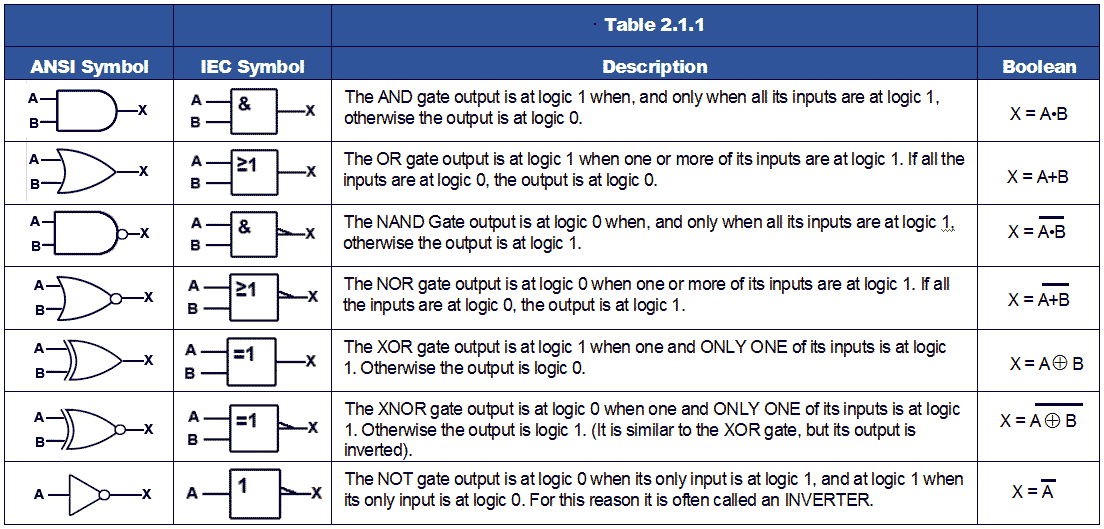
12/2017

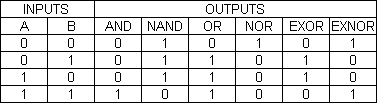
Ashutosh: Computer Languages

Computer Languages

Short notes







**Microprocessor**

**Microprocessor Overview**

Microprocessor is a controlling unit of a microcomputer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it.

Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device. Register array consists of registers identified by letters like B, C, D, E, H, L and accumulator. The control unit controls the flow of data and instructions within the computer.

**How does a Microprocessor Work?**

The microprocessor follows a sequence: Fetch, Decode, and then Execute.

Initially, the instructions are stored in the memory in a sequential order. The microprocessor fetches those instructions from the memory, then decodes it and executes those instructions till STOP instruction is reached. Later, it sends the result in binary to the output port. Between these processes, the register stores the temporarily data and ALU performs the computing functions.

**List of Terms Used in a Microprocessor**

Here is a list of some of the frequently used terms in a microprocessor −

* **Instruction Set** − It is the set of instructions that the microprocessor can understand.
* **Bandwidth** − It is the number of bits processed in a single instruction.
* **Clock Speed** − It determines the number of operations per second the processor can perform. It is expressed in megahertz (MHz) or gigahertz (GHz).It is also known as Clock Rate.
* **Word Length** − It depends upon the width of internal data bus, registers, ALU, etc. An 8bit microprocessor can process 8bit data at a time. The word length ranges from 4 bits to 64 bits depending upon the type of the microcomputer.
* **Data Types** − The microprocessor has multiple data type formats like binary, BCD, ASCII, signed and unsigned numbers.

**Features of a Microprocessor**

Here is a list of some of the most prominent features of any microprocessor −

* **Cost-effective** − The microprocessor chips are available at low prices and results its low cost.
* **Size** − The microprocessor is of small size chip, hence is portable.
* **Low Power Consumption** − Microprocessors are manufactured by using metaloxide semiconductor technology, which has low power consumption.
* **Versatility** − The microprocessors are versatile as we can use the same chip in a number of applications by configuring the software program.
* **Reliability** − The failure rate of an IC in microprocessors is very low, hence it is reliable.

**Microprocessor Classification**

A microprocessor can be classified into three categories –

1. **RISC Processors**
2. **CISC Processors**
3. **Special Processors**

**RISC Processors**

RISC stands for **Reduced Instruction Set Computer**. It is designed to reduce the execution time by simplifying the instruction set of the computer. Using RISC processors, each instruction requires only one clock cycle to execute results in uniform execution time. This reduces the efficiency as there are more lines of code, hence more RAM is needed to store the instructions. The compiler also has to work more to convert high-level language instructions into machine code.

RISC microprocessor architecture uses highly-optimized set of instructions. It is used in portable devices like Apple iPod due to its power efficiency.

**Characteristics of RISC**

The major characteristics of a RISC processor are as follows −

* It consists of simple instructions.
* It supports various datatype formats.
* It utilizes simple addressing modes and fixed length instructions for pipelining.
* It supports register to use in any context.
* One cycle execution time.
* “LOAD” and “STORE” instructions are used to access the memory location.
* It consists of larger number of registers.
* It consists of less number of transistors.

**CISC Processor**

CISC stands for Complex Instruction Set Computer. It is designed to minimize the number of instructions per program, ignoring the number of cycles per instruction. The emphasis is on building complex instructions directly into the hardware.

The compiler has to do very little work to translate a high-level language into assembly level language/machine code because the length of the code is relatively short, so very little RAM is required to store the instructions.

**Architecture of CISC**

Its architecture is designed to decrease the memory cost because more storage is needed in larger programs resulting in higher memory cost. To resolve this, the number of instructions per program can be reduced by embedding the number of operations in a single instruction.

**Characteristics of CISC**

* Variety of addressing modes.
* Larger number of instructions.
* Variable length of instruction formats.
* Several cycles may be required to execute one instruction.
* Instruction-decoding logic is complex.
* One instruction is required to support multiple addressing modes.

**Special Processors**

These are the processors which are designed for some special purposes. Few of the special processors are briefly discussed –

**Coprocessor**

A coprocessor is a specially designed microprocessor, which can handle its particular function many times faster than the ordinary microprocessor.

For example − Math Coprocessor.

**Input/Output Processor**

It is a specially designed microprocessor having a local memory of its own, which is used to control I/O devices with minimum CPU involvement.

**Transputer (Transistor Computer)**

A transputer is a specially designed microprocessor with its own local memory and having links to connect one transputer to another transputer for interprocessor communications. It was first designed in 1980 by Inmos and is targeted to the utilization of VLSI technology.

A transputer can be used as a single processor system or can be connected to external links, which reduces the construction cost and increases the performance.

For example − 16bit T212, 32bit T425, the floating point (T800, T805 & T9000) processors.

**DSP (Digital Signal Processor)**

This processor is specially designed to process the analog signals into a digital form. This is done by sampling the voltage level at regular time intervals and converting the voltage at that instant into a digital form. This process is performed by a circuit called an analogue to digital converter, A to D converter or ADC.

A DSP contains the following components −

* **Program Memory** − It stores the programs that DSP will use to process data.
* **Data Memory** − It stores the information to be processed.
* **Compute Engine** − It performs the mathematical processing, accessing the program from the program memory and the data from the data memory.
* **Input/output** − It connects to the outside world.

Its applications are −

* Sound and music synthesis
* Audio and video compression
* Video signal processing
* 2D and 3d graphics acceleration.

For example − Texas Instrument’s TMS 320 series, e.g., TMS 320C40, TMS320C50.

**Microprocessor - 8085 Architecture**

8085 is pronounced as "eighty-eighty-five" microprocessor. It is an 8bit microprocessor designed by Intel in 1977 using NMOS technology.

It has the following configuration −

* 8bit data bus
* 16bit address bus, which can address upto 64KB
* A 16bit program counter
* A 16bit stack pointer
* Six 8bit registers arranged in pairs: BC, DE, HL
* Requires +5V supply to operate at 3.2 MHZ single phase clock

It is used in washing machines, microwave ovens, mobile phones, etc.

**8085 Microprocessor – Functional Units**

8085 consists of the following functional units –

* **Accumulator -** It is an 8bit register used to perform arithmetic, logical, I/O & LOAD/STORE operations. It is connected to internal data bus & ALU.
* **Arithmetic and logic unit -** As the name suggests, it performs arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8bit data.
* **General purpose register -** There are 6 general purpose registers in 8085 processor, i.e. B, C, D, E, H & L. Each register can hold 8bit data. These registers can work in pair to hold 16bit data and their pairing combination is like BC, DE & HL.
* **Program counter -** It is a 16bitregister used to store the memory address location of the next instruction to be executed. Microprocessor increments the program whenever an instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed.
* **Stack pointer -** It is also a 16bit register works like stack, which is always incremented/decremented by 2 during push & pop operations.
* **Temporary register -** It is an 8bit register, which holds the temporary data of arithmetic and logical operations.
* **Flag register -** It is an 8bit register having five 1bit flip-flops, which holds either 0 or 1 depending upon the result stored in the accumulator.

