CHAPTER 2

Basics of C

OBJECTIVE

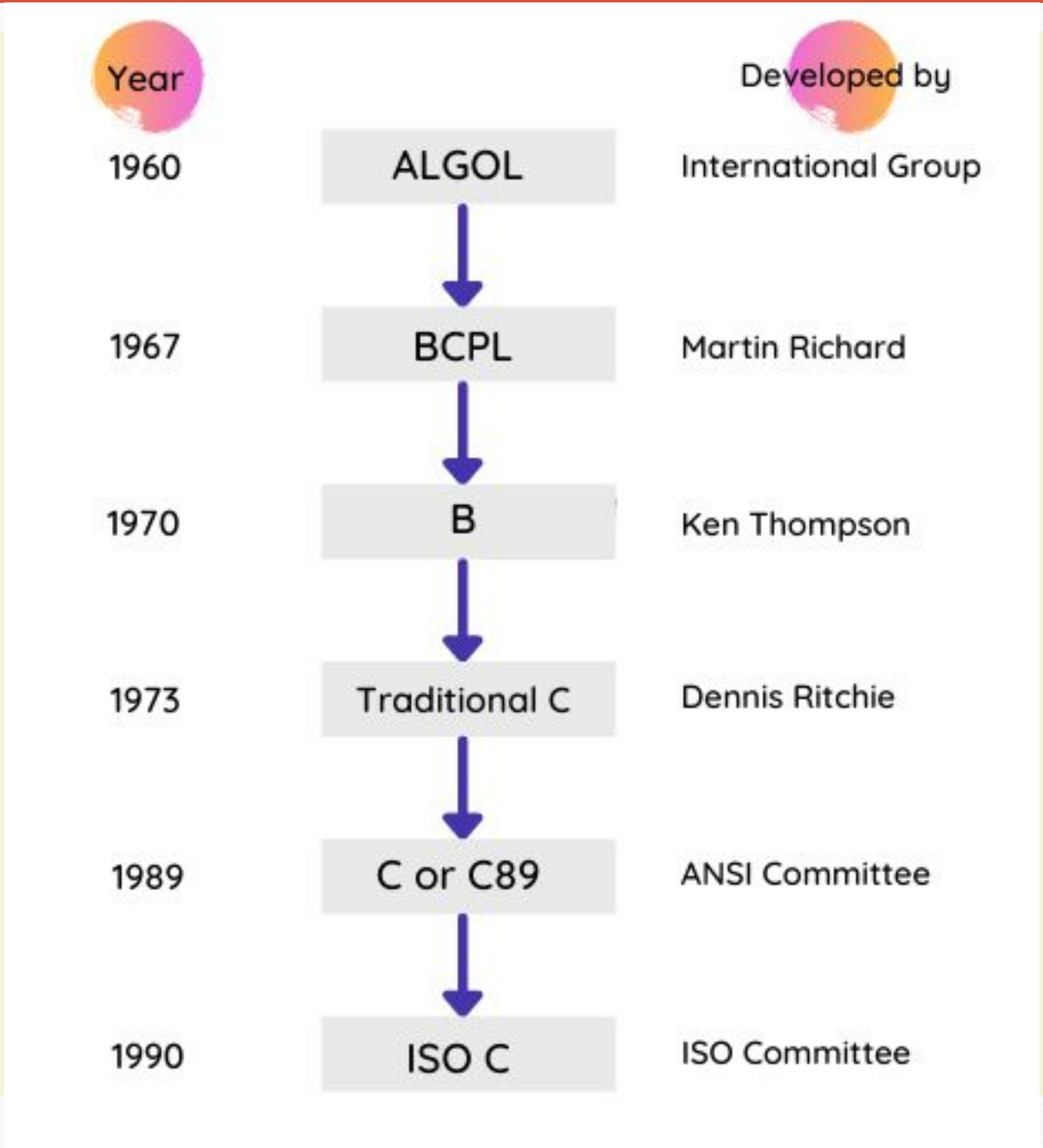
- Understand the basic structure of a program in C .
- Learn the commands used in UNIX/LINUX and MS-DOS for compiling and running a program in C.
- Obtain a preliminary idea of the keywords in C.
- Learn the data types, variables, constants, operators, and expressions in C.
- Understand and grasp the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

INTRODUCTION

- Ken Thompson at Bell Labs, USA, wrote his own variant of this and called it B.
- In due course, the designers of UNIX modified it to produce a programming language called C.
- Dennis Ritchie, also at Bell Labs, is credited for designing C in the early 1970s.
- Subsequently, UNIX was rewritten entirely in C.
- In 1983, an ANSI standard for C emerged, consolidating its international acceptance.
- Ninety percent of the code of the UNIX operating system and of its descendants is written in C.



Year		Developed by
1960	ALGOL	International Group
1967	BCPL	Martin Richard
1970	B	Ken Thompson
1973	Traditional C	Dennis Ritchie
1989	C or C89	ANSI Committee
1990	ISO C	ISO Committee

Changes included in C89 are as follows:

- The addition of truly standard library.
- New preprocessor commands and features.
- Function prototypes which specify the argument types in a function declaration.
- Some new keywords const, volatile and signed.
- Wide characters, wide strings and multi-byte characters.
- Many smaller changes and clarification to conversion rules, declarations and type checking.

In 1990, the ANSI C standard (with a few minor modifications) was made by the International Organization for Standardization (ISO) as ISO/IEC 9899:1990.

This version is sometimes called C90. Therefore, the terms ‘C89’ and ‘C90’ refer to essentially the same language.

C89 is supported by current C compilers, and most C code being written nowadays is based on it. In 1995, amendments to C89 include:

- Three new library headers: iso646.h, wctype.h and wchar.h.
- Some new formatting codes for the printf and scanf family of functions.
- A large number of functions plus some types and constants for multi-byte and wide characters.

In 1999, a more extensive revision to the C standard began. It was completed and approved in 1999. This new version is known as ‘ISO/IEC 9899:1999’ or simply ‘C99’. The followings were included:

- Support for complex arithmetic
- inline functions
- several new data types, including long long int, optional extended integer types, an explicit boolean data type, and a complex type to represent complex numbers
- Variable length arrays
- Better support for non-English characters sets.
- Better support for floating-point types including math functions for all types
- C++ style comments (//)
- new header files, such as stdbool.h and inttypes.h
- type-generic math functions (tgmath.h)
- improved support for IEEE floating point
- variable declaration no longer restricted to file scope or the start of a compound statement

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

INTRODUCTION

- C is the chosen language for systems programming, for the development of 4GL packages such as dbase, and also for the creation of user-friendly interfaces for special applications.
- But application programmers admire C for its elegance, brevity, and the versatility of its operators and control structures.
- C may be termed as a mid-level language,not as low-level as assembly and not as high-level as BASIC.
- 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.

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.

- We can write codes which run quickly, and the program can be very ‘close to the hardware’.
- By that, you can access low level facilities in your computer quite easily, without the compiler or runtime system stopping you from doing something potentially dangerous.

WHY LEARN C?

b . C is a core language :

- In computing, C is a general purpose, cross-platform, block structured procedural, imperative computer programming language.
- There are a number of common and popular computer languages are based on C.
- Having learnt C, it will be much easier to learn languages that are largely or in part based upon C.
- Such languages include C++, Java, and Perl.

WHY LEARN C?

c . C is a small language:

- C has only thirty-two keywords. (and only about twenty of them are in common use)
- This makes it relatively easy to learn compared to bulkier languages.

WHY LEARN C?

d . C is portable.

- C programs written on one system can be run with little or no modification on other systems.
- If modifications are necessary, they can often be made by simply changing a few entries in a header file accompanying the main program.
- The use of compiler directives to the pre-processor makes it possible to produce a single version of a program which can be compiled on several different types of computer.
- In this sense C is said to be very portable.
- The function libraries are standard for all versions of C so they can be used on all systems.

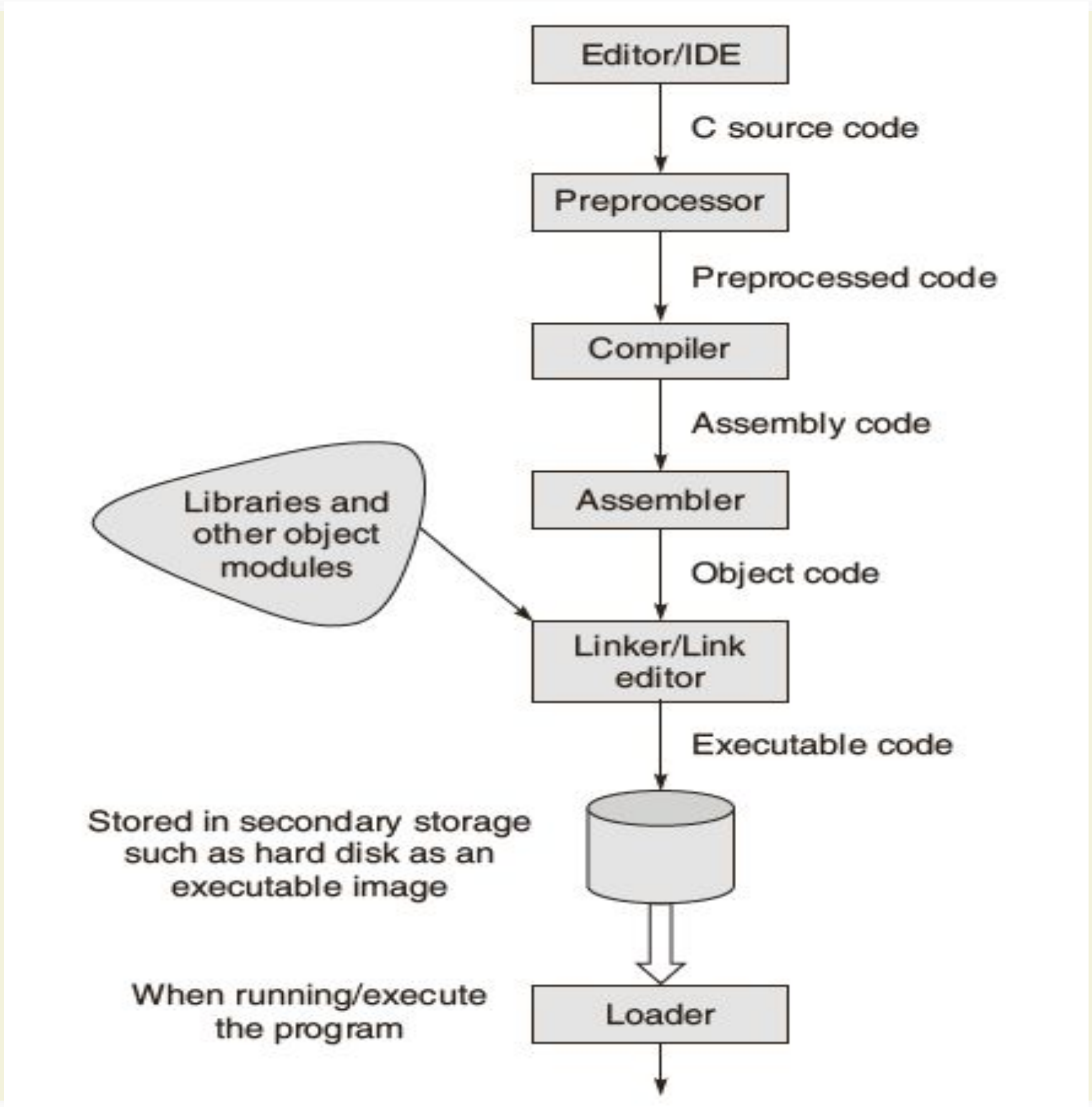
DEVELOPING PROGRAMS IN C

- **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


Editor/IDE
C source code
Preprocessor
Preprocessed code
Compiler
Assembly code
Assembler
Libraries and other object modules
Object code
Linker/Link editor
Executable code
Stored in secondary storage such as hard disk as an executable image
When running/execute the program
Loader

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.

COMPILATION

- 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 fi le 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>.**

COMPILATION : LINUX

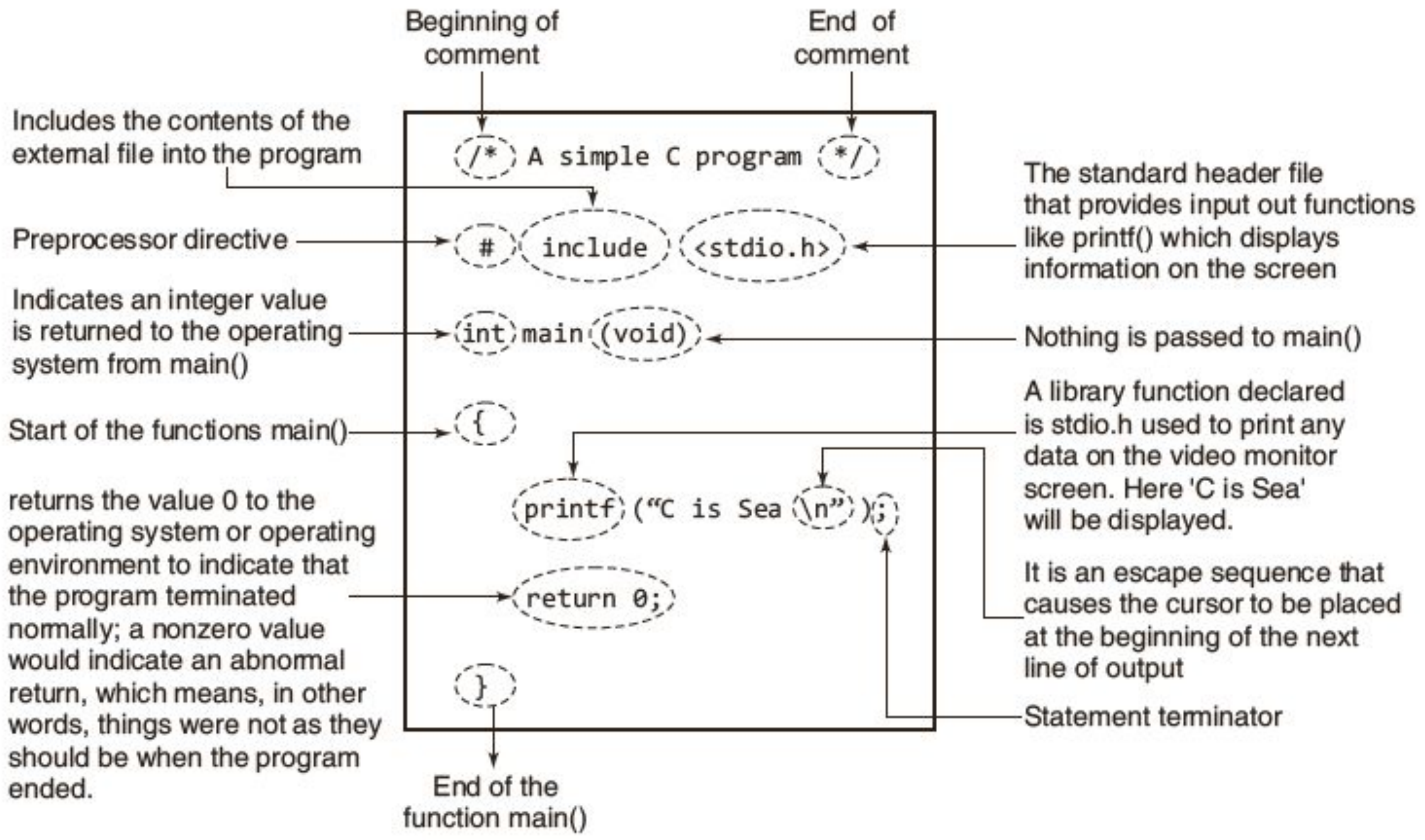
- In the LINUX operating system, a C source program, where first.c is the name of the file, may be compiled by the command

