# Introduction to C-programming

# Acknowledgement

The contents (figures, concepts, graphics, texts etc.) of the slides are gathered and utilized from the books mentioned and the corresponding PPTs available online:

**Books:**

1. **Let Us C,** Yashawant Kanetkar, BPB Publications.
2. **The C Programming Language,** B. W. Kernighan, D. Ritchie, Pearson Education India.

**Web References:**

1. **Problem Solving through Programming in C,** Anupam Basu, NPTEL Video Lectures. Link: https://nptel.ac.in/courses/106/105/106105171/
2. **Compile and Execute C Online (Link:** https://www.onlinegdb.com/)

# Chapter Objectives

- To Learn the elements of C program
- To Learn about variable declarations and data types
- To about Operators and Expressions
- To know about input output operations

# Chapter Topics

- Introduction to C programming
- History of C
- Operators and Expressions
- Data Input and Output

# Introduction to C

- C is a general purpose computing programming language.
- C was invented and was first implemented by *Dennis Ritchie* with the Unix Operating System in 1972.
- C is often called a *middle level* computer language.
- C is a *Structured Language*.

Dennis Ritchie

# History of the C language

- ALGOL 60 (1960): the first programming language with block structures, control constructs, and recursion possibilities.
- BCPL (Basic Combined Programming Language) developed by Martin Richards at Cambridge which foundation for many C elements.
- B (1970), developed by Ken Thompson at the Bell Laboratories for the first UNIX system.
- BCPL and B are type less languages whereas C offers a variety of data types.
- C (1972), written by Dennis Ritchie at Bell Labs for the implementation of UNIX. With the publication of "The C Programming Language" in 1978 by Kernighan and Ritchie evolved into the standard for C.JJ

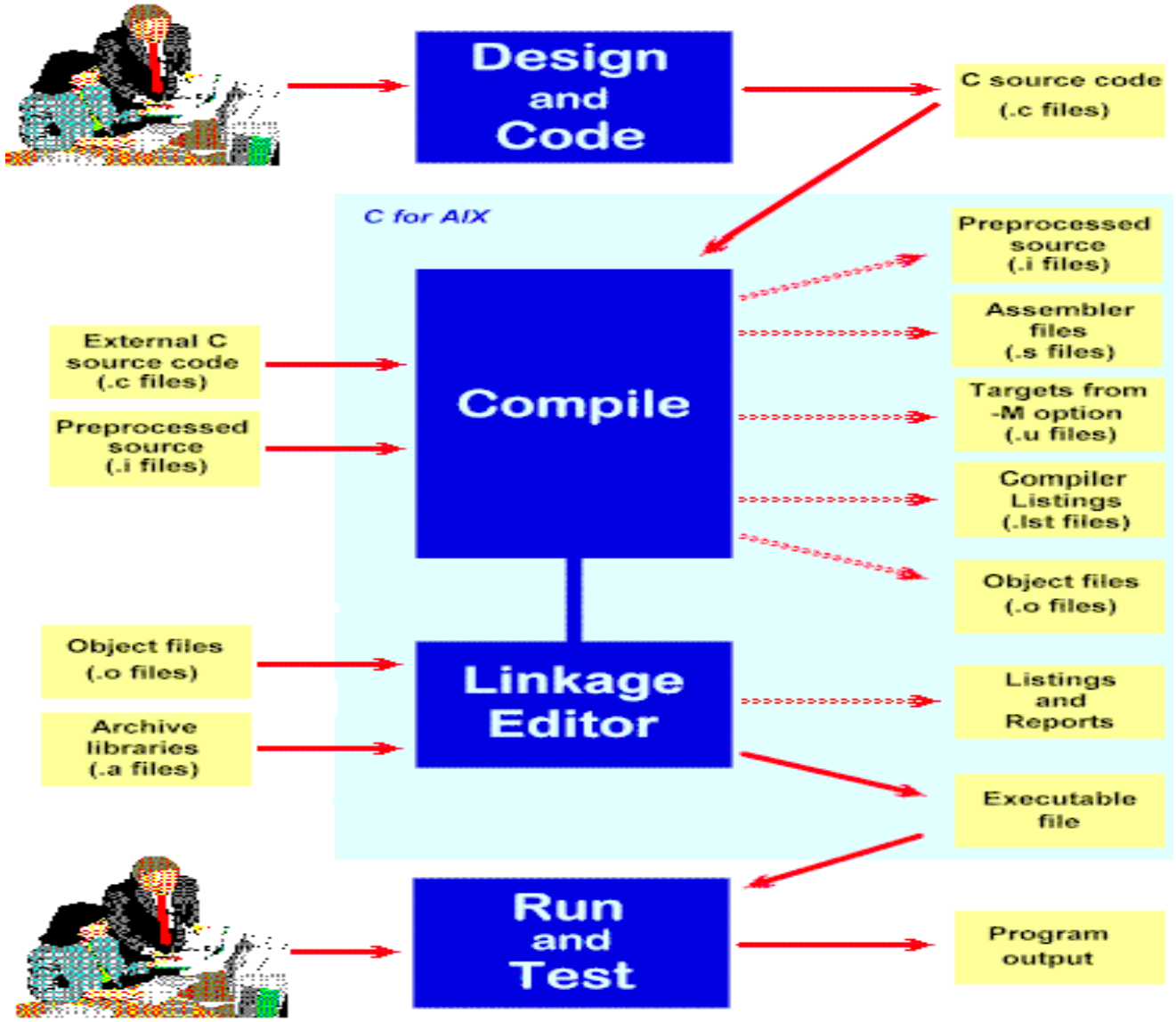
# Usage of C

- C's primary use is for ***system programming***, including implementing ***operating systems*** and ***embedded system*** applications.
- C has also been widely used to implement ***end-user*** applications, although as applications became larger much of that development shifted to other, higher-level languages.
- One consequence of C's wide acceptance and efficiency is that the ***compilers***, libraries, and interpreters of other higher-level languages are often implemented in C.
- You will be able to read and write code for a large number of platforms – even ***microcontrollers*.**