These are the set of 5 flip-flops −

* Sign (S)
* Zero (Z)
* Auxiliary Carry (AC)
* Parity (P)
* Carry (C)

Its bit position is shown in the following table −

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| S | Z |  | AC |  | P |  | CY |

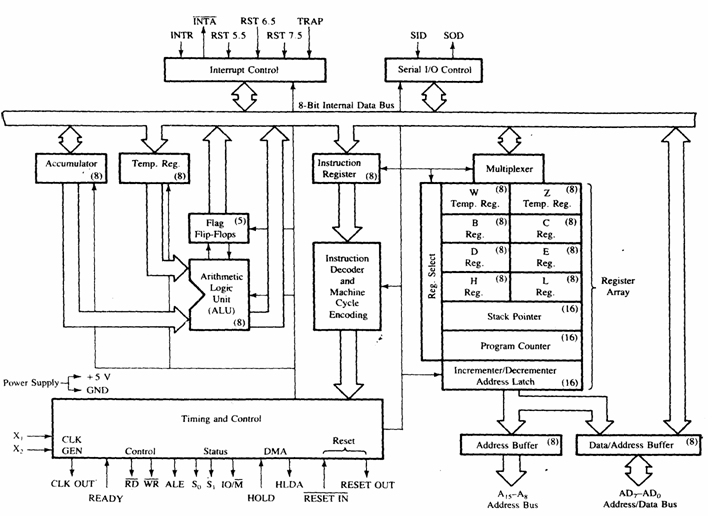
* **Instruction register and decoder -** It is an 8bitregister. When an instruction is fetched from memory then it is stored in the Instruction register. Instruction decoder decodes the information present in the Instruction register.
* **Timing and control unit -** It provides timing and control signal to the microprocessor to perform operations. Following are the timing and control signals, which control external and internal circuits −
  + **Control Signals:** READY, RD’, WR’, ALE
  + **Status Signals:** S0, S1, IO/M’
  + **DMA Signals:** HOLD, HLDA
  + **RESET Signals:** RESET IN, RESET OUT
* **Interrupt control -** As the name suggests it controls the interrupts during a process. When a microprocessor is executing a main program and whenever an interrupt occurs, the microprocessor shifts the control from the main program to process the incoming request. After the request is completed, the control goes back to the main program.

There are 5 interrupt signals in 8085 microprocessor: INTR, RST 7.5, RST 6.5, RST 5.5, TRAP.

* **Serial Input/output control -** It controls the serial data communication by using these two instructions: SID (Serial input data) and SOD (Serial output data).
* **Address buffer and address-data buffer -** The content stored in the stack pointer and program counter is loaded into the address buffer and address data buffer to communicate with the CPU. The memory and I/O chips are connected to these buses; the CPU can exchange the desired data with the memory and I/O chips.
* **Address bus and data bus -** Data bus carries the data to be stored. It is bidirectional, whereas address bus carries the location to where it should be stored and it is unidirectional. It is used to transfer the data & Address I/O devices.

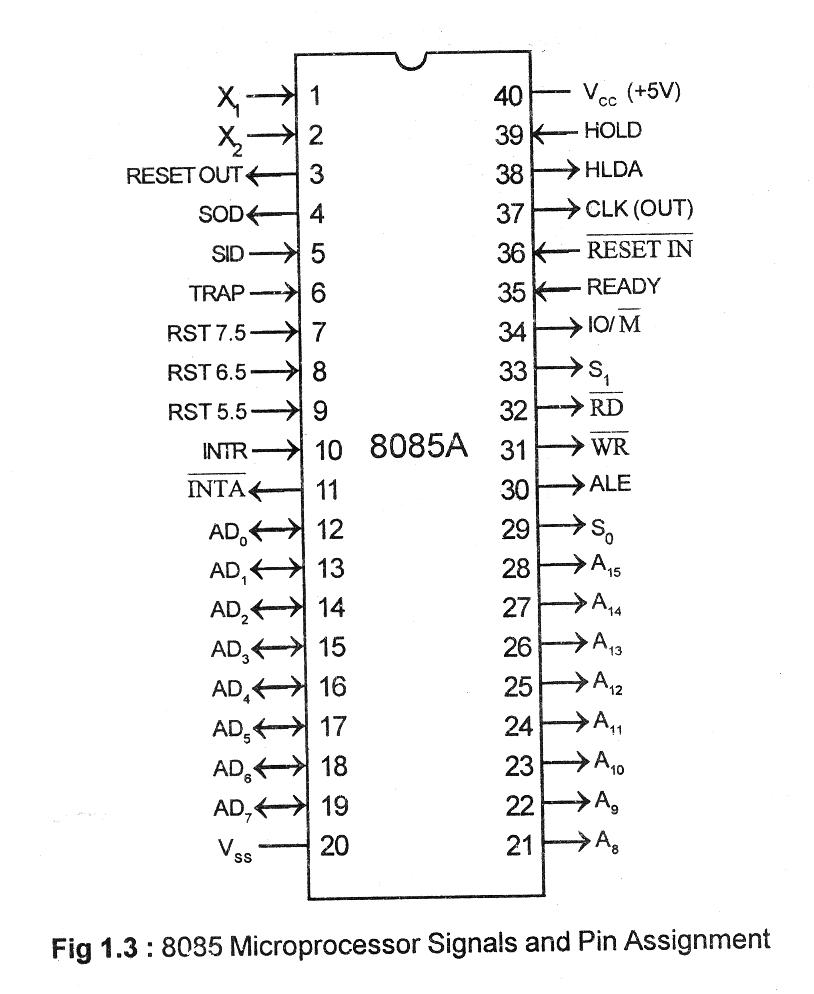
**8085 Architecture**

We have tried to depict the architecture of 8085 with this following image −



**Microprocessor - 8085 Pin Configuration**

The following image depicts the pin diagram of 8085 Microprocessor –



The pins of a 8085 microprocessor can be classified into seven groups −

* **Address bus -** A15A8, it carries the most significant 8bits of memory/IO address.
* **Data bus -** AD7AD0, it carries the least significant 8bit address and data bus.
* **Control and status signals -** These signals are used to identify the nature of operation. There are 3 control signal and 3 status signals.

Three control signals are RD, WR & ALE.

* + **RD −** This signal indicates that the selected IO or memory device is to be read and is ready for accepting data available on the data bus.
  + **WR −** This signal indicates that the data on the data bus is to be written into a selected memory or IO location.
  + **ALE −** It is a positive going pulse generated when a new operation is started by the microprocessor. When the pulse goes high, it indicates address. When the pulse goes down it indicates data.

Three status signals are IO/M, S0 & S1.

* **IO/M -** This signal is used to differentiate between IO and Memory operations, i.e. when it is high indicates IO operation and when it is low then it indicates memory operation.
* **S1 & S0 -** These signals are used to identify the type of current operation.
* **Power supply -** There are 2 power supply signals − VCC & VSS. VCC indicates +5v power supply and VSS indicates ground signal.
* **Clock signals -** There are 3 clock signals, i.e. X1, X2, CLK OUT.
  + **X1, X2 −** A crystal (RC, LC N/W) is connected at these two pins and is used to set frequency of the internal clock generator. This frequency is internally divided by 2.
  + **CLK OUT −** This signal is used as the system clock for devices connected with the microprocessor.
* **Interrupts & externally initiated signals -** Interrupts are the signals generated by external devices to request the microprocessor to perform a task. There are 5 interrupt signals, i.e. TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR. We will discuss interrupts in detail in interrupts section.
  + **INTA −** It is an interrupt acknowledgment signal.
  + **RESET IN −** This signal is used to reset the microprocessor by setting the program counter to zero.
  + **RESET OUT −** This signal is used to reset all the connected devices when the microprocessor is reset.
  + **READY −** This signal indicates that the device is ready to send or receive data. If READY is low, then the CPU has to wait for READY to go high.
  + **HOLD −** This signal indicates that another master is requesting the use of the address and data buses.
  + **HLDA (HOLD Acknowledge) −** It indicates that the CPU has received the HOLD request and it will relinquish the bus in the next clock cycle. HLDA is set to low after the HOLD signal is removed.
* **Serial I/O signals -** There are 2 serial signals, i.e. SID and SOD and these signals are used for serial communication.
  + **SOD (Serial output data line) −** The output SOD is set/reset as specified by the SIM instruction.
  + **SID (Serial input data line) −** The data on this line is loaded into accumulator whenever a RIM instruction is executed.

**8085 Addressing Modes & Interrupts**

Now let us discuss the addressing modes in 8085 Microprocessor.

**Addressing Modes in 8085**

These are the instructions used to transfer the data from one register to another register, from the memory to the register, and from the register to the memory without any alteration in the content. Addressing modes in 8085 is classified into 5 groups –

* **Immediate addressing mode -** In this mode, the 8/16bit data is specified in the instruction itself as one of its operand. For example: MVI K, 20F: means 20F is copied into register K.
* **Register addressing mode -** In this mode, the data is copied from one register to another. For example: MOV K, B: means data in register B is copied to register K.
* **Direct addressing mode -** In this mode, the data is directly copied from the given address to the register. For example: LDB 5000K: means the data at address 5000K is copied to register B.
* **Indirect addressing mode -** In this mode, the data is transferred from one register to another by using the address pointed by the register. For example: MOV K, B: means data is transferred from the memory address pointed by the register to the register K.
* **Implied addressing mode -** This mode doesn’t require any operand; the data is specified by the opcode itself. For example: CMP.

**Interrupts in 8085**

Interrupts are the signals generated by the external devices to request the microprocessor to perform a task. There are 5 interrupt signals, i.e. TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.

Interrupt are classified into following groups based on their parameter −

* **Vector interrupt −** In this type of interrupt, the interrupt address is known to the processor. For example: RST7.5, RST6.5, RST5.5, TRAP.
* **Non-Vector interrupt −** In this type of interrupt, the interrupt address is not known to the processor so, the interrupt address needs to be sent externally by the device to perform interrupts. For example: INTR.
* **Maskable interrupt −** In this type of interrupt, we can disable the interrupt by writing some instructions into the program. For example: RST7.5, RST6.5, RST5.5.
* **Non-Maskable interrupt −** In this type of interrupt, we cannot disable the interrupt by writing some instructions into the program. For example: TRAP.
* **Software interrupt −** In this type of interrupt, the programmer has to add the instructions into the program to execute the interrupt. There are 8 software interrupts in 8085, i.e. RST0, RST1, RST2, RST3, RST4, RST5, RST6, and RST7.
* **Hardware interrupt −** There are 5 interrupt pins in 8085 used as hardware interrupts, i.e. TRAP, RST7.5, RST6.5, RST5.5, INTA.

Note − NTA is not an interrupt, it is used by the microprocessor for sending acknowledgement. TRAP has the highest priority, then RST7.5 and so on.

**Interrupt Service Routine (ISR)**

A small program or a routine that when executed, services the corresponding interrupting source is called an ISR.

