**UNIT – I**

**What is Computer?**

* The term computer is derived from the word “compute” which means “to calculate”.
* A computer is an electronic machine that accepts input, processes it and produces output according to the specified rules.
* Computer is an electronic data processing device which accepts and stores data input, processes the data input and generates the output in a required format.
* It is a device that can perform counting, comparison, and manipulation operations at a very high speed.

**INTRODUCTION TO DIGITAL COMPUTER**

A computer which understands only binary digits i.e. ‘1’ for high volts and ‘0’ for low volts is called **digital computer**. The analog to digital convertor (ADC) converts electrical signals to binary digits whereas the digital to analog converter (DAC) converts binary digits to their equivalent electrical signals as shown below.

****

* Digital computers are the devices capable of solving problems by processing information in discrete form only.
* It operates on data, including magnitudes, letters, and symbols, that are expressed in binary form—i.e., using only the two digits 0 and 1.

**Characteristics of Digital Computer:**

All the digital computers have certain common characteristics irrespective of their types & sizes. Computers can be programmed to perform complex tasks due to following characteristics:

1. **Speed:** A digital computer is an extremely fast machine due to rapid processing as compared to human processing. The computer based processing goes far beyond that of seconds i.e. milliseconds (**10-3** seconds), microseconds (**10-6** seconds), nanoseconds (**10-9** seconds) and picoseconds (**10-12** seconds).
2. **Accuracy:** Digital computers are highly accurate machines and they never make mistakes.
3. **Reliability:** Digital computers can be constructed and programmed in such a manner that they would always yield guaranteed results.
4. **Memory Capability:** The capability of storage makes computer very useful. It saves the effort of inputs every time. The facilities of the storage of data are on the temporary basis as well as on the long term basis.
5. **Versatility:** Digital computers are highly useful everywhere in different respects such as in business/banking/industries, for personal use at home, in schools/colleges for education purpose, in scientific/engineering research, for satellite control and many more applications.

**Functional Components of Digital Computer:**

The digital computer consists of following four major components as shown in block diagram. 

1. **Input Unit** 🡪Input devices (e.g. Keyboard, Mouse, Scanner, Light pen, Joystick etc.)
2. **Processing Unit** 🡪 C. P. U. or Central Processing Unit (A. L. U. & C. U.)
3. **Output Unit** 🡪 Output devices (e.g. Monitor, Printer, Speaker, Plotter etc.)
4. **Memory Unit** 🡪 Primary, Secondary, Cache memories etc.

****

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**Basic Operations of Digital Computer:**

A digital computer is a programmable machine which reads the binary instructions and processes them according to certain requirements of users. For proper operation, a typical digital computer system requires specific functional components as shown in the above block diagram & described below.

* **CPU Operations:** The central processing unit is the “brain” of a digital computer where all kinds of processing is done. This unit takes data from the input devices and processes it according to the set of instructions called “program”. After processing it gives result to the output devices.

Some example processors are: IBM/370 series, CDC 7600, PDP 11 (16 bit), CYBER-175 etc.

CPU has following two major parts:

* + **(1) Arithmetic & Logic Unit (ALU):** ALU is responsible to carry out following tasks:
    - Performs **arithmetic operations** such as:
      * **Addition         (+)**
      * **Subtraction     (-)**
      * **Multiplication (\*)**
      * **Division         (/)**
      * **Modulus (%)**
      * **Increment       (++)**
      * **Decrements     (--)**
    - Performs **logical operations** such as:
      * **Boolean AND (&&)**
      * **Boolean OR (||)**
      * **Boolean NOT       (!)**
  + **(2) Control Unit (CU):** CU is responsible to carry out following tasks:
    - Generates electronic control signals for the synchronization of various operations.
    - Acts as an interface between main memory and ALU.
* **Memory Operations:** The memory is also known as “***storage device***”, where data and programs reside. The information stored in the memory may be used immediately or it can be kept for later use for processing by CPU. The memory can be classified in following categories:
  + **Primary Memory (Main Memory or RAM):** Primary storage of a computer is also known as its main memory. It is made up of ‘semiconductors’. Primary storage can hold information only while computer system is ON. The information is erased as the system turned off. Hence it is also called “***Volatile Memory***”.

It is called **R**andom **A**ccess **M**emory (RAM) because C.P.U. can access any location of memory at any time. Its access time is 10 nano seconds (n.s.) per cycle.

It has limited storage capacity because it is very expensive. Primary storage of modern computers is made up of semi-conductor devices.

Main memory is responsible for:

* + - Holding pieces of program (i.e. instructions), immediate results, recently produced results on which the CPU is currently working.
    - Providing access of current data & programs directly to CPU at a very high speed.
  + **Secondary Memory (Auxiliary Memory or ROM):** Secondary storage of a computer is also known as auxiliary memory. Secondary storage can hold information permanently i.e. information persists even after the computer system is turned OFF. Hence it is also called “***Non-Volatile Memory***”.

It is called **R**ead **O**nly **M**emory (ROM) because the contents of ROMs is decided (i.e. pre-written) by programmers to store various programs/software or manufacturers of ROMs to store software for assemblers, compilers, debugging packages, function tables such as sine, cosine, logarithm, square root, etc.

**NOTE:** It has large storage capacity because it is relatively cheaper than primary storage.

Secondary memory is responsible for:

* + - Holding large data and information for later use when it is not currently required for processing.
    - Providing access to main memory when required for CPU processing at relatively slower speed.
* **Input & output (I/O) Operations:**

I/O devices commonly referred to as peripheral devices and consist of:

* **Input units** (such as keyboards, scanners etc.) for feeding instructions and data into the computer.
* **Output units** (such as printers, monitors, plotters etc.) for displaying results.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**Types of computer languages:**

**Machine Language**: It is also known as ‘low level language’. Machine language is basic language of computer representing data in the form of 1s and 0s. They are machine dependent. This language is not easy to understand by the user i.e. less user friendly.

**Assembly Language**: It is also known as ‘middle level language’. This language uses mnemonics. Assembly language is also machine (hardware) dependent.

A computer can execute program only in machine language hence assembly language requires a translator called ‘**assembler**’. An assembler is used to convert assembly language into machine language and vice-versa.

****

**High level Language**: It is an English-like language which is required to write programs by the user. These languages have predefined rules for writing programs. Examples of high level languages are: C, C++, JAVA, C# etc.

Since a computer cannot understand high level language hence high level languages also require a translator called ‘**compiler**’ or ‘**interpreter**’.

Compiler & interpreter both can convert high-level language in to machine language (& vice versa) but with little difference such as:

* **Compiler** compiles the whole program at a time while **interpreter** compiles a program line-by-line and then execute.
* **Compiler** takes less time for compilation of source code whereas **interpreter** takes more time for compilation of same code.
* In **compiler**, object code can be saved after execution for latter use but object code cannot be saved in the **interpreter**.

**INTRODUCTION TO ‘C’**

**What is C?**

C is a programming language developed at **AT & T’s Bell Laboratories** of **USA** in **1972**. It was designed and written by **Dennis Ritchie**. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc.

**Features of C:**

* C is a high level language having lesser number of keywords than Java or Pascal.
* C is a native language of UNIX, Linux & many of the Windows packages, database programs, graphics libraries are written using C programs.
* C is portable since it provides standard set of libraries that work on the same way with all machines.
* C is modular, as it supports functions to divide the program into sub-programs.

**Getting Started with C:**

There is a close analogy between learning English language and learning C language. Following figure shows this:

|  |  |  |  |
| --- | --- | --- | --- |
| Alphabets | Words | Sentences | Paragraphs |
| Alphabets / Digits / Special symbols | Constants / Variables / Keywords | Instructions | Program |

**Character Sets in C:**

A character denotes any alphabet, digit or special symbol used to represent information. Following figure shows the valid alphabets, numbers and special symbols allowed in C.

|  |  |
| --- | --- |
| Alphabets | A, B, ….., Y, Z and a, b, ……, y, z |
| Digits | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| Special symbols | ~ ‘ ! @ # % ^ & \* ( ) \_ - + = | \ { } [ ] : ; " ' < > , . ? / |

* The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

**Structure of a C program:**

* **Comments**: The comments are usually ignored by the compiler so this part is used only to inform about the necessary descriptions of the program.

Comments are of two types:

1. **// Single line comment**
2. **/\* —– Multi line comments ….\*/**
3. **Preprocessor Directives**

* Before the compiler runs, the preprocessing takes place.
* The preprocessors are prefix with a symbol **#**
* Some examples are: **#include, #define, #ifdef, #pragma,** etc.

1. **Function Prototypes**

* It is used to inform the compiler that the following functions are defined in this program.
* It specifies the name of the function, arguments or parameters and return type and it ends with a semicolon.
* Example: **int sum(int, int);**

1. **Global Variables**

* Global variables are declared above all the functions
* As the name says, they are available to all the functions of that program

1. **Main Function**

* **main()** is the function which is main to the entire program
* Every C program must have a **main()** function
* When the main program returns nothing, i.e. void, syntax used is: **void main() { }**

1. **User-defined Functions**

* User defined functions are defined after the **main()** function as per the function prototype declared in the program.
* These functions may also be defined above (before) the main function. In that case, there is no need for function prototype. For example:

***int sum(int a, int b)***

***{***

***int x;***

***x=a+b;***

***return x;***

***}***

**Basic structure of a C Program:**

***/\* Program to display your name\*/***

***#include <stdio.h> // including the standard IO functions like printf(), scanf()***

***void main() // main() function***

***{ // start of main() function***

***printf(“name”); // body of main() function***

***} // end of main() function***

**Basic Data Types in C:** There are five atomic data types in C: character, integer, floating-point, double floating-point and valueless (**char**, **int**, **float**, **double**, and **void** respectively).

All other data types in C are based upon one of these types. The size and range of these data types may vary between processor types and compilers. However, in all cases a character is 1 byte. The size of an integer is usually the same as the word length of the execution environment of the program. For most 16-bit environments, such as DOS or Windows 3.1, an integer is 16 bits. For most 32-bit environments, such as Windows NT, an integer is 32 bits.