# Characteristics of C

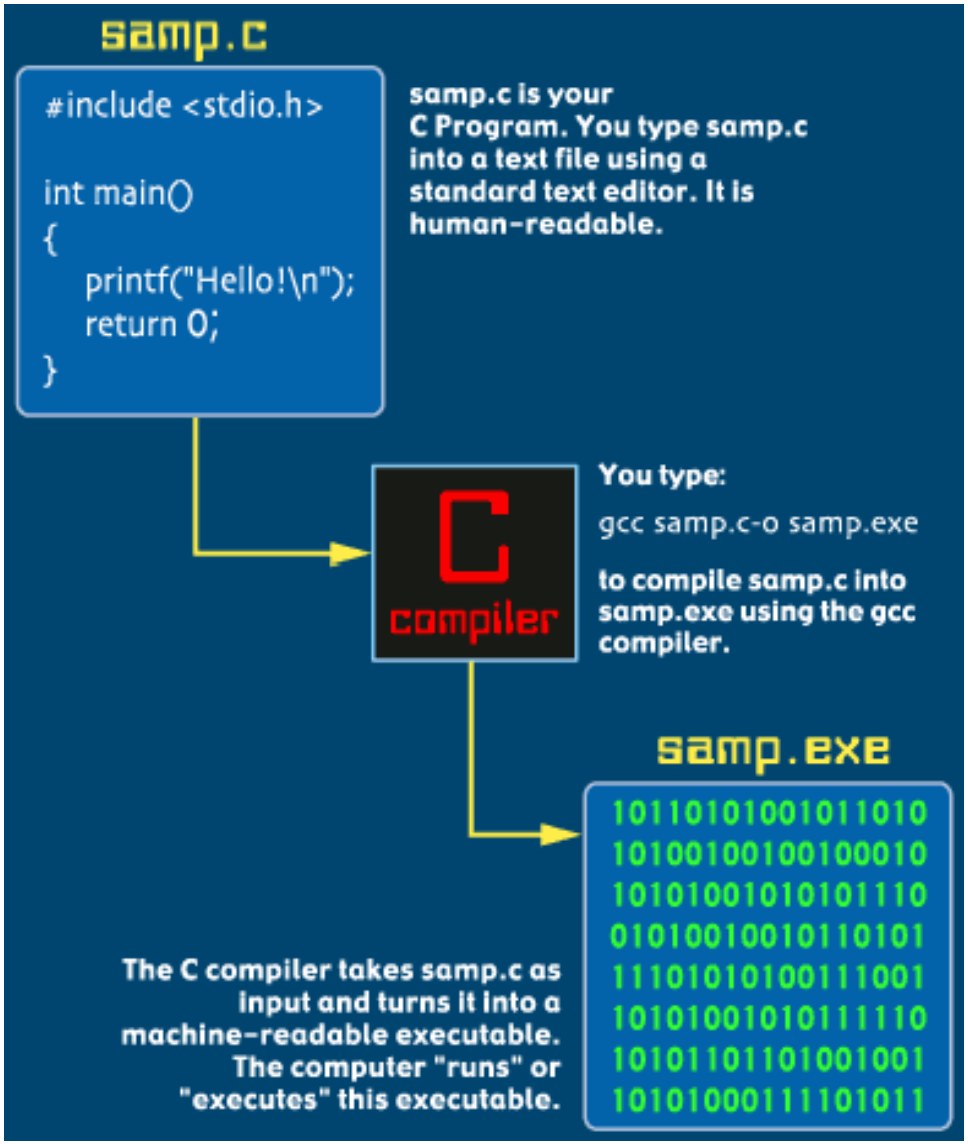
- Portability
  - ❑ Portability means it is easy to adapt software written for one type of computer or operating system to another type.
- Structured programming language
  - ❑ It make use of subroutines by making use of temporary variables.
- Control the memory efficiently
  - ❑ It makes the concept of pointers.
- Various application
  - ❑ Wide usage in all upcoming fields.

# 4 steps of development of C- program



# Compilers

- Commercial Compilers:
  - Microsoft Visual C++
  - Borland C++ Builder
- Freeware Compilers:
  - Borland C++ 5.5 Compiler, also called Turbo C
  - Dev-C++, free IDE (Integrated Development Environment) and compiler for C and C++
  - DJGPP, a DOS based Compiler for C, C++ and Pascal
  - LCC-Win32, free compiler for Windows
  - GCC, the most famous compiler



# The Simplest C Program

- Let's start with the simplest possible C program and use it both to understand the basics of C and the C compilation process.
- Type the following program into a standard text editor. Then save the program to a file named samp.c. If you leave off .c, you will probably get an error when you compile it, so make sure you remember the .c.

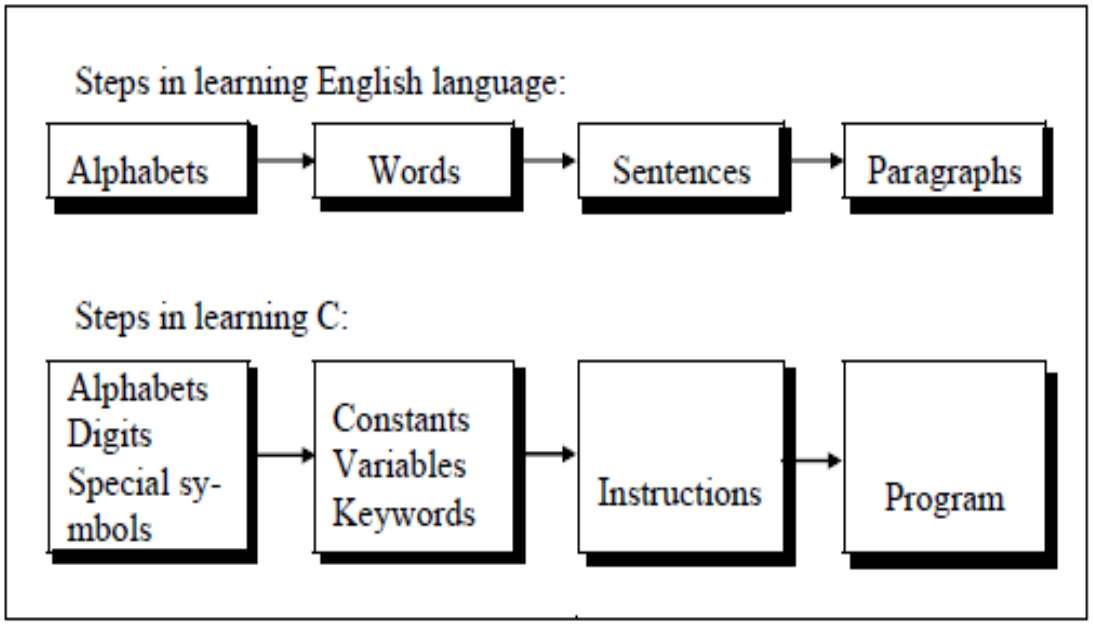
```
 #include <stdio.h>
void main()
{
  printf(“First program!\n");
}
```

# Compilation using ***gcc***

1. Use the command mkdir to create a new directory.
2. Use the text editor vi followed by the name of the file.Ex: vi sample.c
3. Use the insert mode to type the text.Use the Insert key or ‘I’.
4. Save the file using :wq. The file is now saved.
5. Compile using gcc followed by the name of the file. Ex: gcc sample.c
6. View the output using a.out file. Type ./a.out.