* **TRAP -** It is a non-maskable interrupt, having the highest priority among all interrupts. Bydefault, it is enabled until it gets acknowledged. In case of failure, it executes as ISR and sends the data to backup memory. This interrupt transfers the control to the location 0024H.
* **RST7.5 -** It is a maskable interrupt, having the second highest priority among all interrupts. When this interrupt is executed, the processor saves the content of the PC register into the stack and branches to 003CH address.
* **RST 6.5 -** It is a maskable interrupt, having the third highest priority among all interrupts. When this interrupt is executed, the processor saves the content of the PC register into the stack and branches to 0034H address.
* **RST 5.5 -** It is a maskable interrupt. When this interrupt is executed, the processor saves the content of the PC register into the stack and branches to 002CH address.
* **INTR -** It is a maskable interrupt, having the lowest priority among all interrupts. It can be disabled by resetting the microprocessor.

When **INTR signal goes high**, the following events can occur −

The microprocessor checks the status of INTR signal during the execution of each instruction.

When the INTR signal is high, then the microprocessor completes its current instruction and sends active low interrupt acknowledge signal.

When instructions are received, then the microprocessor saves the address of the next instruction on stack and executes the received instruction.

**Microprocessor - 8085 Instruction Sets**

Let us take a look at the programming of 8085 Microprocessor.

Instruction sets are instruction codes to perform some task. It is classified into five categories.

|  |  |  |  |
| --- | --- | --- | --- |
| S. No | Instruction Type | Description | Example |
| 1 | Control Instructions | Following is the table showing the list of Control instructions with their meanings. | NOP, HLT, DI, EI, RIM, SIM |
| 2 | Logical Instructions | Following is the table showing the list of Logical instructions with their meanings. | CMP, CPI, ANA, ANI, XRA, XRI, ORA, ORI, RLC, RRC, RAL, RAR, CMA, CMC, STC. |
| 3 | Branching Instructions | Following is the table showing the list of Branching instructions with their meanings. | JMP(JC,JNC,JP,JM,JZ,JNZ,JPE,JPO), (CC,CNC,CP,CM,CZ,CNZ,CPE,CPO), (RC,RNC,RP,RM,RZ,RNZ,RPE,RPO), PCHL, RST |
| 4 | Arithmetic Instructions | Following is the table showing the list of Arithmetic instructions with their meanings. | ADD, ADC, ADI, ACI, LXI, DAD, SUB, SBB, SUI, SBI, INR, INX, DCR, DCX, DAA |
| 5 | Data Transfer Instructions | Following is the table showing the list of Data-transfer instructions with their meanings. | MOV, MVI, LDA, LDAX, LXI, LHLD, STA, STAX, SHLD, XCHG, SPHL, XTHL, PUSH, POP, OUT, IN |

**Control Instruction**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Operand** | **Meaning** | **Explanation** |
| NOP | None | No operation | No operation is performed, i.e., the instruction is fetched and decoded. |
| HLT | None | Halt and enter wait state | The CPU finishes executing the current instruction and stops further execution. An interrupt or reset is necessary to exit from the halt state. |
| DI | None | Disable interrupts | The interrupt enable flip-flop is reset and all the interrupts are disabled except TRAP. |
| EI | None | Enable interrupts | The interrupt enable flip-flop is set and all the interrupts are enabled. |
| RIM | None | Read interrupt mask | This instruction is used to read the status of interrupts 7.5, 6.5, 5.5 and read serial data input bit. |
| SIM | None | Set interrupt mask | This instruction is used to implement the interrupts 7.5, 6.5, 5.5, and serial data output. |

**Logical Instruction**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Operand** | **Meaning** | **Explanation** |
| CMP | R  M | Compare the register or memory with the accumulator | The contents of the operand (register or memory) are M compared with the contents of the accumulator. |
| CPI | 8-bit data | Compare immediate with the accumulator | The second byte data is compared with the contents of the accumulator. |
| ANA | R  M | Logical AND register or memory with the accumulator | The contents of the accumulator are logically AND with M the contents of the register or memory, and the result is placed in the accumulator. |
| ANI | 8-bit data | Logical AND immediate with the accumulator | The contents of the accumulator are logically AND with the 8-bit data and the result is placed in the accumulator. |
| XRA | R  M | Exclusive OR register or memory with the accumulator | The contents of the accumulator are Exclusive OR with M the contents of the register or memory, and the result is placed in the accumulator. |
| XRI | 8-bit data | Exclusive OR immediate with the accumulator | The contents of the accumulator are Exclusive OR with the 8-bit data and the result is placed in the accumulator. |
| ORA | R  M | Logical OR register or memory with the accumulator | The contents of the accumulator are logically OR with M the contents of the register or memory, and result is placed in the accumulator. |
| ORI | 8-bit data | Logical OR immediate with the accumulator | The contents of the accumulator are logically OR with the 8-bit data and the result is placed in the accumulator. |
| RLC | None | Rotate the accumulator left | Each binary bit of the accumulator is rotated left by one position. Bit D7 is placed in the position of D0 as well as in the Carry flag. CY is modified according to bit D7. |
| RRC | None | Rotate the accumulator right | Each binary bit of the accumulator is rotated right by one position. Bit D0 is placed in the position of D7 as well as in the Carry flag. CY is modified according to bit D0. |
| RAL | None | Rotate the accumulator left through carry | Each binary bit of the accumulator is rotated left by one position through the Carry flag. Bit D7 is placed in the Carry flag, and the Carry flag is placed in the least significant position D0. CY is modified according to bit D7. |
| RAR | None | Rotate the accumulator right through carry | Each binary bit of the accumulator is rotated right by one position through the Carry flag. Bit D0 is placed in the Carry flag, and the Carry flag is placed in the most significant position D7. CY is modified according to bit D0. |
| CMA | None | Complement accumulator | The contents of the accumulator are complemented. No flags are affected. |
| CMC | None | Complement carry | The Carry flag is complemented. No other flags are affected. |
| STC | None | Set Carry | Set Carry |

**Branching Instruction**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Operand** | **Meaning** | **Explanation** |
| **JMP** | **16-bit address** | Jump unconditionally | The program sequence is transferred to the memory address given in the operand. |
| |  |  |  | | --- | --- | --- | | OP | Description | F S | | JC | Jump on Carry | CY=1 | | JNC | Jump on no Carry | CY=0 | | JP | Jump on positive | S=0 | | JM | Jump on minus | S=1 | | JZ | Jump on zero | Z=1 | | JNZ | Jump on no zero | Z=0 | | JPE | Jump on parity even | P=1 | | JPO | Jump on parity odd | P=0 | | **16-bit address** | Jump conditionally | The program sequence is transferred to the memory address given in the operand based on the specified flag of the PSW. |
| |  |  |  | | --- | --- | --- | | OP | Description | F S | | CC | Call on Carry | CY=1 | | CNC | Call on no Carry | CY=0 | | CP | Call on positive | S=0 | | CM | Call on minus | S=1 | | CZ | Call on zero | Z=1 | | CNZ | Call on no zero | Z=0 | | CPE | Call on parity even | P=1 | | CPO | Call on parity odd | P=0 | | **16-bit address** | Unconditional subroutine call | The program sequence is transferred to the memory address given in the operand. Before transferring, the address of the next instruction after CALL is pushed onto the stack. |
| **RET** | **None** | Return from subroutine unconditionally | The program sequence is transferred from the subroutine to the calling program. |
| |  |  |  | | --- | --- | --- | | OP | Description | F S | | RC | Return on Carry | CY=1 | | RNC | Return on no Carry | CY=0 | | RP | Return on positive | S=0 | | RM | Return on minus | S=1 | | RZ | Return on zero | Z=1 | | RNZ | Return on no zero | Z=0 | | RPE | Return on parity even | P=1 | | RPO | Return on parity odd | P=0 | | **None** | Return from subroutine conditionally | The program sequence is transferred from the subroutine to the calling program based on the specified flag of the PSW and the program execution begins at the new address. |
| **PCHL** | **None** | Load the program counter with HL contents | The contents of registers H & L are copied into the program counter. The contents of H are placed as the high-order byte and the contents of L as the loworder byte. |
| **RST** | **0-7** | Restart | The RST instruction is used as software instructions in a program to transfer the program execution to one of the following eight locations.   |  |  | | --- | --- | | Instruction | Restart Address | | RST 0 | 0000H | | RST 1 | 0008H | | RST 2 | 0010H | | RST 3 | 0018H | | RST 4 | 0020H | | RST 5 | 0028H | | RST 6 | 0030H | | RST 7 | 0038H |   The 8085 has additionally 4 interrupts, which can generate RST instructions internally and doesn’t require any external hardware. Following are those instructions and their Restart addresses −   |  |  | | --- | --- | | Interrupt | Restart Address | | TRAP | 0024H | | RST 5.5 | 002CH | | RST 6.5 | 0034H | | RST 7.5 | 003CH | |