  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

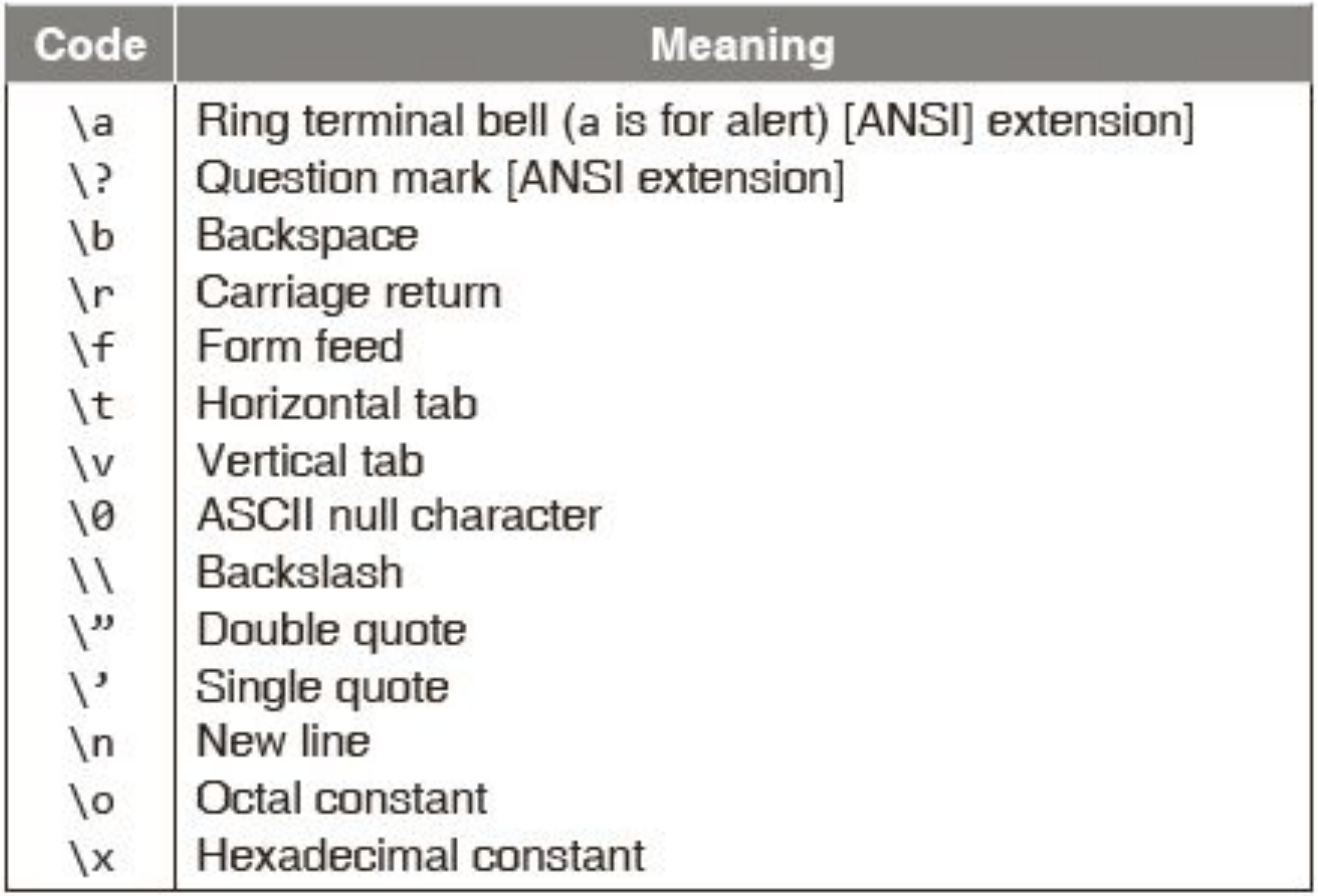
ILLUSTRATED VERSION OF A PROGRAM



```
/* A simple C program */
#include <stdio.h>
int main(void)
{
    printf("C is Sea\n");
    return 0;
}
```

Beginning of comment
End of comment
Includes the contents of the external file into the program
Preprocessor directive
The standard header file that provides input out functions like printf() which displays information on the screen
Indicates an integer value is returned to the operating system from main()
Nothing is passed to main()
Start of the functions main()
A library function declared is stdio.h used to print any data on the video monitor screen. Here 'C is Sea' will be displayed.
returns the value 0 to the operating system or operating environment to indicate that the program terminated normally; a nonzero value would indicate an abnormal return, which means, in other words, things were not as they should be when the program ended.
It is an escape sequence that causes the cursor to be placed at the beginning of the next line of output
Statement terminator
End of the function main()

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

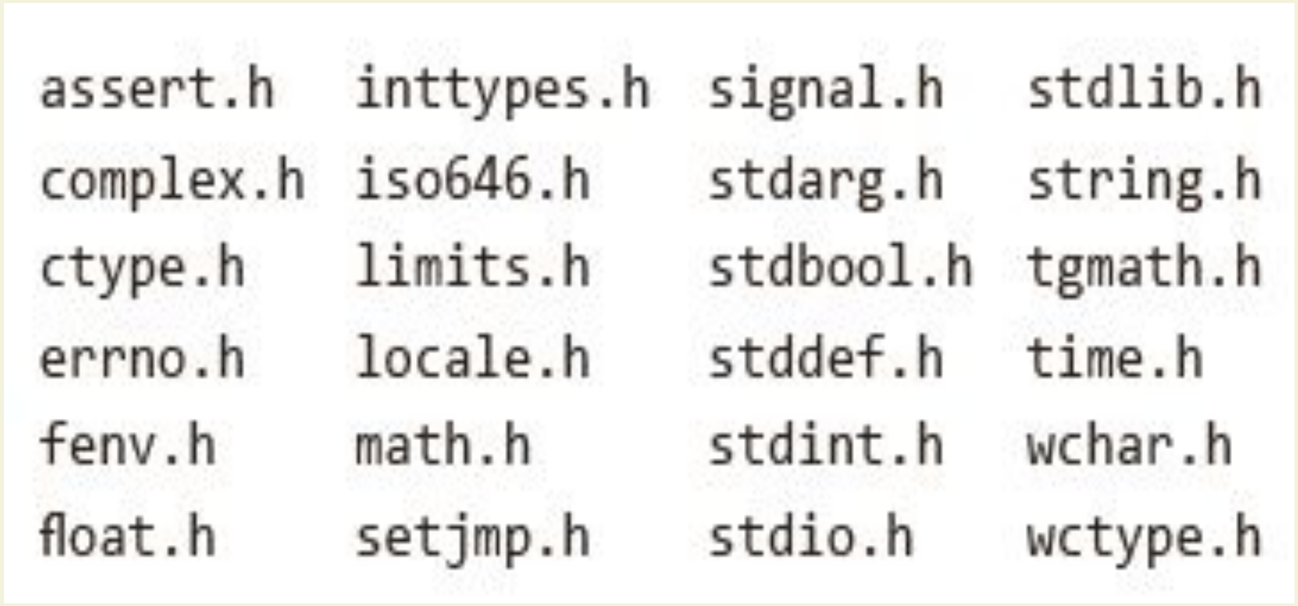
PARTS OF C PROGRAMS

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 c 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[ ]) { /* ... */ }

PHILOSOPHY : MAIN

- On many operating systems, the value returned by main() is used to return an exit status to the environment.
- On Unix, MS-DOS, and Windows systems, the low eight bits of the value returned by main( ) is passed to the command shell or calling program.
- It is extremely common for a program to return a result indication to the operating system.
- Some operating systems require a result code.
- And the return value from main(), or the equivalent value passed in a call to the exit() function, is translated by your compiler into an appropriate code.

PHILOSOPHY : MAIN

There are three and only three completely standard and portable values to return from main() or pass to exit():

- The plain old ordinary integer value 0.
- The constant EXIT_SUCCESS defined in stdlib.h
- The constant EXIT_FAILURE defined in stdlib.h

If you use 0 or EXIT_SUCCESS your compiler's runtime library is guaranteed to translate this into a result code which your operating system considers as successful.

If you use EXIT_FAILURE your compiler's runtime library is guaranteed to translate this into a result code which your operating system considers as unsuccessful

main() is MUST?

- It depends on the environment your program is written for.
- If its a hosted environment, then main function is a must for any standard C program.
- Hosted environments are those where the program runs under an operating system.
- If it is a freestanding environment, then main function is not required.
- Freestanding environments are those where the program does not depend on any host and can have any other function designated as startup function.
- Freestanding implementation need not support complete support of the standard libraries; Usually only a limited number of I/O libraries will be supported and no memory management functions will be supported.
- Examples of freestanding implementations are embedded systems and the operating system kernel

main() is MUST?

The following will give a linker error in all compilers:

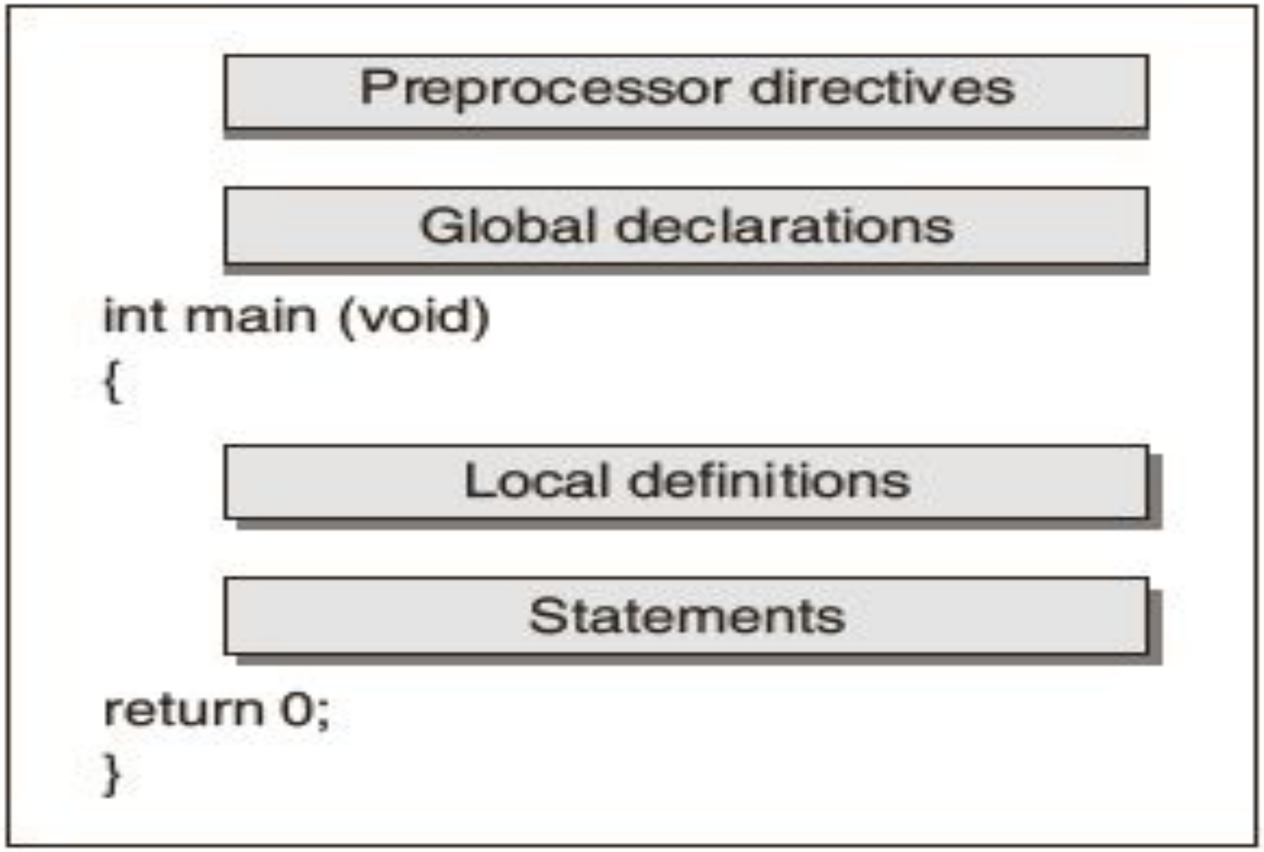
```
MAIN()
{
 printf(“hello, world\n”);
}
```

Along with the user supplied main() function all C programs include something often called the run-time support package which is actually the code that the operating system executes when starting up your program.