# Language

- Means of Communication
- Chinese – Chinese, English – Chinese, Nepali-Chinese



# General Structure of a C Program

- **Opening comment block**
  - Should always contain author's name, name of the file, and purpose of the program
- **Preprocessor directives**
  - include all header files for needed libraries
  - define any useful constants
- **Function prototypes**
  - Must be declared before the function is called
- **Main function**
  - Program execution begins at the start of main() and ends when it reaches the end of main()
  - Any additional functions called **main** are ignored.
- **Other function definitions**

# Internal Structure of a C Program

A C source program is a collection of one or more directives, declarations, and statements contained in one or more source files**.**

- **Statements**: Specify the action to be performed.
- **Directives**: Instruct the preprocessor to act on the text of the program.
- **Declarations**: Establish names and define linkage characteristics such as scope, data type, and linkage.
- **Definitions**: Are declarations that allocate storage for data objects or define a body for functions. An object definition allocates storage and may optionally initialize the object.

# General Programming Rules

- All C Statements are free-form
  - Can begin and end on any line and in any column
- C statements are always terminated with a semicolon “;”.
- Blank lines are ignored
- White space (blanks, tabs and new lines) must separate keywords from other things
- Comments –

  All text enclosed within “/* ----- */”

  Text on the same line following “//”

  Examples:

```
// This is a comment
/* So is
        this. */
```

## Variables and Constants

- Most important concept for problem solving using computers
- All temporary results are stored in terms of variables
  - The value of a variable can be changed.
  - The value of a constant do not change.
- Where are they stored?
  - In main memory.

Results of any calculations are stored in computer's memory cells.
To make the retrieval and usage of these values easy, these memory cells are given names.
Since the values stores in each memory location/cell may change the names given to those locations are called variable names. Refer book

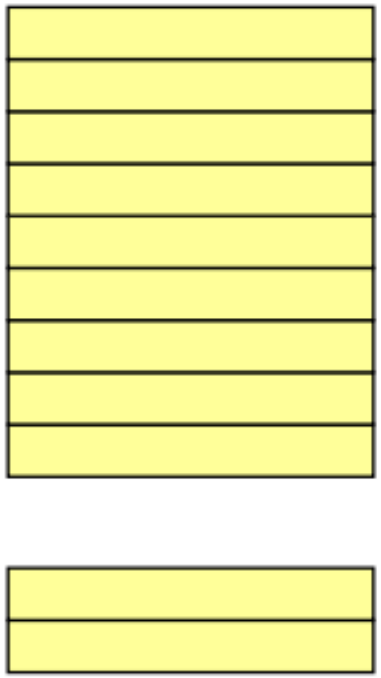
Source: Pallab Dasgupta,IIT Khragpur

## Contd.

- **How does memory look like (logically)?**
  - **As a list of storage locations, each having a unique address.**
  - **Variables and constants are stored in these storage locations.**
  - **A variable is like a *bin***
    - **The contents of the *bin* is the *value* of the variable**
    - **The variable name is used to refer to the value of the variable**
    - **A variable is mapped to a *location* of the memory, called its *address***

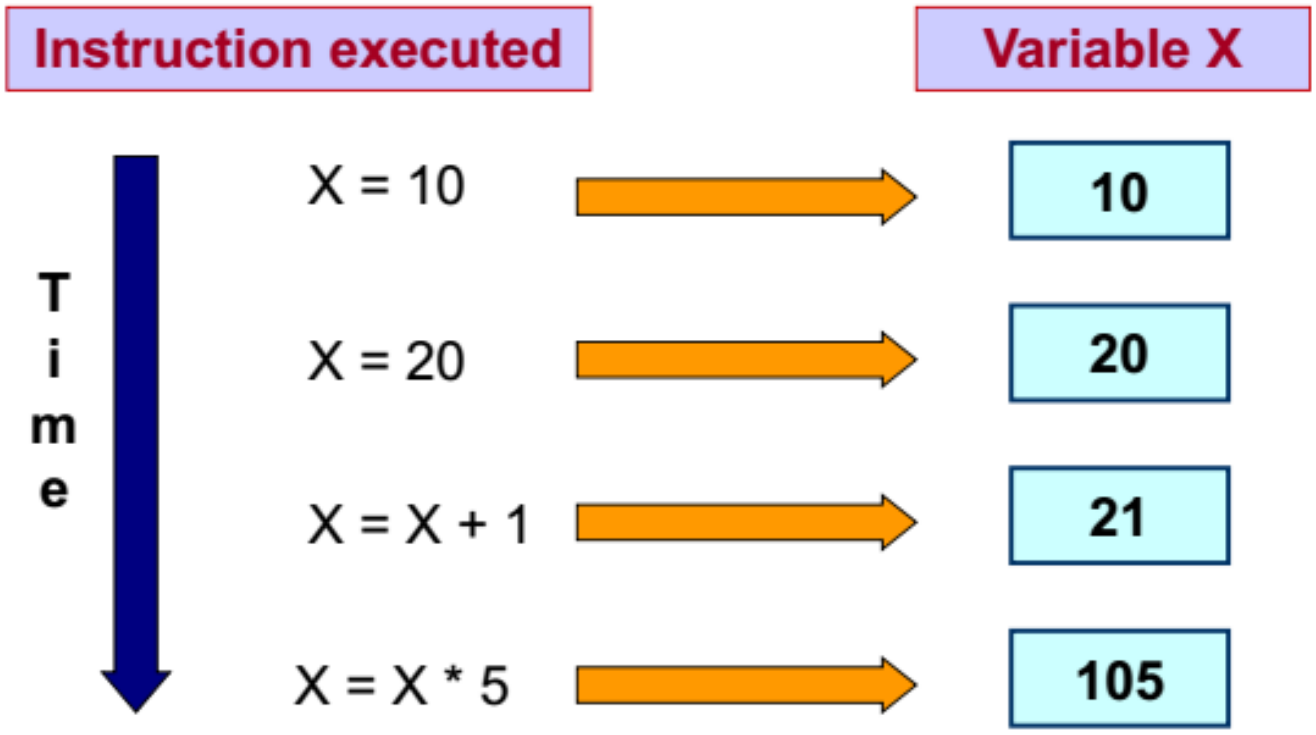
**Source: Pallab Dasgupta,IIT Khragpur**

