   | 037         | 0            | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| x ^ y   | 036         | 0            | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| x << 2  | 044         | 0            | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| x >> 2  | 002         | 0            | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

# CONDITIONAL OPERATOR

- The conditional operator has three expressions.
  - It has the general form **expression1 ? expression2 : expression3**
  - First, expression1 is evaluated; it is treated as a logical condition.
  - If the result is non-zero, then expression2 is evaluated and its value is the final result. Otherwise, expression3 is evaluated and its value is the final result.
- For example, `int m = 1, n = 2, min;`  
`min = (m < n ? m : n); /* min is assigned a value 1 */`
- In the above example, because m is less than n, `m < n` expression evaluates to be true, therefore, min is assigned the value m, i.e., 1.



# COMMA OPERATOR

- This operator allows the evaluation of multiple expressions, separated by the comma, from left to right in order and the evaluated value of the rightmost expression is accepted as the final result. The general form of an expression using a comma operator is
- Expression M = (expression1, expression2, ..., expression N);
- where the expressions are evaluated strictly from left to right and their values discarded, except for the last one, whose type and value determine the result of the overall expression.

# Tokens

Say we have the following piece of code,

```
if (x<5)
 x = x + 2;
else
 x = x + 10;
```

Here the tokens that will be generated are

|              |           |
|--------------|-----------|
| Keywords :   | if , else |
| Identifier : | x         |
| Constants :  | 2, 10, 5  |
| Operators :  | +, =      |
| Separator :  | ;         |

# sizeof OPERATOR

- C provides a useful operator, sizeof, for calculating the size of any data item or type. It takes a single operand that may be a type name (e.g., int) or an expression (e.g., 100) and returns the size of the specified entity in bytes. The outcome is totally machine-dependent.

## □ For example:

```
#include <stdio.h>

int main()
{
 printf("char size = %d bytes\n", sizeof(char));
 printf("short size = %d bytes\n", sizeof(short));
 printf("int size = %d bytes\n", sizeof(int));
 printf("long size = %d bytes\n", sizeof(long));
 printf("float size = %d bytes\n", sizeof(float));
 printf("double size = %d bytes\n", sizeof(double));
 printf("1.55 size = %d bytes\n", sizeof(1.55));
 printf("1.55L size = %d bytes\n", sizeof(1.55L));
 printf("HELLO size = %d bytes\n", sizeof("HELLO"));
 return 0;
}
```

When run, the program will produce the following output (on the programmer's PC):

```
char size = 1 bytes
short size = 2 bytes
int size = 2 bytes
long size = 4 bytes
float size = 4 bytes
double size = 8 bytes
1.55 size = 8 bytes
1.55L size = 10 bytes
HELLO size = 6 bytes
```

# EXPRESSION EVALUATION: PRECEDENCE & ASSOCIATIVITY

- Evaluation of an expression in C is very important to understand. Unfortunately there is no 'BODMAS' rule in C language as found in algebra.
- The precedence of operators determines the order in which different operators are evaluated when they occur in the same expression. Operators of higher precedence are applied before operators of lower precedence.

| Operaors                                                                                      | Associativity |
|-----------------------------------------------------------------------------------------------|---------------|
| ( ) [ ] . ++ (postfix) -- (postfix)                                                           | L to R        |
| ++ (prefix) -- (prefix) !~ sizeof(type)<br>+ (unary) - (unary) & (address) *<br>(indirection) | R to L        |
| * / %                                                                                         | L to R        |
| + -                                                                                           | L to R        |
| << >>                                                                                         | L to R        |
| < <= > >=                                                                                     | L to R        |
| == !=                                                                                         | L to R        |
| &                                                                                             | L to R        |
| ^                                                                                             | L to R        |
|                                                                                               | L to R        |
| &&                                                                                            | L to R        |
|                                                                                               | L to R        |
| ?:                                                                                            | R to L        |
| = += -= *= /= %= >>= <<= &= ^=  =                                                             | R to L        |
| , (comma operator)                                                                            | L to R        |

