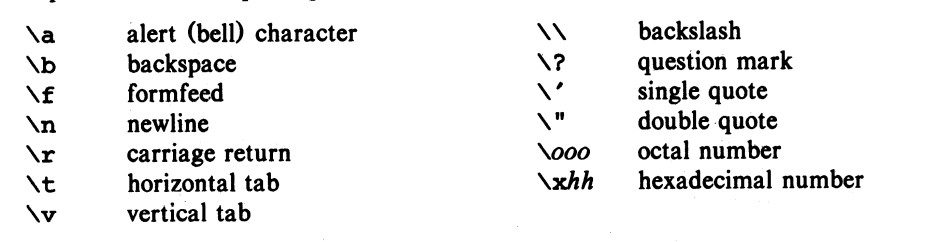
***Techniques for Programming in C***

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

******

1. **Interchanging 2 Variables**

temp = x;

    x = y;

  y = temp;

// OTHER WAY Without using third variable;

x=x+y;

y=x-y;

x=x-y;

1. **Checking Prime Number or Not**

if((a ==2 || a == 3 || a == 5 || a== 7) || a%2!=0 && a%3!=0 && a%5!=0 && a%7!=0 && a!=1){

            printf("%d is a Prime Number \n", a);

    }

    else{

        printf("%d is not a Prime Number", a);

    }

**OR**

int a;

    printf("Enter the number if you want to know whether it is prime or not? : ");

    scanf("%d", &a);

    int div = 0;

    for(int i=1; i<=a; i++){

        if(a%i==0){

            div++;

        }

    }

    if(div==2){

        printf("%d is prime", a);

    }

    else{

        printf("%d is not prime", a);

    }

1. **Divisors of an integer**

for(int i=1; i<=n; i++){

        if(n%i==0){

            printf("%d \n", i);

        }

  }

1. **Counting Digits**

#include<stdio.h>

int main(){

    int n, count=0;

    printf("Enter Integer: ");

    scanf("%d", &n);

    while(n!=0){

        n/=10;

        count++;

    }

    printf("Count is %d", count);

  return 0;

}

1. **Condition for Leap Year (V.IMP)**

year%400==0 || (year%4==0 && year%100!=0)

1. **Reversing the DIGITS**

while(n!=0){

        remainder = n%10;

        rev = rev\*10 + remainder;

        n /= 10;

    }

1. **Sum of Digits**

while(n!=0){

        remainder = n%10;

        sum  = sum + remainder;

        n = n/10;

    }

1. **Getting first digit**

while(n>=10){

        n = n/10;

    }

    firstnum = n;

1. **Roots of Quadratic Equation**

float d = b\*b - 4\*a\*c;

if(d<0){

        printf("Roots are imaginary... \n");

        float RP = -b/(2\*a);

        float IMG = sqrt(-d)/(2\*a);

        printf("The roots are  %.2f+%.2fi and %.2f-%.2fi", RP, IMG, RP, IMG);

    }

    else if(d==0){

        printf("Roots are equal... \n");

        printf("Roots --> %.2f", -b/(2\*a));

    }

    else if(d>0){

        printf("Roots are distinct");

        float R\_1 = (-b+sqrt(d))/(2\*a);

        float R\_2 = (-b-sqrt(d))/(2\*a);

        printf("Root-1 = %.2f and Root-2 = %.2f", R\_1, R\_2);

    }

1. **Alphabet or Special Char**

char c;

    printf("Enter: ");

    scanf("%c", &c);

    if((c>=32 && c<=64) || (c>=91 && c<=96) || (c>=123 && c<=127)){

        printf("It is special Character");

    }

    else if((c>=65 && c<=90) || (c>=97 && c<=122)){

        printf("It is an alphabet");

    }

1. **Vowel or Consonant**

char c;

    printf("Enter any character to check whether entered character is a vowel or consonant --> ");

    scanf("%c", &c);

    if(c==65|| c==69 || c== 73 || c==79 || c==85 || c==97 || c==101 || c==105 || c==111 || c==117 ){

        printf("Entered character is a vowel");

    }

    else{

        printf("Entered character is a consonant");

    }

1. **Ascending Order**

for(int i=0; i<n; i++){             // n is the number of elements

        for(int j = i+1; j<n; j++){

            if(arr[j]<arr[i]){

                temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

  }

1. **Descending Order**

for(int i=0; i<n; i++){             // n is the number of elements

        for(int j = i+1; j<n; j++){

            if(arr[j]<arr[i]){

                temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

  }

1. **HCF (Highest Common Factor)**

#include<stdio.h>

int main(){

    int a, b, hcf;

    printf("Enter the value of a and b: ");

    scanf("%d %d", &a, &b);

    if(a<b){

        // min num --> a

        for(int i=1; i<=a; i++){

            if(a%i==0 && b%i==0){

                hcf = i;

            }

        }

    }

    else if(a>b){

        // min num --> b

        for(int i=1; i<=b; i++){

            if(a%i==0 && b%i==0){

                hcf = i;

            }

        }

    }

    printf("The HCF is %d", hcf);

1. **LCM (Lowest Common Multiple)**

For this simply use the HCF program and use the formula that a x b = HCF x LCM

1. **Smallest Element and its position**

int arr[100];

    int position = 0;

    int n;

    printf("Enter how many elements u gonna put in the array: ");

    scanf("%d", &n);

    for(int i=0; i<n; i++){

        printf("Enter element %d --> ", i+1);

        scanf("%d", &arr[i]);

    }

    int small = arr[0];

    for(int i=0; i<n; i++){

        if(arr[i]<small){

            small = arr[i];

            position = i+1;

        }

    }

    printf("The smallest element entered is %d \n", small);

    printf("Position of that element is %d", position);