# Memory map



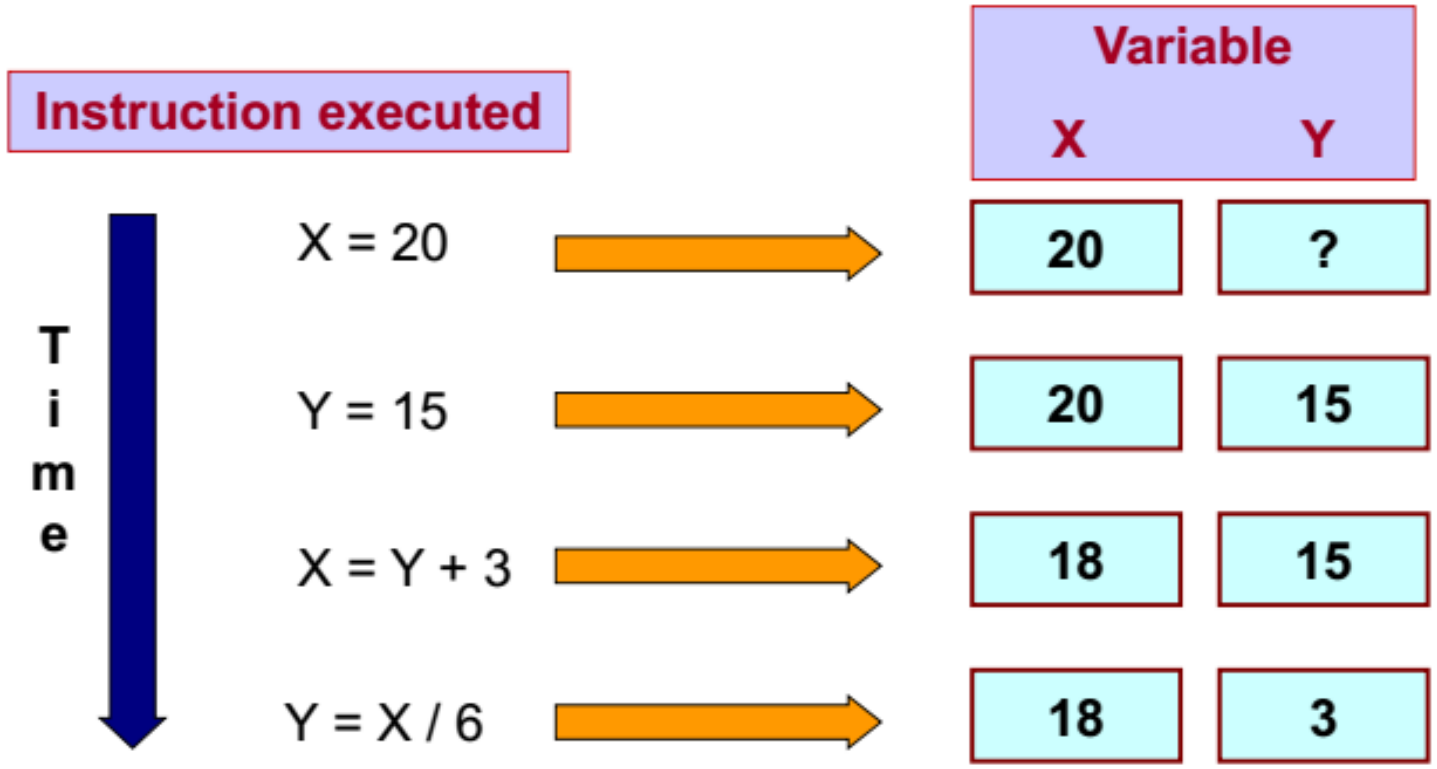

Every variable is mapped to a particular memory address

# Variables in Memory




Source: Pallab Dasgupta,IIT Khragpur

# Variables in Memory (contd.)




Source: Pallab Dasgupta,IIT Khragpur

# Types of C variables - Basic Data Types

- **Variable:** An entity that may vary during program execution. Variable names are names given to locations in memory. These locations can contain integer, real or character constants.

- **Variable type:** Depends on the types of constants that a particular language can handle. This is because a particular type of variable can hold only the same type of constant.

- For example, an integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant

# Types of C variables - Basic Data Types

**char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

**int:** As the name suggests, an int variable is used to store an integer.

**float:** It is used to store decimal numbers (numbers with floating point value) with single precision.

**double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

# Basic Data Types

Integral Types

| | |
|---|---|
| **char** | Stored as 8 bits. Unsigned 0 to 255.<br>Signed -128 to 127. |
| **short int** | Stored as 16 bits. Unsigned 0 to 65535.<br>Signed -32768 to 32767. |
| **int** | Same as either short or long int. |
| **long int** | Stored as 32 bits. Unsigned 0 to 4294967295.<br>Signed -2147483648 to 2147483647 |

***Unsigned binary numbers*** are, by definition, **positive numbers** and thus do not require an arithmetic sign.

An *m*-bit unsigned number represents all numbers in the range **0 to $2^m - 1$**. For example, the range of 8-bit unsigned binary numbers is from 0 to $255_{10}$ in decimal.

# Data types

| Type | Size (bits) | Range |
|---|---|---|
| char or signed | 8 | -128 to 127 |
| char unsigned | 8 | 0 to 256 |
| char | 16 | -32768 to 32767 |
| int or signed | 16 | 0 to 65535 |
| int unsigned | 8 | -128 to 127 |
| int | 8 | 0 to 255 |
| short int or signed | 32 | -2147483648 to 1247483647 |
| short int unsigned | 32 | 0 to 4294967295 |
| short int | 32 | 3.4 E –38 to 3.4 E + 38 |
| long int or signed | 64 | 1.7 E –308 to 1.7 E + 308 |
| long int unsigned | 80 | 3.4 E –4832 to 1.1 E +4932 |
| long int | | |
| float | | |
| double | | |
| long | | |
| double | | |

# Rules For Constructing Variable Names

- A variable name is any combination of 1 to 31 alphabets, digits or underscores.
- Do not create unnecessarily long variable names as it adds to your typing effort.
- The first character in the variable name must be an alphabet or underscore.
- No commas or blanks are allowed within a variable name.
- No special symbol other than an underscore (as in gross_sal) can be used in a variable name.
- Ex.: si_int, m_hra, pop_e_89
- These rules remain same for all the types of primary and secondary variables. Naturally, the question follows... how is C able to differentiate between these variables? This is a rather simple

# Variable Declarations in C

- C compiler is able to distinguish between the variable names by making it compulsory for you to declare the type of any variable name that you wish to use in a program. This type declaration is done at the beginning of the program. Following are the examples of type declaration statements:

Ex.: int si, m_hra ;

float bassal ;

char code ;

- All variables must be declared before using them in a program.
- Variable declarations can also be used to initialize the variable.
- We can declare variables in a single statement with the list of variables separated by commas.
  - int lower, upper, step;
  - float fahr, celsius;
  - int i=0;
  - char backslash = '\\'