**Arithmetic Instruction**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Operand** | **Meaning** | **Explanation** |
| ADD | R  M | Add register or memory, to the accumulator | The contents of the register or memory are added to the contents of the accumulator and the result is stored in the accumulator.  **Example** − ADD K. |
| ADC | R  M | Add register to the accumulator with carry | The contents of the register or memory & M the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator.  **Example** − ADC K |
| ADI | 8-bit data | Add the immediate to the accumulator | The 8-bit data is added to the contents of the accumulator and the result is stored in the accumulator.  **Example** − ADI 55K |
| ACI | 8-bit data | Add the immediate to the accumulator with carry | The 8-bit data and the Carry flag are added to the contents of the accumulator and the result is stored in the accumulator.  **Example** − ACI 55K |
| LXI | Reg. pair, 16bit data | Load the register pair immediate | The instruction stores 16-bit data into the register pair designated in the operand.  **Example** − LXI K, 3025M |
| DAD | Reg. pair | Add the register pair to H and L registers | The 16-bit data of the specified register pair are added to the contents of the HL register.  **Example** − DAD K |
| SUB | R  M | Subtract the register or the memory from the accumulator | The contents of the register or the memory are subtracted from the contents of the accumulator, and the result is stored in the accumulator.  **Example** − SUB K |
| SBB | R  M | Subtract the source and borrow from the accumulator | The contents of the register or the memory & M the Borrow flag are subtracted from the contents of the accumulator and the result is placed in the accumulator.  **Example** − SBB K |
| SUI | 8-bit data | Subtract the immediate from the accumulator | The 8-bit data is subtracted from the contents of the accumulator & the result is stored in the accumulator.  **Example** − SUI 55K |
| SBI | 8-bit data | Subtract the immediate from the accumulator with borrow | The contents of register H are exchanged with the contents of register D, and the contents of register L are exchanged with the contents of register E.  **Example** – XCHG |
| INR | R  M | Increment the register or the memory by 1 | The contents of the designated register or the memory are incremented by 1 and their result is stored at the same place.  **Example** − INR K |
| INX | R | Increment register pair by 1 | The contents of the designated register pair are incremented by 1 and their result is stored at the same place.  **Example** − INX K |
| DCR | R  M | Decrement the register or the memory by 1 | The contents of the designated register or memory are decremented by 1 and their result is stored at the same place.  **Example** − DCR K |
| DCX | R | Decrement the register pair by 1 | The contents of the designated register pair are decremented by 1 and their result is stored at the same place.  **Example** − DCX K |
| DAA | None | Decimal adjust accumulator | The contents of the accumulator are changed from a binary value to two 4-bit BCD digits.  If the value of the low-order 4-bits in the accumulator is greater than 9 or if AC flag is set, the instruction adds 6 to the low-order four bits.  If the value of the high-order 4-bits in the accumulator is greater than 9 or if the Carry flag is set, the instruction adds 6 to the high-order four bits.  **Example** – DAA |

**Data Transfer Instruction**

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Operand** | **Meaning** | **Explanation** |
| MOV | Rd, Sc  M, Sc  Dt, M | Copy from the source (Sc) to the destination(Dt) | This instruction copies the contents of the source register into the destination register without any alteration.  **Example** − MOV K, L |
| MVI | Rd, data  M, data | Move immediate 8-bit | The 8-bit data is stored in the destination register or memory.  **Example** − MVI K, 55L |
| LDA | 16-bit address | Load the accumulator | The contents of a memory location, specified by a 16-bit address in the operand, are copied to the accumulator.  **Example** − LDA 2034K |
| LDAX | B/D Reg. pair | Load the accumulator indirect | The contents of the designated register pair point to a memory location. This instruction copies the contents of that memory location into the accumulator.  **Example** − LDAX K |
| LXI | Reg. pair, 16-bit data | Load the register pair immediate | The instruction loads 16-bit data in the register pair designated in the register or the memory.  **Example** − LXI K, 3225L |
| LHLD | 16-bit address | Load H and L registers direct | The instruction copies the contents of the memory location pointed out by the address into register L and copies the contents of the next memory location into register H.  **Example** − LHLD 3225K |
| STA | 16-bit address | 16-bit address | The contents of the accumulator are copied into the memory location specified by the operand.  This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address.  **Example** − STA 325K |
| STAX | 16-bit address | Store the accumulator indirect | The contents of the accumulator are copied into the memory location specified by the contents of the operand.  **Example** − STAX K |
| SHLD | 16-bit address | Store H and L registers direct | The contents of register L are stored in the memory location specified by the 16-bit address in the operand and the contents of H register are stored into the next memory location by incrementing the operand.  This is a 3-byte instruction, the second byte specifies the low-order address and the third byte specifies the high-order address.  **Example** − SHLD 3225K |
| XCHG | None | Exchange H and L with D and E | The contents of register H are exchanged with the contents of register D, and the contents of register L are exchanged with the contents of register E.  **Example** – XCHG |
| SPHL | None | Copy H and L registers to the stack pointer | The instruction loads the contents of the H and L registers into the stack pointer register. The contents of the H register provide the high-order address and the contents of the L register provide the low-order address.  **Example** – SPHL |
| XTHL | None | Exchange H and L with top of stack | The contents of the L register are exchanged with the stack location pointed out by the contents of the stack pointer register.  The contents of the H register are exchanged with the next stack location (SP+1).  **Example** – XTHL |
| PUSH | Reg. pair | Push the register pair onto the stack | The contents of the register pair designated in the operand are copied onto the stack in the following sequence.  The stack pointer register is decremented and the contents of the high order register (B, D, H, A) are copied into that location.  The stack pointer register is decremented again and the contents of the low-order register (C, E, L, flags) are copied to that location.  **Example** − PUSH K |
| POP | Reg. pair | Pop off stack to the register pair | The contents of the memory location pointed out by the stack pointer register are copied to the low-order register (C, E, L, status flags) of the operand.  The stack pointer is incremented by 1 and the contents of that memory location are copied to the high-order register (B, D, H, A) of the operand.  The stack pointer register is again incremented by 1.  **Example** – POPK |
| OUT | 8-bit port address | Output the data from the accumulator to a port with 8bit address | The contents of the accumulator are copied into the I/O port specified by the operand.  **Example** − OUT K9L |
| IN | 8-bit port address | Input data to accumulator from a port with 8-bit address | The contents of the input port designated in the operand are read and loaded into the accumulator.  **Example** − IN5KL |

**8085 – Demo Programs**

Now, let us take a look at some program demonstrations using the above instructions −

**Adding Two 8-bit Numbers**

Write a program to add data at 3005H & 3006H memory location and store the result at 3007H memory location.

**Problem demo −**

(3005H) = 14H

(3006H) = 89H

**Result −**

14H + 89H = 9DH

The program code can be written like this –

**LXI H 3005H** : *"HL points 3005H"*

**MOV A, M** : *"Getting first operand"*

**INX H** : *"HL points 3006H"*

**ADD M** : *"Add second operand"*

**INX H** : *"HL points 3007H"*

**MOV M, A** : *"Store result at 3007H"*

**HLT** : *"Exit program"*

**Exchanging the Memory Locations**

Write a program to exchange the data at 5000M& 6000M memory location.

**LDA 5000M :** *"Getting the contents at5000M location into accumulator"*

**MOV B, A :** *"Save the contents into B register"*

**LDA 6000M :** *"Getting the contents at 6000M location into accumulator"*

**STA 5000M :** *"Store the contents of accumulator at address 5000M"*

**MOV A, B :** *"Get the saved contents back into A register"*

**STA 6000M :** *"Store the contents of accumulator at address 6000M"*

**Arrange Numbers in an Ascending Order**

Write a program to arrange first 10 numbers from memory address 3000H in an ascending order.

**MVI B, 09 :** *"Initialize counter"*

**START :** *"LXI H, 3000H: Initialize memory pointer"*

**MVI C, 09H :** *"Initialize counter 2"*

**BACK: MOV A, M :** *"Get the number"*

**INX H :** *"Increment memory pointer"*

**CMP M :** *"Compare number with next number"*

**JC SKIP :** *"If less, don’t interchange"*

**JZ SKIP :** *"If equal, don’t interchange"*

**MOV D, M**

**MOV M, A**

**DCX H**

**MOV M, D**

**INX H :** *"Interchange two numbers"*

**SKIP:DCR C :** *"Decrement counter 2"*

**JNZ BACK :**  *"If not zero, repeat"*

**DCR B :**  *"Decrement counter 1"*

**JNZ START**

**HLT :** *"Terminate program execution"*

**Some High Level Languages**

**Introduction**

* In today's era, programmer use lots of languages for programming in which High-Level Languages have lots of contribution.
* To overcome the drawbacks of low-level Languages these High-Level Languages are developed.
* High-Level Languages are written in statements. Examples of High-Level Languages are FORTRAN, COBOL, Program JAVA, C# etc
* These languages are procedure oriented
* A program written in these languages in one computer can easily be used on another computer.
* The programs written in these languages are easier and faster let's discuss some languages.

**FORTRAN**

* Abbreviated as 'Formula Translation'.
* Developed by IBM nearly in the 1950s.
* Used for scientific, engineering calculation and for mathematical operations.

**COBOL**

* Abbreviated as 'Common Business Oriented Language'.
* Developed by U.S. Government committe in 1960.
* Widely used in past for business applications and data processing.
* It supports a limited number of numeric operations.
* Object Oriented version of COBOL is Visual COBOL.

**BASIC**

* Abbreviated as 'Beginner's All Purpose Symbolic Instruction Code'.
* Developed by Dartmouth college in 1965.
* Used for scientific and engineering operations.
* It is very easy and simple language.

**Visual Basic**

* It is generally called scripting language which is used for combining small program written in BASIC Language.
* This tool is mostly used for development of Windows applications.
* Development of Windows apps in C is difficult that's why this language is created.
* It is object Oriented programming language.

**C language**

* Developed by Dennis Ritchie and Brain Kernighan at AT and T's Bell laboratory in 1972.
* It is a general purpose language.
* This language first used to write UNIX operating system.
* This language also used to write system software, drivers and Commercial software packages.
* C++ is an extension of C Language.
* C++ is developed by Bjarne Stroustrup in the year 1980.
* C++ is object Oriented language.

**C# language**

* Developed by Microsoft and this language appeared nearly in 2000.
* The Recent stable version is C# 6.0 released in 2015.
* This language also pronounced as See Sharp.
* Used for development of software, GUI application and also for games development.
* This language is the hybrid of C and C++.
* Windows Mobile apps are developed using this language.

**PROLOG**

* Abbreviated as 'Programming Logic'.
* Developed by University of Marseilles in 1972.
* Main application of this language is for Artificial Intelligence.
* This language is capable of handling large number of databases.

**LISP**

* Abbreviated as 'list processing'.
* Developed by McCarthy in 1960.
* This language is mostly used in the artificial intelligence.
* This language is suitable for I/O handling and graphics.

**SNOBOL**

* Abbreviated as 'String Oriented Symbolic Language'.
* Developed nearly in 1960.
* Used for text processing.

**LOGO**

* Developed at MIT by Seymour Papert in nearly 1960.
* Used in universities for solving complex scientific problems.
* Due to its graphics feature it can be used for educational purposes.