1. **Getting 2nd last digit**

 int n, num;

    printf("Enter the value of n: ");

    scanf("%d", &n);

    while(n>=10){

        n = n/10;

    }

    num = n;

    printf("firstnum is %d \n", num);

1. **Printing nth digit of number**

#include<stdio.h>

#include<math.h>

int main(){

    int n, rem;

    printf("Enter the value of n: ");

    scanf("%d", &n);

    int k;

    printf("Which digit u wanna know: ");

    scanf("%d", &k);

    while(n>=pow(10, (k-1))){

        rem = n%10;

        n = n/10;

    }

    if(k==1){

        printf("%dst digit is %d", k, rem);

    }

    else if(k==2){

        printf("%dnd digit is %d", k, rem);

    }

    else if(k==3){

        printf("%drd digit is %d", k, rem);

    }

    else{

        printf("%dth digit is %d", k, rem);

    }

    return 0;

}

1. **Printing Value in sin(x), cos(x) or other trigon**

float x;

    printf("Enter Value of x: ");

    scanf("%f", &x);

    if(x==0)

     printf("Value of sin(1/x) is Not Possible");

    else{

      printf("Value of sin(1/x): %.4f", sin((1/x)\*3.14159/180)); // Always use \*3.14159/180 to convert radian to degree

    }

1. **Taking Modulus of a Number**

#include<stdio.h>

#include<stdlib.h>

int main(){

    int a;

    printf("Enter a number: ");

    scanf("%d", &a);

    printf("The value of 'a' is %d", abs(a));

    return 0;

}

1. **Printing Random Number**

#include<stdio.h>

#include<stdlib.h>

**int** main(){

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

**return** 0;

}

1. **Printing Random Number after Every Run!**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

**int** main(){

**srand**(**time**(0));

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

**return** 0;

}

1. **Printing Random Number in a range**

int lower = 1, upper = 10, count = 1;   //  You can also take limit using scanf

    int num;

    srand(time(0));

    for (int i = 0; i < count; i++) {

        num = (rand() % (upper - lower + 1)) + lower;

    }

1. **Decimal Number to Binary Number**

#include<stdio.h>

#include<stdlib.h>

int main(){

    int a[10],n,i;

    printf("Enter the number jo aap convert karna chahte ho: ");

    scanf("%d", &n);

    for(i=0;n>0;i++){

        a[i]=n%2;

        n=n/2;

    }

    printf("Binary of Given Number is = ");

    for(i=i-1;i>=0;i--){

    printf("%d", a[i]);

    }

    return 0;

}

**OR**

#include<stdio.h>

#include<stdlib.h>

int main(){

    int a[10],n,i;

    printf("Enter the number jo aap convert karna chahte ho: ");

    scanf("%d", &n);

    for(i=0;n>0;i++){

        a[i]=n%2;

        n=n/2;

    }

    printf("Binary of Given Number is = ");

    for(i=i-1;i>=0;i--){

    printf("%d", a[i]);

    }

    return 0;

}

1. **Binary to Decimal**

#include<stdio.h>

#include<math.h>

int main(){

    int rem, dec = 0, i = 0;

    int n;

    printf("Enter the binary number: ");

    scanf("%d", &n);

    while(n!=0){

        rem = n%10;

        n/=10;

        dec = dec + rem\*pow(2, i);

        i++;

    }

    printf("Decimal mein ans --> %d", dec);

    return 0;

}

**OR**

#include<stdio.h>

void main(){

    int n , p=1;

    int dec = 0, i = 1, d;

    printf("Input a binary number: ");

    scanf("%d", &n);

    int n1 = n;

    for (int j = n; j>0; j= j/10){

        d = j % 10;

            if(i==1){

                  p=p\*1;

            }

            else{

                 p=p\*2;

            }

       dec= dec + (d\*p);

       i++;

    }

        printf("The equivalent Decimal  Number: %d", dec);

}

1. **Binary to Octal**

#include<stdio.h>

#include<math.h>

int main(){

    int sum = 0;

    int count = 0;

    int n;

    printf("Enter the binary number: ");

    scanf("%d", &n);

    int k = n;

    while(n!=0){

        count++;

        n = n/10;

    }

    int arr[count];

    for(int i=0; i<count; i++){

        int l = k%10;

        arr[i] = l;

        k = k/10;

    }

    for(int i=0; i<count; i++){

        // printf("%d", arr[i]);

        sum = (int)(sum + arr[i]\*pow(2, i));

    }

    // printf("Decimal --> %d \n", sum);

    int octal = sum;

    int sum\_count = 0;

    while(sum!=0){

        sum\_count++;

        sum = sum/10;

    }

    int oct[sum\_count];

    for(int i=0; i<sum\_count; i++){

        int l = octal%8;

        oct[i] = l;

        octal = octal/10;

    }

    for(int i=(sum\_count-1); i>=0;i--){

        printf("%d", oct[i]);

    }

    return 0;

}

1. **Octal to Decimal**

#include<stdio.h>

#include<math.h>

int main(){

    int sum = 0;

    int count = 0;

    int n;

    printf("Enter the octal Number: ");

    scanf("%d", &n);

    int k = n;

    while(n!=0){

        count++;

        n = n/10;

    }

    int arr[count];

    for(int i=0; i<count; i++){

        int l = k%10;

        arr[i] = l;

        k = k/10;

    }

    for(int i = 0; i<count; i++){

        // printf("%d", arr[i]);

        sum = (int)(sum + arr[i]\*pow(8, i));

    }

    printf("The decimal is %d", sum);

    return 0;

}

* **Swapping Characters Like in ARJUN swapping A and R, J and U, N being alone not swapped**

**int i=0; int j=i+1;**

**while(i<n-1)**

**{**

**swap(s[i],s[j]);**

**i=j+1;**

**j=i+1;**

**}**

1. **Perfect Number:**

An integer number is said to be “perfect number” if its factors, including 1(but not the number itself), sum to the number.