Following table shows all data types defined by the ANSI/ISO C Standard:

**Type Typical Size in Bits Range**

**char 8 −127 to 127**

**unsigned char 8 0 to 255**

**signed char 8 −127 to 127**

**int 16 or 32 −32,767 to 32,767**

**unsigned int 16 or 32 0 to 65,535**

**signed int 16 or 32 same as int**

**short int 16 −32,767 to 32,767**

**unsigned short int 16 0 to 65,535**

**signed short int 16 same as short int**

**long int 32 −2,147,483,647 to 2,147,483,647**

**signed long int 32 same as long int**

**unsigned long int 32 0 to 4,294,967,295**

**float 32 Six digits of precision**

**double 64 Ten digits of precision**

**long double 80 Ten digits of precision**

**Tokens in C:** Tokens are classified as:

* **Keywords**
* **Identifiers (Variables)**
* **Constants**
* **Special Symbols**
* **Operators**
* **Keywords in C:**
* Keywords are the words whose meaning has already been explained to the C compiler. The keywords **cannot** be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer.
* The keywords are also called ‘**Reserved words**’. There are only 32 keywords available in C. following table lists all the keywords used in C:

|  |  |  |  |
| --- | --- | --- | --- |
| **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** |

* **Identifiers:** Identifiers are the variable names declared inside the program.

**Definition:** ***A variable is an entity that may change whereas a constant is an entity that doesn’t change.***

**Variables in C:** An entity that may vary during program execution is called a variable. Variable names are names given to locations in memory. These locations can contain integer, real or character constants. In any language, the types of variables that it can support depend on the types of constants that it 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.

**NOTE:**

1. **The rules for constructing different types of constants are different. However, for constructing variable names of all types the same set of rules apply.**
2. **These rules remain same for all the types of primary and secondary variables. The ‘type’ preceding the ‘variable names’ distinguishes them from each other.**

* **Rules for Constructing Variable Names**
* A variable name is any combination of 1 to 31 alphabets, digits or underscores i.e. \_. Some compilers allow variable names whose length could be up to 247 characters.
* 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 (e.g. **gross\_salary**) can be used in a variable name.
* They are case sensitive i.e **Var\_name** is different from **var\_name**

**Some valid identifiers are**:

|  |  |  |  |
| --- | --- | --- | --- |
| **avg** | **Value** | **area\_of\_circle** | **x1** |
| **x2** | **Chennai** | **Sum1** | **abcd** |

**Some invalid identifiers are:**

|  |  |  |  |
| --- | --- | --- | --- |
| **123** | **%** | **(value)** | **27th** |
| **1st** | **ab-d** | **Sum of total** | **x&y** |

* **Constants in C:** C constants can be divided into two major categories:

1. **Primary Constants:**

* **Integer Constant**
* **Real Constant**
* **Character Constant**

1. **Secondary Constants:**

* **Array**
* **Pointer**
* **Structure**
* **Union**
* **Enum, etc.**

For constructing these different types of constants certain rules have been laid down. These rules are as under:

* **Rules for Constructing Integer Constants**
* An integer constant must have at least one digit.
* It must not have a decimal point.
* It can be either positive or negative.
* If no sign precedes an integer constant it is assumed to be positive.
* No commas or blanks are allowed within an integer constant.
* The allowable range for integer constants is **-32768 to +32767**.

Truly speaking, the range of an Integer constant depends upon the compiler. For a **16-bit compiler** like Turbo C or Turbo C++, this range is –32768 to +32767.

Examples of integer constants: **426, +782, -8000, -7605** etc.

* **Rules for Constructing Real Constants**

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

Following rules must be observed while constructing real constants expressed in fractional form:

* A real constant must have at least one digit.
* It must have a decimal point.
* It could be either positive or negative.
* Default sign is positive.
* No commas or blanks are allowed within a real constant.

Examples of valid real constants (in fractional form): **+325.34, 426.0, -32.76, -48.5792** etc.

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large. In exponential form of representation, the real constant is represented in two parts. The part appearing before ‘e’ is called mantissa, whereas the part following ‘e’ is called exponent.

Following rules must be observed while constructing real constants expressed in exponential form:

* The mantissa part and the exponential part should be separated by a letter e.
* The mantissa part may have a positive or negative sign.
* Default sign of mantissa part is positive.
* The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.
* Range of real constants expressed in exponential form is **-3.4e38 to 3.4e38**.

Examples of valid real constants (in exponential form): **+3.2e-5, 4.1e8, -0.2e+3, -3.2e-5** etc.

* **Rules for Constructing Character Constants**
* A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
* The maximum length of a character constant can be 1 character.

Examples of valid character constants: **'A', 'I', '5' '=', '+'** etc.

* **Special Symbols:** Special symbols are the characters used other than alphabets & digits such as:

**~ ‘ ! @ # % ^ & \* ( ) \_ - + = | \ { } [ ] : ; " ' < > , . ? /** etc.

* **Operators in C:** To perform various operations, a large verity of operators is available in C.

**Arithmetic Operator**  **Action Relational Operator**  **Action**

**− Subtraction == Equal to**

**+ Addition != Not equal to**

**\* Multiplication > Greater than**

**/ Division < Less than**

**% Modulus >= Greater than or equal to**

**<= Less than or equal to**

**Logical Operator**  **Action Bitwise Operator**  **Action**

**&& Logical AND & Bitwise AND**

**|| Logical OR | Bitwise OR**

**! Logical NOT ^ Bitwise XOR ~ One’s complement**

**<< Shift left**

**>> Shift right**

**Unary Operator**  **Action**

**- Unary minus**

**++ Increment**

**– – Decrement**

**Assignment Operator**  **Action**

**= “Assignment” of value on the right side of it to the variable on the left side.**

**Conditional operator Action**

**? : Condition test? True : false**

**e. g. x = (a>b)? a : b; means if the condition a> b is true then a is assigned to x, if false then b is assigned to x.**

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

**The First C Program:** Armed with the knowledge about the types of variables, constants, keywords & operators, the next logical step is to combine them to form instructions.

Before we begin with our first C program, do remember the following rules that are applicable to all C programs:

* 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 should be entered in small case letters since C is a case sensitive language..
* 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 semicolon (**;)**. Thus **;** acts as a statement terminator.

Let us now write down our first C program. It would simply calculate simple interest for a set of values representing principle, number of years and rate of interest.

**/\* 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 ) ;***

***}***

Now a few useful tips about the program...

1. Comment about the program should be enclosed within **/**\* \***/**. For example, the first statement in our program is comment.
2. Though comments are not necessary, it is a good practice to begin a program with a comment indicating the purpose of the program.
3. 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* *;***

1. Sometimes it is worthwhile mentioning the purpose of the statement (or a set of statements) using a comment. e.g.

**/\* formula for simple interest \*/**

***si = p \* n \* r / 100 ;***

1. **main( )** is a collective name given to a set of statements. Technically speaking **main( )** is a function. ***Every C program must start with this function.*** Every function has a pair of parentheses **( )** associated with it. All statements that belonging to a functionare enclosed within a pair of braces { } as shown below.

***main( )***

***{***

***statement 1 ;***

***statement 2 ;***

***statement 3 ;***

***}***

1. Any variable used in the program must be declared before using it. e.g.,

**int p, n ;**

**float r, si ;**

1. Any C statement always ends with a ; For example,

**float r, si;**

**r = 8.5;**

1. In the statement,

**si = p \* n \* r / 100 ;**

**\*** and **/** are the arithmetic operators. (The arithmetic operators available in C are **+**, **-**, **\***, **/** and **%**).

1. 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 such as **printf( )**.

* The general form of **printf( )** function is:

**printf ( "format strings", list of variables ) ;**

Where **format strings** can contain **format specifiers** and/or **other strings**.

**Format Specifiers:**

* Format specifiers are used to input & output data types.
* The  **scanf()** & **printf()** functions accepts a wide variety of format specifiers, some of them are shown in the following table:

|  |  |
| --- | --- |
| **Data type** | **Format Specifier** |
| **short signed int** | **%d or %I** |
| **short unsigned int** | **%u** |
| **long singed int** | **%ld** |
| **long unsigned int** | **%lu** |
| **unsigned hexadecimal int** | **%x** |
| **unsigned octal int** | **%o** |
| **float** | **%f** |
| **double** | **%lf** |
| **signed character** | **%c** |
| **unsigned character** | **%c** |
| **string** | **%s** |

**Operator precedence & associativity:** C provides various operators for smooth operations. These operators having their own order of execution in mixed operators.

The order of execution is known as precedence. In same precedence order, the direction of execution is called associativity. The associativity may be left to right or right to left.

In left to right association the evaluation start from left and in right to left association evaluation start from right. Various operators and their precedence are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator name** | **Symbol** | **Associativity** | **Precedence** |
| Special | **( ) [ ] ->** | Left to right | **1st** |
| Unary | **! ~ ++ -- - & \* sizeof type cast** | **Right to left** | **2nd** |
| Arithmetic | **\* / %** | Left to right | **3rd** |
| Arithmetic | **+ -** | Left to right | **4th** |
| Bitwise shift | **<< >>** | Left to right | **5th** |
| Relational | **< > <= >=** | Left to right | **6th** |
| Equality | **== !=** | Left to right | **7th** |
| Bitwise AND | **&** | Left to right | **8th** |
| Bitwise EX-OR | **^** | Left to right | **9th** |
| Bitwise OR | **|** | Left to right | **10th** |
| Logical AND | **&&** | Left to right | **11th** |
| Logical OR | **||** | Left to right | **12th** |
| Conditional | **?:** | **Right to left** | **13th** |
| Assignment | **= += -= \*= /=** | **Right to left** | **14th** |
| Comma | **,** | Left to right | **15th** |

**Mixed operators & type conversions:** C permits use of mixed type of operands together. It permits mixing of constant and variable of different types in expressions. The conversion result will be one of the following:

* **Implicit conversion** (i.e. automatic)
* **Explicit conversion** (i.e. user can convert it explicitly)

In implicit or automatic (by default) conversion method, lower priority-order data type gets converted into higher priority-order data type and result will be higher data type as following conversion sequence.

The arrow shows order of conversions in mixed type of operators.

long double

double

float

int

character

short

long int

**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 as mentioned below.

* An arithmetic operation between an integer and integer always yields an integer result.
* An operation between a real and real always yields a real result.
* 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.

Following table illustrates this:

|  |  |  |  |
| --- | --- | --- | --- |
| **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.

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 **int**s whereas others are **float**s. As we know, during evaluation of the expression the **int**s would be promoted to **float**s 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**.

Observe the results of the arithmetic statements shown in following figure. It has been assumed that **k** is an **int** variable and **a** is a **float** variable.

|  |  |  |  |
| --- | --- | --- | --- |
| **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** |

**Computer System & Programming in C**

**(RCS-101) (PART-II)**

* **Use of Logical Operators:** These operators are used in conditional operations as follows:

1. **Logical AND (&&):** In logical AND (&&) operation of two input statements, the output results as TRUE (logic-1) only when both the statements are TRUE (logic-1) otherwise, the output results as FALSE (logic-0).
2. **Logical AND (&&): I**n logical OR ( || ) operation of two input statements, the output results as FALSE (logic-0) only when both the statements are FALSE (logic-0) otherwise, the output results as TRUE (logic-1).
3. **Logical NOT (!):** Logical NOT (!) can be operated on single input only. It reverses the result i.e. TRUE to FALSE and vice-versa.