# Characters Representation

- Characters are represented at the machine level as an integer
- Computers use ASCII (American Standard Code for Information Interchange) code
- The values used as code range from 0 to 255
- A-Z (upper case) are represented by 65-90
- a-z (lower case) are represented by 97-122
- Constants can be defined using the *#define* construct (symbolic constant)
  - #define LOWER 0
  - #define UPPER 300
  - #define STEP 20

# C Tokens

- Character Set :
  - Letters : A..Z, a...z and Digits : 0...9
  - Special characters : , . : ; ? ’ “ ! | / \ ~ -_ $ % # & ^ * + - < > ( ) { } [ ]
  - White space : blank space, horizontal tab, vertical tab,carriage return, new line, form feed.
- C tokens : individual units.
  - Keyword – float, while, for, int,….
  - Identifier – main( ) , amount, sum, …
  - Constants – -13.5, 500, …
  - Strings – “ABC”, “MCA”, …
  - Operators – + - * % …
  - Special Symbols – [ ] { }…

# C Tokens

- There are several rules that you must follow when naming constants and variables:

| Names... | Example |
|---|---|
| CANNOT start with a number | 2i |
| CAN contain a number elsewhere | h2o |
| CANNOT contain any arithmetic operators... | r*s+t |
| CANNOT contain any other punctuation marks... | #@x%£!!a |
| CAN contain or begin with an underscore | _height_ |
| CANNOT be a C keyword | struct |
| CANNOT contain a space | im stupid |
| CAN be of mixed cases | XSquared |

- All variables must be declared before using them in a program and Variable declarations can also be used to initialize the variable. (not good practice)
- We can declare variables in a single statement with the list of variables separated by commas. Eg. int lower, upper, step;

# Identifiers, Constants & Variables

Identifiers

- Refers to names of variables, functions,…
- User defined names.
- Uppercase and lowercase.

Constants

- Fixed values.
- Does not changes during execution of program.
- Numeric constant – Integer (decimal, octal, hexadecimal) and Real
- Character constant :
  - Single character constant
  - String constant
  - Backslash character constant

Variables

- Data name used to store data value.
- May take different values at different times during execution.
- Chosen by the programmer.
- Letters, digits and '_' are allowed.

# Keywords used in C

- These are reserved words of the C language
- Fixed meaning, cannot be changed
- Basic building block
- Lowercase

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

# FIRST C Program

```
/* Calculation of simple interest */
main( )
{
int p, n ;
float r, si ;
p = 1000 ;
n = 3 ;
r = 8.5 ;
/* formula for simple interest */
si = p * n * r / 100 ;
printf ( "%f" , si ) ;
}
```

- Each instruction in a C program is written as a separate statement. Therefore a complete C program would comprise of a series of statements.
- The statements in a program must appear in the same order in which we wish them to be executed; unless of course the logic of the problem demands a deliberate ‘jump’ or transfer of control to a statement, which is out of sequence.
- Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.
- All statements are entered in small case letters.
- C has no specific rules for the position at which a statement is to be written. That’s why it is often called a free-form language.
- Every C statement must end with a ;. Thus ; acts as a statement terminator.

- Now a few useful tips about the program...

− Comment about the program should be enclosed within /* */. For example, the first two statements in our program are comments.

− Though comments are not necessary, it is a good practice to begin a program with a comment indicating the purpose of the program, its author and the date on which the program was written.

− Any number of comments can be written at any place in the program. For example, a comment can be written before the statement, after the statement or within the statement as shown below:

```
/* formula */ si = p * n * r / 100 ;
si = p * n * r / 100 ; /* formula */
si = p * n * r / /* formula */ 100 ;
```

− Sometimes it is not so obvious as to what a particular statement in a program accomplishes. At such times it is worthwhile mentioning the purpose of the statement (or a set of statements) using a comment. For example:

```
/* formula for simple interest */
si = p * n * r / 100 ;
```

− The normal language rules do not apply to text written within /* .. */. Thus we can type this text in small case, capital or a combination. This is because the comments are solely given for the understanding of the programmer or the fellow programmers and are completely ignored by the compiler.

− Comments cannot be nested. For example,
/* Cal of SI /* Author sam date 01/01/2002 */ */is invalid.

− A comment can be split over more than one line, as in,
/* This is
a jazzy
comment */

Such a comment is often called a multi-line comment.

− main( ) is a collective name given to a set of statements. This name has to be main( ), it cannot be anything else. All statements that belong to main( ) are enclosed within a pair of braces { } as shown below.

```
main( )
{
statement 1 ;
statement 2 ;
statement 3 ;
}
```

− Technically speaking main( ) is a function. Every function has a pair of parentheses ( ) associated with it.

− Any variable used in the program must be declared before using it. For example,

```
int p, n ;
float r, si ;
```

− Any C statement always ends with a ;

For example,

```
float r, si ;
r = 8.5 ;
```

− In the statement,

si = p * n * r / 100 ;

* and / are the arithmetic operators. The arithmetic operators available in C are +, -, * and /. C is very rich in operators. There are about 45 operators available in C. Surprisingly there is no operator for exponentiation... a slip, which can be forgiven considering the fact that C has been developed by an individual, not by a committee.

− Once the value of si is calculated it needs to be displayed on the screen. Unlike other languages, C does not contain any instruction to display output on the screen. All output to screen is achieved using readymade library functions. One such function is printf( ). We have used it display on the screen the value contained in si.

The general form of printf( ) function is,

printf ( "<format string>", <list of variables> ) ;

<format string> can contain,

%f for printing real values

%d for printing integer values

%c for printing character values

In addition to format specifiers like %f, %d and %c the format string may also contain any other characters. These characters are printed as they are when the printf( ) is executed.

Following are some examples of usage of printf( ) function:

```
printf ( "%f", si ) ;
printf ( "%d %d %f %f", p, n, r, si ) ;
printf ( "Simple interest = Rs. %f", si ) ;
printf ( "Prin = %d \nRate = %f", p, r ) ;
```

The output of the last statement would look like this...

Prin = 1000

Rate = 8.5

What is '\n' doing in this statement? It is called newline and it takes the cursor to the next line. Therefore, you get the output split over two lines. '\n' is one of the several Escape Sequences available in C.

printf( ) can not only print values of variables, it can also print the result of an expression. An expression is nothing but a valid combination of constants, variables and operators. Thus, 3, 3 + 2, c and a + b * c – d all are valid expressions. The results of these expressions can be printed as shown below:

```
printf ( "%d %d %d %d", 3, 3 + 2, c, a + b * c – d ) ;
```

Note that 3 and c also represent valid expressions.