E.g., 6 is a perfect number because 6=1+2+3].

1. **Abundant Number:**

A number n is said to be Abundant Number if sum of all the proper divisors of the number denoted by sum(n) is greater than the value of the number n. And the difference between these two values is called the abundance.   
Mathematically, if below condition holds the number is said to be Abundant number:

sum(n)> n

abundance = sum(n) – n

1. **Hexadecimal Number**

Hexadecimal Digits are a-f, A-F or 0-9

1. **Majority Element**

A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

1. **Palindrome**

If Number entered is equal to reversed number, then it is palindrome

1. **Harshad Number**

If a number is divisible by the sum of its digits, then it will be known as a Harshad Number. Some Harshad numbers are 8, 54, 120 etc.

1. **Pronic Number**  A Pronic number is a number which is the product of two consecutive integers, that is, a number of the form n(n + 1).
2. **Ugly Number** Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … shows the first 11 ugly numbers. By convention, 1 is included

if((n==1)||(n%2==0 || n%3==0 || n%5==0) && (n%4!=0 && n%6!=0 && n%7!=0 && n%8!=0 && n%9!=0)){

        printf("%d is ugly", n);

    }

1. **Deficient Number**

Deficient Number if sum of all the divisors of the number denoted by divisors Sum(n) is less than twice the value of the number n. The Difference between these two values is called deficiency

1. **Fermat Number**

Fermat numbers are non-negative odd numbers which is valid for all values of k>=0. Only the first seven terms of the sequence are known till date. 2^2^k+1

1. **Mersenne Number**  A Mersenne prime is a prime number that is one less than a power of two. That is, it is a prime number of the form Mn = 2^n − 1 for some integer n.
2. **Hexadecimal Digit** Hexadecimal Digits are a-f, A-F or 0-9
3. **Abundant Number**

An Abundant Number (also known as excessive number) is a number in the number theory. which itself is smaller than the sum of all its proper divisors.

1. **Friendly Pair** Friendly Pair or Amicable numbers are two different numbers related in a way such that the sum of the proper divisors of each is equal to the other number.
2. **Pell Number**

Pn = 2\*Pn-1 + Pn-2

with seeds P0 = 0 and P1 = 1

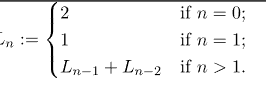
1. **Disarium Number**

A number is Disarium when sum of digits raised to power of respective positions is equal to number itself. E.g. 89, 135, 518 1^1 + 3^2 + 5^3 = 135

1. **Armstrong Number**

abcd... = a^n + b^n + c^n + d^n + …. (n is number of digits) E.g. 153 = 1\*1\*1 + 5\*5\*5 + 3\*3\*3

1. **Lucas Number**

****



1. **Automorphic Number**

A number is called Automorphic number if and only if its square ends in the same digits as the number itself.

1. **Fibonacci Series**

Fn = Fn-1 + Fn-2

F0 = 0 and F1 = 1

1. **Happy Number**

A **happy number** is a number which eventually reaches 1 when replaced by the sum of the square of each digit. For instance, 13 is a happy number because 1^2 + 3^2 = 1 and 1^2 + 0^2 = 1{\displaystyle 1^{2}+3^{2}=10}1^2{\displaystyle 1^{2}+0^{2}=1}1^21ff

A number which is not happy is called sad or unhappy number

1. **Keith Number**

Write a program in C to check if a number is Keith or not(with explanation). A n digit number x is called Keith number if it appears in a special sequence (defined below) generated using its digits. The special sequence has first n terms as digits of x and other terms are recursively evaluated as sum of previous n terms.

Input : x = 197 Output : Yes

197 has 3 digits, so n = 3

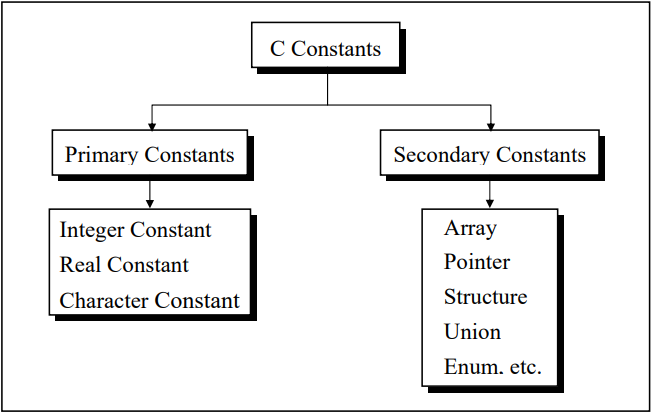
The number is Keith because it appears in the special sequence that has first three terms as 1, 9, 7 and remaining terms evaluated using sum of previous 3 terms. E.g. 1, 9, 7, 17, 33, 57, 107, 197, .....

1. **Narcissistic Number**

Narcissistic Number is a number that is the sum of its own digits each raised to the power of the number of digits

1. **Karpekar Number**

Number whose square in that base can be split in 2 parts that add upto the original number again!

 ***(IMP POINTS)***

**Rules for Constructing Integer Constants**

(a) An integer constant must have at least one digit.

(b) It must not have a decimal point.

(c) It can be either positive or negative. If no sign precedes an integer constant it is assumed to be positive.

(d) No commas or blanks are allowed within an integer constant.

(e) The allowable range for integer constants is -32768 to 32767.

**Rules for Constructing Real Constants**

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

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

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

In exponential form of representation, the real constant is represented in two parts. The part appearing before ‘e’ is called mantissa, whereas the part following ‘e’ is called exponent.