**NOTE:** In C language, **TRUE stands for Boolean value 1** and **FALSE stands for Boolean value 0**.

E. g.: **int x = 10, y = 5, z;**

**z = (x<5) && (y>10);** //Result is false (F) hence z=0; because false is equivalent to 0

**T F**

**z = (x<5) || (y>10);** //Result is true (T) hence z=1; because true is equivalent to 1

**T F**

Here is table for AND, OR, & NOT operations:

|  |  |  |  |
| --- | --- | --- | --- |
| **X** | **Y** | **X&&Y** | **X||Y** |
| **0** | **0** | **0** | **0** |
| **0** | **1** | **0** | **1** |
| **1** | **0** | **0** | **1** |
| **1** | **1** | **1** | **1** |
| **NOT (!) 🡪 reverse the bit** | | | |

* **Use of Bitwise Operators:** Bitwise operators are same as the logical operators but the operation is performed bit by bit.

**Example:** Let two binary sequences are given below as (10101010)**2** and (11001101)**2**

Then the result will be:

1. **Bitwise AND:**

1 0 1 0 1 0 1 0

**&** 1 1 0 0 1 1 0 1

**1 0 0 0 1 0 0 0**

1. **Bitwise OR:**

1 0 1 0 1 0 1 0

**|** 1 1 0 0 1 1 0 1

**1 1 1 0 1 1 1 1**

**NOTE:** If we apply bitwise operators on two characters. Then the compiler computes 8 bit ASCII code then applies bit by bit operation.

e. g. **char ch1=’A’, ch2=’B’, ch;**

**ch=ch1&ch2;**

Then the result will be ***the result of ASCII value of A and ASCII value of B then convert both in 8 bit binary code and compute bitwise AND (&) then resultant value is finally assigned to character ch***.

1. **Bitwise left shift (<<):** This operator shifts bit sequence to the left, bit by bit, as required and inserts a default 0 in empty position.

e. g. **101111<<1** will shift all bits one position to the left, hence, the result will be **011110**

**101111<<2** will shift all bits two positions to the left, hence, the result will be **111100**

Therefore, we can say in general:

**101111<<2 = 011110<<1 = 111100**

1. **Bitwise right shift (>>):** This operator shifts bit sequence to the right, bit by bit, as required and inserts a default 0 in empty position.

e. g. **101111>>1** shifts all the bits one position to the right, hence, the result will be **010111**

**101111>>2** shifts all the bits two positions to the right, hence, the result will be **001011**

Therefore, we can say in general:

**101111>>2 = 010111<<1 = 001011**

**NOTE:** In general, left shift returns multiplication by two, whereas, right shift returns division by two.

* **Assignment operators ( =, +=, -=, \*=, /= etc):** In assignment operation, the object value of right hand side variables are copied into address of left hand side variable.

i. e. **a = 10;**

**b = a;**

Here, 10 is copied into address of b which means now b is equal to 10.

**NOTE:** The assignment **a=a+1;** is equivalent to **a+=1;** and so on.

* **Increment and decrement operators (++ and --):** There are two methods for increment and decrement operators; Prefix and postfix.

1. **Pre-fix Notation:** In prefix notation, first increment or decrement occurs then the operation takes place.

Syntax: **++x**, **--y** etc.

1. **Post-fix Notation:** In postfix notation, first operation is performed then increment or decrement takes place.

Syntax: **x++**, **y--** etc.

**NOTE THAT:** **a++;** is equivalent to **a=a+1;**

Similarly, **a--;** is equivalent to **a=a-1;**

For example: **a = 10;**

**b = ++a;**

In this case, prefix notation first increases the value of **a** by one and then assigns it to **b**.

Thus, the value of **a** and **b** both are same i.e. **11**.

Now, consider postfix notation:

**a=10;**

**b=a++;**

Here, first, the value of **a** is assigned to **b** and then the value of **a** increases by one.

Thus, the values of **a** and **b** are **11** and **10** respectively.

* **Conditional operator (? :) :**

Conditional operator is also known as **ternary operator**, since its syntax divided in three parts.

i. e. **expression1 ? expression2 : expresion3**

*Condition test T F*

It means if **expression1** is true then return **expression2** otherwise **expresion3**.

Let **a=100;**

**b=50;**

**x = (a>b) ? a : b;**

Here, it will return **x = 100**.

* **Comma operator ( , ):**

Comma operator is used to link operators together. The expression gets evaluated from left to right and the rightmost expression is the result.

For example: Expression **X = (a = 10, b = 20, a + b);** will return **30** to **X** i. e. **X = 30**.

**UNIT - 3**

**Control Instructions in C:** As the name suggests the ‘control instructions’ enable us to specify the order in which the various instructions in a program are to be executed by the computer. In other words, the control instructions determine the ‘flow of control’ in a program. There are 3 types of control instructions in C:

**(a) Sequence Control Instructions**

**(b) Selection or Decision & Case Control Instructions**

**(c) Repetition or Loop Control Instructions**

**(a) Sequence Control Instructions:** The Sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program. By default the instructions in a program are executed sequentially. (*Refer to class notes for example programs*).

**(b) Selection or Decision Control Instructions:** 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. A decision control instruction can be implemented in C using:

* **if** statement
* **if-else** statement
* **switch case default** statements
* ***if* Statement:** C uses the keyword **if** to implement the decision control instruction. In **if** method any condition can be tested whether by **single relational operation** or by **an expression**. When the condition meets **TRUE** then body of **if** will get executed, otherwise body of **if** will be skipped and control goes to remaining statements of **main** body.

The general form of **if** statement looks like this:

***if ( this condition is true )***

***execute this statement ;***

The keyword **‘if’** tells the compiler that the condition following the keyword **if** (*always enclosed within a pair of parentheses*) is a decision control instruction. If the condition, whatever it is, is **TRUE**, then the statement is executed. If the condition is not true then the statement is not executed.

**//**Demo of **if condition**

***void main()***

***{***

***int a=10,b=20;***

***if(a<100)***

***{***

***printf(“ a is =%d\t”,a);***

***}***

***printf(“b is =%d”,b);***

***}***

**Output: *a is = 10 b is = 20* //execute both statements because if body is TRUE.**

***void main()***

***{***

***int a=10,b=20;***

***if(a>100)***

***{***

***printf(“ a is =%d\t”,a);***

***}***

***printf(“b is =%d”,b);***

***}***

**Output: *b is 20* //because if body will not execute due to FALSE condition.**

**NOTE: In nested if situations, ‘if’ condition can be given inside if at any level.**

* ***if else* Statements:** In **if** statement only TRUE condition considered but in **if else** statement FALSE condition of if statement goes to **else** statement as alternate.

**Syntax:**

***if (test condition)***

***{***

***Statements;* // execute when ‘if’ condition meets TRUE**

***}***

***else***

***{***

***Statements;* //execute when ‘if’ condition meets FALSE**

***}***

**Example of leap year problem: WAP in C to check whether the given year is leap or not.**

**Logic for Solution:** *Any year is called “leap year” if it is completely divisible by 4 i.e. remainder becomes zero.*

***//Program to check given year is leap or not***

***void main()***

***{***

***int year;***

***printf(“ENTER THE YEAR:\n”);***

***scanf(“%d”,&year);***

***if (year%4==0)***

***{***

***printf(“\n LEAP YEAR”);***

***}***

***else***

***{***

***printf(“\n NOT LEAP YEAR”);***

***}***

***}***

**Output (1st Run):**

**ENTER THE YEAR: *2000***

***NOT LEAP YEAR* //because condition meets FALSE and else body executed**

**Output (2nd Run):**

**ENTER THE YEAR: *2008***

***LEAP YEAR*  //because condition meets TRUE and if body executed**

**NOTE: In nested if else, the if and else statements can be given inside if and else body at any level.**

* **Various Forms of *if*:**The if statement can take any of the following forms:
* ***if ( condition )***

***do this ;***

* ***if ( condition )***

***{***

***do this ;***

***and this ;***

***}***

* ***if ( condition )***

***do this ;***

***else***

***do this ;***

* ***if ( condition )***

***{***

***do this ;***

***and this ;***

***}***

***else***

***{***

***do this ;***

***and this ;***

***}***

* ***if ( condition )***

***do this ;***

***else***

***{***

***if ( condition )***

***do this ;***

***else***

***{***

***do this ;***

***and this ;***

***}***

***}***

* ***if ( condition )***

***{***

***if ( condition )***

***do this ;***

***else***

***{***

***do this ;***

***and this ;***

***}***

***}***

***else***

***do this ;***

**NOTE: For example programs, do refer to your class notes.**

**Disadvantages of *if* statements:** As we see,in **if** statements there are number of alternatives. If we need one alternative at a time, we would require more and more **if** statements. Hence the complexity will increase with number of alternatives. **switch case** statements avoid this type of problem.

* ***switch case default* Statements:** The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switch-case-default**, since these three keywords go together to make up the control statement.

**Syntax of switch:**

***switch (expression)***

***{***

***case constant1:***

***statement1;***

***break;***

***case constant2:***

***statement2;***

***break;***

***:***

***:***

***:***

***case constantN:***

***statementN;***

***break;***

***default:***

***default statement;***

***}***

Where ***constant1……….constantN*** are results of expression tested. Only one statement (or set of statements) out of ***statement1……….statementN*** will execute because expression returns only one result at a time. Hence it requires **break** after every **case** statement. If result will not match any case then control will be transferred to **default** body and then back in **main** body.