# Format Specifiers

| Format Specifier | Meaning |
|---|---|
| %c | Single character |
| %d | Signed decimal integer |
| %e | Floating-point number, e notation |
| %E | Floating-point number, E notation |
| %f | Floating-point number, decimal notation |
| %g | Causes %f or %e to be used, whichever is shorter |
| %G | Causes %f or %E to be used, whichever is shorter |
| %i | Signed decimal integer |
| %o | Unsigned octal integer |
| %p | Pointer |
| %s | Character string |
| %u | Unsigned decimal integer |
| %x | Unsigned hex integer using digits 0-f |
| %X | Unsigned hex integer using digits 0-F |
| %% | Print a percent sign |

# contd…Format Specifiers

| Format Specifier | Meaning |
|---|---|
| %i | Signed decimal integer |
| %o | Unsigned octal integer |
| %p | Pointer |
| %s | Character string |
| %u | Unsigned decimal integer |
| %x | Unsigned hex integer using digits 0-f |
| %X | Unsigned hex integer using digits 0-F |
| %% | Print a percent sign |

# Receiving Input -Formatted Input Function

- In the program discussed above we assumed the values of p, n and r to be 1000, 3 and 8.5. Every time we run the program we would get the same value for simple interest. If we want to calculate simple interest for some other set of values then we are required to make the relevant change in the program, and again compile and execute it. Thus the program is not general enough to calculate simple interest for any set of values without being required to make a change in the program.
- To make the program general the program itself should ask the user to supply the values of p, n and r through the keyboard during execution. This can be achieved using a function called scanf( ).
- This function is a counter-part of the printf( ) function. printf( ) outputs the values to the screen whereas scanf( ) receives them from the keyboard.

# Receiving Input -Formatted Input Function

This is illustrated in the program shown below.

```
/* Calculation of simple interest */
/* Author gekay Date 25/05/2004 */
main( )
{
int p, n ;
float r, si ;
printf ( "Enter values of p, n, r" ) ;
scanf ( "%d %d %f", &p, &n, &r ) ;
si = p * n * r / 100 ;
printf ( "%f" , si ) ;
}
```

# Receiving Input -Formatted Input Function

- The first printf( ) outputs the message 'Enter values of p, n, r' on the screen. Here we have not used any expression in printf( ) which means that using expressions in printf( ) is optional.
- Note that the ampersand (&) before the variables in the scanf( ) function is a must. & is an 'Address of' operator. It gives the location number used by the variable in memory. When we say &a, we are telling scanf( ) at which memory location should it store the value supplied by the user from the keyboard.
- Note that a blank, a tab or a new line must separate the values supplied to scanf( ). Note that a blank is creating using a spacebar, tab using the Tab key and new line using the Enter key. This is shown below:

  Ex.: The three values separated by blank
  1000 5 15.5
  Ex.: The three values separated by tab.
  1000 5 15.5
  Ex.: The three values separated by newline.
  1000
  5
  15.5

# Receiving Input -Formatted Input Function

```
/* Just for fun. Author: Bozo */
main( )
{
int num ;
printf ( "Enter a number" ) ;
scanf ( "%d", &num ) ;
printf ( "Now I am letting you on a secret..." ) ;
printf ( "You have just entered the number %d", num ) ;
}
```

# Formatted Input Function

## scanf( )

- int scanf(char *format, args....) -- reads from stdin and puts input in address of variables specified in argument list.

- Returns, number of charecters read.
- The address of variable or a pointer to one is required by scanf. Example: scanf(``%d",&i);
- We can just give the name of an array or string to scanf since this corresponds to the start address of the array/string.

Example:

```
char string[80];
scanf(``%s",string);
```

# Formatted Output Function

## printf()

- The printf function is defined as follows:
  int printf(char *format, arg list ...) --
  prints to stdout the list of arguments according specified format string.
  Returns number of characters printed.
- The format string has 2 types of object:
- *ordinary characters* -- these are copied to output.
- *conversion specifications* -- denoted by % and listed in Table.

# C Instructions

There are basically three types of instructions in C:

a) Type Declaration Instruction
b) Arithmetic Instruction
c) Control Instruction

The purpose of each of these instructions is given below:

Type declaration instruction

− To declare the type of variables used in a C program.

Arithmetic instruction

− To perform arithmetic operations between constants and variables.

Control instruction

− To control the sequence of execution of various statements in a C program

The elementary C programs would usually contain only the type declaration and the arithmetic instructions;

# Type Declaration Instruction

- This instruction is used to declare the type of variables being used in the program. Any variable used in the program must be declared before using it in any statement. The type declaration statement is written at the beginning of main( ) function.

  Ex.: int bas ;

  float rs, grosssal ;

  char name, code ;

- There are several subtle variations of the type declaration instruction. These are discussed below:

(a)While declaring the type of variable we can also initialize it as shown below.

```
int i = 10, j = 25 ;
float a = 1.5, b = 1.99 + 2.4 * 1.44
```

# Type Declaration Instruction

- The order in which we define the variables is sometimes important sometimes not. For example,

  int i = 10, j = 25 ;

  is same as

  int j = 25, i = 10 ;

  However,

  float a = 1.5, b = a + 3.1 ;

  is alright, but

  float b = a + 3.1, a = 1.5 ;

  is not. This is because here we are trying to use a variable even before defining it.

(c)The following statements would work

int a, b, c, d ;

a = b = c = 10 ;

However, the following statement would not work

int a = b = c = d = 10 ;

Once again we are trying to use b (to assign to a) before defining it.

# Arithmetic Instruction

- A C arithmetic instruction consists of a variable name on the left hand side of = and variable names & constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators like +, -, *, and /.