The run-time support package then expects to call the user supplied function main(), if there is no user supplied main() then the linker cannot finish the installation of the runtime support package.

In this case the user had supplied MAIN() rather than main(). “MAIN” is a perfectly valid C function name but it isn’t “main”.

STRUCTURE : C PROGRAM



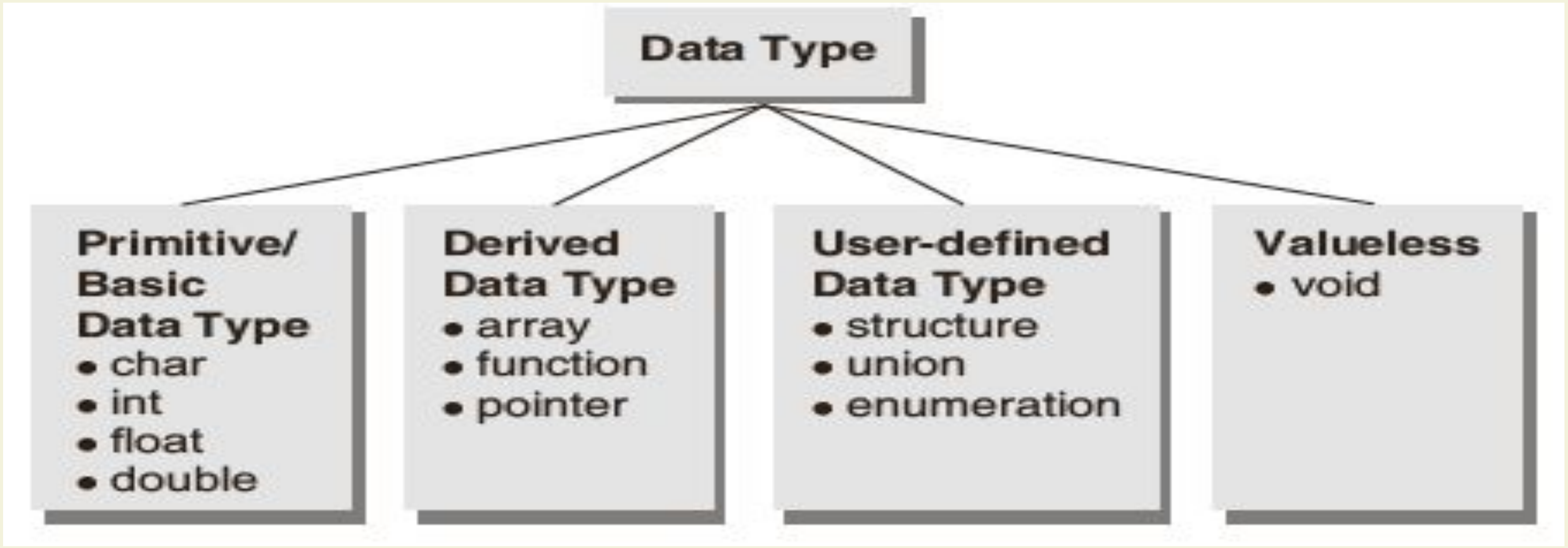
DECLARATION & DEFINITION

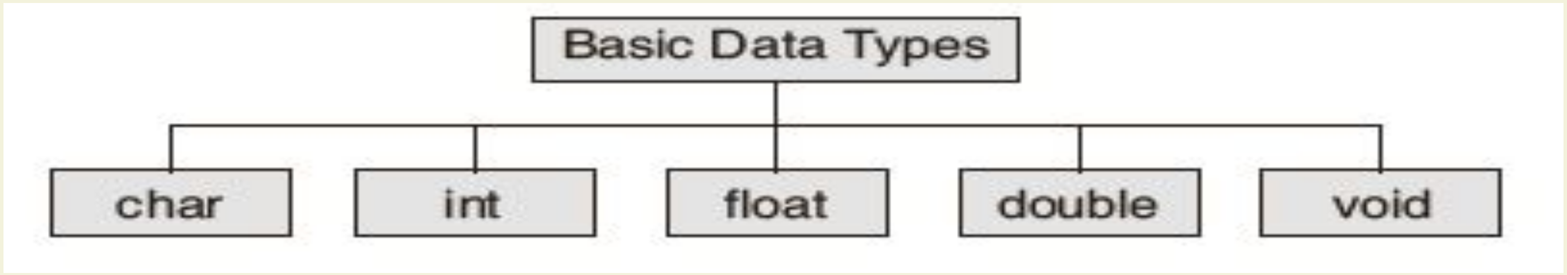
- Declaration means describing the type of a data object to the compiler but not allocating any space for it.
  - A *declaration announces the properties of a data* object or a function. If a variable or function is declared and then later make reference to it with data objects that do not match the 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.

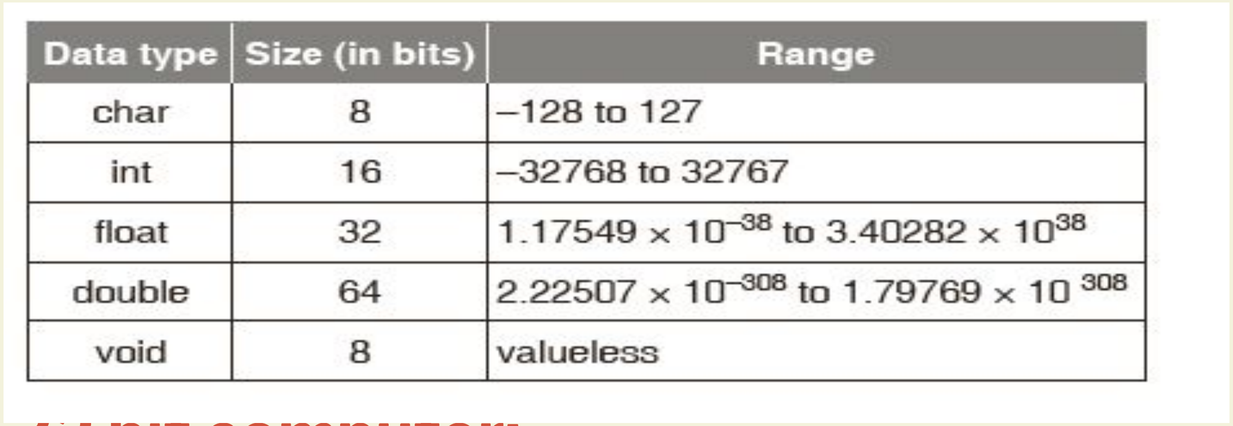
CLASSIFICATION :DATA TYPE





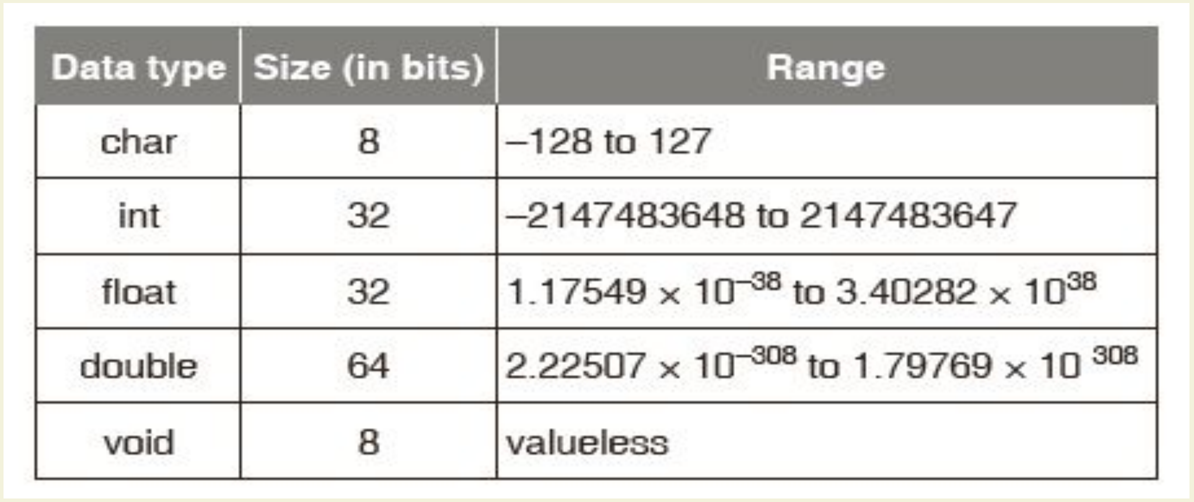
BASIC DATA TYPES:SIZE & RANGE

- 16 bit computer:



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

- The C standard does not state how much precision the float, double types provide, since different computers may store floating point numbers in different ways.
- According to IEEE, the precisions for float and double are 6 and 15 respectively.
- The void type has no values and only one operation, assignment.
- The void type specifies an empty set of values.
- It is used as the type returned by functions that generate no value.
- The void type never refers to an object and therefore, is not included in any reference to object types.
- According to ISO/IEC draft, "The void type comprises an empty set of values; it is an incomplete type that cannot be completed."

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.

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.

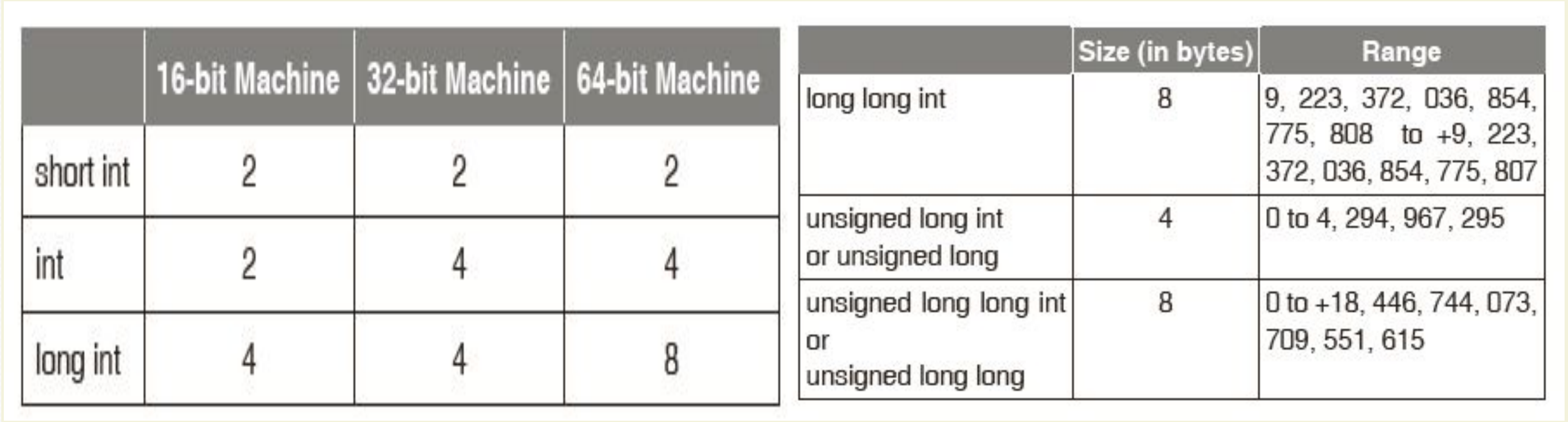
- The specifier short, when placed in front of the int declaration, tells the C compiler that the particular variable being declared is used to store fairly small integer values.
- The motivation for using short variables is primarily one of conserving memory space, which can be an issue in situations in which the program needs a lot of memory and the amount of available memory is limited.
- In any ANSI C compiler, the sizes of short int, int, and long int are restricted by the following rules.
  - The minimum size of a short int is two bytes.
  - The size of an int must be greater than or equal to that of a short int.
  - The size of a long int must be greater than or equal to that of an int.
  - The minimum size of a long int is four bytes.

- In most of the DOS based compilers that work on 16-bit computers, the size of a short int and an int is the same, which is two bytes.
- In such compilers, a long int occupies four bytes.
- On the other hand, in the 32-bit machine compilers such as GNU C(gcc), an int and long int take four bytes while a short int occupies two bytes.
- For UNIX based compilers, a short int takes two bytes, while a long int takes four bytes.
- The long qualifier is also used with the basic data type double.
- In older compilers this qualifier was used with float, but it is not allowed in the popular compilers of today.
- As mentioned earlier, it may be noted here that the sign qualifiers can be used only with the basic data types int and char.

- C99 provides two additional integer types long long int and unsigned long long int.
- For long long, the C99 standard specified at least 64 bits to support.