**IMPORTANT NOTE:**

1. The limitation of **switch** statement is that, in **switch**, we can use ***only integer***, ***character*** or ***any expression which output either character*** *or* ***integer value***.
2. The character case must be given into single quotation mark (i.e. **‘A’**).
3. Each **constant** in each **case** must be different from all the others.
4. We can execute more than one case into single case as given below:

**case ‘A’:**

**case ‘a’:**

Consider the following program:

**/\*Demonstration of switch-case-default\*/**

**void main( )**

**{**

**int i = 2 ;**

**clrscr();**

**switch ( i )**

**{**

**case 1 :**

**printf ( "I am in case 1 \n" ) ;**

**break ;**

**case 2 :**

**printf ( "I am in case 2 \n" ) ;**

**break ;**

**case 3 :**

**printf ( "I am in case 3 \n" ) ;**

**break ;**

**default :**

**printf ( "I am in default \n" ) ;**

**}**

**getch();**

**}**

The output of this program would be:

**I am in case 2**

**Use of *break* & *default* in switch case statements:**

* C uses the keyword **break** to terminate the current execution and send control to outside executing **case** body whereas keyword **default** is used to control if there are unmatched **case**s of **switch** expression.
* In **switch** structure, **break** & **default** areoptional statementsbut required to improve the programming efficiency.
  1. If **break** is not used then control executes all cases until last break statement will meet or encounters end of **switch**.
  2. If unmatched **case** is found then control executes **default** section and then terminate. **default** can be used anywhere in control body but only once.
* ***Tips and Traps (*switch statement*):***

A few useful tips about the usage of **switch** and a few pitfalls to be avoided:

1. **The earlier program that used switch may give you the wrong impression that you can use only cases arranged in ascending order, 1, 2, 3 and default. You can in fact put the cases in any order you please. Here is an example of scrambled case order:**

**void main( )**

**{**

**int i = 22 ;**

**switch ( i )**

**{**

**case 121 :**

**printf ( "I am in case 121 \n" ) ;**

**break ;**

**case 7 :**

**printf ( "I am in case 7 \n" ) ;**

**break ;**

**case 22 :**

**printf ( "I am in case 22 \n" ) ;**

**break ;**

**default :**

**printf ( "I am in default \n" ) ;**

**}**

**getch();**

**}**

The output of this program would be:

**I am in case 22**

1. **You are also allowed to use char values in case and switch as shown in the following program:**

**void main( )**

**{**

**char c = 'x' ;**

**clrscr();**

**switch ( c )**

**{**

**case 'v' :**

**printf ( "I am in case v \n" ) ;**

**break ;**

**case 'a' :**

**printf ( "I am in case a \n" ) ;**

**break ;**

**case 'x' :**

**printf ( "I am in case x \n" ) ;**

**break ;**

**default :**

**printf ( "I am in default \n" ) ;**

**}**

**getch();**

**}**

The output of this program would be: **I am in case x**

***In fact here when we used ‘v’, ‘a’, ‘x’ they are actually replaced by the ASCII values (118, 97, 120) of these character constants.***

1. **At times we may want to execute a common set of statements for multiple cases. How this can be done is shown in the program example of “Mathematical Calculator”.**
2. **Even if there are multiple statements to be executed in each case there is no need to enclose them within a pair of braces (unlike if, and else).**
3. **Every statement in a switch must belong to some case or the other. If a statement doesn’t belong to any case the compiler won’t report an error. However, the statement would never get executed. For example, in the following program the printf( ) never goes to work.**

**void main( )**

**{**

**int i, j ;**

**clrscr();**

**printf ( "Enter value of i" ) ;**

**scanf ( "%d”, &i ) ;**

**switch ( i )**

**{**

**printf ( "Hello" ) ;**

**case 1 :**

**j = 10 ;**

**break ;**

**case 2 :**

**j = 20 ;**

**break ;**

**}**

**getch();**

**}**

1. **If we have no default case, then the program simply falls through the entire switch and continues with the next instruction (if any,) that follows the closing brace of switch.**
2. **Is switch a replacement for if? Yes and no. Yes, because it offers a better way of writing programs as compared to if, and no because in certain situations we are left with no choice but to use if. The disadvantage of switch is that one cannot have a case in a switch which looks like: case i <= 20 : All that we can have after the case is an int constant or a char constant or an expression that evaluates to one of these constants. Even a float is not allowed. The advantage of switch over if is that it leads to a more structured program and the level of indentation is manageable,** **more so if there are multiple statements within each case of a switch.**
3. **We can check the value of any expression in a switch. Thus the following switch statements are legal.**

**switch ( i + j \* k )**

**switch ( 23 + 45 % 4 \* k )**

**switch ( a < 4 && b > 7 )**

**Expressions can also be used in cases provided they are constant expressions (Cases can never have variable expressions).** Thus **case 3+7:** is **correct**, however, **case a+b:** or **case a+3:** is **incorrect.**

1. **A float expression cannot be tested using a switch.**
2. **Multiple cases cannot use same expressions. Thus the following switch is illegal:**

**switch ( a )**

**{**

**case 3 :**

**...**

**case 1 + 2 :**

**...**

**}**

**Example: WAP in C for mathematical calculator for addition & subtraction operations.**

**Answer:**

**/\*Program for Mathematical Calculator\*/**

**void main()**

**{**

**int x=50,y=20,z;**

**char ch;**

**clrscr();**

**printf ("\n ENTER A FOR ADDITION");**

**printf ("\n ENTER S FOR SUBTRACTION");**

**printf ("\n ENTER YOUR CHOICE\t");**

**scanf ("%c",&ch);**

**switch (ch)** //switch character value given through keyboard

**{**

**case 'A':**

**case ‘a’:**

**case ‘+’:**

**z=x+y;**

**printf("\nAddition of x and y is = %d",z);**

**break;**

**case 'S':**

**case ‘s’:**

**case ‘-’:**

**z=x-y;**

**printf("\nSubtraction of x and y is =%d",z);**

**break;**

**default:**

**printf("\n ENTER CORRECT CHOICE");**

**break;**

**}**

**getch();**

**}**

The output will be as shown below:

**ENTER A FOR ADITION**

**ENTER S FOR SUBTRACTION**

**ENTER YOUR CHOICE A**

**Addition of x and y is =70** //Here value of ch is A hence case ‘A’ will executed.

**Advantages & Disadvantages** of ***switch*** over ***if*:**

Is switch a replacement for if? Yes and no. Yes, because it offers a better way of writing programs as compared to if, and no because in certain situations we are left with no choice but to use if.

The disadvantage of switch is that one cannot have a case in a switch which looks like:

**case i <= 20 :**

All that we can have after the **case** is an **int** **constant** or a **char** **constant** or an expression that evaluates to one of these constants. Even a **float** is not allowed.

The advantage of switch over if is that it leads to a more structured programming and the level of indentation is manageable, more so if there are multiple statements within each case of a switch.

* **Repetition or Loop Control Instructions:** The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This is done through a **loop control instruction**. The loop control instruction helps computer to execute a group of statements repeatedly.

**The Loop Control Structure:**

Loops are used to execute any statements more than one times as required. It is better to write any statement in loop body than write statement more than one times. There are two methods used in loop structures: i.e. **entry control loop** and **exit control loop**.

1. **Entry Control Loop:** In entry control loops, the condition is checked first (and is **TRUE**) then statements associated to it get executed until condition turns **FALSE**.
2. **Exit Control Loop:** In exit control loops, irrespective of the condition, the associated statements get executed first then the condition is checked; now:
3. If it is **TRUE**, execution continues until condition becomes **FALSE**.
4. If it is **FALSE**, the execution terminates immediately.

It means in **case ii**, loop will execute at least once even if condition is **FALSE**.

* C provides three looping constructs by way of which we can repeat a part of a program. They are:
* Using a **while loop**
* Using a **do-while loop**
* Using a **for loop**

Here we will learn syntax and examples of each constructs in all loops there are three sections:

1. ***Initialization***
2. ***Test condition***
3. ***Increment / decrement***

***All these sections can have either integer or floating point values (depending on the programmer).***

* ***while* loop:** In ***while*** loop, ***initial condition*** is given before the *loop body* followed by ***condition test*** in loop and ***increment*** or ***decrement*** occurs inside the loop body.

The syntax is shown below:

***Initialization;***

***while (test-condition)***

***{***

***Statements;***

***Increment or decrement;***

***}***

**Example:** **/\*Program to print Kashi IT five times using while loop\*/**

**void main()**

**{**

**int i;**

**i=0;**

**while (i<5)**

**{**

**printf(“ Kashi IT \n”);**

**i++;**

**}**

**getch();**

**}**

The output will be:

**Kashi IT**

**Kashi IT**

**Kashi IT**

**Kashi IT**

**Kashi IT**

***Tips and Traps (*while loop*):***

1. The statements within the **while** loop would keep on getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the **while** loop.
2. In place of the condition there can be any other valid expression. So long as the expression evaluates to a non-zero value the statements within the loop would get executed.
3. The condition being tested may use relational or logical operators as shown in the following examples:

**while ( i <= 10 )**

**while ( i >= 10 && j <= 15 )**

**while ( j > 10 && ( b < 15 || c < 20 ) )**

1. The statements within the loop may be a single line or a block of statements. In the first case the parentheses are optional. For example:

**while ( i <= 10 )**

**i = i + 1;**

is same as

**while ( i <= 10)**

**{**

**i = i + 1;**

**}**

1. As a rule the while must test a condition that will eventually become false, otherwise the loop would be executed forever, indefinitely.

**void main( )**

**{**

**int i = 1 ;**

**while (i <= 10 )**

**printf(“ %d\n", i ) ;**

**}**

This is an indefinite loop, since **i** remain equal to 1 forever. The correct form would be as given under:

**void main( )**

**{**

**int i = 1 ;**

**while (i <= 10 )**

**{**

**printf(“ %d\n", i ) ;**

**i=i + 1;**

**}**

1. Instead of incrementing the loop counter, we can even decrement it.

**void main( )**

**{**

**int i = 10 ;**

**while (i >= 1 )**

**{**

**printf(“ %d\n", i ) ;**

**i=i - 1;**

**}**

1. It is not necessary that a loop counter must only be an **int**. It can even be a **float**.

**void main( )**

**{**

**float a=10.0 ;**

**while (a <= 10.5 )**

**{**

**printf(“ %f\n", a ) ;**

**a=a + 0.1;**

**}**

1. Even floating point loop counters can be decremented. Once again the increment and decrement could be by any value, not necessarily 1.
2. **+=** is a compound assignment operator. Hence, **i = i + 10** can also be written as **i += 10**. Other compound assignment operators are **-=, \*=, / =** and **%=**.
3. Consider the following program segments:

**/\*Demo of post-increment operator\*/**

**void main( )**

**{**

**int i = 0 ;**

**while (i++ < 10)**

**{**

**printf(“ %d\n", i ) ;**

**}**

In the statement **while (i++ <= 10)**, firstly the comparison of value of **i** with 10 is performed, and then the increment of **i** takes place. Since the increment of **i** happens after its usage, here the **++** operator is called a **post-increment operator**. When the control reaches **printf( )**, **i** has already been incremented.

**/\*Demo of pre-increment operator\*/**

**void main( )**

**{**

**int i = 0 ;**

**while (++i < 10)**

**{**

**printf(“ %d\n", i ) ;**

**}**

In the statement **while (++i <= 10)**, firstly increment of **i** takes place, then the comparison of value of **i** with 10 is performed. Since the increment of **i** happens before its usage, here the **++** operator is called a **pre-increment operator**.

* ***for* loop:** The ***for*** loop allows us to specify three things about a loop in a single line; i.e.