* + The mantissa part and the exponential part should be separated by a letter e.
  + The mantissa part may have a positive or negative sign. Default sign of mantissa part is positive.
  + The exponent must have at least one digit, which must be a positive or negative integer.
  + Default sign is positive.
  + Range of real constants expressed in exponential form is -3.4e38 to 3.4e38. **Ex.: +3.2e-5 4.1e8 -0.2e+3 -3.2e-5**

**Rules for Constructing Character Constants**

A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.

Both the inverted commas should point to the left. For example, ’A’ is a valid character constant whereas ‘A’ is not. The maximum length of a character constant can be 1 character. Ex.: 'A' , 'I' , '5'

**Variable names are names given to locations in memory**.

These locations can contain integer, real or character constants. In any language, the types of variables that it can support depend on the types of constants that it can handle. This is because a particular type of variable can hold only the same type of constant. For example, an integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

**Rules for Constructing Variable Names**

* A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters.
* Do not create unnecessarily long variable names as it adds to your typing effort.
* The first character in the variable name must be an alphabet or underscore.
* No commas or blanks are allowed within a variable name.
* No special symbol other than an underscore can be used in a variable name.

**C Keywords**

Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. Some C compilers allow you to construct variable names that exactly resemble the keywords. However, it would be safer not to mix up the variable names and the keywords. The keywords are also called ‘Reserved words’.

**; acts as a statement terminator.**

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

**C Instructions**

There are basically three types of instructions in C:

* Type Declaration Instruction 🡪 To declare the type of variables used in a C program
* Arithmetic Instruction 🡪 To perform arithmetic operations between constants and variables.
* Control Instruction 🡪 To control the sequence of execution of various statements in a C program.

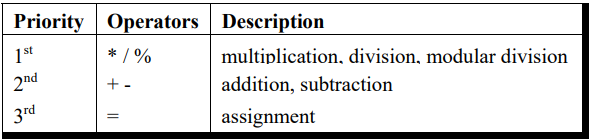
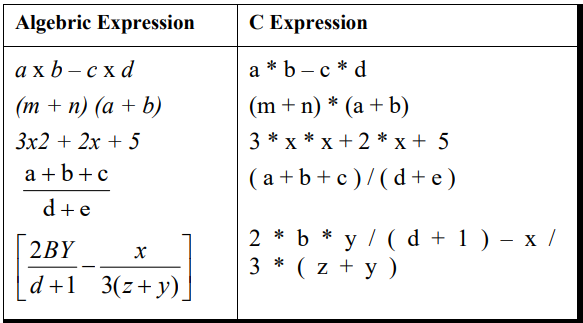
\*, /, -, + are the arithmetic operators. = is the assignment operator. 2, 5 and 3200 are integer constants. 3.2 and 0.0056 are real constants.

The variables and constants together are called ‘operands’

**A C arithmetic statement could be of three types**.

These are as follows:

* **Integer mode arithmetic statement** - This is an arithmetic statement in which all operands are either integer variables or integer constants. Ex.: int i, king, issac, noteit ; i = i + 1 ; king = issac \* 234 + noteit - 7689 ;
* **Real mode arithmetic statement** - This is an arithmetic statement in which all operands are either real constants or real variables. Ex.: float qbee, antink, si, prin, anoy, roi ; qbee = antink + 23.123 / 4.5 \* 0.3442 ; si = prin \* anoy \* roi / 100.0 ;
* **Mixed mode arithmetic statement** - This is an arithmetic statement in which some of the operands are integers and some of the operands are real. Ex.: float si, prin, anoy, roi, avg ; int a, b, c, num ; si = prin \* anoy \* roi / 100.0 ; avg = ( a + b + c + num ) / 4 ;
* It is very important to understand how the execution of an arithmetic statement takes place. Firstly, the right-hand side is evaluated using constants and the numerical values stored in the variable names. This value is then assigned to the variable on the left-hand side.

** **

**Control Instructions in C**

There are four types of control instructions in C. They are:

(a) Sequence Control Instruction

(b) Selection or Decision Control Instruction

(c) Repetition or Loop Control Instruction

(d) Case Control Instruction

The Sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program. Decision and Case control instructions allow the computer to take a decision as to which instruction is to be executed next. The Loop control instruction helps computer to execute a group of statements repeatedly.

A decision control instruction can be implemented in C using:

(a) The if statement

(b) The if-else statement

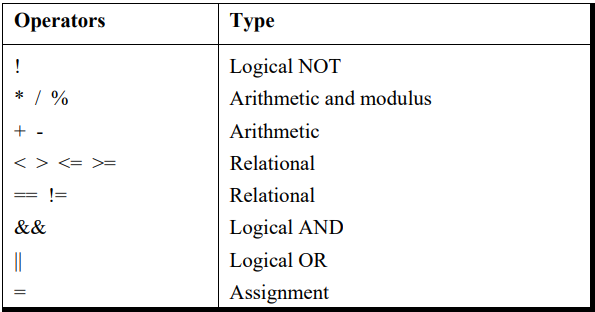
(c) The conditional operators

**🡪** Use of Logical Operators C allows usage of three logical operators, namely, &&, || and !. These are to be read as ‘AND’ ‘OR’ and ‘NOT’ respectively.

* **The ! Operator**

!(y < 10 ) This means “not y less than 10”. In other words, if y is less than 10, the expression will be false, since ( y < 10 ) is true. We can express the same condition as ( y >= 10 ).

The NOT operator is often used to reverse the logical value of a single variable, as in the expression if ( ! flag ) This is another way of saying if ( flag == 0 ) Does the NOT operator sound confusing? Avoid it if you want, as the same thing can be achieved without using the NOT operator.