**JAVA**

* It is developed by Sun Microsystems now which is owned by Oracle corporation.
* It is the object Oriented programming language.
* This language can be run on any Java virtual machine.
* This language is suitable for running on any desktop computer, servers, Internet, microprocessor and lots more.
* Now a days all android apps are developed using JAVA.

**SQL**

* Abbreviated as 'Structured Query Language'.
* This language is developed initially at IBM.
* Used for database related application and used by database companies like ORACLE , SYBASE etc.
* This language became standard of American National Standard Institute(ANCI) for database query in 1989.

**Some Facts**

* First High Level Language developed and used was FORTRAN.
* First Computer Programmer was a women whose name is Adam Lovelace and ADA programming language is named after her.
* First website is made using Hyper Text Markup Language(HTML).
* First Version of UNIX operating system was written using C Language.

**C**

**Language**

**C LANGUAGE OVERVIEW**

C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972. In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard. The UNIX operating system, the C compiler, and essentially all UNIX applications programs have been written in C. C has now become a widely used professional language for various reasons.

* Easy to learn
* Structured language
* It produces efficient programs.
* It can handle low-level activities.
* It can be compiled on a variety of computer platforms.

**Facts about C**

* C was invented to write an operating system called UNIX.
* C is a successor of B language which was introduced around 1970.
* The language was formalized in 1988 by the American National Standard Institute ANSI.
* The UNIX OS was totally written in C by 1973.
* Today C is the most widely used and popular System Programming Language.
* Most of the state-of-the-art software’s have been implemented using C.
* Today's most popular Linux OS and RBDMS MySQL have been written in C.

**C - ENVIRONMENT SETUP**

Before you start doing programming using C programming language, you need the following two software’s available on your computer, a) Text Editor and b) The C Compiler.

**Text Editor:** This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The files you create with your editor are called source files and contain program source code. The source files for C programs are typically named with the extension **".c".**

**The C Compiler:** The source code written in source file is the human readable source for your program. It needs to be "compiled", to turn into machine language so that your CPU can actually execute the program

**C - PROGRAM STRUCTURE**

Before we study basic building blocks of the C programming language, let us look a bare minimum C program structure so that we can take it as a reference in upcoming chapters.

**C Hello World Example**

A C program basically consists of the following parts:

* Preprocessor Commands
* Functions
* Variables
* Statements & Expressions
* Comments

Let us look at a simple code that would print the words "Hello World":

*#include <stdio.h>*

*int main()*

*{*

*/\* my first program in C \*/*

*printf("Hello, World! \n");*

*return 0;*

*}*

Let us look various parts of the above program:

The first line of the program #include <stdio.h> is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.

The next line int main is the main function where program execution begins.

The next line /\*...\*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.

The next line printf... is another function available in C which causes the message "Hello, World!" to be displayed on the screen. The next line return 0; terminates mainfunction and returns the value 0.

**Compile & Execute C Program:**

Lets look at how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.
2. Save the file as hello.c
3. Open a command prompt and go to the directory where you saved the file.
4. Type gcc hello.c and press enter to compile your code.
5. If there are no errors in your code, the command prompt will take you to the next line and would generate a.out executable file.
6. Now, type a.out to execute your program.
7. You will be able to see "Hello World" printed on the screen.

$ gcc hello.c

$ ./a.out Hello, World!

Make sure that gcc compiler is in your path and that you are running it in the directory containing source file hello.c.

**C - BASIC SYNTAX**

You have seen a basic structure of C program, so it will be easy to understand other basic building blocks of the C programming language.

**Tokens in C**

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens:

printf("Hello, World! \n");

The individual tokens are:

printf ( "Hello, World! \n" ) ;

**Semicolons ;**

In C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity. For example, following are two different statements:

printf("Hello, World! \n"); return 0;

**Comments**

Comments are like helping text in your C program and they are ignored by the compiler. They start with /\* and terminates with the characters \*/ as shown below:

/\* my first program in C \*/

You cannot have comments within comments and they do not occur within a stribng or character literals.

**Identifiers**

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore \_ followed by zero or more letters, underscores, and digits 0to9. C does not allow punctuation characters such as @, $, and % within identifiers. C is a **case sensitive** programming language. Thus, Manpower and manpower are two different identifiers in C. Here are some examples of acceptable identifiers:

mohd zara abc move\_name a\_123 myname50 \_temp j a23b9 retVal

**Keywords**

The following list shows the reserved words in C. These reserved words may not be used as constant or variable or any other identifier names.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| auto | Else | long | switch | Do |
| break | Enum | register | typedef | int |
| case | Extern | return | union | struct |
| char | Float | short | unsigned | double |
| const | For | signed | void | \_packed |
| continue | Goto | sizeof | volatile |  |
| default | If | static | while |  |

**Whitespace in C**

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it. Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:

int age;

There must be at least one whitespace character usuallyaspace between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement:

fruit = apples + oranges; // get the total fruit

No whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

**C - DATA TYPES**

In the C programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The types in C can be classified as follows:

|  |  |
| --- | --- |
| S.N. | Types and Description |
| 1 | **Basic Types:** They are arithmetic types and consists of the two types: a integer types and b floating point types. |
| 2 | **Enumerated types:** They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program. |
| 3 | **The type void:** The type specifier void indicates that no value is available. |
| 4 | **Derived types:** They include a Pointer types, b Array types, c Structure types, d Union types and e Function types. |

The array types and structure types are referred to collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see basic types in the following section, whereas, other types will be covered in the upcoming chapters.

**Integer Types**

Following table gives you details about standard integer types with its storage sizes and value ranges:

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| int | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions **sizeoftype** yields the storage size of the object or type in bytes.

**Floating-Point Types**

Following table gives you details about standard floating-point types with storage sizes and value ranges and their precision:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Storage size** | **Value range** | **Precision** |
| float | 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| double | 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs.

**The void Type**

The void type specifies that no value is available. It is used in three kinds of situations:

|  |  |
| --- | --- |
| **S.N.** | **Types and Description** |
| 1 | **Function returns as void**  There are various functions in C which do not return value or you can say they return void. A function with no return value has the return type as void. For example, void exit intstatus; |
| 2 | **Function arguments as void**  There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, int randvoid; |
| 3 | **Pointers to void**  A pointer of type void \* represents the address of an object, but not its type. For example a memory allocation function **void \*malloc** sizetsize; returns a pointer to void which can be casted to any data type. |

The void type may not be understood to you at this point, so let us proceed and we will cover these concepts in the upcoming chapters.

**C - VARIABLES**

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in previous chapter, there will be the following basic variable types:

|  |  |
| --- | --- |
| **Type** | **Description** |
| char | Typically a single octet (one byte). This is an integer type. |
| int | The most natural size of integer for the machine. |
| float | A single-precision floating point value. |
| double | A double-precision floating point value. |
| void | Represents the absence of type. |

C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

**Variable Definition in C:**

A variable definition means to tell the compiler where and how much to create the storage for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

type variable\_list;

Example:

* int i, j, k;
* char c, ch;
* float f, salary;
* double d;

The line int i, j, k; both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int. Variables can be initialized assignedaninitialvalue in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

type variable\_name = value;

Some examples are:

* extern int d = 3, f = 5; // declaration of d and f.
* int d = 3, f = 5; // definition and initializing d and f.
* byte z = 22; // definition and initializes z.
* char x = 'x'; // the variable x has the value 'x'.

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL allbyteshavethevalue0; the initial value of all other variables is undefined.

**Variable Declaration in C:**

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program. A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use extern keyword to declare a variable at any place. Though you can declare a variable multiple times in your C program but it can be defined only once in a file, a function or a block of code.

**Example**

Try the following example, where a variable has been declared at the top, but it has been defined inside the main function:

*#include <stdio.h>*

*// Variable declaration:*

*extern int a, b;*

*extern int c;*

*extern float f;*

*int main ()*

*{*

*/\* variable definition: \*/*

*int a, b;*

*int c;*

*float f;*

*/\* actual initialization \*/*

*a = 10;*

*b = 20;*

*c = a + b;*

*printf("value of c : %d \n", c);*

*f = 70.0/3.0;*

*printf("value of f : %f \n", f);*

*return 0;*

*}*

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

value of c : 30 value of f : 23.333334

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example:

*// function declaration*

*int func();*

*int main()*

*{*

*// function call*

*int i = func();*

*}*

*// function definition*

*int func()*

*{*

*return 0;*

*}*

**Lvalues and Rvalues in C:**

There are two kinds of expressions in C:

**1. lvalue :** An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.

**2. rvalue :** An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:

int g = 20;

But following is not a valid statement and would generate compile-time error:

10 = 20;

**C - CONSTANTS AND LITERALS**

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The **constants** are treated just like regular variables except that their values cannot be modified after their definition.

**Integer literals**

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

* 212 /\* Legal \*/
* 215u /\* Legal \*/
* 0xFeeL /\* Legal \*/
* 078 /\* Illegal: 8 is not an octal digit \*/
* 032UU /\* Illegal: cannot repeat a suffix \*/

Following are other examples of various type of Integer literals:

* 85 /\* decimal \*/
* 0213 /\* octal \*/
* 0x4b /\* hexadecimal \*/
* 30 /\* int \*/
* 30u /\* unsigned int \*/
* 30l /\* long \*/
* 30ul /\* unsigned long \*/

**Floating-point literals**

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form. While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E. Here are some examples of floating-point literals:

* 3.14159 /\* Legal \*/
* 314159E-5L /\* Legal \*/
* 510E /\* Illegal: incomplete exponent \*/
* 210f /\* Illegal: no decimal or exponent \*/
* .e55 /\* Illegal: missing integer or fraction \*/

**Character constants**

Character literals are enclosed in single quotes, e.g., 'x' and can be stored in a simple variable of char type. A character literal can be a plain character e.g., ′x′, an escape sequence e.g., ′\t′, or a universal character e.g.,′\u02C0′.