1. Setting a loop counter to an initial value.
2. Testing the loop counter to determine whether its value has reached the number of repetitions desired.
3. Increasing the value of loop counter each time the program segment within the loop has been executed.

All the above three sections are separated by (**;**) as shown in syntax below:

***for (initialization; test condition; increment or decrement)***

***{***

***statements;***

***}***

**Example:** **/\*Program to print Kashi IT five times using for loop\*/**

**void main()**

**{**

**int i;**

**clrscr();**

**for (i=1; i<=5; i++)**

**{**

**printf (“ Kashi IT \n”);**

**}**

**getch();**

**}**

The output will be:

**Kashi IT**

**Kashi IT**

**Kashi IT**

**Kashi IT**

**Kashi IT**

**IMPORTANT NOTE:**

1. We can give any initial value as well as decrement or increment values as per requirement.

For example, above loop is similar to: **for (i=9; i>=0; i--)**

1. We can give more than one initial conditions and increment/decrement values separated by comma (**,**) as well as more than one expression with relational operator. Hence, the other form of for loop is as shown below:

**for (int i=0, int j=0; (i<10)&&(j<6); i++, j--)**

1. It is important to note that the initialization, testing and increment/decrement part of a **for** loop can be replaced by any valid expression. Thus the following **for** loops are perfectly ok.
2. **for ( i = 10 ; i ; i -- )**

**printf ( "%d", i ) ;**

1. **for ( i < 4 ; j = 5 ; j = 0 )**

**printf ( "%d", i ) ;**

**Nesting of Loops:** The way **if** statements can be nested, similarly **while**s and **for**s can also be nested.

To understand how nested loops work; look at the program given below:

**/\* Demonstration of nested loops \*/**

**void main( )**

**{**

**int r, c, sum;**

**for ( r = 1 ; r <= 3 ; r++ ) /\* outer loop \*/**

**{**

**for ( c = 1 ; c <= 2 ; c++ ) /\* inner loop \*/**

**{**

**sum = r + c ;**

**printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;**

**}**

**}**

**getch();**

**}**

When you run this program you will get the following output:

**r = 1 c = 1 sum = 2**

**r = 1 c = 2 sum = 3**

**r = 2 c = 1 sum = 3**

**r = 2 c = 2 sum = 4**

**r = 3 c = 1 sum = 4**

**r = 3 c = 2 sum = 5**

**Explanation:** Here, for each value of **r** the inner loop is cycled through twice, with the variable **c** taking values from 1 to 2. The inner loop terminates when the value of **c** exceeds 2, and the outer loop terminates when the value of **r** exceeds 3.

Instead of using two statements, one to calculate **sum** and another to print it out, we can compact this into one single statement by saying:

**printf ( "r = %d c = %d sum = %d\n", r, c, r + c ) ;**

**NOTE:** The way **for** loops have been nested here, similarly, two **while** loops can also be nested. Not only this, a **for** loop can occur within a **while** loop, or a **while** within a **for**.

* ***do while* loop:** There is a minor difference between the working of **while** and **do while** loops. This difference is the place where the condition is tested. The **while** tests the condition before executing any of the statements within the **while** loop. As against this, the **do while** tests the condition after having executed the statements within the loop. The syntax is as given below.

**Initialization;**

**do**

**{**

**Statements:**

**}**

**while (test condition);** //(**;**) is necessary at last.

**/\*Program to print Kashi IT five times using do while loop\*/**

**void main()**

**{**

**int i;**

**i=0;**

**do**

**{**

**printf(“ Kashi IT \n”);**

**i++;**

**}**

**while(i<10);**

**getch();**

**}**

**Output:** print **Kashi IT** five times.

* For infinite loop user can control by explicit method using own concept. In for loop, the following statement will execute infinite time. **for ( ; ; )**
* For other loops infinite execution gives a condition which will never meet. Such as:

**int i=1;**

**while(i>0)**

**{**

**printf(“Kashi IT\n”);**

**i++;**

**} //Print infinite times because i always greater than 0**

**IMPORTANT NOTE:** ***Programmers should always avoid this type of condition.***

**NOTE: For more example programs, do refer to your class notes.**

**The *break* Statement:**

We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword **break** allows us to do this. When **break** is encountered inside any loop, control automatically passes to the first statement after the loop. i. e. **break** statement terminates the loop at any specific condition. A **break** is usually associated with an **if**. As an example, let’s consider the following example:

**void main()**

**{**

**int i;**

**for(i=0;i<10;i++)**

**{**

**printf(“ %d\t ”,i);**

**if(i==5)**

**break;**

**}**

**getch();**

**}**

**Output:** Print numbers from 0 to 5 and halt when i==5

i.e. **0 1 2 3 4 5**

**Example 2: Write a program in C to determine whether a number is prime or not.**

* ***Logic:*** *A prime number is one, which is divisible only by 1 or itself.*

All we have to do to test whether a number is prime or not, is to divide it successively by all numbers from 2 to one less than itself. If remainder of any of these divisions is zero, the number is not a prime. If no division yields a zero then the number is a prime number. Following program implements this logic.

**void main( )**

**{**

**int num, i ;**

**printf ( "Enter a number " ) ;**

**scanf ( "%d", &num ) ;**

**i = 2 ;**

**while ( i <= num - 1 )**

**{**

**if ( num % i == 0 )**

**{**

**printf ( "Not a prime number" ) ;**

**break ;**

**}**

**i++ ;**

**}**

**if ( i == num )**

**printf ( "Prime number" ) ;**

**getch();**

**}**

In this program the moment **num % i** turns out to be zero, (i.e. **num** is exactly divisible by **i**) the message “**Not a prime number**” is printed and the control breaks out of the **while** loop. Why does the program require the **if** statement after the **while** loop at all? Well, there are two ways the control could have reached outside the **while** loop:

* 1. It jumped out because the number proved to be **not a prime**.
  2. The loop came to an end because the value of **i** became equal to **num**.

When the loop terminates in the case **b**, it means that there was no number between **2** to **num - 1** that could exactly divide **num**. i.e, **num** is indeed a prime. If this is true, the program should print out the message “**Prime number**”.

The keyword **break**, breaks the control only from the **while** in which it is placed.

Consider the following program, which illustrates this fact.

**void main( )**

**{**

**int i = 1 , j = 1 ;**

**while ( i++ <= 100 )**

**{**

**while ( j++ <= 200 )**

**{**

**if ( j == 150 )**

**break ;**

**else**

**printf ( "%d %d\n", i, j ) ;**

**}**

**}**

**getch();**

**}**

In this program when **j** equals 150, **break** takes the control outside the inner **while** only, since it is placed inside the inner **while**.

**The *continue* Statement:**

In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword **continue** allows us to do this. When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop.

**NOTE:** **continue** statement avoid only the condition tested and remain continue the execution.

For example:

**void main()**

**{**

**int i;**

**for(i=0;i<10;i++)**

**{**

**if(i==5)**

**continue;**

**printf(“ %d\t ”,i);**

**}**

**getch();**

**}**

**Output:** Print no 0 to 9 except 5 as:

**0 1 2 3 4 6 7 8 9**

since continue use to skip any statement such as skip 5 in above program..

* A **continue** is usually associated with an **if**. As an example, let's consider the following program.

**void main( )**

**{**

**int i, j ;**

**for ( i = 1 ; i <= 2 ; i++ )**

**{**

**for ( j = 1 ; j <= 2 ; j++ )**

**{**

**if ( i == j )**

**continue ;**

**printf ( "\n%d\t %d\n", i, j ) ;**

**}**

**}**

**getch();**

**}**

The output of the above program would be...

**1 2**

**2 1**

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**Computer System & Programming in C**

**(RCS-201) (PART-III)**

**Modular programming:** Modular programming is also known as **function programming** in C because it consists of various small blocks of code known as ‘***module***’ or ‘***function***’.

**Functions in C:** **A** **function is a self contained block of code (or set of instructions) to carry out a particular task**.

There are two types of functions available in C:

1. **Built-in** or **Pre-defined** or **Library** functions
2. **User defined** functions

***Built-in functions are predefined in library and their prototype has already been declared in header files.***

***e.g. printf(), scanf(), getch() etc.***

Here, we will discuss only **user defined functions**. Following points illustrate it:

* User defined function behave similar to built-in functions but it consists of declaration part before definition and calling in the function.
* All the user defied functions should have its parent function called **main** function.
* However user defined function may call any other user defined function.
* If any C program has only one function then it is definitely **main** function.

**Syntax for user defined function:** There are three parts of a user defined function:

1. **Function declaration** or **Function prototype**
2. **Function definition**
3. **Function call**
4. **Function Declaration:** Functionsneed to be declared before their use like variables.The syntax for function declaration is given as:

**return \_type function\_name (argument\_list** with data types**);**

**OR**

**return \_type function\_name (parameter\_list** with data types**);**

Here,

1. **return\_type** 🡪 ***the type of value returned by function***.
2. **function\_name** 🡪 ***unique name identifying function.***
3. **argument\_list** or **parameter\_list** 🡪 ***list of types & names of parameters (separated by commas)***.
4. **return\_type** and **argument\_list** may be **int**, **float**, **char**, **void**, **double**, **long double** etc.
5. The declaration should end with semicolon (**;**).

Examples: **int factorial (int);**

**float addition (float , float);**

**NOTE:**

1. Declarationpart may be avoided if definition starts before **main** body.
2. Function name should be valid identifier; not any built-in function name.
3. **Function Definition:** Defining a function means **writing the actual code for function which performs the specific task**. Definition of any user defined function always given outside the main body the syntax of definition is as follows:

**return \_type function\_name (argument\_list** with data types**)**

**{ *// start of function body***

**statement1;**

**:**

**.**

**:**

**statementN;**

**} *// end of function body***

For Example:

***// function definition*** *for* ***computing factorial of a number*** *and* ***return to main body***

**int factorial (int x)**

**{**

**if (x==1)**

**return (1);**

**else**

**return (x\*factorial (x-1));**

**}**

1. **Function Call:** In order to use the defined function in a program, we need to access it by calling it using a **function call**. User defined functions may call any function by its name and with or without parameter as they are declared and defined. **There should be no mismatching in each part**.

The function which is called in other function is known as **called (or “callee”) function** whereas the function which has called that function, is known as **calling (or “caller”) function**.