**The Conditional Operators**

The conditional operators ? and : are sometimes called ternary operators since they take three arguments. In fact, they form a kind of foreshortened if-then-else.

**expression 1 ? expression 2 : expression 3**

What this expression says is: “if expression 1 is true (that is, if its value is non-zero), then the value returned will be expression 2, otherwise the value returned will be expression 3”

1. It’s not necessary that the conditional operators should be used only in arithmetic statements. This is illustrated in the following examples:

Ex.: **int i ;**

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

**( i == 1 ? printf ( "Amit" ) : printf ( "All and sundry" ) ) ;**

Ex.: **char a = 'z' ;**

**printf ( "%c" , ( a >= 'a' ? a : '!' ) ) ;**

1. The conditional operators can be nested as shown below.

**int big, a, b, c ;**

**big = ( a > b ? ( a > c ? 3: 4 ) : ( b > c ? 6: 8 ) ) ;**

1. Check out the following conditional expression: a > b ? g = a : g = b ; This will give you an error ‘Value Required’. The error can be overcome by enclosing the statement in the : part within a pair of parenthesis. This is shown below: a > b ? g = a : ( g = b ) ; In absence of parentheses the compiler believes that b is being assigned to the result of the expression to the left of second =. Hence it reports an error.
2. The limitation of the conditional operators is that after the ? or after the : only one C statement can occur. In practice rarely is this the requirement. Therefore, in serious C programming conditional operators aren’t as frequently used as the if-else.

**Why Use Functions**

1. Writing functions avoids rewriting the same code over and over.
2. Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

In the programs the moment closing brace ( } ) of the called function was encountered the control returned to the calling function. No separate return statement was necessary to send back the control. This approach is fine if the called function is not going to return any meaningful value to the calling function. In the above program, however, we want to return the sum of x, y and z. Therefore, it is necessary to use the return statement.

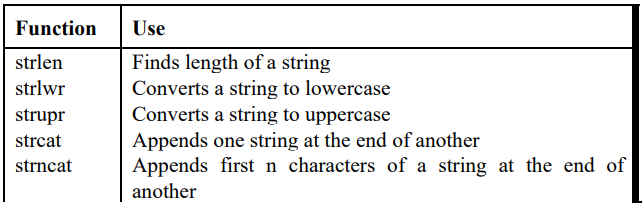
**ARRAYS**

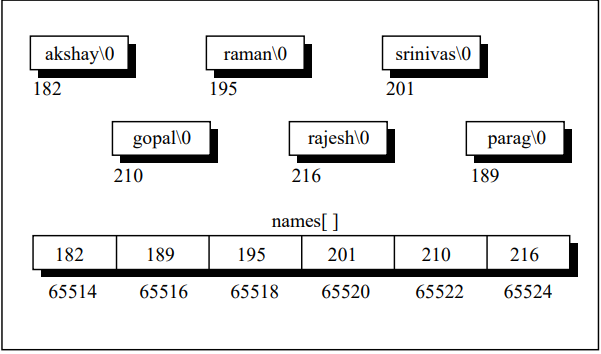
**Array[p][q][r]**

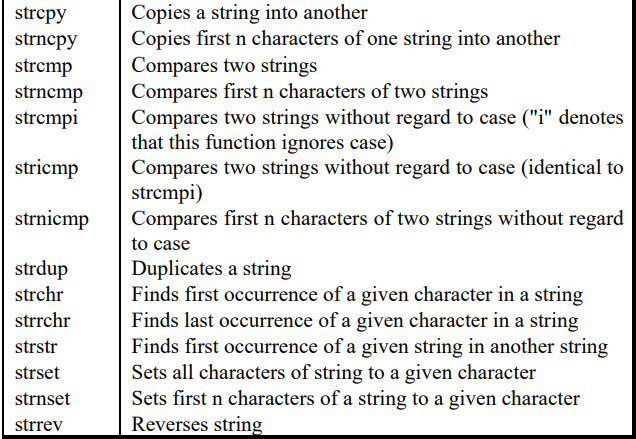
**\*( \*( \*( Array + p ) + q ) + r )**

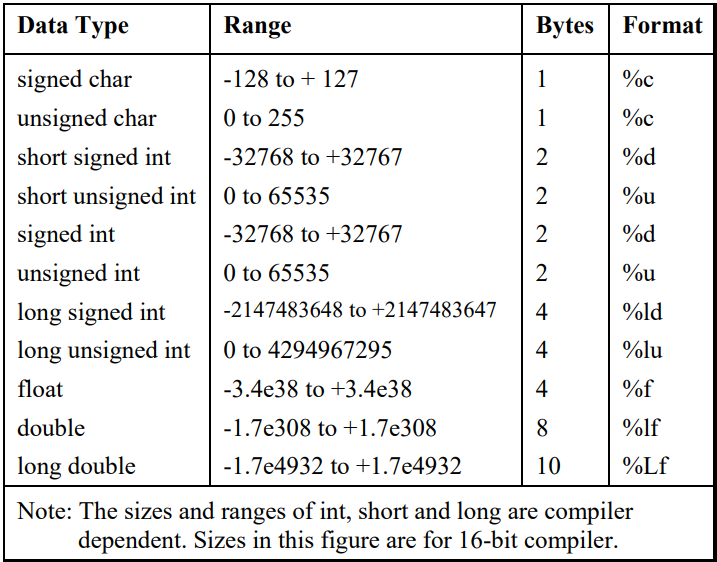
arr[2][3][1]

\*( \*( \*( arr + 2 ) + 3 ) + 1 )