There are certain characters in C when they are preceded by a backslash they will have special meaning and they are used to represent like newline \n or tab \t. Here, you have a list of some of such escape sequence codes:

|  |  |  |
| --- | --- | --- |
| **Escape sequence** | **Hex value in ASCII** | **Character represented** |
| **\a** | 07 | [Alert (Beep, Bell)](https://en.wikipedia.org/wiki/Bell_character) (added in C89)[[1]](https://en.wikipedia.org/wiki/Escape_sequences_in_C#cite_note-Rationale_2003_C-1) |
| **\b** | 08 | [Backspace](https://en.wikipedia.org/wiki/Backspace) |
| **\f** | 0C | [Formfeed](https://en.wikipedia.org/wiki/Formfeed) |
| **\n** | 0A | [Newline](https://en.wikipedia.org/wiki/Newline) (Line Feed); see notes below |
| **\r** | 0D | [Carriage Return](https://en.wikipedia.org/wiki/Carriage_return) |
| **\t** | 09 | [Horizontal Tab](https://en.wikipedia.org/wiki/Tab_key) |
| **\v** | 0B | [Vertical Tab](https://en.wikipedia.org/wiki/Tab_key) |
| **\\** | 5C | [Backslash](https://en.wikipedia.org/wiki/Backslash) |
| **\'** | 27 | [Single quotation mark](https://en.wikipedia.org/wiki/Apostrophe) |
| **\"** | 22 | Double [quotation mark](https://en.wikipedia.org/wiki/Quotation_mark) |
| **\?** | 3F | [Question mark](https://en.wikipedia.org/wiki/Question_mark) (used to avoid [trigraphs](https://en.wikipedia.org/wiki/Digraphs_and_trigraphs#c)) |
| **\nnn**[**note 1**](https://en.wikipedia.org/wiki/Escape_sequences_in_C#endnote_Note1) | Any | The byte whose numerical value is given by nnn interpreted as an [octal](https://en.wikipedia.org/wiki/Octal) number |
| **\xhh…** | Any | The byte whose numerical value is given by hh… interpreted as a [hexadecimal](https://en.wikipedia.org/wiki/Hexadecimal) number |

**String literals**

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters. You can break a long line into multiple lines using string literals and separating them using whitespaces. Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

**Defining Constants**

There are two simple ways in C to define constants:

1. Using **#define** preprocessor.

2. Using **const** keyword.

**The const Keyword**

You can use **const** prefix to declare constants with a specific type as follows:

const type variable = value;

**C - STORAGE CLASSES**

A storage class defines the scope visibility and life time of variables and/or functions within a C Program. These specifiers precede the type that they modify. There are following storage classes which can be used in a C Program

* auto
* register
* static
* extern

**The auto Storage Class**

The **auto** storage class is the default storage class for all local variables.

{ int mount; auto int month; }

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

**The register Storage Class**

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size usuallyoneword and can't have the unary '&' operator applied to it asitdoesnothaveamemorylocation.

{ register int miles; }

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

**The static Storage Class**

The static storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared. In C programming, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

**The extern Storage Class**

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined. When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another file. The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions.

**C - OPERATORS**

This tutorial will explain the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

**Arithmetic Operators**

Following table shows all the arithmetic operators supported by C language. Assume variable A holds 10 and variable B holds 20 then: Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands. | A + B = 30 |
| − | Subtracts second operand from the first. | A − B = -10 |
| \* | Multiplies both operands. | A \* B = 200 |
| / | Divides numerator by de-numerator. | B / A = 2 |
| % | Modulus Operator and remainder of after an integer division. | B % A = 0 |
| ++ | Increment operator increases the integer value by one. | A++ = 11 |
| -- | Decrement operator decreases the integer value by one. | A-- = 9 |

**Relational Operators**

Following table shows all the relational operators supported by C language. Assume variable A holds 10 and variable B holds 20, then: Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not. If yes, then the condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true. | (A <= B) is true. |

**Logical Operators**

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then: Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. | !(A && B) is true. |

**Bitwise Operators**

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13 then: Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12, i.e., 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) = 61, i.e., 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49, i.e., 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = -61, i.e,. 1100 0011 in 2's complement form. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 = 240 i.e., 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 = 15 i.e., 0000 1111 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **Q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Assignment Operators**

There are following assignment operators supported by C language: Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator. Assigns values from right side operands to left side operand | C = A + B will assign the value of A + B to C |
| += | Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand. | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | Bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

**Misc Operators &map; sizeof & ternary**

There are few other important operators including sizeof and ? : supported by C Language. Show Examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof() | Returns the size of a variable. | sizeof(a), where a is integer, will return 4. |
| & | Returns the address of a variable. | &a; returns the actual address of the variable. |
| \* | Pointer to a variable. | \*a; will give actual address of the variable. |
| ? : | Conditional Expression. | If Condition is true ? then value X : otherwise value Y |

**Operators Precedence in C**

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first. Show Examples

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

**C - DECISION MAKING**

C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null** then it is assumed as **false** value.

C programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **S.N.** | **Statement & Description** |
| 1 | **if statement**  An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | **if...else statement**  An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false. |
| 3 | **nested if statements**  You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| 4 | **switch statement**  A **switch** statement allows a variable to be tested for equality against a list of values. |
| 5 | **nested switch statements**  You can use one **switch** statement inside another **switch** statement(s). |
| 6 | The ? : Operator Condition ? true value : false value; |

**C - LOOPS**

There may be a situation when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:

C programming language provides following types of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **S.N.** | **Loop Type & Description** |
| **while loop** | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| **for loop** | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| **do...while loop** | It is more like a while statement, except that it tests the condition at the end of the loop body. |
| **nested loops** | You can use one or more loops inside any other while, for, or do..while loop. |

**Loop Control Statements:**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| **break statement** | Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| **continue statement** | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| **goto statement** | Transfers control to the labeled statement. |

**C - FUNCTIONS**

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function **strcat** to concatenate two strings, function **memcpy** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure, etc.

**Defining a Function:**

The general form of a function definition in C programming language is as follows:

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

*{*

*body of the function*

*}*

A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

**Return Type:** A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void.**

**Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.

**Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

**Function Body:** The function body contains a collection of statements that define what the function does.

**C - SCOPE RULES**

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable can not be accessed. There are three places where variables can be declared in C programming language:

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

Let us explain what are local and global variables and formal parameters.

**Local Variables**

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

**Global Variables**

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

**Formal Parameters**

Function parameters, formal parameters, are treated as local variables within that function and they will take preference over the global variables.

**C - ARRAYS**

C programming language provides a data structure called the **array**, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

A specific element in an array is accessed by an **index**.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

**Declaring Arrays**

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

*type arrayName [ arraySize ];*

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type.

**Initializing Arrays**

You can initialize array in C either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above:

**Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable.

**C - POINTERS**

A **pointer** is a variable, whose value is the address of another variable, i.e., direct address of the memory location. The general form of a pointer variable declaration is:

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The **asterisk \*** you used to declare a pointer is the same asterisk that you use for multiplication.

*int \*ip; /\* pointer to an integer \*/*

*double \*dp; /\* pointer to a double \*/*

*float \*fp; /\* pointer to a float \*/*

*char \*ch /\* pointer to a character \*/*

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

**How to use Pointers?**

unary operator \* that returns the value of the variable located at the address specified by its operand.

**C - STRINGS**

The string in C programming language is actually a one-dimensional **array of characters** which is terminated by a **null** character **'\0'**. Thus a null-terminated string contains the characters that comprise the string followed by a null. The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows:

char greeting[] = "Hello";

Following is the memory presentation of above-defined string in C/C++:

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

C supports a wide range of functions that manipulate null-terminated strings:

|  |  |
| --- | --- |
| Function | Purpose |
| Strcpy(s1,s2); | Copies string s2 into string s1. |
| Strcat(s1,s2); | Concatenates string s2 onto the end of string s1. |
| Strlen(s1); | Returns the length of string s1. |
| strcmps1,s2; | Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| Strchr(s1,ch); | Returns a pointer to the first occurrence of character ch in string s1. |
| Strstr(s1,s2); | Returns a pointer to the first occurrence of string s2 in string s1. |

**C - STRUCTURES**

C arrays allow you to define type of variables that can hold several data items of the same kind but **structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.

**Defining a Structure**

To define a structure, you must use the **struct** statement. The struct statement defines a **new data type**, with more than one member for your program. The format of the struct statement is this:

*struct [structure tag]*

*{*

*member definition;*

*member definition;*

*...*

*member definition;*

*} [one or more structure variables];*

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

**Accessing Structure Members**

The member access operator is coded as a **period** (.) between the structure variable name and the structure member that we wish to access.

**C - UNIONS**

A union is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, ***but only one member can contain a value at any given time*.** Unions provide an efficient way of using the same memory location for multi-purpose.

**Defining a Union**

To define a union, you must use the **union** statement in very similar was as you did while defining structure. The union statement defines a new data type, with more than one member for your program. The format of the union statement is as follows:

*union [union tag]*

*{*

*member definition;*

*member definition;*

*...*

*member definition;*

*} [one or more union variables];*

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. The memory occupied by a union will be large enough to hold the largest member of the union.

**Accessing Union Members**

The member access operator is coded as a **period (.)** between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type.

**C - BIT FIELDS**

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows:

*struct*

*{*

*unsigned int widthValidated;*

*unsigned int heightValidated;*

*} status;*

This structure requires 8 bytes of memory space but in actual we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situation. If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes.

**Bit Field Declaration**

The declaration of a bit-field has the following form inside a structure:

*struct*

*{*

*type [member\_name] : width ;*

*};*

The following table describes the variable elements of a bit field:

|  |  |
| --- | --- |
| Elements | Description |
| type | An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int. |
| member\_name | The name of the bit-field. |
| width | The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. |

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit-field with a width of 3 bits.

**C - TYPEDEF**

The C programming language provides a keyword called **typedef**, which you can use to give a type a new name. Following is an example to define a term **BYTE** for one-byte numbers:

*typedef unsigned char BYTE;*

After this type definitions, the identifier BYTE can be used as an abbreviation for the type **unsigned char**, for example:.

BYTE b1, b2;

You can use typedef to give a name to user defined data type as well.