  Ex.: int ad ;

  float kot, deta, alpha, beta, gamma ;

  ad = 3200 ;

  kot = 0.0056 ;

  deta = alpha * beta / gamma + 3.2 * 2 / 5 ;
- Here, *, /, -, + are the arithmetic operators. = is the assignment operator. 2, 5 and 3200 are integer constants. 3.2 and 0.0056 are real constants. ad is an integer variable. kot, deta, alpha, beta, gamma are real variables.
- The variables and constants together are called ‘operands’ that are operated upon by the ‘arithmetic operators’ and the result is assigned, using the assignment operator, to the variable on left-hand side

# Arithmetic Instruction

A C arithmetic statement could be of three types. These are as follows:

(a) Integer mode arithmetic statement - This is an arithmetic statement in which all operands are either integer variables or integer constants.

```
Ex.: int i, king, issac, noteit ;
i = i + 1 ;
king = issac * 234 + noteit - 7689 ;
```

- Real mode arithmetic statement - This is an arithmetic statement in which all operands are either real constants or real variables.

```
Ex.: float qbee, antink, si, prin, anoy, roi ;
qbee = antink + 23.123 / 4.5 * 0.3442 ;
si = prin * anoy * roi / 100.0 ;
```

- Mixed mode arithmetic statement - This is an arithmetic statement in which some of the operands are integers and some of the operands are real.

```
Ex.: float si, prin, anoy, roi, avg ;
int a, b, c, num ;
si = prin * anoy * roi / 100.0 ;
avg = ( a + b + c + num ) / 4 ;
```

# Arithmetic Instruction

- It is very important to understand how the execution of an arithmetic statement takes place. Firstly, the right hand side is evaluated using constants and the numerical values stored in the variable names. This value is then assigned to the variable on the left-hand side.
- Though Arithmetic instructions look simple to use one often commits mistakes in writing them. Let us take a closer look at these statements. Note the following points carefully.
- C allows only one variable on left-hand side of =. That is, z = k * l is legal, whereas k * l = z is illegal.
- In addition to the division operator C also provides a modular division operator. This operator returns the remainder on dividing one integer with another. Thus the expression 10 / 2 yields 5, whereas, 10 % 2 yields 0. Note that the modulus operator (%) cannot be applied on a float. Also note that on using % the sign of the remainder is always same as the sign of the numerator. Thus –5 % 2 yields –1, whereas, 5 % -2 yields 1.

# Arithmetic Instruction

- An arithmetic instruction is often used for storing character constants in character variables.

```
char a, b, d ;
a = 'F' ;
b = 'G' ;
d = '+' ;
```

- When we do this the ASCII values of the characters are stored in the variables. ASCII values are used to represent any character in memory. The ASCII values of ‘F’ and ‘G’ are 70 and 71 (refer the ASCII Table in Appendix E).
- Arithmetic operations can be performed on ints, floats and chars.
- Thus the statements, are perfectly valid, since the addition is performed on the ASCII values of the characters and not on characters themselves. The ASCII values of ‘a’ and ‘b’ are 97 and 98, and hence can definitely be added.

```
char x, y ;
int z ;
x = 'a' ; y = 'b' ; z = x + y ;
```

# Arithmetic Instruction

- No operator is assumed to be present. It must be written explicitly. In the following example, the multiplication operator after b must be explicitly written.
  a = c.d.b(xy) usual arithmetic statement
  b = c * d * b * ( x * y ) C statement
- Unlike other high level languages, there is no operator for performing exponentiation operation. Thus following statements are invalid.
  a = 3 ** 2 ; b = 3 ^ 2 ;
- If we want to do the exponentiation we can get it done this way:

```
#include <math.h>
main( )
{
int a;
a = pow ( 3, 2 ) ;   //math.h
printf ( "%d", a ) ;  //stdio.h
T= First(2,3);
}
```

Function – function declaration
function definition
function calling
In-built/Predefined, user defined

```
int First(int, int); - declaration
int First( a, b)
{
 int c;
 c= a+b;
  return c;
```

- Here pow( ) function is a standard library function. It is being used to raise 3 to the power of 2. #include <math.h> is a preprocessor directive.

# Integer and Float Conversions

- In order to effectively develop C programs, it will be necessary to understand the rules that are used for the implicit conversion of floating point and integer values in C. These are mentioned below.

(a) An arithmetic operation between an integer and integer always yields an integer result.

(b)An operation between a real and real always yields a real result.

(c)An operation between an integer and real always yields a real result. In this operation the integer is first promoted to a real and then the operation is performed. Hence the result is real.

| Operation | Result | Operation | Result |
|---|---|---|---|
| 5 / 2 | 2 | 2 / 5 | 0 |
| 5.0 / 2 | 2.5 | 2.0 / 5 | 0.4 |
| 5 / 2.0 | 2.5 | 2 / 5.0 | 0.4 |
| 5.0 / 2.0 | 2.5 | 2.0 / 5.0 | 0.4 |

# Type Conversion in Assignments

- It may so happen that the type of the expression and the type of the variable on the left-hand side of the assignment operator may not be same.
- In such a case the value of the expression is promoted or demoted depending on the type of the variable on left-hand side of =.
- For example, consider the following assignment statements.

  ```
  int i ;
  float b ;
  i = 3.5 ;
  b = 30 ;
  ```
- Here in the first assignment statement though the expression's value is a float (3.5) it cannot be stored in i since it is an int. In such a case the float is demoted to an int and then its value is stored. Hence what gets stored in i is 3. Exactly opposite happens in the next statement. Here, 30 is promoted to 30.000000 and then stored in b, since b being a float variable cannot hold anything except a float value.

# Type Conversion in Assignments

- Instead of a simple expression used in the above examples if a complex expression occurs, still the same rules apply. For example, consider the following program fragment.

  ```
  float a, b, c ;
  int s ;
  s = a * b * c / 100 + 32 / 4 - 3 * 1.1 ;
  ```

- Here, in the assignment statement some operands are ints whereas others are floats. As we know, during evaluation of the expression the ints would be promoted to floats and the result of the expression would be a float. But when this float value is assigned to s it is again demoted to an int and then stored in s.

# Type Conversion in Assignments

| Arithmetic Instruction | Result | Arithmetic Instruction | Result |
|---|---|---|---|
| k = 2 / 9 | 0 | a = 2 / 9 | 0.0 |
| k = 2.0 / 9 | 0 | a = 2.0 / 9 | 0.2222 |
| k = 2 / 9.0 | 0 | a = 2 / 9.0 | 0.2222 |
| k = 2.0 / 9.0 | 0 | a = 2.0 / 9.0 | 0.2222 |
| k = 9 / 2 | 4 | a = 9 / 2 | 4.0 |
| k = 9.0 / 2 | 4 | a = 9.0 / 2 | 4.5 |
| k = 9 / 2.0 | 4 | a = 9 / 2.0 | 4.5 |
| k = 9.0 / 2.0 | 4 | a = 9.0 / 2.0 | 4.5 |

- Observe the results of the arithmetic statements shown in Figure. It has been assumed that k is an integer variable and a is a real variable.
- Note that though the following statements give the same result, 0, the results are obtained differently.

  k = 2 / 9 ;

  k = 2.0 / 9 ;
- In the first statement, since both 2 and 9 are integers, the result is an integer, i.e. 0. This 0 is then assigned to k. In the second statement 9 is promoted to 9.0 and then the division is performed. Division yields 0.222222. However, this cannot be stored in k, k being an int. Hence it gets demoted to 0 and then stored in k.