****



****

**See Storage Classes on pg-238 and Conditional and Miscellaneous Directives on pg-275 in Let Us C**

**SUMMARY:**

1. The three primary constants and variable types in C are integer, float and character.
2. A variable name can be of maximum 31 characters. Do not use a keyword as a variable name.
3. An expression may contain any sequence of constants, variables and operators.
4. Operators having equal precedence are evaluated using associativity. Left to right associativity means that the left operand of a operator must be unambiguous whereas right to left associativity means that the right operand of an operator must be unambiguous.
5. Input/output in C can be achieved using scanf( ) and printf( ) functions.
6. There are three ways for taking decisions in a program. First way is to use the if-else statement, second way is to use the conditional operators and third way is to use the switch statement.
7. The default scope of the if statement is only the next statement. So, to execute more than one statement they must be written in a pair of braces.
8. An if block need not always be associated with an else block. However, an else block is always associated with an if statement.
9. If the outcome of an if-else ladder is only one of two answers then the ladder should be replaced either with an else-if clause or by logical operators.
10. && and || are binary operators, whereas, ! is a unary operator.
11. In C every test expression is evaluated in terms of zero and non-zero values. A zero value is considered to be false and a non-zero value is considered to be true.
12. Assignment statements used with conditional operators must be enclosed within a pair of parentheses.
13. A break statement takes the execution control out of the loop
14. A continue statement skips the execution of the statements after it and takes the control to the beginning of the loop.
15. A do-while loop is used to ensure that the statements within the loop are executed at least once.
16. The operators +=, -=, \*=, /=, %= are compound assignment operators. They modify the value of the operand to the left of them.
17. When we need to choose one among number of alternatives, a switch statement is used.
18. The switch keyword is followed by an integer or an expression that evaluates to an integer.
19. The case keyword is followed by an integer or a character constant.
20. The control falls through all the cases unless the break statement is given.
21. The usage of the goto keyword should be avoided as it usually violets the normal flow of execution.
22. An array is also known as a subscripted variable.
23. An array is similar to an ordinary variable except that it can store multiple elements of similar type.
24. Compiler doesn’t perform bounds checking on an array.
25. The array variable acts as a pointer to the zeroth element of the array. In a 1-D array, zeroth element is a single value, whereas, in a 2-D array this element is a 1-D array.
26. On incrementing a pointer it points to the next location of its type.
27. Array elements are stored in contiguous memory locations and so they can be accessed using pointers. Only limited arithmetic can be done on pointers.
28. If the array is declared as a global one or as static in a function, then all elements are initialized to zero if they aren't initialized already.
29. **Address of a floating-point variable is always a whole number.** True.
30. malloc( ) function can be used to allocate space in memory on the fly during execution of the program.
31. Though scanf( ) can be used to receive multi-word strings, gets( ) can do the same job in a cleaner way.
32. Both printf( ) and puts( ) can handle multi-word strings
33. Strings can be operated upon using several standard library functions like strlen( ), strcpy( ), strcat( ) and strcmp( ) which can manipulate strings. More importantly we imitated some of these functions to learn how these standard library functions are written.
34. Though in principle a 2-D array can be used to handle several strings, in practice an array of pointers to strings is preferred since it takes less space and is efficient in processing strings.
35. A structure is usually used when we wish to store dissimilar data together. Structure elements can be accessed through a structure variable using a dot (.) operator.
36. Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.
37. All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator. It is possible to pass a structure variable to a function either by value or by address.
38. It is possible to create an array of structures.
39. We can use different variations of the primary data types, namely signed and unsigned char, long and short int, float, double and long double. There are different format specifications for all these data types when they are used in scanf( ) and printf( ) functions.
40. The maximum value a variable can hold depends upon the number of bytes it occupies in memory. By default all the variables are signed. We can declare a variable as unsigned to accommodate greater value without increasing the bytes occupied.
41. We can make use of proper storage classes like auto, register, static and extern to control four properties of the variable—storage, default initial value, scope and life.
42. The preprocessor directives enable the programmer to write programs that are easy to develop, read, modify and transport to a different computer system.
43. We can make use of various preprocessor directives such as #define, #include, #ifdef - #else - #endif, #if and #elif in our program. The directives like #undef and #pragma are also useful although they are seldom used.
44. C is a Middle Level Language.
45. Tokens are the smallest elements of a program, which are meaningful to the compiler. The following are the types of tokens: Keywords, Identifiers, Constant, Strings, Operators, etc.
46. Max Length of variables in C 🡪 32
47. Integer division results in Truncating the fractional part.
48. Explicit type conversion is known as Casting.
49. p++ executes faster than p+1 because p++ is a single instruction
50. Header files in C contain Library functions.
51. The printf() function retunes which value when an error occurs Negative value
52. Symbolic constants can be defined using constant.
53. Null character is represented by \0
54. C supports 3 looping constructs.
55. Number of relational operators in C are 6
56. A link is an analysing tool in C.
57. The continue command cannot be used with for.
58. A multidimensional array can be expressed in terms of array of pointers rather than as pointers to a group of a contiguous array.
59. A pointer to a pointer is a form of multiple indirections and a chain of pointers
60. Pointers are of unsigned integer data types.
61. Using C language programmers can write their own library functions
62. C language is used in development for Databases, Graphic Applications, Word Processors etc.
63. 31 characters used to distinguish Identifier or Names of Functions and Global variables.

**C++ BASICS**

**Programmers code in various Programming Languages to perform Computational Tasks**

1. **Machine level Language**: Machine code or machine language is a set of instructions executed directly by a computer's central processing unit (CPU). Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory. Every program directly executed by a CPU is made up of a series of such instructions.
2. **Assembly level Language:** An assembly language (or assembler language) is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.
3. **High level Language**: High-level language is any programming language that enables development of a program in much simpler programming context and is generally independent of the computer's hardware architecture. High-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization.