E.g. Let above function to be called inside the main body as shown below:

**void main()**

**{**

**int a, fact;**

**printf (“enter the number”);**

**scanf(“%d”,&a);**

**fact = factorial (a); *//calling of factorial( )***

**printf (“the factorial of number is=%d”, fact);**

**getch ();**

**}**

* **Types of functions:** Functions can be categorized in following ways:

1. **Functions with NO arguments & NO return value:** These types of functions neither receive any input (i.e. arguments) from the calling function nor they return any value. They simply perform their specified task when called. Therefore their ***return\_type*** & ***argument\_list*** is declared as **void**. So the syntax for such type of functions is:

**void function\_name (void) OR void function\_name ( )**

//Program to calculate sum of given two numbers

**#include<stdio.h>**

**#include<conio.h>**

**void add (); // Function declaration *with NO arguments & NO return value***

**void main()**

**{**

**add (); //calling of function**

**getch ();**

**}**

**// Function definition**

**void add ()**

**{**

**int x, y, sum;**

**printf (“Enter any two numbers”);**

**scanf(“%d%d”, &x, &y);**

**sum = x+y;**

**printf (“The sum is=%d”, sum);**

**}**

1. **Functions with arguments & NO return value:** These types of functions only receive the arguments (parameters) from the calling function but they do not return any value. After receiving value they perform their specified task when called and display the result, if required.

The syntax for such type of functions is:

**void function\_name (argument\_list) OR void function\_name (parameter\_list)**

//Program to calculate sum of given two numbers

**#include<stdio.h>**

**#include<conio.h>**

**void add (int, int); // Function declaration *with arguments & NO return value***

**void main()**

**{**

**int a, b;**

**printf (“Enter any two numbers”);**

**scanf(“%d%d”, &a, &b);**

**add (a,b); //calling of function**

**getch ();**

**}**

**// Function definition**

**void add (int x, int y)**

**{**

**int sum;**

**sum = x+y;**

**printf (“The sum is=%d”, sum);**

**}**

1. **Functions with arguments & return value:** These types of functions receive the arguments (parameters) from the calling function and also return value i.e. after performing the specified task they return the computed result to the calling function.

The syntax for such type of functions is:

**return\_type function\_name (argument\_list)**

**OR**

**return\_type function\_name (parameter\_list)**

//Program to calculate sum of given two numbers

**#include<stdio.h>**

**#include<conio.h>**

**int add (int, int); // Function declaration *with arguments & return value***

**void main()**

**{**

**int a, b, result;**

**printf (“Enter any two numbers”);**

**scanf(“%d%d”, &a, &b);**

**result = add (a,b); //calling of function**

**printf (“The sum is=%d”, sum);**

**getch ();**

**}**

**// Function definition**

**void add (int x, int y)**

**{**

**int sum;**

**sum = x+y;**

**return sum;**

**}**

1. **Functions with NO arguments but return a value:** These types of functions neither receive any input (i.e. arguments) from the calling function nor they return any value. They simply perform their specified task when called. Therefore their ***return\_type*** & ***argument\_list*** is declared as **void**. So the syntax for such type of functions is:

**void function\_name (void) OR void function\_name ( )**

//Program to calculate sum of given two numbers

**#include<stdio.h>**

**#include<conio.h>**

**void add (); // Function declaration *with NO arguments & but return value***

**void main()**

**{**

**int result;**

**result = add (); //calling of function**

**printf (“The sum is=%d”, result);**

**getch ();**

**}**

**// Function definition**

**void add ()**

**{**

**int x, y, sum;**

**printf (“Enter any two numbers”);**

**scanf(“%d%d”, &x, &y);**

**sum = x+y;**

**return sum;**

**}**

**Recursive Functions: A function which can call itself is known as** **recursive function**. These functions are used to solve certain problems which are recursive in nature such as computing factorial of a number or Fibonacci series etc.

**//Program to compute factorial of given number using recursion**

**#include<stdio.h>**

**#include<conio.h>**

**int factorial (int); // Function declaration**

**void main()**

**{**

**int a, fact;**

**printf (“enter the number”);**

**scanf(“%d”,&a);**

**fact = factorial (a); // Calling factorial( ) function**

**printf (“the factorial of number is=%d”, fact);**

**getch ();**

**}**

**int factorial (int x) // Function definition**

**{**

**if (x==1)**

**return (1);**

**else**

**return (x\*factorial (x-1));**

**}**

**Scope Rules:** A **main** program could contain many functions as well as any function can contain any other function also within them (i.e. nested functions). Scope rules are the rules that govern the visibility region (i.e. accessibility) for a piece of code (or entity) in the program.

Scope rules are:

1. The scope of an entity is the region where it is declared (i.e. ***local*** or ***global***).
2. A global entity is visible to all, including the function in which that entity is declared.
3. An entity declared in the scope of another entity is always a different entity, even if, their names are identical.

**NOTE:** If a function has variable declarations inside its own body, it referred to as **local variables** (unknown to other functions)henceno other function can access those variables; whereas variables which are declared outside all the functions and can be accessed by all functions referred to as **global variables**.

***Refer to class notes for more programs…***

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

**UNIT - IV**

**Arrays:** Array is a ***collection of similar (homogenous) data type*** items where a single variable holds only one value at a time in the memory. Arrays ***require contiguous memory locations*** to store data.

Array declaration syntax: ***data\_type array\_name [size];***

* ***data\_type*** may be any fundamental data type such as int, char, float, double etc.
* ***array\_name*** is any valid identifier.
* ***size*** should be any integer constant value.

For example: ***int num [10];***

* Here, ***num*** is an integer array which can hold maximum 10 integers starting from location ***0*** *to* ***(size - 1)*** i.e. 0 to 9 in this case.
* The address of num starts from ***index 0*** and next element is stored in just contiguous (adjacent) memory location as shown below:

For example: ***int num [10] = {18, 128, 20, 35, 254, 65, 35, 70, 83, 105};***

All 10 elements shown as:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***18*** | ***128*** | ***20*** | ***35*** | ***254*** | ***65*** | ***35*** | ***70*** | ***83*** | ***105*** |

**num[0] num[1] num[2] num[3] num[4] num[5] num[6] num[7] num[8] num[9]**

**NOTE:** In the case of character array, the last element is special null character (**‘\0’**) are stored. It means if we declare character array of size 10 then only 9 characters can be stored starting from index 0 to 8 and at last null character ‘\0’stored at index 9 as shown below (detailed study of character arrays is given under the topic STRINGS):

***For example: char ch[10] = {‘A’, ’B’, ’C’, ’D’, ’E’, ’F’, ’G’, ’H’, ’I’, ’\0’};***

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **‘A’** | **‘B’** | **‘C’** | **‘D’** | **‘E’** | **‘F’** | **‘G’** | **‘H’** | **‘I’** | **‘\0’** |

**ch[0] ch[1] ch[2] ch[3] ch[4] ch[5] ch[6] ch[7] ch[8] ch[9]**

This is called linear array or one dimension. Memory occupied by linear array is computed as:

***Memory occupied by linear array = size of data type \* size of array***

In above case, ***num*** occupies **2\*10 = 20 bytes** and ***ch*** occupies **1\*10=10** **bytes** contiguous memory blocks.

**NOTE:** For the computation of **ith location address** following formula applies:

**Address of ith location = base address + data type size \* (i - 1)**

**base address** is **address for 0th element** and **data type size** 🡪 (for **int** = **2 bytes**, **char** = **1 byte**, **float** = **4 bytes**, **double** = **8 bytes**, **long double** = **10 bytes**).

**Program:** Program that prints out the memory locations in which the elements of this array are stored.

***main( )***

***{***

***int num[ ] = { 24, 34, 12, 44, 56, 17 } ;***

***int i ;***

***for ( i = 0 ; i <= 5 ; i++ )***

***{***

***printf ( "\nelement no. %d\t ", i ) ;***

***printf ( "address = %u", &num[i] ) ;***

***}***

***}***

The output of this program would look like this:

***element no. 0 address = 65512***

***element no. 1 address = 65514***

***element no. 2 address = 65516***

***element no. 3 address = 65518***

***element no. 4 address = 65520***

***element no. 5 address = 65522***

**Program: *//Program to access the elements of array***

***void main( )***

***{***

***int num[ ] = { 24, 34, 12, 44, 56, 17 } ;***

***int i ;***

***for ( i = 0 ; i <= 5 ; i++ )***

***{***

***printf ( "\n address = %u \t", &num[i] ) ;***

***printf ( "element = %d", num[i] ) ;***

***}***

***}***

The output of this program would be:

***address = 65512 element = 24***

***address = 65514 element = 34***

***address = 65516 element = 12***

***address = 65518 element = 44***

***address = 65520 element = 56***

***address = 65522 element = 17***

**// program for accessing array elements**

***void main()***

***{***

***int i;***

***int num[5]={1,2,3,4,5};***

***for (i=0;i<5;i++)***

***printf(“%d\t”, num[i]);***

***}***

**Output:** ***1 2 3 4 5***

Similarly, we can store element in the array using scanf () function.

**Program:** Write a program to find average marks obtained by a class of 30 students in a test.

***main( )***

***{***

***int avg, sum = 0, i ;***

***int marks[30] ;* /\* array declaration \*/**

***printf ( "\nEnter marks of 30 students:" ) ;***

***for ( i = 0 ; i <= 29 ; i++ )***

***{***

***scanf ( "%d", &marks[i] ) ;* /\* store data in array \*/**

***}***

***for ( i = 0 ; i <= 29 ; i++ )***

***sum = sum + marks[i] ;* /\*read & add data from array\*/**

***avg = sum / 30 ;***

***printf ( "\nAverage marks = %d", avg ) ;***

***}***

**Multidimensional array:** C permits multi-dimensional arrays. It is possible for arrays to have two or more dimensions. **The two-dimensional array is also called a** **“matrix”**.

Two dimensional arrays can be manipulated by matrix of ***m* x *n size*** by declaring as: ***a[m][n];***

E.g. ***int num[2][5]*** can be initialize as:

***num[2][5] = { {12, 32, 63, 94, 75}, {68, 87, 28, 29, 70} };***

**NOTE:** **In the same way character & floating point arrays can also be initialized.**

**Storing two dimensional array elements in the memory:** The above 2-D array can be shown as below by matrix form.

**Column🡪 0 1 2 3 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **12** | **32** | **63** | **94** | **75** |
| **68** | **87** | **28** | **29** | **70** |

**Row 0**

**Row 1**

**NOTE:** The 2-D arrays can be stored in the memory either by row-major order or column-major order. Since elements are stored only in contiguous locations in memory. Hence, in row-major order elements are stored row by row and in column-major order elements are stored column by column.

Above array can be stored in memory as below:

1. **Row-major order:**

num[0][0] =

num[0][1] =

num[0][2] =

num[0][3] =

num[0][4] =

num[1][0] =

num[1][1] =

num[1][2] =

num[1][3] =

num[1][4] =

And **column major order** as shown below:

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

num[0][0]

num[1][0]

num[0][1]

num[1][1]

num[0][2]

num[1][2]

num[0][3]

num[1][3]

num[0][4]

num[1][4]

**NOTE: In C language, by default array elements are stored in row-major order.**

**Initializing a 2-Dimensional Array:** 2-D arrays can be initialized at run time (dynamically) or at compile time (statically) as required.