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 or unsigned long	4	0 to 4, 294, 967, 295
unsigned long long int or unsigned long long	8	0 to +18, 446, 744, 073, 709, 551, 615

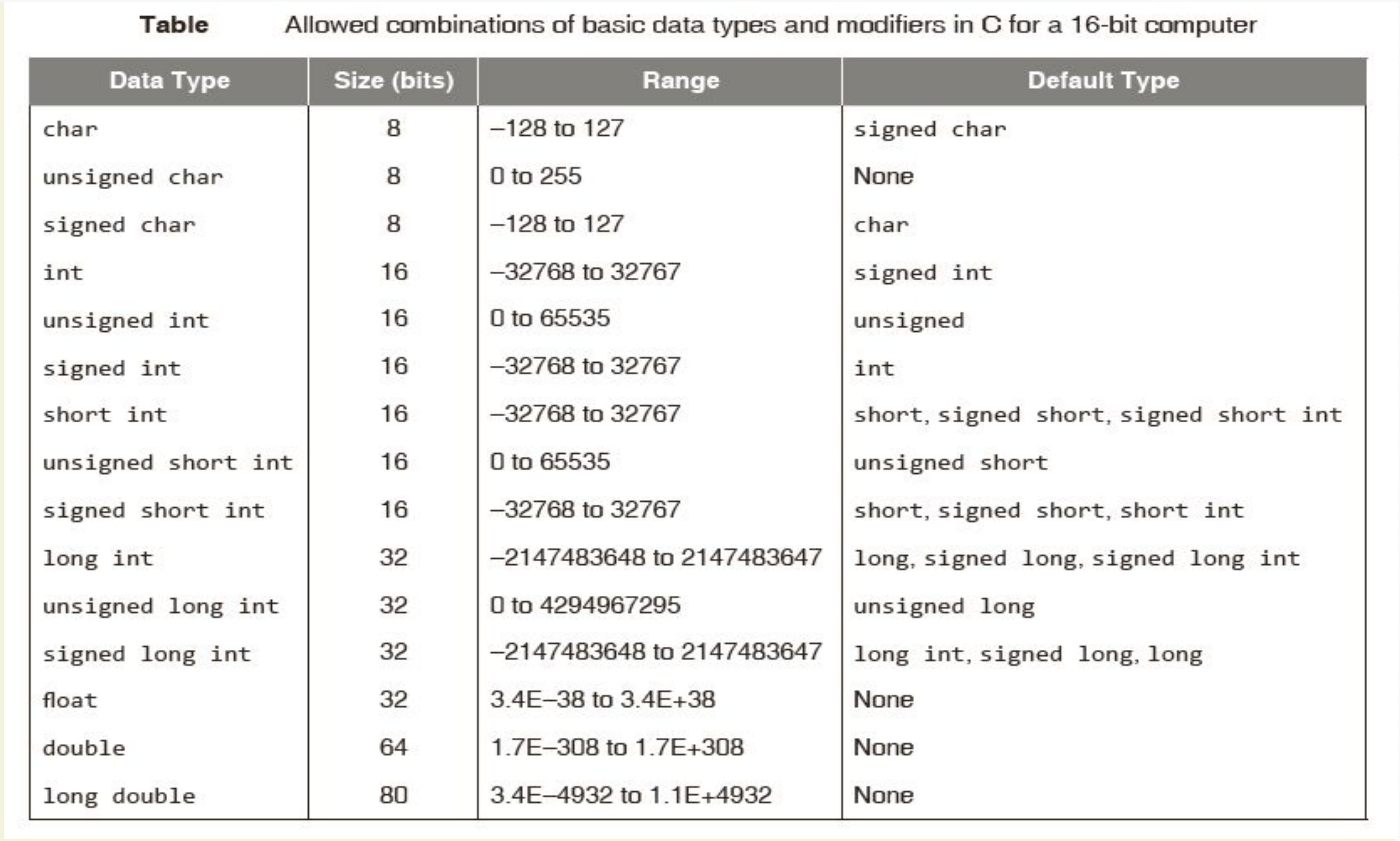


Table Allowed combinations of basic data types and modifiers in C for a 16-bit computer

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

Type Qualifiers

- The C89 Committee added to C two type qualifiers, const and volatile; and C99 adds a third, restrict.
- Type qualifiers control how variables may be accessed or modified.
- They specify which variables will never (const) change and those variables that can change unexpectedly (volatile).
- Both keywords require that an associated data type be declared for the identifier, for example

  ```
  const float pi = 3.14156;
  ```

  specifies that the variable pi can never be changed by the program.
- Any attempt by code within the program to alter the value of pi will result in a compile time error.
- The value of a const variable must be set at the time the variable is declared. Specifying a variable as const allows the compiler to perform better optimization on the program because of the data type being known.

```
#include <stdio.h>
int main(void)
{
   const int value = 42;
   /* constant, initialized integer variable */
   value = 100;
   /* wrong! - will cause compiler error */
   return 0;
}
```

- const does not turn a variable into a constant.
- A variable with const qualifier merely means the variable cannot be used for assignment.
- This makes the value read only through that variable; it does not prevent the value from being modified some other ways e.g. through pointer.

- The volatile keyword indicates that a variable can unexpectedly change because of events outside the control of the program.
- This usually is used when some variable within the program is linked directly with some hardware component of the system.
- The hardware could then directly modify the value of the variable without the knowledge of the program.
- For example, an I/O device might need to write directly into a program or data space.
- Meanwhile, the program itself may never directly access the memory area in question.
- In such a case, we would not want the compiler to optimize-out this data area that never seems to be used by the program, yet must exist for the program to function correctly in a larger context.
- It tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and forces every reference to such an object to be a genuine reference.

- The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables.
- Anyone for whom this is not a concern can safely ignore this feature of the language.

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 (;).

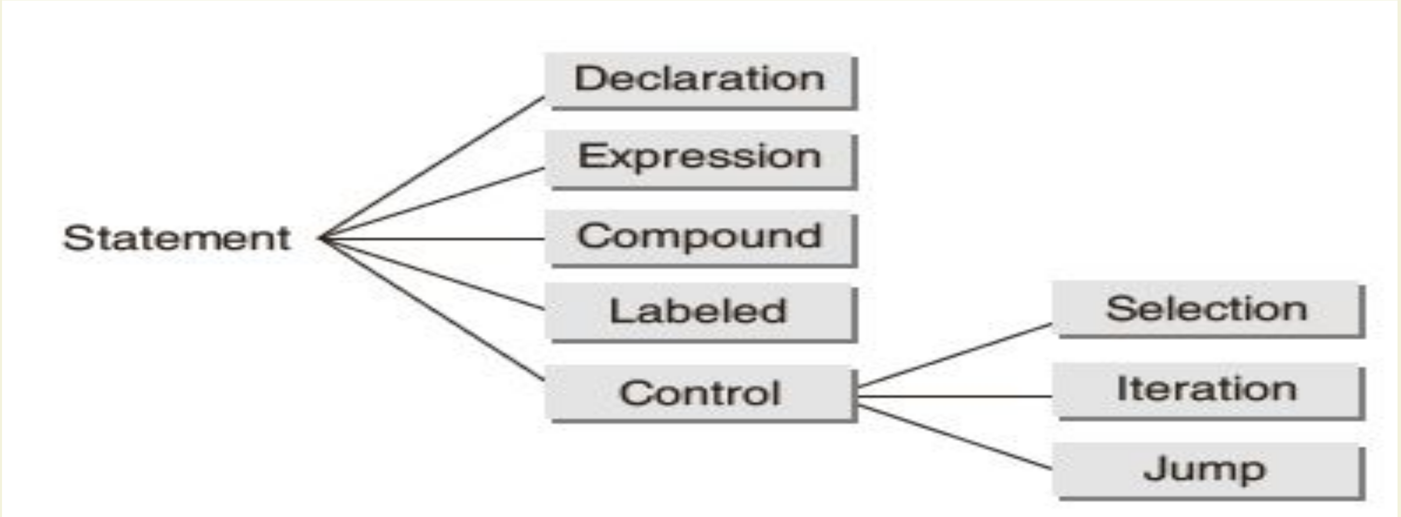


Figure Different types of program statements available in C

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.

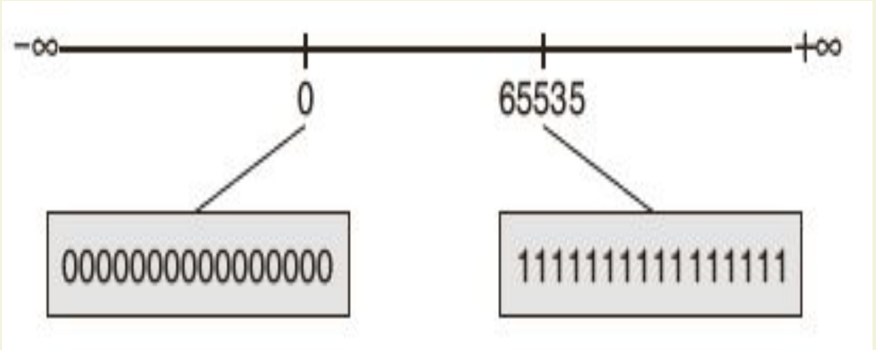


Figure (a) Range of an unsigned integer stored in a 16-bit word

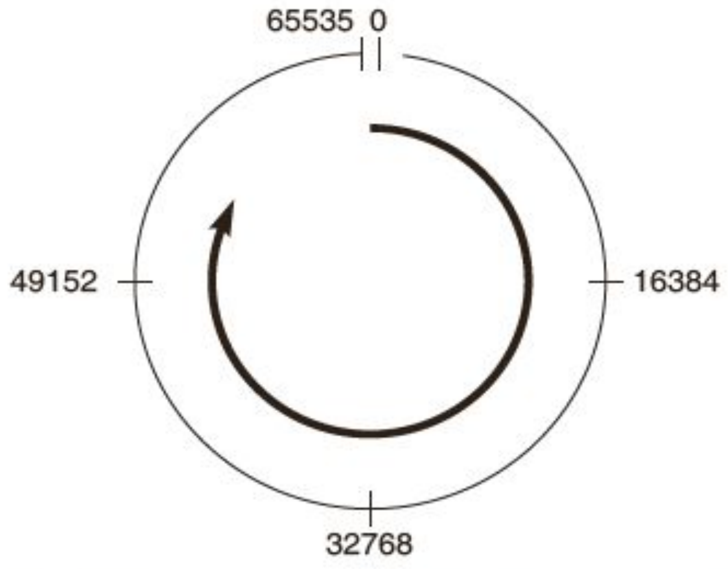
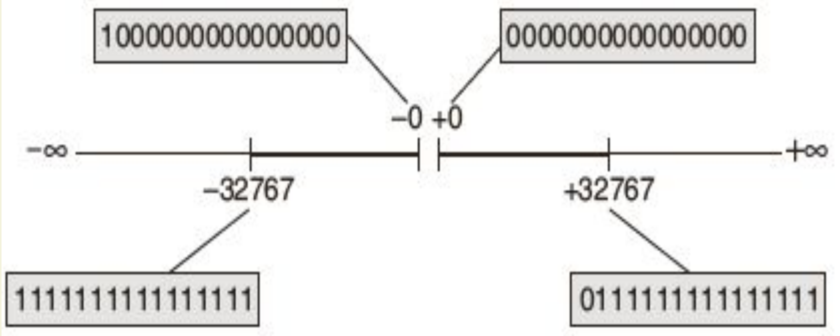


Figure (b) Cyclic view of the range of an unsigned integer stored in a 16-bit word

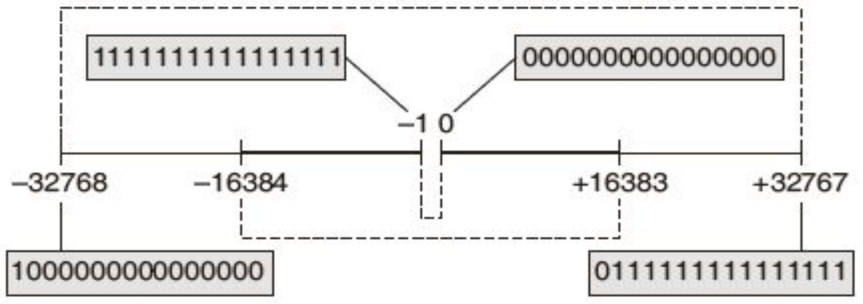
HOW THE INTEGERS ARE STORED IN MEMORY

- 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 $M \le N$). 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).

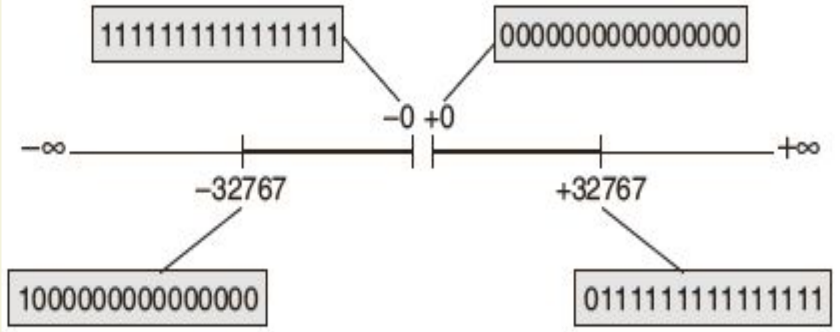
INTEGER STORING:PROCESS



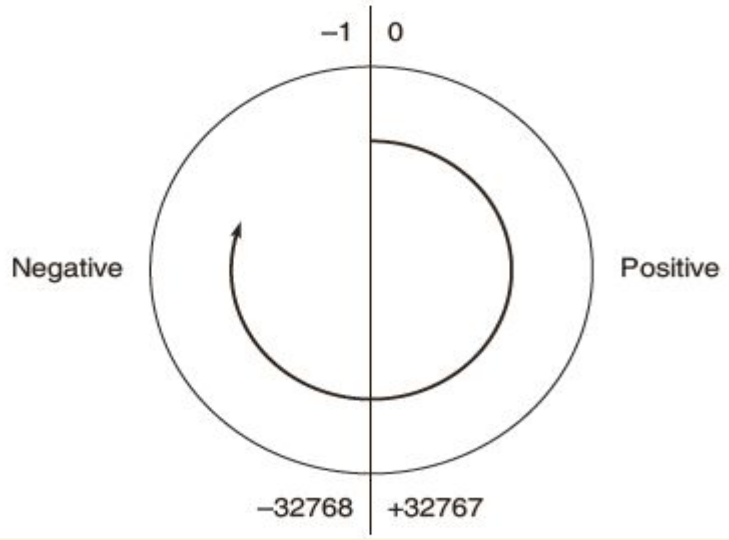
Range of a signed integer stored in a 16-bit word in sign and magnitude form