| Operaors                                                                                      | Associativity |
|-----------------------------------------------------------------------------------------------|---------------|
| ( ) [ ] . ++ (postfix) -- (postfix)                                                           | L to R        |
| ++ (prefix) -- (prefix) !~ sizeof(type)<br>+ (unary) - (unary) & (address) *<br>(indirection) | R to L        |
| * / %                                                                                         | L to R        |
| + -                                                                                           | L to R        |
| << >>                                                                                         | L to R        |
| < <= > >=                                                                                     | L to R        |
| == !=                                                                                         | L to R        |
| &                                                                                             | L to R        |
| ^                                                                                             | L to R        |
|                                                                                               | L to R        |
| &&                                                                                            | L to R        |
|                                                                                               | L to R        |
| ?:                                                                                            | R to L        |
| = += -= *= /= %= >>= <<= &= ^=  =                                                             | R to L        |
| , (comma operator)                                                                            | L to R        |

# EXAMPLE : OPERATOR PRECEDENCE

```
#include <stdio.h>
int main()
{
 int a;
 int b = 4;
 int c = 8;
 int d = 2;
 int e = 4;
 int f = 2;
 a = b + c / d + e * f;
 /* result without parentheses */
 printf("The value of a is = %d \n", a);
 a = (b + c) / d + e * f;
 /* result with parentheses */
 printf("The value of a is = %d \n", a);
 a = b + c / ((d + e) * f);
 /* another result with parentheses */
 printf("The value of a is = %d \n", a);
 return 0;
}
```

## Output:

The value of a is = 16

The value of a is = 14

The value of a is = 6

# LVALUES AND RVALUES

- An lvalue is an expression to which a value can be assigned .
- An rvalue can be defined as an expression that can be assigned to an lvalue.
- The lvalue expression is located on the left side of an assignment statement, whereas an rvalue is located on the right side of an assignment statement.
- The address associated with a program variable in C is called its lvalue; the contents of that location are its rvalue, the quantity that is supposed to be the value of the variable.
- The rvalue of a variable may change as program execution proceeds; but never its lvalue. The distinction between lvalues and rvalues becomes sharper if one considers the assignment operation with variables a and b.

# LVALUES AND RVALUES

## ■ For example :

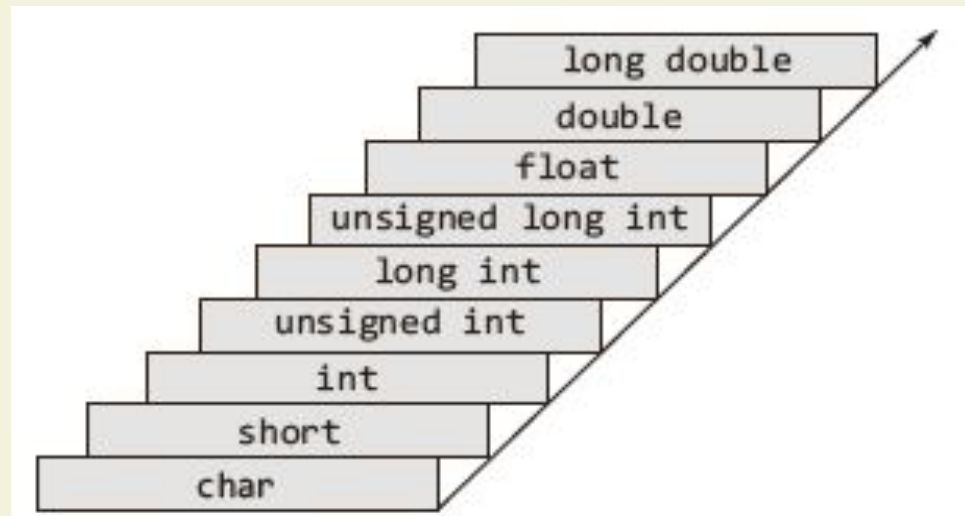
- `a = b;`
- `b`, on the right-hand side of the assignment operator, is the quantity to be found at the address associated with `b`, i.e., an rvalue. `a` is assigned the value stored in the address associated with `b`. `a`, on the left-hand side, is the address at which the contents are altered as a result of the assignment. `a` is an lvalue. The assignment operation deposits `b`'s rvalue at `a`'s lvalue.