* **Dynamic initialization:**

***void main( )***

***{***

***int stud[4][2] ;* /\* 2-D array declaration \*/**

***int i, j ;***

***for ( i = 0 ; i <= 3 ; i++ )***

***{***

***printf ( "\n Enter roll no. and marks" ) ;***

***scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;* /\* store data in array \*/**

***}***

***for ( i = 0 ; i <= 3 ; i++ )***

***printf ("\n%d \t%d", stud[i][0], stud[i][1] ) ;* /\*read data from array\*/**

***}***

* **Static initialization:**

***void main( )***

***{***

***int stud[4][2] = {***

***{ 1234, 56 },***

***{ 1212, 33 },***

***{ 1434, 80 },***

***{ 1312, 78 }***

***} ;***

or even this would work...

***int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;***

There are two parts to the program—in the first part through a **for** loop we read in the values of roll no. and marks, whereas, in second part through another **for** loop we print out these values. Following figure shows the arrangement of respective row & column values of matrix.

|  |  |  |  |
| --- | --- | --- | --- |
| **column No. 0** | | **column No. 1** | |
| **row no. 0** | **1234** | | **56** |
| **row no. 1** | **1212** | | **33** |
| **row no. 2** | **1434** | | **80** |
| **row no. 3** | **1312** | | **78** |

**NOTE:** **It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.**

Thus the declarations,

***int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;***

***int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;*** are perfectly acceptable,

Whereas,

***int arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ;***

***int arr[ ][ ] = { 12, 34, 23, 45, 56, 45 } ;*** would never work.

**Varying size arrays:**

1. The size of array is fixed but in some compilers, it is permitted to give size of array at run time. This type of array is known as varying size array. Varying or unknown size array are used to implement stack and queue data structures. This saves the memory at run time.
2. In some cases, it is not known to programmer that how much length should declare to array. Such type of situation can be handled by varying size array.

***Refer to class notes for more programs…***

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

**Strings:** The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. **Character arrays are also called strings**.

A string constant is a one-dimensional array of characters terminated by a null ( ‘**\0**’ ). For example,

**char name[ ] = { 'R', 'A', 'J', 'E', 'E', 'V', '\0' } ;**

**Each character in the array occupies one byte of memory and the last character is always ‘\0’**. What character is this? It looks like two characters, but it is actually only one character, with the \ indicating that what follows it is something special. ‘\0’ is called null character. Note that **‘\0’** and **‘0’** are not same. **ASCII value of ‘\0’ is 0**, whereas **ASCII value of ‘0’ is 48**.

**The terminating null (‘\0’) is important, because it is the only way the functions that work with a string can know where the string ends.**

Example figure of character array has been shown (under the topic ARRAYS) for the way a character array is stored in memory. Note that the elements of the character array are stored in contiguous memory locations.

C provides a shortcut for initializing strings. For example, the string used above can also be initialized as,

**char name[ ] = "RAJEEV" ;**

We can use any case (upper case / lower case) or mixture of cases while initializing strings as:

**char name[ ] = "Rajeev" ;**

**Note that, in this declaration ‘\0’ is not necessary. C inserts the null character automatically.**

**Program to demonstrate printing of a string in three different ways:**

1. By testing end of character array value:

**void main( )**

**{**

**char name[ ] = "Rajeev" ;**

**int i = 0 ;**

**while ( i <= 5 )**

**{**

**printf ( "%c", name[i] ) ;**

**i++ ;**

**}**

**}**

Output: **Rajeev**

1. As we know that each character array always ends with a **‘\0’**. Therefore, can also write the while loop without using the final value i.e.

**void main( )**

**{**

**char name[ ] = "Rajeev" ;**

**int i = 0 ;**

**while ( name[i] != `\0' )**

**{**

**printf ( "%c", name[i] ) ;**

**i++ ;**

**}**

**}**

Output: **Rajeev**

1. printf( ) function has got a very simple way of doing it, as shown below:

**void main( )**

**{**

**char name[ ] = "Rajeev" ;**

**printf ( "%s", name ) ;**

**}**

Output: **Rajeev**

**NOTE: The %s used in printf( ) is a format specification for printing out a string.**

The same specification can be used to receive a string from the keyboard, as shown below.

**void main( )**

**{**

**char name[25] ;**

**printf ( "Enter your name " ) ;**

**scanf ( "%s", name ) ;**

**printf ( "Hello %s!", name ) ;**

**}**

While entering the string using **scanf( )** we must be careful about two things:

* The length of the string should not exceed the dimension of the character array. This is because the C compiler doesn’t perform bounds checking on character arrays.
* **scanf( )** is not capable of receiving multi-word strings. Therefore names such as ‘Rajeev Kumar’ would be unacceptable. The way to overcome this limitation is by using the in-built function **gets( )**. The usage of functions **gets( )** and its counterpart **puts( )** is shown below:

**void main( )**

**{**

**char name[25] ;**

**printf ( "Enter your full name " ) ;**

**gets ( name ) ;**

**puts ( "Hello!" ) ;**

**puts ( name ) ;**

**}**

The program and the output are self-explanatory except for the fact that, **puts( )** can display only one string at a time (hence the use of two **puts( )** in the program above).

**NOTE: gets( )** is capable of receiving only one string at a time but the plus point with **gets( )** is that it can receive a multi-word string.

**Standard Library String Functions:** With every C compiler a large set of useful string handling library functions are provided. Following figure lists the more commonly used functions along with their purpose.

|  |  |
| --- | --- |
| **Function** | **Use** |
| **strlen** | **Finds length of a string** |
| **strlwr** | **Converts a string to lowercase** |
| **strupr** | **Converts a string to uppercase** |
| **strcat** | **Appends one string at the end of another** |
| **strncat** | **Appends first n characters of a string at the end of another** |
| **strcpy** | **Copies a string into another** |
| **strncpy** | **Copies first n characters of one string into another** |
| **strcmp** | **Compares two strings** |
| **strncmp** | **Compares first n characters of two strings** |
| **strcmpi** | **Compares two strings without regard to case ("i" denotes that this function ignores case)** |
| **stricmp** | **Compares two strings without regard to case (identical to strcmpi)** |
| **strnicmp** | **Compares first n characters of two strings without regard to case** |
| **strdup** | **Duplicates a string** |
| **strchr** | **Finds first occurrence of a given character in a string** |
| **strrchr** | **Finds last occurrence of a given character in a string** |
| **strstr** | **Finds first occurrence of a given string in another string** |
| **strset** | **Sets all characters of string to a given character** |
| **strnset** | **Sets first n characters of a string to a given character** |
| **strrev** | **Reverses string** |

Out of the above list we shall discuss the functions **strlen( )**, **strcpy( )**, **strcat( )** and **strcmp( )**, since these are the most commonly used functions.

**strlen( )** 🡪 This function counts the number of characters present in a string. Its usage is illustrated in the following program.

**void main( )**

**{**

**char arr[ ] = "Introduction" ;**

**int len1, len2 ;**

**len1 = strlen ( arr ) ;**

**len2 = strlen ( "String Handling" ) ;**

**printf ( "\nstring = %s \t length = %d", arr, len1 ) ;**

**printf ( "\nstring = %s \t length = %d", " String Handling", len2 ) ;**

**}**

The output would be...

**string = Introduction length = 12**

**string = String Handling length = 15**

**strcpy( ) 🡪** This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. Here is an example of **strcpy( )** in action...

**#include<string.h>**

**void main( )**

**{**

**char str1[20], str2[20];**

**printf ( "Enter the string:\t") ;**

**gets(str1);**

**printf ( "\n The copied string is: %s", strcpy (str2, str1)) ;**

**getch();**

**}**

And here is the output...

**Enter the string: Windows**

**The copied string is: Windows**

**NOTE:** On supplying the base addresses, **strcpy( )** goes on copying the characters in source string into the target string till it doesn't encounter the end of source string (**‘\0’**). It is our responsibility to see to it that the target string’s dimension is big enough to hold the string being copied into it.

**strcat( ) 🡪** This function concatenates the source string at the end of the target string. Here is an example of **strcat( )** at work.

**void main( )**

**{**

**char source[ ] = "Students!" ;**

**char target[30] = "Hello" ;**

**strcat ( target, source ) ;**

**printf ( "source string = %s", source ) ;**

**printf ( "\n target string = %s", target ) ;**

**}**

And here is the output...

**source string = Students!**

**target string = HelloStudents!**

**NOTE:** The target string should be made big enough to hold the final string.

**strcmp( ) 🡪** This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, **strcmp( )** returns a value zero. If they’re not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is a program which puts **strcmp( )** in action.

**#include<string.h>**

**void main( )**

**{**

**char str1[20], str2[20];**

**printf ( "Enter the first string:\t") ;**

**gets(str1);**

**printf ( "Enter the second string:\t") ;**

**gets(str2);**

**if(strcmp (str1,str2)==0)**

**printf ( "\n\n Strings are equal") ;**

**else**

**printf ( "\n\n Strings are unequal") ;**

**getch();**

**}**

And here is the output...

Enter the first string: **HELLO**

Enter the first string: **Hello**

**Strings are unequal**

**#include<string.h>**

**void main( )**

**{**

**char str1[20], str2[20];**

**printf ( "Enter the first string:\t") ;**

**gets(str1);**

**printf ( "Enter the second string:\t") ;**

**gets(str2);**

**if(strcmpi (str1,str2)==0)**

**printf ( "\n\n Strings are equal") ;**

**else**

**printf ( "\n\n Strings are unequal") ;**

**getch();**

**}**

And here is the output...

Enter the first string: **HELLO**

Enter the first string: **Hello**

**Strings are equal**

**#include<string.h>**

**void main( )**

**{**

**char string1[ ] = "Jerry" ;**

**char string2[ ] = "Ferry" ;**

**int i, j, k ;**

**i = strcmp ( string1, "Jerry" ) ;**

**j = strcmp ( string1, string2 ) ;**

**k = strcmp ( string1, "Jerry boy" ) ;**

**printf ( "\n%d\t %d\t %d", i, j, k ) ;**

**}**

And here is the output...