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



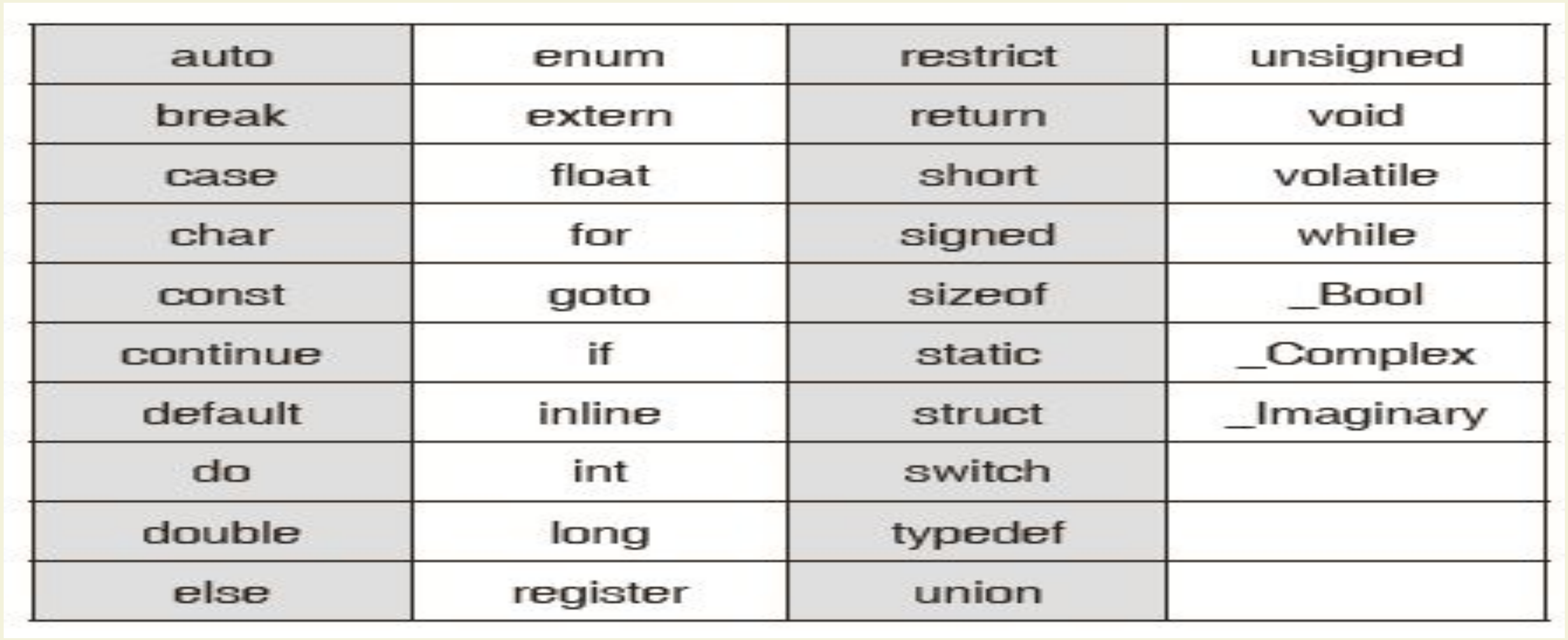
Range of a signed integer stored in 16-bit word in one's complement form



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.**

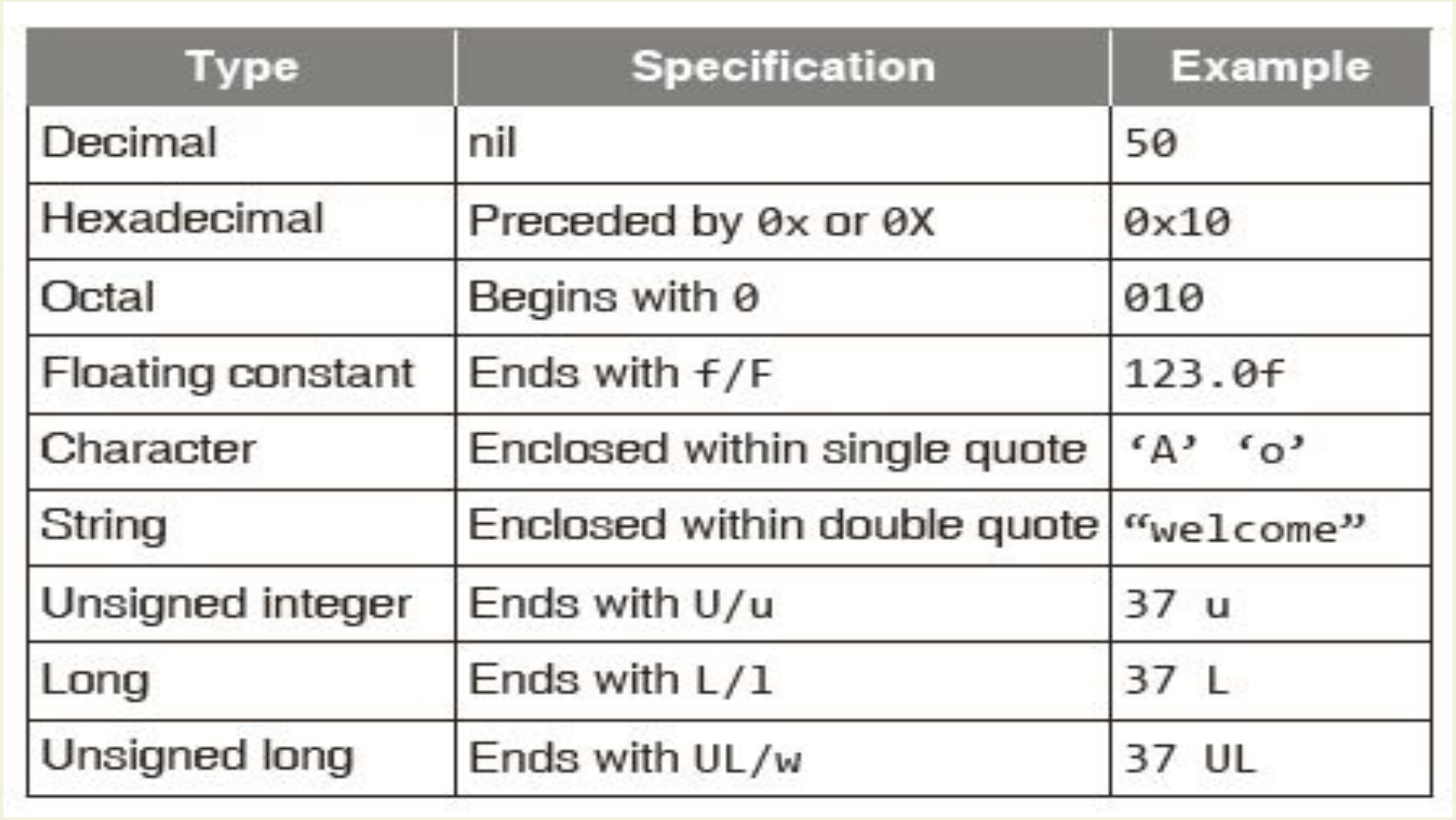


auto	enum	restrict	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool
continue	if	static	_Complex
default	inline	struct	_Imaginary
do	int	switch	
double	long	typedef	
else	register	union	

CONSTANT

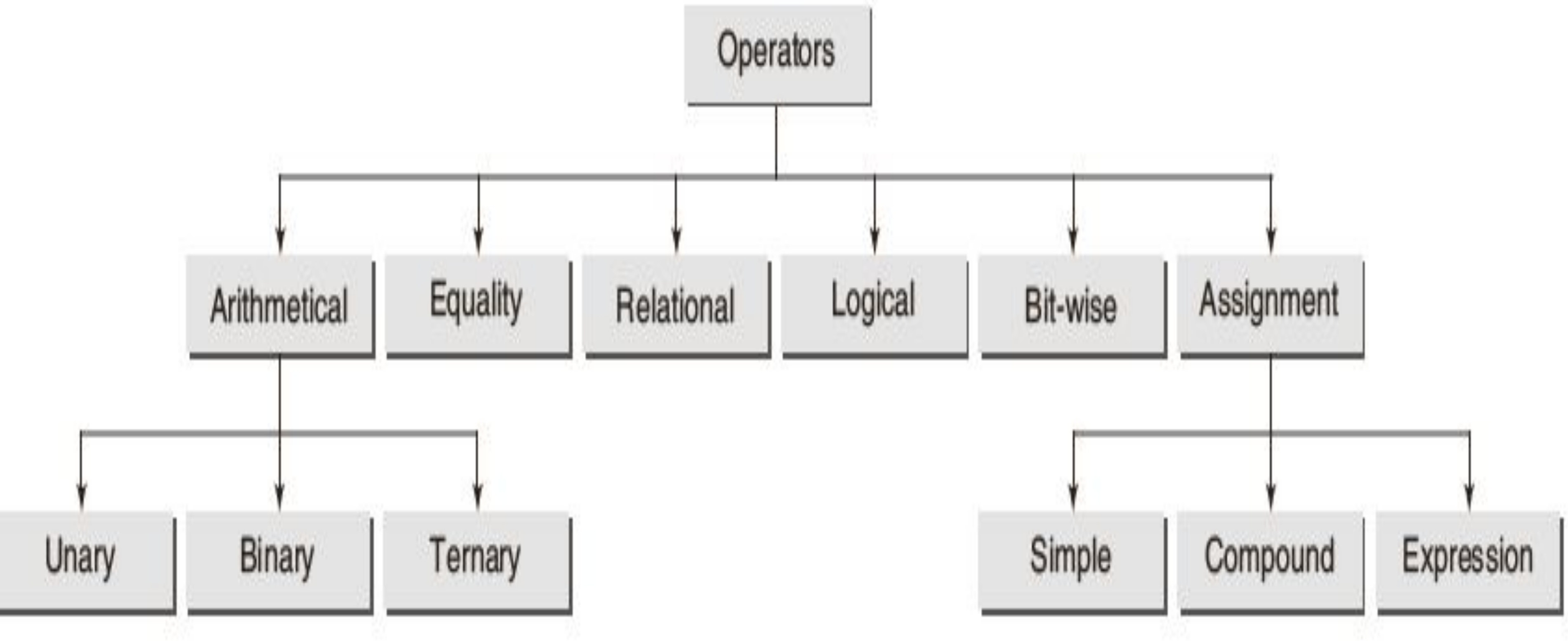
- A constant is an explicit data value written by the programmer. Thus, it is a value known to the compiler at compiling time.
- In ANSI C, a decimal integer constant is treated as an unsigned long if its magnitude exceeds that of the signed long. An octal or hexadecimal integer that exceeds the limit 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

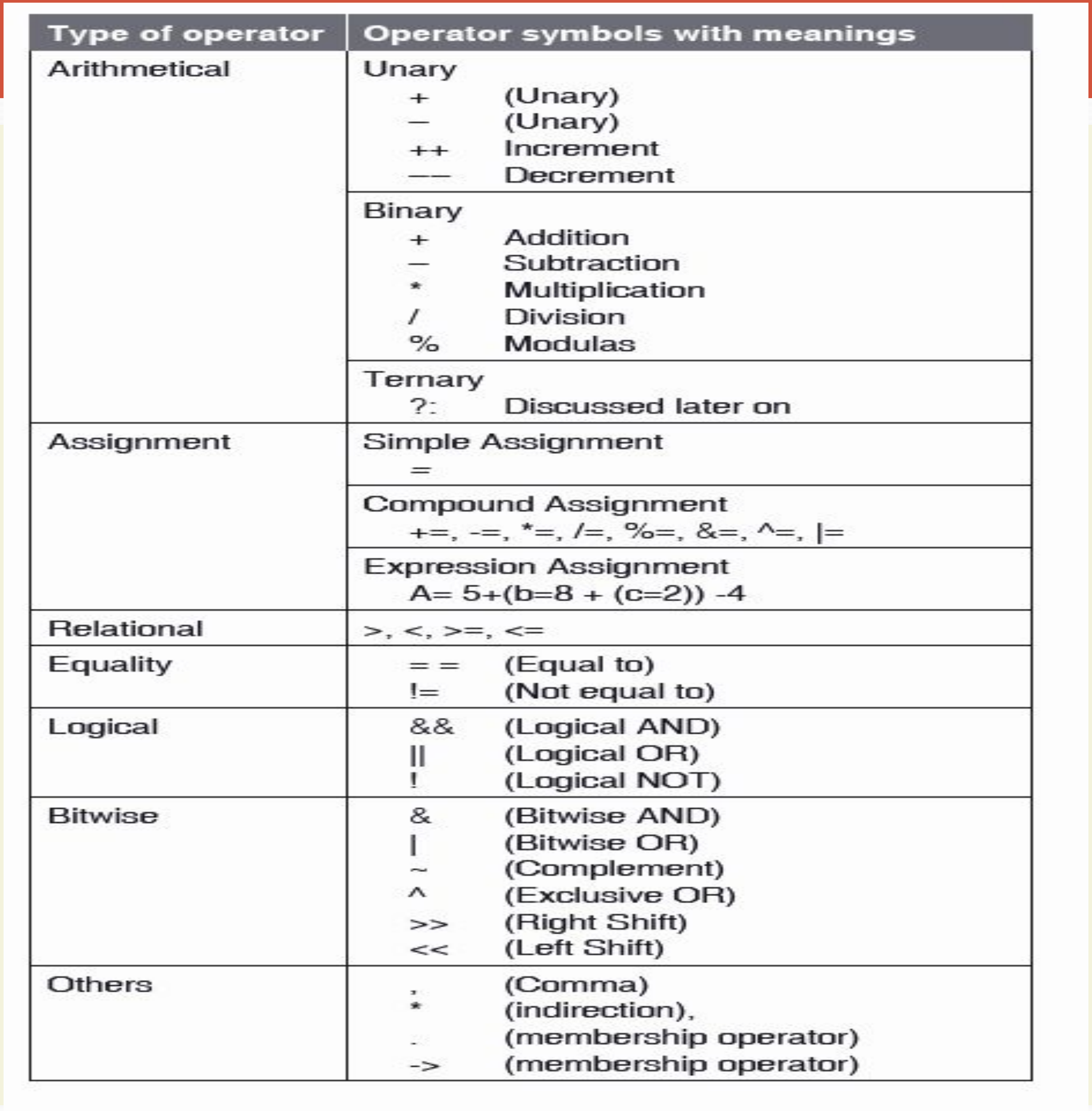


Type	Specification	Example
Decimal	nil	50
Hexadecimal	Preceded by 0x or 0X	0x10
Octal	Begins with 0	010
Floating constant	Ends with f/F	123.0f
Character	Enclosed within single quote	‘A’ ‘o’
String	Enclosed within double quote	“welcome”
Unsigned integer	Ends with U/u	37 u
Long	Ends with L/l	37 L
Unsigned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