| Lvalue                                                              | Rvalue                                              |
|---------------------------------------------------------------------|-----------------------------------------------------|
| Consider the following assignment statement:<br><code>a = b;</code> |                                                     |
| Refers to the address that 'a' represents.                          | Means the content of the address that b represents. |
| is known at compile time.                                           | is not known until runtime.                         |
| Says where to store the value.                                      | Tells what is to be stored.                         |
| Cannot be an expression or a constant                               | Can be an expression or a constant                  |



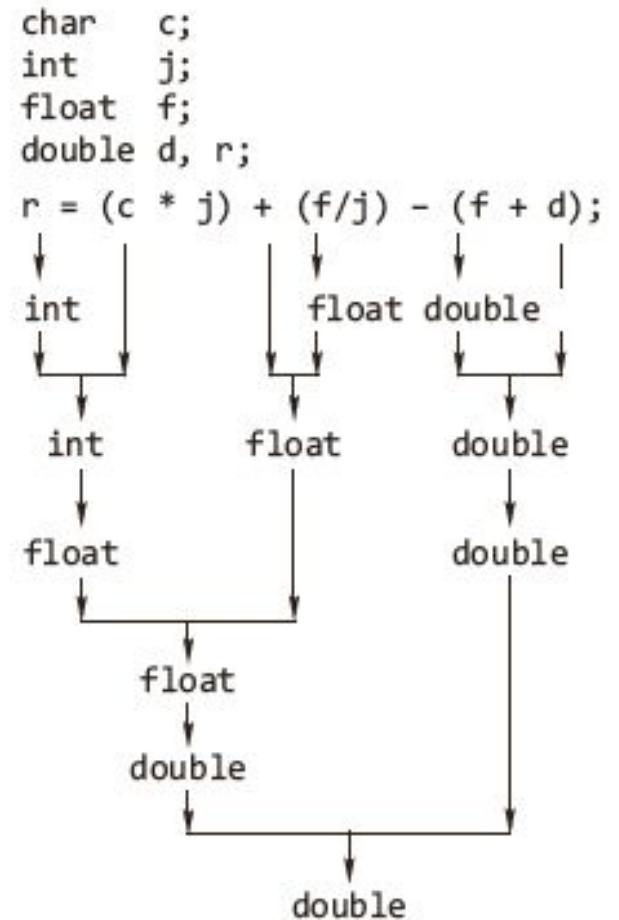
# TYPE CONVERSION

- Though the C compiler performs *automatic type conversions*, the programmer should be aware of what is going on so as to understand how C evaluates expressions.
  - When a C expression is evaluated, the resulting value has a particular data type. If all the variables in the expression are of the same type, the resulting type is of the same type as well. For example, if x and y are both of int type, the expression x + y is of int type as well.
  - The smallest to the largest data types conversion with respect to size is along the arrow as shown below:



# RULE:TYPE CONVERSION

- float operands are converted to double.
- char or short (signed or unsigned) are converted to int (signed or unsigned).
- If any one operand is double, the other operand is also converted to double, and that is the type of the result; or
- If any one operand is long, the other operand is treated as long, and that is the type of the result;
- If any one operand is of type unsigned, the other operand is converted to unsigned, or the only remaining possibility is that Both operands must be int, and that is also the type of the result.



Conversion of types in a mixed expression

# COMPLEX NUMBERS

- A complex number is a number with a real part and an imaginary part. It is of the form  $a + bi$  where  $i$  is the square root of minus one, and  $a$  and  $b$  are real numbers.  $a$  is the real part, and  $bi$  is the imaginary part of the complex number. A complex number can also be regarded as an ordered pair of real numbers  $(a, b)$ .
- According to C99, three complex types are supported:
  - float complex
  - double complex
  - long double complex
- C99 implementations support three imaginary types also :
  - float imaginary
  - double imaginary
  - long double imaginary

# COMPLEX NUMBERS

- To use the complex types, the `complex.h` header file must be included. The `complex.h` header file defines some macros and several functions that accept complex numbers and return complex numbers.
- To use the complex types, the `complex.h` header file must be included. The `complex.h` header file defines some macros and several functions that accept complex numbers and return complex numbers. In particular, the macro `I` represents the square root of  $-1$ . It enables to do the following:
  - `double complex c1 = 3.2 + 2.0 * I;`
  - `float imaginary c2 = -5.0 * I;`