**High Level Language are categorized into 2 types 🡪 1) Procedure Oriented Programming (POP)**

**2) Object Oriented Programming (OOP)**

* **POP 🡪** Procedure oriented programming basically consist of writing a list of instruction or actions for the computer to follow and organizing this instruction into groups known as functions.

The Disadvantage of the POP languages are:

**1. Global data access**

**2. It does not model real word problem very well**

**3. No data hiding**

**CHARACTERISTICS OF POP**

1. Emphasis is on doing things(algorithm).
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data.
4. Data move openly around the system from function to function.
5. Function transforms data from one form to another.
6. Employs top-down approach in program design

* **OOP** 🡪“Object oriented programming as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand”.

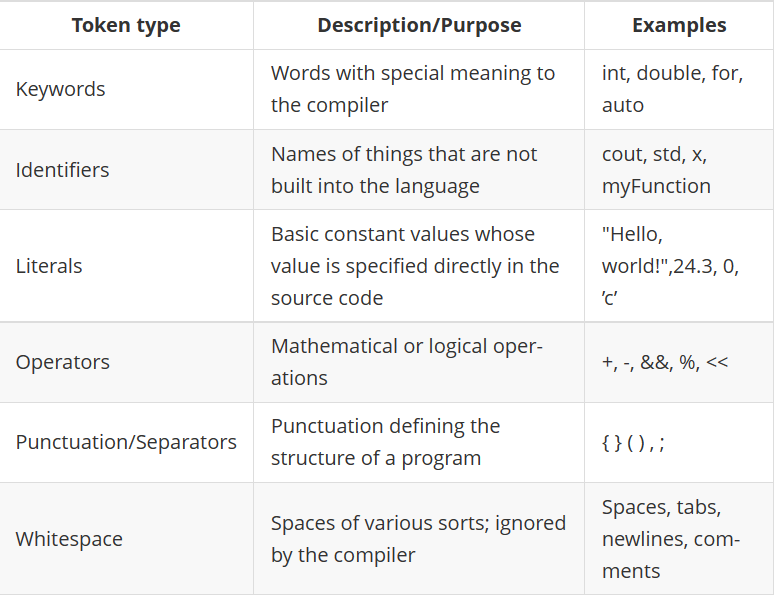
**CHARACTERISTICS OF OOP**

1. Emphasis is on doing rather than procedure.
2. Programs are divided into what are known as objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and can’t be accessed by external functions.
6. Objects may communicate with each other through functions.
7. New data and functions can be easily added.
8. Follows bottom-up approach in program design.

**------------------------------------------------------------------------------------------------------------------------------------**

* **Why C++**
* **Conciseness**: programming languages allow us to express common sequences of com­ mands more concisely. C++ provides some especially powerful shorthands.
* **Maintainability**: modifying code is easier when it entails just a few text edits, instead of rearranging hundreds of processor instructions. C++ is object oriented, which further improves maintainability.
* **Portability**: diﬀerent processors make diﬀerent instructions available. Programs writ­ ten as text can be translated into instructions for many diﬀerent processors; one of C++’s strengths is that it can be used to write programs for nearly any processor.

Tokens are the minimal chunk of program that have meaning to the compiler – the smallest meaningful symbols in the language. Our code displays all 6 kinds of tokens, though the usual use of operators is not present here:



Namespaces: In C++, identiﬁers can be deﬁned within a context – sort of a directory of names – called a namespace. When we want to access an identiﬁer deﬁned in a namespace, we tell the compiler to look for it in that namespace using the scope resolution operator (::). Here, we’re telling the compiler to look for cout in the std namespace, in which many standard C++ identiﬁers are deﬁned.

* A signed integer is one that can represent a negative number; an unsigned integer will never be interpreted as negative, so it can represent a wider range of positive numbers. Most compilers assume signed if unspeciﬁed.
* There are actually 3 integer types: short, int, and long, in non-decreasing order of size (int is usually a synonym for one of the other two). You generally don’t need to worry about which kind to use unless you’re worried about memory usage or you’re using really huge numbers. The same goes for the 3 ﬂoating point types, float, double, and long double, which are in non-decreasing order of precision (there is usually some imprecision in representing real numbers on a computer).
* The sizes/ranges for each type are not fully standardized; those shown above are the ones used on most 32-bit computers. An operation can only be performed on compatible types. You can add 34 and 3, but you can’t take the remainder of an integer and a ﬂoating-point number.

References are just pointers that are dereferenced every time they are used. Just like point­ers, you can pass them around, return them, set other references to them, etc. The only differences between using pointers and using references are:

* References are sort of pre-dereferenced – you do not dereference them explicitly.
* You cannot change the location to which a reference points, whereas you can change the location to which a pointer points. Because of this, references must always be initialized when they are declared.
* When writing the value that you want to make a reference to, you do not put an & before it to take its address, whereas you do need to do this for pointers.

Structures are a way of storing many different values in variables of potentially different types under the same name. Classes define types of data structures and the functions that operate on those data structures.

* **Identifiers 🡪** Identifiers refers to the name of variable, functions, array, class etc. created by programmer. Each language has its own rule for naming the identifiers.

1. Only alphabetic chars, digits and underscore are permitted.

2. The name can’t start with a digit.

3. Upper case and lower-case letters are distinct.

4. A declared keyword can’t be used as a variable name.

In ANSI C the maximum length of a variable is 32 chars but in C++ there is no bar

Accessible by any function

Global Variable

Accessible by function A

Function B

Local Variable

Function A

Local Variable

Accessible by function B

**Object-Oriented-Programming**

An object-oriented programming approach is a collection of objects and each object consists of corresponding data structures and procedures. The program is reusable and more maintainable. The important aspect in OOP is a class which has similar syntax that of structure.