CLASSIFICATION:OPERATORS IN C



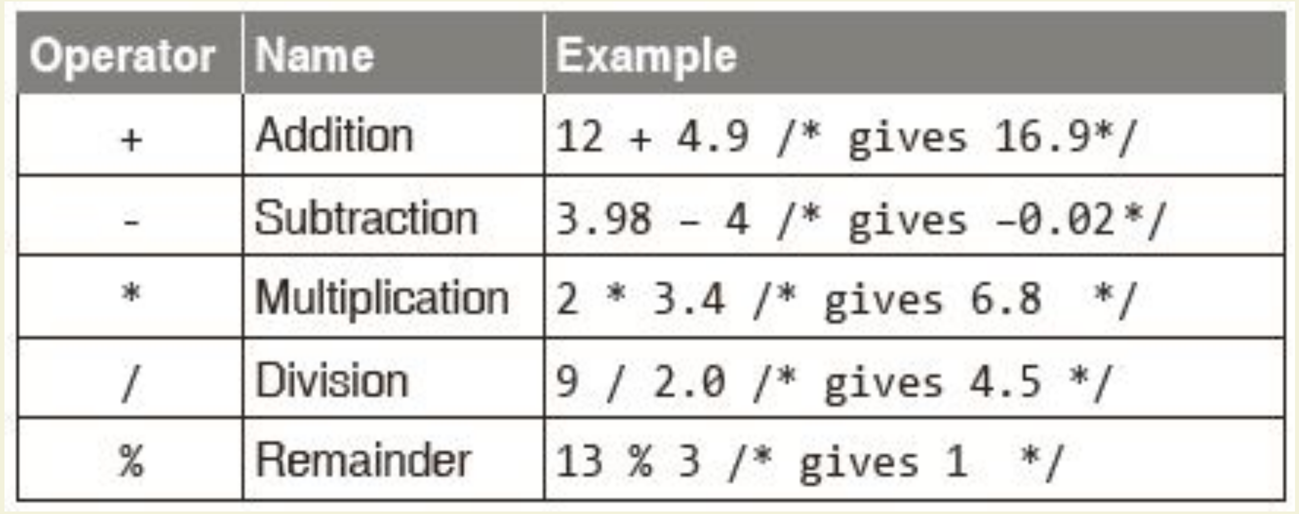
DIFFERENT OPERATORS



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

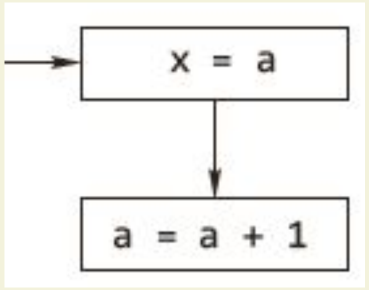
UNARY OPERATION

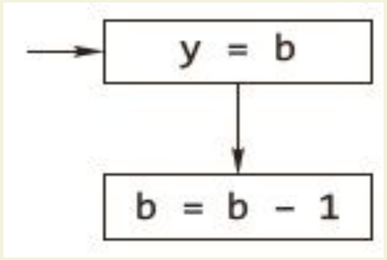
- ***Unary operators:*** The unary '–' operator negates the value of its operand (clearly, a signed number). A numeric constant is assumed positive unless it is preceded by the negative operator. That is, there is no unary '+'. It is implicit. Remember that -x does not change the value of x at the location 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.

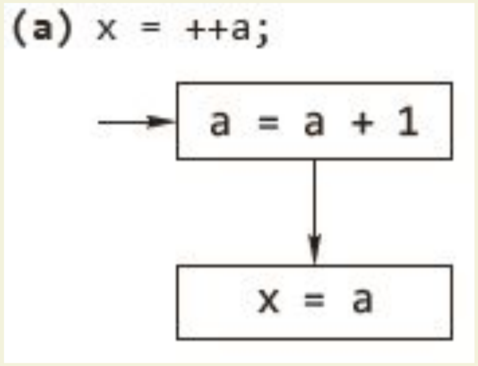


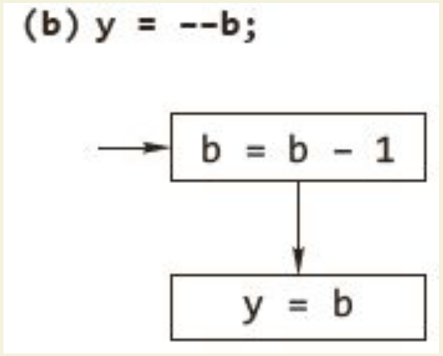


PREFIX

Prefix :

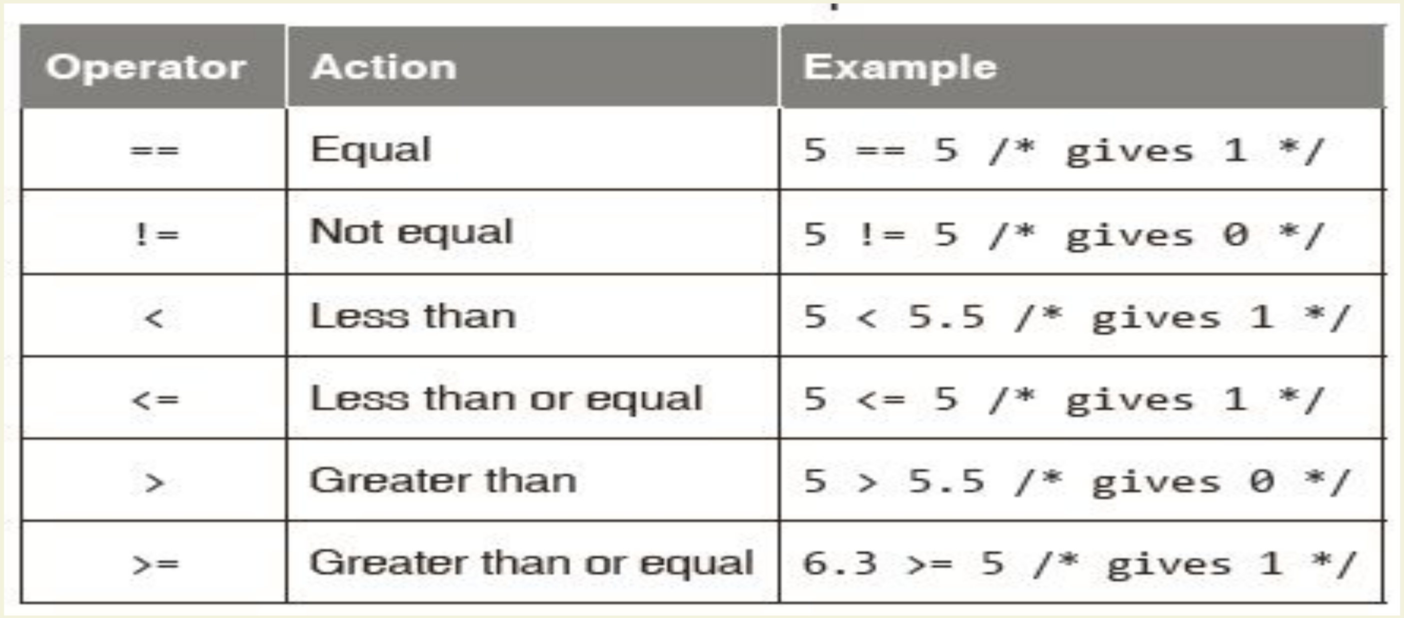
- **(a) x = ++a;**
  - *First action: increment value of a* by 1 and store result in memory location for variable a.
  - *Second action: store value of a* in memory location for variable x.
- **(b) y = ––b;**
  - *First action: decrement value of b* by 1 and put result in memory location for variable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





RELATIONAL OPERATORS

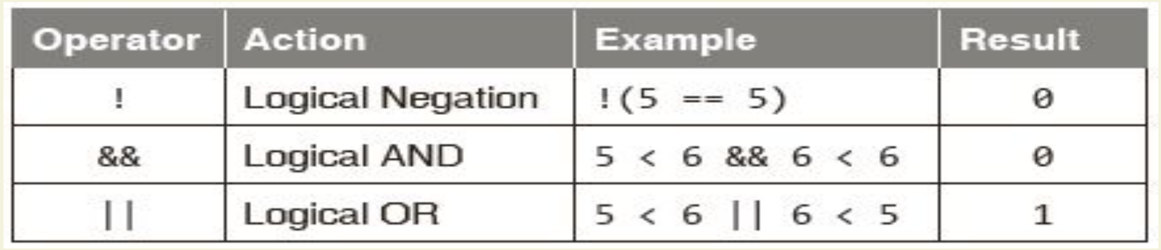
- C provides six relational operators for comparing numeric quantities. Relational operators evaluate to 1, representing the *true outcome, or* 0, representing the *false outcome.*



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.

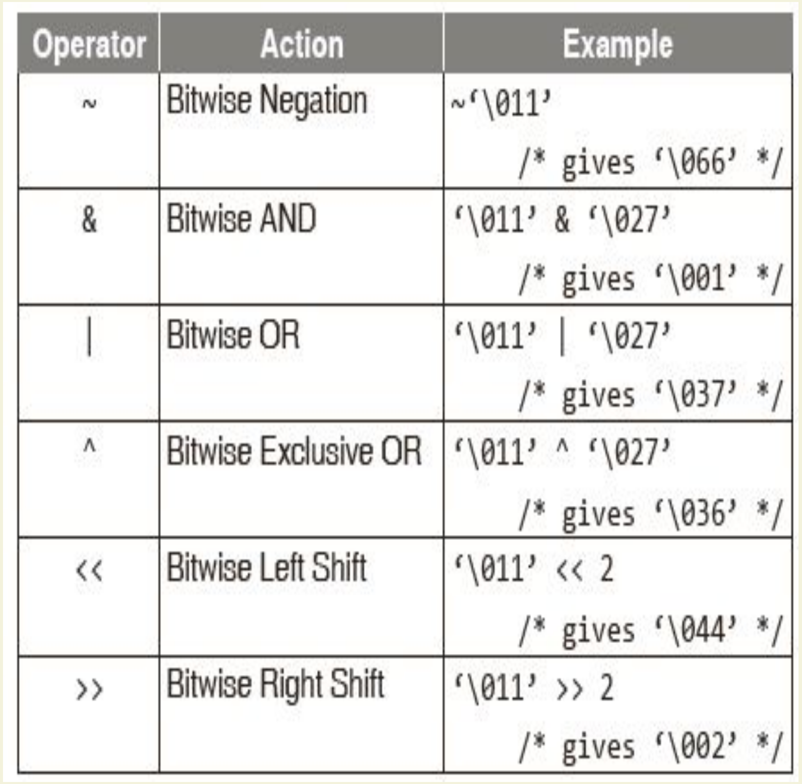


Operator	Action	Example	Result
!	Logical Negation	`!(5 == 5)`	0
&&	Logical AND	`5 < 6 && 6 < 6`	0
||	Logical OR	`5 < 6 || 6 < 5`	1

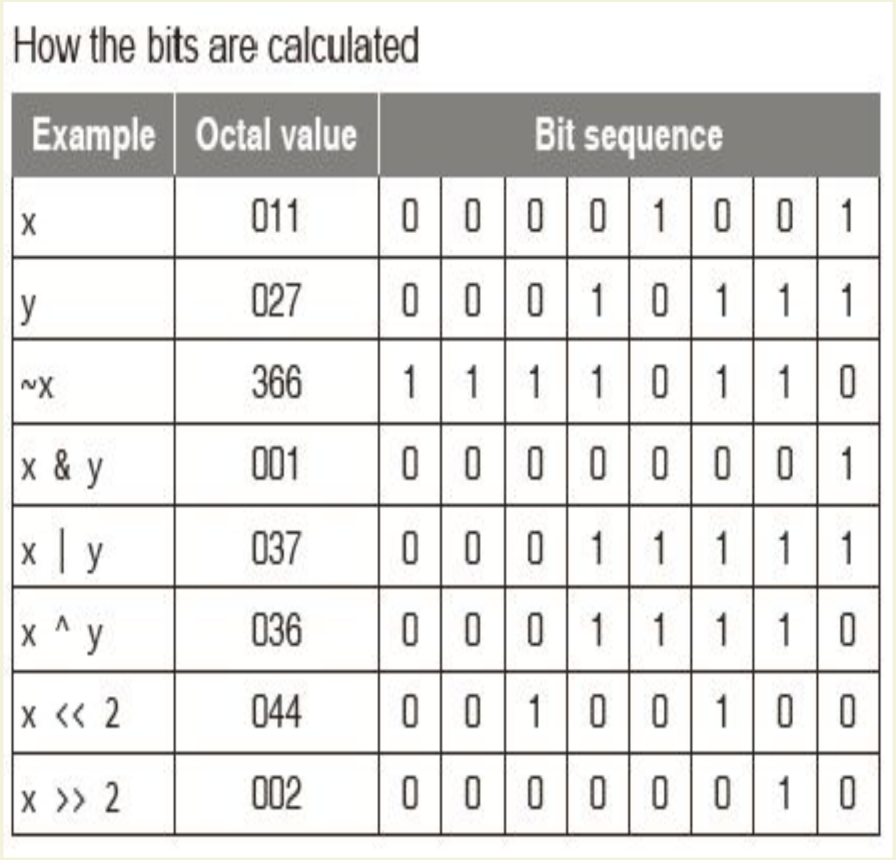
BIT WISE OPERATORS

- 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' /* gives '\066' */
&	Bitwise AND	'\011' & '\027' /* gives '\001' */
|	Bitwise OR	'\011' | '\027' /* gives '\037' */
^	Bitwise Exclusive OR	'\011' ^ '\027' /* gives '\036' */
<<	Bitwise Left Shift	'\011' << 2 /* gives '\044' */
>>	Bitwise Right Shift	'\011' >> 2 /* 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

BIT WISE OPERATORS

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

  ________

  11011100  = 220 (In decimal)
```

Twist in Bitwise Complement Operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n + 1). To understand this, you should have the knowledge of 2's complement.

BIT WISE OPERATORS

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

  ________

  11011100  = 220 (In decimal)
```

Twist in Bitwise Complement Operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n + 1). To understand this, you should have the knowledge of 2's complement.