# Hierarchy of Operations

- While executing an arithmetic statement, which has two or more operators, we may have some problems as to how exactly does it get executed. For example, does the expression 2 * x - 3 * y correspond to (2x)-(3y) or to 2(x-3y)? Similarly, does A / B * C correspond to A / (B * C) or to (A / B) * C? To answer these questions satisfactorily one has to understand the 'hierarchy' of operations. The priority or precedence in which the operations in an arithmetic statement are performed is called the hierarchy of operations. The hierarchy of commonly used operators is shown in Figure

| Priority | Operators | Description |
|---|---|---|
| 1st | * / % | multiplication, division, modular division |
| 2nd | + - | addition, subtraction |
| 3rd | = | assignment |

# Hierarchy of Operations

- Now a few tips about usage of operators in general.

(a) If there are more than one set of parentheses, the operations within the innermost parentheses would be performed first, followed by the operations within the second innermost pair and so on.

(b) We must always remember to use pairs of parentheses. A careless imbalance of the right and left parentheses is a common error. Best way to avoid this error is to type ( ) and then type an expression inside it.

# Hierarchy of Operations

Example: Determine the hierarchy of operations and evaluate the following expression:

i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

Stepwise evaluation of this expression is shown below:

| | |
|---|---|
| i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8 | |
| i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8 | operation: * |
| i = 1 + 4 / 4 + 8 - 2 + 5 / 8 | operation: / |
| i = 1 + 1+ 8 - 2 + 5 / 8 | operation: / |
| i = 1 + 1 + 8 - 2 + 0 | operation: / |
| i = 2 + 8 - 2 + 0 | operation: + |
| i = 10 - 2 + 0 | operation: + |
| i = 8 + 0 | operation : - |
| i = 8 | operation: + |

Note that 6 / 4 gives 1 and not 1.5. This so happens because 6 and 4 both are integers and therefore would evaluate to only an integer constant. Similarly 5 / 8 evaluates to zero, since 5 and 8 are integer constants and hence must return an integer value

# Hierarchy of Operations

Example 1.2: Determine the hierarchy of operations and evaluate the following expression:

kk = 3 / 2 * 4 + 3 / 8 + 3

Stepwise evaluation of this expression is shown below:

| | |
|---|---|
| kk = 3 / 2 * 4 + 3 / 8 + 3 | |
| kk = 1 * 4 + 3 / 8 + 3 | operation: / |
| kk = 4 + 3 / 8 + 3 | operation: * |
| kk = 4 + 0 + 3 | operation: / |
| kk = 4 + 3 | operation: + |
| kk = 7 | operation: + |

Note that 3 / 8 gives zero, again for the same reason mentioned in the previous example.

All operators in C are ranked according to their precedence. And mind you there are as many as 45 odd operators in C, and these can affect the evaluation of an expression in subtle and unexpected ways if we aren't careful. Unfortunately, there are no simple rules that one can follow, such as “BODMAS” that tells algebra students in which order does an expression evaluate.

# Associativity of Operators

- When an expression contains two operators of equal priority the tie between them is settled using the associativity of the operators. Associativity can be of two types—Left to Right or Right to Left.
- Left to Right associativity means that the left operand must be unambiguous. Unambiguous in what sense? It must not be involved in evaluation of any other sub-expression.
- Similarly, in case of Right to Left associativity the right operand must be unambiguous. Let us understand this with an example.

  Consider the expression

  a = 3 / 2 * 5 ;
- Here there is a tie between operators of same priority, that is between / and *. This tie is settled using the associativity of / and *. But both enjoy Left to Right associativity.

| Operator | Left | Right | Remark |
|---|---|---|---|
| / | 3 | 2 or 2 * 5 | Left operand is unambiguous, Right is not |
| * | 3 / 2 or 2 | 5 | Right operand is unambiguous, Left is not |

# Associativity of Operators

| Operator | Left | Right | Remark |
|---|---|---|---|
| / | 3 | 2 or 2 * 5 | Left operand is unambiguous, Right is not |
| * | 3 / 2 or 2 | 5 | Right operand is unambiguous, Left is not |

- Since both / and * have L to R associativity and only / has unambiguous left operand (necessary condition for L to R associativity) it is performed earlier.
- Consider one more expression
- a = b = 3 ;
- Here both assignment operators have the same priority and same associativity (Right to Left). Consider Figure

| Operator | Left | Right | Remark |
|---|---|---|---|
| = | a | b or b = 3 | Left operand is unambiguous, Right is not |
| = | b or a = b | 3 | Right operand is unambiguous, Left is not |

# Associativity of Operators

| Operator | Left | Right | Remark |
|---|---|---|---|
| = | a | b or b = 3 | Left operand is unambiguous, Right is not |
| = | b or a = b | 3 | Right operand is unambiguous, Left is not |

- Since both = have R to L associativity and only the second = has unambiguous right operand (necessary condition for R to L associativity) the second = is performed earlier.

# Associativity of Operators

- Consider yet another expression

z = a * b + c / d ;

- Here * and / enjoys same priority and same associativity (Left to Right). Figure 1.12 shows for each operator which operand is unambiguous and which is not.
- Here since left operands for both operators are unambiguous Compiler is free to perform * or / operation as per its convenience since no matter which is performed earlier the result would be same.

| Operator | Left | Right | Remark |
|---|---|---|---|
| * | a | b | Both operands are unambiguous |
| / | c | d | Both operands are unambiguous |

# Rules of Associativity

| Operator | Associativity |
|---|---|
| ( ) [ ] -> | Left to right |
| ! ~ ++ -- - (type) * & sizeof() | Right to left |
| * / % | Left to right |
| + - | Left to right |
| << >> | Left to right |
| < <= > >= | Left to right |
| == != | Left to right |
| & | Left to right |
| \| | Left to right |
| ^ | Left to right |
| && | Left to right |
| \|\| | Left to right |
| ?: | Right to left |
| = += -= /= %= &= \|= ^= | Right to left |
| , | Left to right |

# Control Instructions in C

- The ‘Control Instructions’ enable us to specify the order in which the various instructions in a program are to be executed by the computer.
- The control instructions determine the ‘flow of control’ in a program. There are four types of control instructions in C.

**(a) Sequence Control Instruction:** The Sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program.

**(b)Repetition or Loop Control Instruction:** The Loop control instruction helps computer to execute a group of statements repeatedly.

**(c) Selection or Decision Control Instruction:**

**(d) Case Control Instruction:**

Decision and Case control instructions allow the computer to take a decision as to which instruction is to be executed next.

# Control Instructions in C

- Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation.
- This kind of situation is dealt in C programs using a decision control instruction.
- C has three major decision making instructions—
  - the if statement,
  - the if-else statement, and
  - the switch statement.

# Thank You!