**typedef vs #define**

**#define** is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences:

* **typedef** is limited to giving symbolic names to types only, whereas **#define** can be used to define alias for values as well, e.g., you can define 1 as ONE, etc.
* **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the preprocessor.

**C - INPUT & OUTPUT**

When we are saying **Input** that means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

When we are saying **Output** that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen as well as you can save that data in text or binary files.

**The Standard Files**

C programming language treats all the devices as files. So devices such as the display are addressed in the same way as files and following three file are automatically opened when a program executes to provide access to the keyboard and screen.

|  |  |  |
| --- | --- | --- |
| Standard File | File Pointer | Device |
| Standard input | Stdin | Keyboard |
| Standard output | Stdout | Screen |
| Standard error | Stderr | Your screen |

The file points are the means to access the file for reading and writing purpose. This section will explain you how to read values from the screen and how to print the result on the screen.

**The getchar() & putchar() functions**

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one characters from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen.

**The gets() & puts() functions**

The **char \*gets(char \*s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF(End of file).

The **int puts(const char \*s)** function writes the string **‘s’** and **‘a’** trailing newline to **stdout**.

**The scanf() and printf functions**

The **int scanf(const char ∗ format,...)** function reads input from the standard input stream **stdin** and scans that input according to **format** provided.

The **int printf(const char ∗ format,...)** function writes output to the standard output stream **stdout** and produces output according to a format provided.

The **format** can be a simple constant string, but you can specify **%s, %d, %c, %f** etc. to print or read **strings, integer, character or float** respectively. There are many other formatting options available which can be used based on requirements. For a complete detail you can refer to a man page for these function.

**C - FILE I/O**

A file represents a **sequence of bytes**, does not matter if it is a **text file or binary file**. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through important calls for the file management.

**Opening Files**

You can use the **fopen** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

*FILE \*fopen( const char \* filename, const char \* mode );*

Here, **filename** is string literal which you will use to name your file and access **mode** can have one of the following values:

|  |  |
| --- | --- |
| Mode | Description |
| r | Opens an existing text file for reading purpose. |
| w | Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file. |
| a | Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content. |
| r+ | Opens a text file for reading and writing both. |
| w+ | Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist. |
| a+ | Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended. |

If you are going to handle **binary** files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

**Closing a File**

To close a file, use the fclose function. The prototype of this function is:

*int fclose( FILE \*fp );*

The **fclose** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file stdio.h.

**Writing a File**

Following is the simplest function to write individual **characters** to a stream:

*int fputc( int c, FILE \*fp );*

The function **fputc** writes the character value of the argument **c** to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error. You can use the following functions to write a null-terminated **string** to a stream:

*int fputs( const char \*s, FILE \*fp );*

The function **fputs** writes the string **s** to the output stream referenced by fp. It returns a nonnegative value on success, otherwise EOF is returned in case of any error. You can use

**int fprintf(FILE ∗fp ,const char ∗format,...)** function as well to write a string into a file.

**Reading a File**

Following is the simplest function to read a single **character** from a file:

*int fgetc( FILE \* fp );*

The **fgetc** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns EOF. The following functions allow you to read a **string** from a stream:

*char \*fgets( char \*buf, int n, FILE \*fp );*

The functions fgets reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use **int fscanf(FILE ∗fp, const char ∗format,...)** function to read strings from a file but it stops reading after the first space character encounters.

**Binary I/O Functions**

There are following two functions which can be used for binary input and output:

*size\_t fread(void \*ptr, size\_t size\_of\_elements, size\_t number\_of\_elements, FILE \*a\_file);*

*size\_t fwrite(const void \*ptr, size\_t size\_of\_elements, size\_t number\_of\_elements, FILE \*a\_file);*

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

**C - PREPROCESSORS**

The **C Preprocessor** is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a pound symbol **#**. It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

|  |  |
| --- | --- |
| Directive | Description |
| #define | Substitutes a preprocessor macro |
| #include | Inserts a particular header from another file |
| #undef | Undefines a preprocessor macro |
| #ifdef | Returns true if this macro is defined |
| #ifndef | Returns true if this macro is not defined |
| #if | Tests if a compile time condition is true |
| #else | The alternative for #if |
| #elif | #else an #if in one statement |
| #endif | Ends preprocessor conditional |
| #error | Prints error message on stderr |
| #pragma | Issues special commands to the compiler, using a standardized method |

**Predefined Macros**

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

|  |  |
| --- | --- |
| Macro | Description |
| \_\_DATE\_\_ | The current date as a character literal in "MMM DD YYYY" format |
| \_\_TIME\_\_ | The current time as a character literal in "HH:MM:SS" format |
| \_\_FILE\_\_ | This contains the current filename as a string literal. |
| \_\_LINE\_\_ | This contains the current line number as a decimal constant. |
| \_\_STDC\_\_ | Defined as 1 when the compiler complies with the ANSI standard. |

**C - HEADER FILES**

A header file is a file with extension **.h** which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

You request the use of a header file in your program by including it, with the C preprocessing directive **#include** like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be very much error-prone and it is not a good idea to copy the content of header file in the source files, specially if we have multiple source file comprising our program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in header files and include that header file wherever it is required.

**Include Syntax**

Both user and system header files are included using the preprocessing directive #include. It has following two forms:

*#include <file>*

This form is used for **system header files**. It searches for a file named file in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

*#include "file"*

This form is used for **header files of your own program.** It searches for a file named file in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

**C - TYPE CASTING**

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the **cast operator** as follows:

*(type\_name) expression*

Integer Promotion Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character in an int:

*#include <stdio.h>*

*main()*

*{*

*int i = 17;*

*char c = 'c'; /\* ascii value is 99 \*/*

*int sum;*

*sum = i + c;*

*printf("Value of sum : %d\n", sum );*

*}*

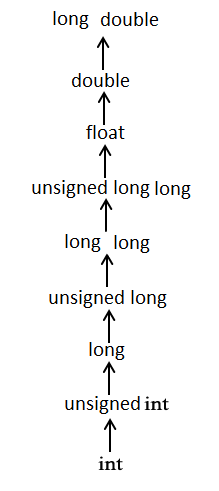
When the above code is compiled and executed, it produces the following result:

Value of sum : 116

Here, value of sum is coming as 116 because compiler is doing integer promotion and converting the value of 'c' to ascii before peforming actual addition operation.

**Usual Arithmetic Conversion**

The **usual arithmetic conversions** are implicitly performed to cast their values in a common type. Compiler first performs integer promotion, if operands still have different types then they are converted to the type that appears highest in the following hierarchy:



The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take following example to understand the concept:

#include <stdio.h>

main() { int i = 17; char c = 'c'; /\* ascii value is 99 \*/ float sum;

sum = i + c; printf("Value of sum : %f\n", sum );

}

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

Value of sum : 116.000000

**C - ERROR HANDLING**

As such C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and sets an error code **errno** is set which is global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. As a good practice, developer should set errno to 0 at the time of initialization of the program. A value of 0 indicates that there is no error in the program.

**The errno, perror and strerror**

The C programming language provides **perror** and **strerror** functions, which can be used to display the text message associated with **errno**.

* The **perror** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
* The **strerror** function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

*#include <stdio.h>*

*#include <errno.h>*

*#include <string.h>*

*extern int errno ;*

*int main ()*

*{*

*FILE \* pf;*

*int errnum;*

*pf = fopen ("unexist.txt", "rb");*

*if (pf == NULL)*

*{*

*errnum = errno;*

*fprintf(stderr, "Value of errno: %d\n", errno);*

*perror("Error printed by perror");*

*fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));*

*}*

*Else*

*{ fclose (pf); }*

*return 0;*

*}*

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

Value of errno: 2 Error printed by perror: No such file or directory Error opening file: No such file or directory

**C - VARIABLE ARGUMENTS**

Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments i.e. parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the usage:

#include <stdio.h>

#include <stdarg.h>

double average(int num,...)

{

va\_list valist;

double sum = 0.0;

int i; /\* initialize valist for num number of arguments \*/

va\_start(valist, num); /\* access all the arguments assigned to valist \*/

for (i = 0; i < num; i++)

{

sum += va\_arg(valist, int);

} /\* clean memory reserved for valist \*/

va\_end(valist);

return sum/num;

}

int main()

{

printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));

printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));

}

When the above code is compiled and executed, it produces the following result. It should be noted that the function average has been called twice and each time first argument represents the total number of variable arguments being passed. Only ellipses will be used to pass variable number of arguments.

Average of 2, 3, 4, 5 = 3.500000 Average of 5, 10, 15 = 10.000000

**C - MEMORY MANAGEMENT**

The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.

|  |  |
| --- | --- |
| Function | Description |
| void \*calloc(int num, int size); | This function allocates an array of **num** elements each of whose size in bytes will be **size**. |
| void free(void ∗address); | This function release a block of memory block specified by address. |
| void \*malloc(int num); | This function allocates an array of **num** bytes and leave them initialized. |
| void \*realloc(void ∗address, int newsize); | This function re-allocates memory extending it upto **newsize**. |

**C - COMMAND LINE ARGUMENTS**

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program specially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example, which checks if there is any argument supplied from the command line and take action accordingly:

*#include <stdio.h>*

*int main( int argc, char \*argv[] )*

*{*

*if( argc == 2 )*

*{*

*printf("The argument supplied is %s\n", argv[1]);*

*}*

*else if( argc > 2 )*

*{*

*printf("Too many arguments supplied.\n");*

*}*

*Else*

*{*

*printf("One argument expected.\n");*

*}*

*}*

When the above code is compiled and executed with a single argument, it produces the following result.

$./a.out testing

The argument supplied is testing Processing math: 100%

**C++**

**Language**

**C++ Overview**

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by **Bjarne Stroustrup** starting in **1979** at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

**C++ Compiler:**

This is actual C++ compiler, which will be used to compile your source code into final executable program.

Most C++ compilers don't care what extension you give your source code, but if you don't specify otherwise, many will use .cpp by default

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have respective Operating Systems.

**C++ Program Structure:**

Let us look at a simple code that would print the words *Hello World*.