BIT WISE OPERATORS

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

```
Decimal         Binary           2's complement
   0            00000000           -(11111111+1) = -00000000 = -0(decimal)
   1            00000001           -(11111110+1) = -11111111 = -256(decimal)
   12           00001100           -(11110011+1) = -11110100 = -244(decimal)
   220          11011100           -(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.
```

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

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;
```

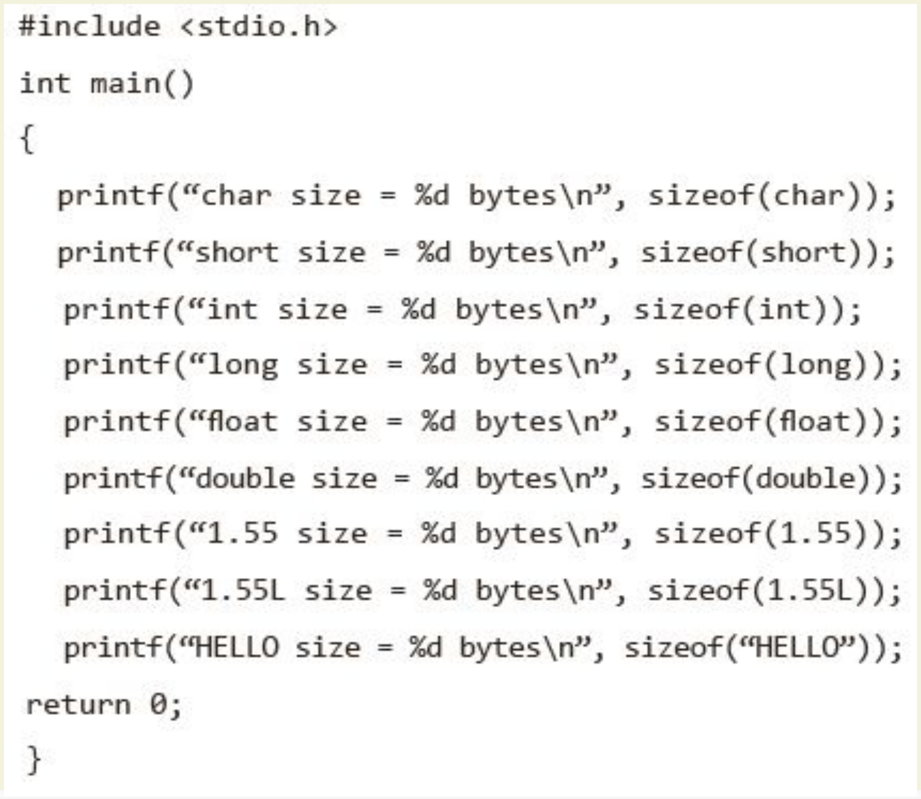
Here the tokens that will be generated are

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

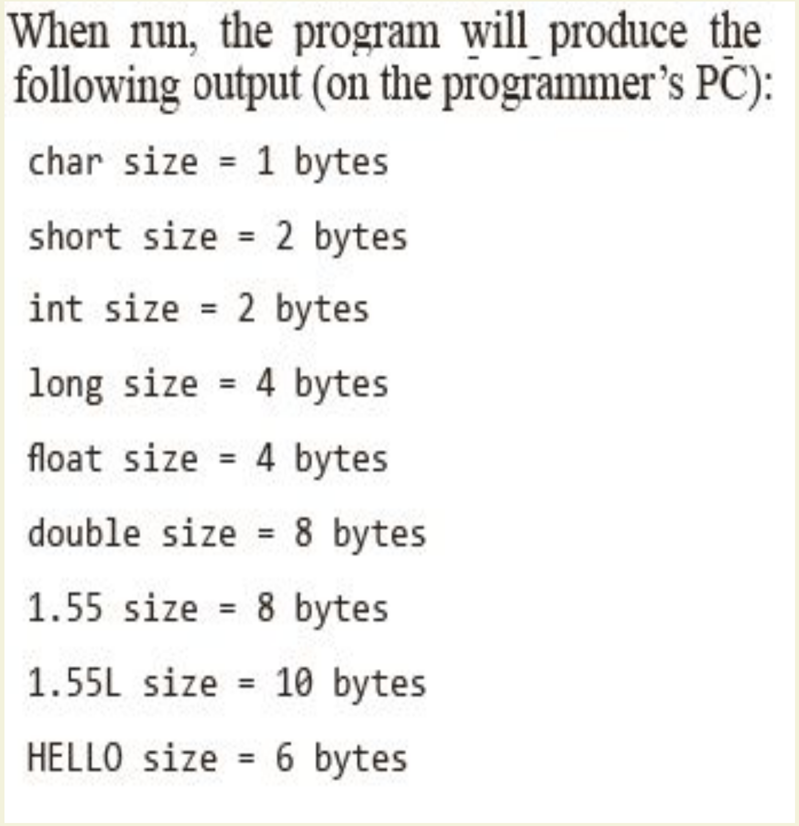
SIZEOF OPERATOR

- 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;
}
```

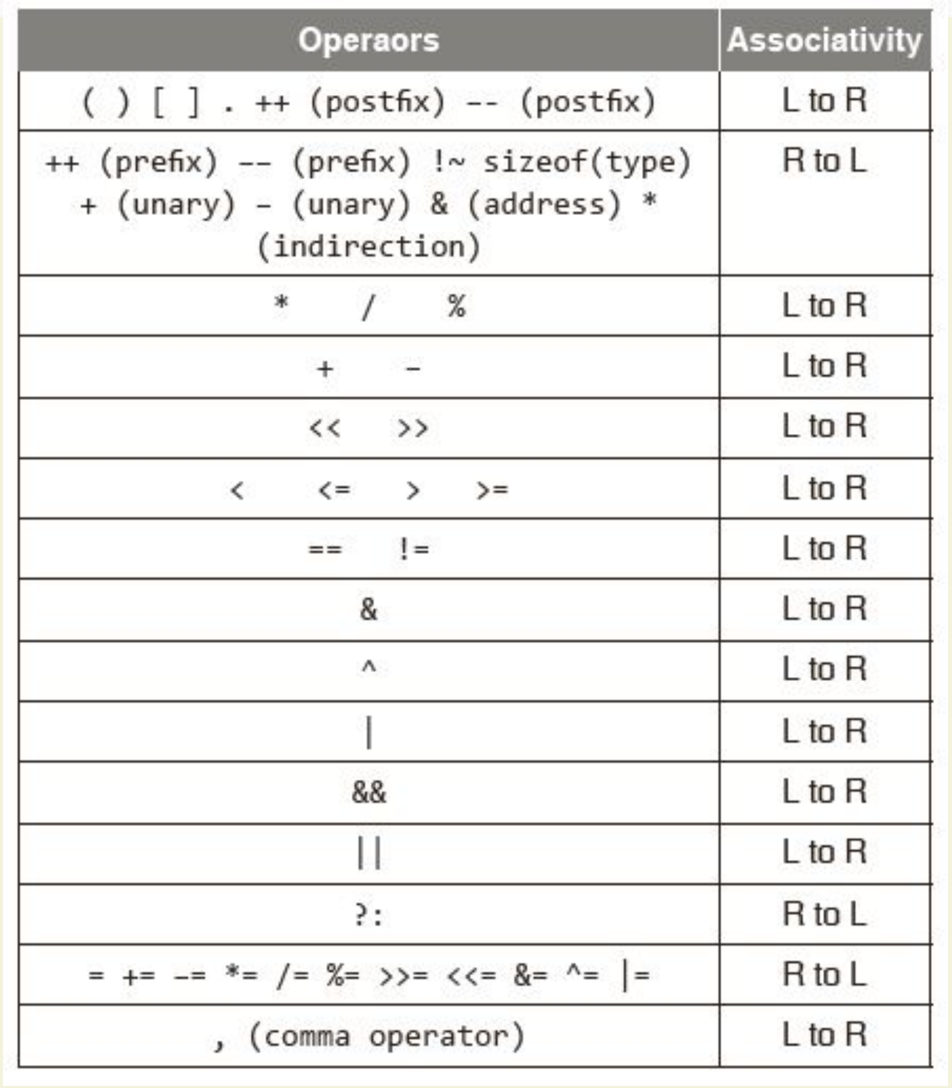


When run, the program will produce the following output (on the programmer’s PC):

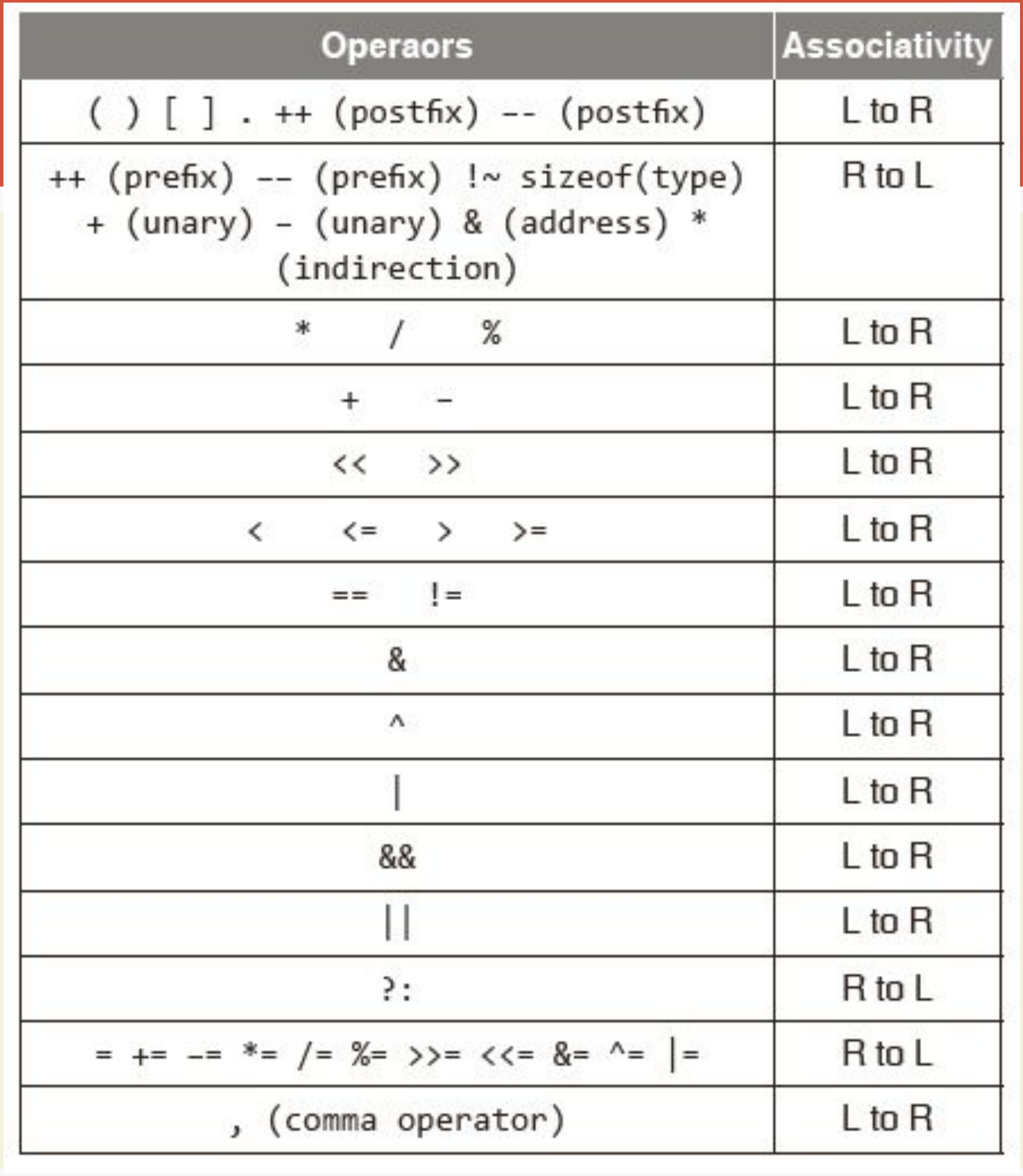
```
char size = 1 bytes
short size = 2 bytes
int size = 2 bytes
long size = 4 bytes
float size = 4 bytes
double size = 8 bytes
1.55 size = 8 bytes
1.55L size = 10 bytes
HELLO size = 6 bytes
```

EXPRESSION EVOLUATION: 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.