**0 4 -32**

In the first call to **strcmp( )**, the two strings are identical—“Jerry” and “Jerry”—and the value returned by **strcmp( )** is zero. In the second call, the first character of “Jerry” doesn't match with the first character of “Ferry” and the result is 4, which is the numeric difference between ASCII value of ‘J’ and ASCII value of ‘F’. In the third call to **strcmp( )** “Jerry” doesn’t match with “Jerry boy”, because the null character at the end of “Jerry” doesn’t match the blank in “Jerry boy”. The value returned is -32, which is the value of null character minus the ASCII value of space, i.e., ‘\0’ minus ‘ ’, which is equal to -32.

**NOTE:** The exact value of mismatch will rarely concern us. All we usually want to know is whether or not the first string is alphabetically before the second string. If it is, a negative value is returned; if it isn’t, a positive value is returned. Any non-zero value means there is a mismatch.

***Refer to class notes for more programs…***

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

**Why we need Structures?** Quite often we deal with entities that are collection of dissimilar data types. For example, suppose we want to store data about a book. We might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books are to be stored, then we can follow two approaches:

1. Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.
2. Use a structure variable.

Let us begin with a program that uses arrays.

**void main( )**

**{**

**char name[3] ;**

**float price[3] ;**

**int pages[3], i ;**

**printf ( "\nEnter names, prices and no. of pages of 3 books\n" ) ;**

**for ( i = 0 ; i <= 2 ; i++ )**

**scanf ( "%c %f %d", &name[i], &price[i], &pages[i] );**

**printf ( "\nAnd this is what you entered\n" ) ;**

**for ( i = 0 ; i <= 2 ; i++ )**

**printf ( "%c %f %d\n", name[i], price[i], pages[i] );**

**}**

The program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc. To solve this problem, C provides a special data type—the structure.

**Structures:** A structure contains a number of data types grouped together. These data types may or may not be of the same type. Hence, **structure may be recognized as a collection of dissimilar data types.** The following example illustrates the use of this data type.

**void main( )**

**{**

**struct book**

**{**

**char name ;**

**float price ;**

**int pages ;**

**} ;**

**struct book b1, b2, b3 ;**

**printf ( "\nEnter names, prices & no. of pages of 3 books\n" ) ;**

**scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;**

**scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;**

**scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;**

**printf ( "\nAnd this is what you entered" ) ;**

**printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;**

**printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;**

**printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;**

**}**

This program demonstrates two fundamental aspects of structures:

* Declaration of a structure
* Accessing of structure elements

**Declaring a Structure:** In our example program, the following statement declares the structure type:

**struct book**

**{**

**char name ;**

**float price ;**

**int pages ;**

**} ;**

This statement defines a new data type called **struct book**. Each variable of this data type will consist of a character variable called **name**, a float variable called **price** and an integer variable called **pages**. The general form of a structure declaration statement is given below:

**struct <structure name>**

**{**

**structure element 1 ;**

**structure element 2 ;**

**structure element 3 ;**

**......**

**......**

**} ;**

Once the new structure data type has been defined one or more variables can be declared to be of that type. For example the variables **b1**, **b2**, **b3** can be declared to be of the type **struct book**, as,

**struct book b1, b2, b3 ;**

This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, 7 bytes—one for **name**, four for **price** and two for **pages**. These bytes are always in adjacent memory locations.

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

**struct book**

**{**

**char name[10] ;**

**float price ;**

**int pages ;**

**} ;**

**struct book b1 = { "Basic", 130.00, 550 } ;**

**struct book b2 = { "Physics", 150.80, 800 } ;**

Note the following points while declaring a structure type:

* The closing brace in the structure type declaration must be followed by a semicolon.
* It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the ‘form’ of the structure.

**Accessing Structure Elements:** After declaration of the structure type and the structure variables, now let us see how it can be accessed?

Structures use **dot (.) operator** for accessing its elements. So to refer to **pages** of the structure defined in our sample program we have to use,

**b1.pages**

Similarly, to refer to **price** we would use,

**b1.price**

**NOTE: Before the dot there must always be a structure variable and after the dot there must always be a structure element.**

**Array of Structures:** In our sample program, to store data of 100 books we would be required to use 100 different structure variables from **b1** to **b100**, which is definitely not very convenient. A better approach would be to use an array of structures. Following program shows how to use an array of structures:

**/\* Usage of an array of structures \*/**

**void main( )**

**{**

**struct book**

**{**

**char name ;**

**float price ;**

**int pages ;**

**} ;**

**struct book b[100] ;**

**int i ;**

**for ( i = 0 ; i <= 99 ; i++ )**

**{**

**printf ( "\nEnter name, price and pages " ) ;**

**scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;**

**}**

**for ( i = 0 ; i <= 99 ; i++ )**

**printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;**

**}**

**Union:** A union is a data type similar to structure but the difference lies in their storage allocation schemes. In structure, each member has its separate memory locations whereas union shares common memory space.

Hence, in structure, the number of locations allocated would be equal to the number of members in the structure.

**Size of structure variable is equal to the addition of its all the members size**

In the case of union, only one location which is large enough to collect the largest data type member in the union gets allocated (but only one member can remain in memory at a given time).

**Size of union variable is equal to its largest member’s size**

E.g. If we are storing following three types of data i.e. one **integer**, one **float** & one **character**, in a structure & union then they will require memory spaces respectively as:

**Size of structure variable = size of (int + float + char) = 2 bytes + 4 bytes + 1 byte = 7 bytes**

**Size of union variable = size of largest data type i.e. float = 4 bytes**

**Enumerated data types:** Enumerated data type offers us a way of inventing our own data type.

***Refer to class notes for more programs…***

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

**Standard C Preprocessors:** The C preprocessor is a program that processes the source program before passing it to the compiler. It is like a language defined within a language. The preprocessor directly substitutes the value or expression in source program resulting in fast execution.

C Preprocessor

Compiler

Object code

Source Program

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following preprocessor directives here:

* **Macro expansion**
* **File inclusion**
* **Conditional Compilation**
* **Miscellaneous directives**

Let us understand these features of preprocessor one by one.

* **Macro Expansion:** Macro expansion is a preprocessor statement, which is used to make the programs more readable & fast executing. Each preprocessor begins with **#define** statement and not terminated by semicolon.

Have a look at the following program:

**#define UPPER 25**

**void main( )**

**{**

**int i ;**

**for ( i = 1 ; i <= UPPER ; i++ )**

**printf ( "\n%d", i ) ;**

**}**

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main( )** through the statement,

**#define UPPER 25**

This statement is called ‘macro definition’. During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25. Here is another example of macro definition.

**#define PI 3.1415**

**void main( )**

**{**

**float r = 6.25 ;**

**float area ;**

**area = PI \* r \* r ;**

**printf ( "\nArea of circle = %f", area ) ;**

**}**

**UPPER** and **PI** in the above programs are often called ‘**macro templates**’, whereas, **25** and **3.1415** are called their corresponding ‘**macro expansions**’.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure completed, the program is handed over to the compiler.

**In C programming it is mandatory to use capital letters for macro template.** This makes it easy for programmers to pick out all the macro templates when reading through the program.

**Note that a macro template and its macro expansion are separated by blanks.** A space between **#** and **define** is optional. **Remember that a macro definition is never to be terminated by a semicolon.**

A **#define** directive is many a times used to define operators as shown below:

**#define AND &&**

**#define OR ||**

* **File Inclusion:** The C preprocessor defines the concept of file inclusion.

***Refer to class notes for more illustrations…***

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

**20 most important questions**

**Subject: CSP [RCS-201]**

**Branch: CSE/EN**

**Sem. / Session :: 2nd / 2016-17**

**UNIT-1**

1. Explain block diagram of digital computer with its functional units.
2. Define & differentiate between following:
3. System software & application software
4. Machine language & high level language
5. Compiler & interpreter
6. Standard library functions & user defined functions
7. (a) What do you mean by algorithm? What are the properties of a good algorithm? Write an algorithm to find the summation of all the integers between 10 and 500 which are divisible by 3 and 5 both.

(b) Explain components of flow chart with diagrams. Draw a flowchart for finding largest of three positive integers.

1. Convert the following:
2. **(AB584.32C)16 = ( ? )8 = ( ? )10 = ( ? )5**
3. **(564.21)8 = ( ? )10 = ( ? )2 = ( ? )16**
4. **(1101.101)10 = ( ? )2 = ( ? )8**
5. **(11001.10111)2 = ( ? )5 = ( ? )9**

**UNIT-02**

1. Write short notes on the following:
2. Keywords
3. Constants
4. Variables
5. ASCII code
6. How many types of operators are there in C? Explain each operator with at least two examples.
7. What do you mean by operator precedence? Define conversion rules with examples.
8. (a) What are data types? Explain primary & secondary data types in brief.

(b) Explain different types of storage classes with examples.

**UNIT-03**

1. (a) Briefly describe the working of nested if-else. Also write a program in C to read a 4-digit number if it is even then add up the digits otherwise multiply them and print the result.

(b) What are the different ways of passing parameters to the function? Explain with example.

1. What is the advantage of switch statement over if else statement? Write a program in C using switch statement to find the value of Y for a given value of N between 1 to 4:

|  |  |
| --- | --- |
| If N = 1 | Y = (ax + b)2 |
| If N = 2 | Y = ax2 + b3 |
| If N = 3 | Y = -ax + b |
| If N = 4 | Y = a2 + x |

1. Write short notes on the following with the help of examples in reference of C language:
2. Continue & break statements
3. Entry & exit control loops

Write C programs to calculate the sum of following series:

1. F(x) = x1 – x3 + x5 - x7 + …………………… up to nth terms.
2. S = 1! + 2! + 3! + ………………. + N!
3. What is recursive function? Write a program in C to:
4. Find out factorial of any number using function with recursion.
5. Generate Fibonacci series (0 1 1 2 3 5 8 13.........) up to nth terms.

**UNIT-04**

1. (a) What do you mean by array? Write a program in C to which returns the maximum and minimum element of a given matrix of size m x n.

(b) Write a program in C to print the multiplication of two matrices of size 3 x 3.

1. What are the fundamental, derived & user define data types in C? Explain.
2. What are the differences between structure and union? Write a program in C that accepts the roll number and name of students of a class size of 100 students along with the marks obtained by them in physics, chemistry & mathematics. Print roll number and name of top 10 students in the order of merit. The merit is based on the sum of the marks obtained in the three subjects.
3. List any five string handling & mathematical library functions with their working example.

**UNIT-05**

1. (a) What is a pointer? How it can be used to access any other variable, explain with the help of suitable example.

(b) Differentiate between call by value and call by reference with suitable example of swapping of two numbers.

1. What do you mean by preprocessor directives? Distinguish between #ifdef and #if directives with suitable examples.
2. Why we use files in C? List various file handling operations (functions) in C, with their format and use.
3. Write short notes on
4. Library functions
5. Macros
6. String manipulation functions
7. Conditional compilation

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**