#include <iostream >

using nam espace std;

// m ain() is where program execution begins.

int m ain()

{

cout << "Hello World"; // prints Hello World

return 0;

}

**Comments in C++**

C++ supports single line and multi-line comments. All characters available inside any comment are ignored by compiler.

C++ multi-line comments start with /\* and end with \*/.

A single line comment start with //, extending to the end of the line.

**C++ Primitive Built-in Types:**

C++ offer the programmer a rich assortment of built-in as well as user-defined data types.

Following table list down seven basic C++ data types:

|  |  |
| --- | --- |
| **Type** | **Keyword** |
| Boolean | Bool |
| Character | Char |
| Integer | Int |
| Floating point | Float |
| Double floating point | Double |
| Valueless | Void |
| Wide character | wchar\_t |

**C++ Variable Scope:**

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

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

**C++ Constants/Literals:**

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

**C++ Modifier Types:**

C++ allows the **char, int,** and **double** data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.

The data type modifiers are listed here:

* signed
* unsigned
* long
* short

The modifiers **signed, unsigned, long,** and **short** can be applied to integer base types. In addition, **signed** and **unsigned** can be applied to char, and **long** can be applied to double.

The modifiers **signed** and **unsigned** can also be used as prefix to **long** or **short** modifiers. For example **unsigned long int**.

C++ allows a shorthand notation for declaring **unsigned, short,** or **long** integers. You can simply use the word **unsigned, short,** or **long**, without the int. The int is implied. For example, the following two statements both declare unsigned integer variables.

**Storage Classes in C++:**

A storage class defines the scope *visibility* and life time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes which can be used in a C++ Program

* auto
* register
* static
* extern
* mutable

**C++ Operators:**

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides following type of operators:

* Arithmetic Operators +, − , ∗ , + + , − −
* Relational Operators == , ! = , > . < , >= , <=
* Logical Operators &&, ||, !
* Bitwise Operators & |, ^, ~, <<, >>
* Assignment Operators = , + = , − = , ∗ = , / = ,
* Misc Operators sizeof, & cast, comma, conditional etc.

**C++ Loop Types:**

C++ programming language provides the following types of loops to handle looping requirements.

Click the following links to check their detail.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| while loop | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| for loop | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| do...while loop | Like a while statement, except that it tests the condition at the end of the loop body |
| nested loops | You can use one or more loop inside any another while, for or do..while loop. |

**C++ Decision Making:**

C++ programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| if statement | An if statement consists of a boolean expression followed by one or more statements. |
| if...else statement | An if statement can be followed by an optional else statement, which executes when the boolean expression is false. |
| switch statement | A switch statement allows a variable to be tested for equality against a list of values. |
| nested if statements | You can use one if or else if statement inside another if or else if statement*s*. |
| nested switch statements | You can use one swicth statement inside another switch statement*s*. |

**C++ Functions:**

The general form of a C++ function definition is as follows:

return\_type function\_nam e( param eter list )

{

body of the function

}

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

* **Return Type**: A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body:** The function body contains a collection of statements that define what the function does.

**Numbers in C++:**

To utilize these functions you need to include the math header file **<cmath>**.

|  |  |  |
| --- | --- | --- |
| S.N. | Function | Purpose |
| 1 | **double cos(double);** | This function takes an angle (as a double) and returns the cosine. |
| 2 | **double sin(double);** | This function takes an angle (as a double) and returns the sine. |
| 3 | **double tan(double);** | This function takes an angle (as a double) and returns the tangent. |
| 4 | **double log(double);** | This function takes a number and returns the natural log of that number. |
| 5 | **double pow(double, double);** | The first is a number you wish to raise and the second is the power you wish to raise it t |
| 6 | **double hypot(double, double);** | If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse. |
| 7 | **double sqrt(double);** | You pass this function a number and it gives you the square root. |
| 8 | **int abs(int);** | This function returns the absolute value of an integer that is passed to it. |
| 9 | **double fabs(double);** | This function returns the absolute value of any decimal number passed to it. |
| 10 | **double floor(double);** | Finds the integer which is less than or equal to the argument passed to it. |

**C++ Arrays:**

**Declaring Arrays**

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [ arraySize ];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type.

**Initializing Arrays**

You can initialize C++ array elements either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } cannot be larger than the number ofelements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

Following is an example, which will show array declaration, assignment and accessing arrays in C++:

**Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

|  |  |
| --- | --- |
| Concept | Description |
| Multi-dimensional arrays | C++ supports multidimensional arrays. The simplest form of the multi-dimensional array is the two-dimensional array. |
| Pointer to an array | You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |
| Passing arrays to functions | You can pass to the function a pointer to an array by specifying the array's name withoutan index. |
| Return array from functions | C++ allows a function to return an array. |

**C++ Strings:**

C++ provides following two types of string representations:

* The C-style character string.
* The string class type introduced with Standard C++.

The C-style character string as follows:

actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

C++ supports a wide range of functions that manipulate null-terminated strings:

|  |  |  |
| --- | --- | --- |
| S.N. | Function | Purpose |
| 1 | **strcpy(s1, s2);** | Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);** | Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);** | Returns the length of string s1. |
| 4 | **strcmp(s1, s2);** | Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);** | Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);** | Copies string s2 into string s1. |

**C++ Classes & Objects**

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces { }. A class definition must be followed either by a semicolon or a list of declarations.

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

**Define C++ Objects:**

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types.

**Accessing the Data Members:**

The public data members of objects of a class can be accessed using the direct member access operator (**.**).

**C++ Inheritance:**

Inheritance allows us to define a class in terms of another class which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

This existing class is called the **base** class, and the new class is referred to as the **derived** class.

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class*es*. A class derivation list names one or more base classes and has the form:

*class derived-class: access-specifier base-class*

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is **private by default**.

**Multiple Inheritance**

*class derived-class: access baseA, access baseB....*

Consider a base class **Shape** and its derived class **Rectangle** as follows:

#include <iostream >

using nam espace std;

// Base class

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

int m ain(void)

{

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

**C++ Overloading**

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

Following is the example where same function **print** is being used to print different data types:

#include <iostream >

using nam espace std;

class printData

{

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int m ain(void)

{

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

**operator overloading**

#include <iostream>

using namespace std;

class Time

{

private:

int hours; // 0 to 23

int minutes; // 0 to 59

public:

// required constructors

Time(){

hours = 0;

minutes = 0;

}

Time(int h, int m){

hours = h;

minutes = m;

}

// method to display time

void displayTime()

{

cout << "H: " << hours << " M:" << minutes <<endl;

}

// overloaded prefix ++ operator

Time operator++ ()

{

++minutes; // increment this object

if(minutes >= 60)

{

++hours;

minutes -= 60;

}

return Time(hours, minutes);

}

// overloaded postfix ++ operator

Time operator++( int )

{

// save the orignal value

Time T(hours, minutes);

// increment this object

++minutes;

if(minutes >= 60)

{

++hours;

minutes -= 60;

}

// return old original value

return T;

}

};

int main()

{

Time T1(11, 59), T2(10,40);

++T1; // increment T1

T1.displayTime(); // display T1

++T1; // increment T1 again

T1.displayTime(); // display T1

T2++; // increment T2

T2.displayTime(); // display T2

T2++; // increment T2 again

T2.displayTime(); // display T2

return 0;

}

**Polymorphism in C++**

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes and area method has been implemented by the two sub-classes with different implementation.

#include <iostream >

using nam espace std;

class Shape

{

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

int area()

{

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape

{

public:

Rectangle( int a=0, int b=0)

{

Shape(a, b);

}

int area ()

{

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

class Triangle: public Shape

{

public:

Triangle( int a=0, int b=0)

{

Shape(a, b);

}

int area ()

{

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// Main function for the program

int m ain( )

{

Shape \* shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

The reason for the incorrect output is that the call of the function area is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area in the Shape class with the keyword **virtual** so that it looks like this:

class Shape {

protected:

int width, height;

public:

Shape( int a=0, int b=0)

{

width = a;

height = b;

}

virtual int area()

{

cout << "Parent class area :" <<endl;

return 0;

}

};

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

Rectangle class area

Triangle class area

**Data Abstraction in C++:**

Data abstraction refers to, providing only **essential information** to the outside word and hiding their background details ie. to represent the needed information in program without presenting the details.

Data abstraction is a programming and designtechnique that relies on the separation of interface and implementation.

For example, in C++ we use **classes** to define our own abstract data types *ADT*. You can use the **cout** object of class **ostream** to stream data to standard output like this:

#include <iostream >

using nam espace std;

int main( )

{

cout << "Hello C++" <<endl;

return 0;

}

Here, you don't need to understand how **cout** displays the text on the user's screen. You need only know the public interface and the underlying implementation of cout is free to change.

**Data Encapsulation in C++:**

**Program statements *code*:** This is the part of a program that performs actions and they are called functions.

**Program data:** The data is the information of the program which affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**. And class is way to implement encapsulation.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private.

**C++ Files and Streams:**

The **iostream** standard library **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

To read and write from a file requires another standard C++ library called **fstream** which defines three new data types:

**Data Type Description**

**ofstream** This data type represents the output file stream and is used to create files and to write information to files.

**ifstream** This data type represents the input file stream and is used to read information from files.

**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

Following is the C++ program, which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

#include <fstream >

#include <iostream >

using nam espace std;

int m ain ()

{

char data[100];

// open a file in write m ode.

ofstream outfile;

outfile.open("afile.dat");

cout << "Writing to the file" << endl;

cout << "Enter your nam e: ";

cin.getline(data, 100);

// write inputted data into the file.

outfile << data << endl;

cout << "Enter your age: ";

cin >> data;

cin.ignore();

// again write inputted data into the file.

outfile << data << endl;

// close the opened file.

outfile.close();

// open a file in read m ode.

ifstream infile;

infile.open("afile.dat");

cout << "Reading from the file" << endl;

infile >> data;

// write the data at the screen.

cout << data << endl;

// again read the data from the file and display it.

infile >> data;

cout << data << endl;

// close the opened file.

infile.close();

return 0;

}