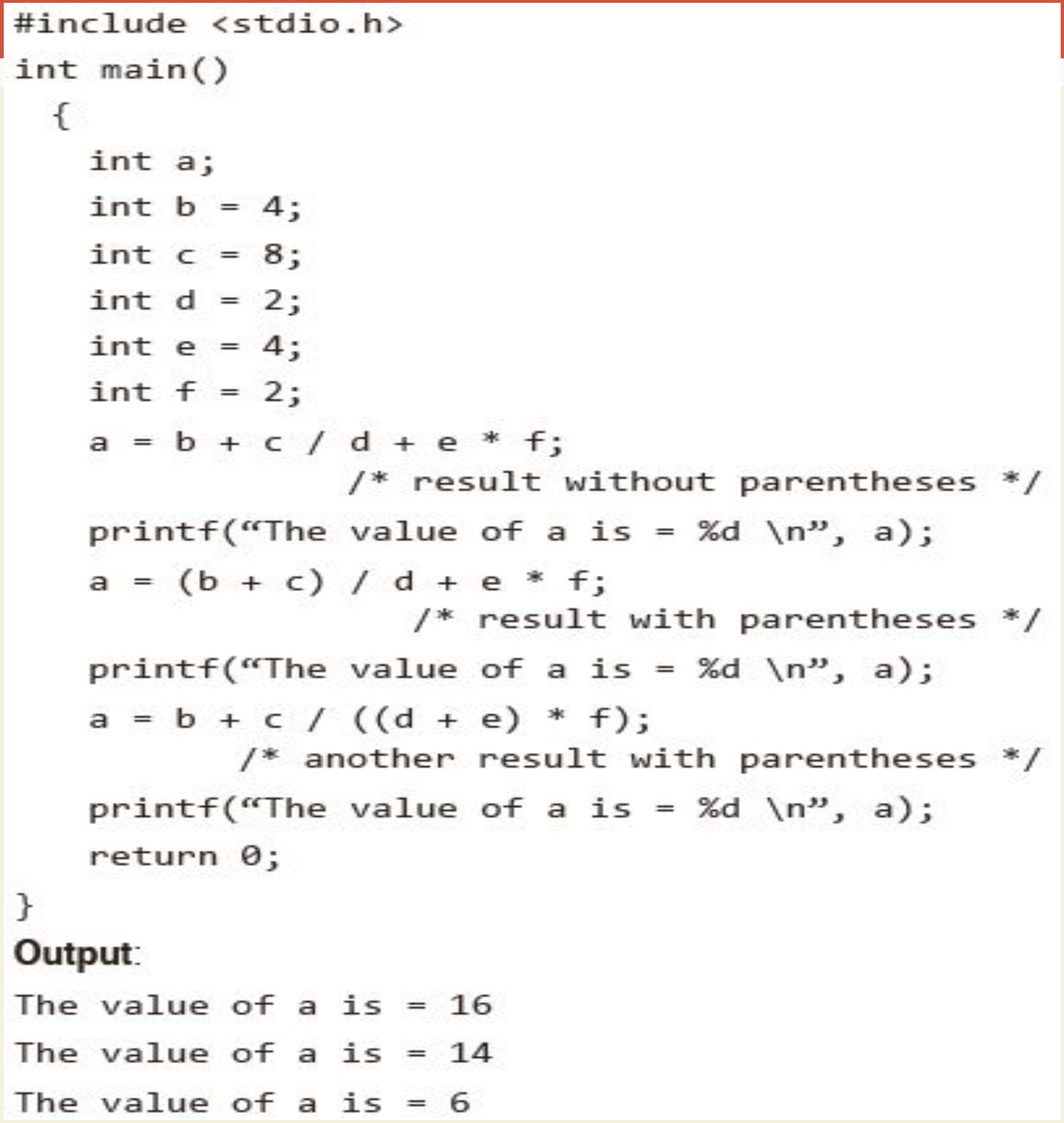


Operaors	Associativity
( ) [ ] . ++ (postfix) -- (postfix)	L to R
++ (prefix) -- (prefix) !~ sizeof(type) + (unary) - (unary) & (address) * (indirection)	R to L
* / %	L to R
+ -	L to R
<< >>	L to R
< <= > >=	L to R
== !=	L to R
&	L to R
^	L to R
|	L to R
&&	L to R
||	L to R
?:	R to L
= += -= *= /= %=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



Operaors	Associativity
( ) [ ] . ++ (postfix) -- (postfix)	L to R
++ (prefix) -- (prefix) !~ sizeof(type) + (unary) - (unary) & (address) * (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:

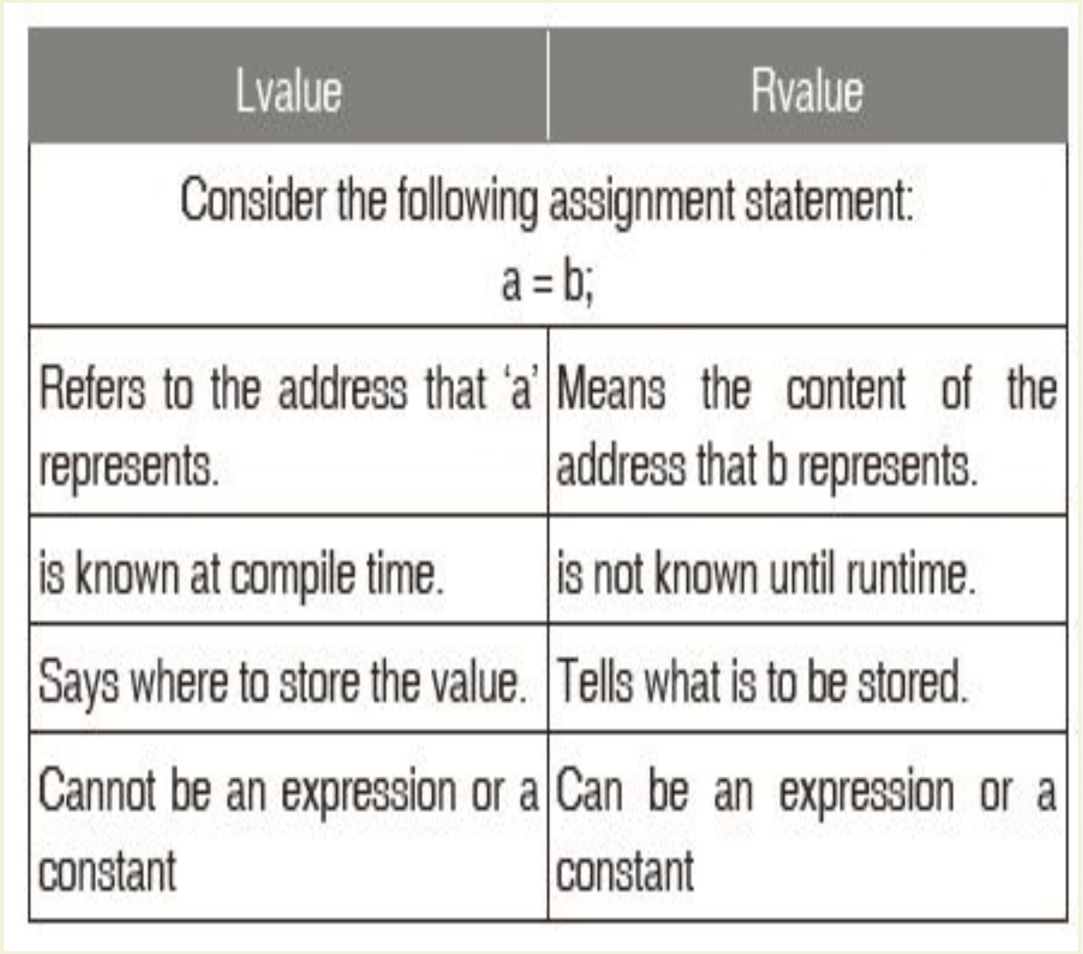
```
The value of a is = 16
The value of a is = 14
The value of a is = 6
```

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

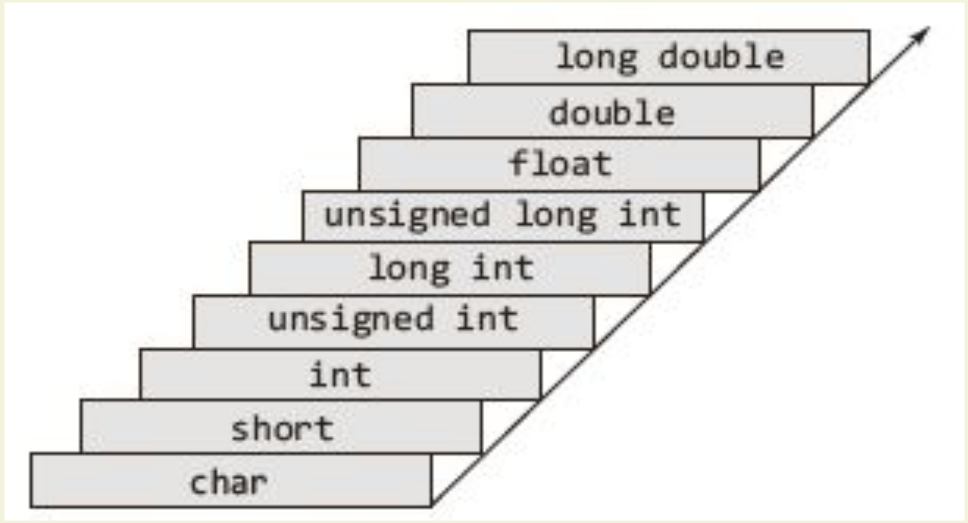
- For example :
  - a = b;
  - b, on the right-hand side of the assignment operator, is the quantity to be found at the address associated with b, i.e., an rvalue. a is assigned the value stored in the address associated with b. a, on the left-hand side, is the address at which the contents are altered 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: a = b;	
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

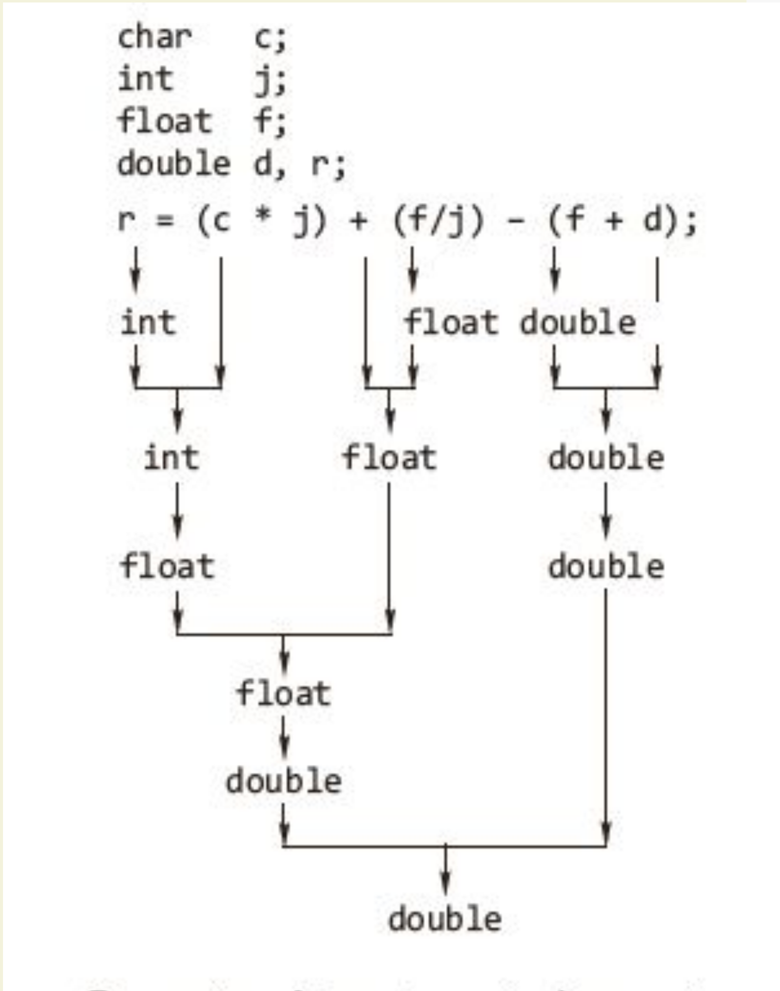
TYPE CONVERSION

- 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

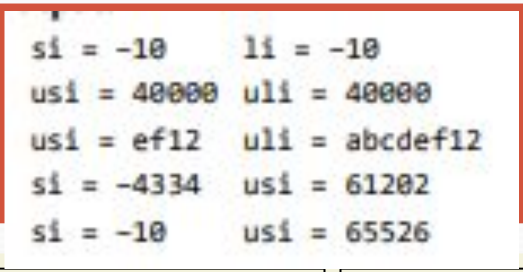
- float operands are converted to double.
- char or short (signed or unsigned) are converted to int (signed or unsigned).
- If any one operand is double, the other operand is also converted to double, and that is the type of the result; or
- If any one operand is long, the other operand is treated as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



```
char   c;
int    j;
float  f;
double d, r;

r = (c * j) + (f/j) - (f + d);
```

Conversion of types in a mixed expression


```
si = -10     li = -10
usi = 40000 uli = 40000
usi = ef12   uli = abcdef12
si = -4334   usi = 61202
si = -10     usi = 65526
```

```
#include <stdio.h>
int main()
{
 short int si;
 long int li;
 unsigned short int usi;
 unsigned long int uli;
 si = -10;
 li = si; /* sign extension - li should
be -10 */
 printf("si = %8hd li = %8ld\n",si,li);
 usi = 40000U;
/* unsigned decimal constant */
 uli = usi;
```

```
 /* zero extension - uli should be 40000
*/
 printf("usi = %8hu uli = %8lu\n",usi,uli);
 uli = 0xabcdef12; /* sets most bits ! */
usi = uli;
/* will truncate - discard more
sigfi cant bits */
 printf("usi = %8hx uli = %8lx\n",usi,uli);
 si = usi; /* preserves bit pattern */
 printf("si = %8hd usi = %8hu\n",si,usi);
 si = -10;
 usi = si; /* preserves bit pattern */
 printf("si = %8hd usi = %8hu\n",si,usi);
 return 0;
}
```

```
#include <stdio.h>
int main()
{
        int si;
        unsigned int usi;
        char ch = ‘a’;
       // Most significant bit will be zero
        si = ch; // will give small +ve integer
        usi = ch;
        printf(“c = %c\n si = %d\n usi = %u\n”, ch,
        si,usi);
        ch = ‘\377’; /* set all bits to 1 */ //(char)-1, which is typically '\377'.
        si = ch; /* sign extension makes negative */
        usi = ch;
        printf(“si = %d\n usi = %u\n”,si,usi);
        return 0;
}
```

Conversions of float and double

- ANSI C considers all floating point constants to be implicitly double precision, and operations involving such constants therefore take place in double precision.
- To force single precision arithmetic in ANSI C, use the f or F suffix on floating point constants. To force long double precision on constants, use the l or L suffix.
- For example, 3.14l is long double precision, 3.14 is double precision, and 3.14f is single precision in ANSI C.
- if you try to make a float variable exceed its limits For example, suppose you multiply 1.0e38f by 1000.0f (overflow) or divide 1.0e-37f by 1.0e8f
- (underflow), The result depends on the system.
- Either could cause the program to abort and to print a runtime error message. Or overflows may be replaced by a special value, such as the largest possible float value, underflows might be replaced by 0.
- Other systems may not issue warnings or may offer you a choice of responses. If this matter concerns you, check the rules for your system. If you can't find the information, don't be afraid of a little trial and error.

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;
  - fl oat imaginary c2= -5.0 * I;