**Class** 🡪 It is a collection of data and member functions that manipulate data. The data components of class are data members and functions that manipulate the data are **member functions**.

It can also called as blue print or prototype that defines the variables and functions common to all objects of certain kind. It is also known as user defined data type or ADT(abstract data type) A class is declared by the keyword class.

**Object or Attributes or Characteristics 🡪** Instance of a class is called as object. Objects are the basic run-time entities in an object-oriented system. The fundamental idea behind object-oriented approach is to combine both data and function into a single unit and these units are called objects.

**Access Control 🡪** Access specifier or access modifiers are the labels that specify type of access given to members of a class. These are used for data hiding. These are also called as visibility modes. There are three types of access specifiers

1. **Private** - If the data members are declared as private access, then they cannot be accessed from other functions outside the class. It can only be accessed by the functions declared within the class. It is declared by the key word **private**.

2. **Public** - If the data members are declared public access then they can be accessed from other functions out side the class. It is declared by the key word **public**.

3. **Protected** -The access level of protected declaration lies between public and private. This access specifier is used at the time of inheritance.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The way you structure a program C++ is:

1. Split it up into a set of tasks and subtasks .
2. Make functions for the tasks .
3. Instruct the computer to perform them in sequence.

With large amounts of data and/or large numbers of tasks, this makes for complex and unmaintainable programs.

To manage this complexity, it’s nicer to package up self-sufficient, modular pieces of code. OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.

There are lots of definitions for OOP, but 3 primary features of it are:

* Encapsulation: grouping related data and functions together as objects and defining an interface to those objects.
* Inheritance: allowing code to be reused between related types .
* Polymorphism: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type .

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**ENCAPSULATION**

Encapsulation just refers to packaging related stuff together. We’ve already seen how to package up data and the operations it supports in C++: with classes. If someone hands us a class, we do not need to know how it actually works to use it; all we need to know about is its public methods/data – its interface.

Interfaces abstract away the details of how all the operations are actually performed, allowing the programmer to focus on how objects will use each other’s interfaces – how they interact.

This is why C++ makes you specify public and private access specifiers: by default, it assumes that the things you define in a class are internal details which someone using your code should not have to worry about. The practice of hiding away these details from client code is called “data hiding,” or making your class a “black box.”

One way to think about what happens in an object-oriented program is that we define what objects exist and what each one knows, and then the objects send messages to each other (by calling each other’s methods) to exchange information and tell each other what to do.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**POLYMORPHISM**

Polymorphism means “many shapes.” It refers to the ability of one object to have many types. If we have a function that expects a Vehicle object, we can safely pass it a Car object, because every Car is also a Vehicle. Likewise for references and pointers: anywhere you can use a Vehicle , you can use a Car .

**VIRTUAL FUNCTIONS**

A virtual function or virtual method is a function or method whose behaviour can be overridden within an inheriting class by a function with the same signature. This concept is an important part of the polymorphism.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**DATA ABSTRACTION** :

Abstraction refers to the act of representing essential features without including the back ground details or explanations. Classes use the concept of abstraction and are defined as size, width and cost and functions to operate on the attributes.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**DYNAMIC BINDING :**

Binding refers to the linking of a procedure call to the code to the executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with a polymorphic reference depends upon the dynamic type of that reference

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**MESSAGE PASSING :**

An object oriented program consists of a set of objects that communicate with each other. A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and information to be sent.

**Example 🡪** Employee**.**Salary(name), Employee is object, Salary is the Message and name is the information

**BENEFITS OF OOPS**

* Through inheritance, we can eliminate redundant code extend the use of existing
* Classes.
* We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
* The principle of data hiding helps the programmer to build secure program that cannot be invaded by code in other parts of a programs.
* It is possible to have multiple instances of an object to co-exist without any interference.
* It is possible to map object in the problem domain to those in the program.
* It is easy to partition the work in a project based on objects.
* The data-centered design approach enables us to capture more detail of a model can implemental form.
* Object-oriented system can be easily upgraded from small to large system.
* Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
* Software complexity can be easily managed.

**OBJECT ORIENTED LANGUAGE 🡪**

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

* 1. **Object-based programming** is the style of programming that primarily supports encapsulation and object identity. Major feature that are required for object based programming are: • Data encapsulation • Data hiding and access mechanisms • Automatic initialization and clear-up of objects • Operator overloading Languages that support programming with objects are said to the objects-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.
  2. **Object-oriented programming** language incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statements:

Object-based features + inheritance + dynamic binding

**APPLICATIONS**

* Real-time system
* Simulation and modelling
* Object-oriented data bases
* Hypertext, Hypermedia
* AI and expert systems
* Neural networks and parallel programming
* Decision support and office automation systems
* CIM/CAM/CAD system

The object-oriented paradigm sprang from the language, has matured into design, and has recently moved into analysis. It is believed that the richness of OOP environment will enable the software industry to improve not only the quality of software system but also its productivity. Object-oriented technology is certainly going to change the way the software engineers think, analyse, design and implement future system.

**IMP POINTS**

1. **Accessing Members: -** Dot operator is used to access members of class
2. The identifier cout (pronounced “C out”) is actually an object. The operator << is called the insertion or put to operator. >> is the extraction or get from operator.
3. A C++ instruction that tells the computer to do something is called a **statement.**
4. HelloATG 🡪 Identifier, HelloATG() 🡪 Function
5. It’s perfectly all right to use variables of different data types in the same arithmetic expression.
6. Remainder operator is also known as modulus operator.
7. Dividing a program into functions makes the program easier to conceptualize and may reduce the size of the program.
8. A function body must be followed by **parentheses**
9. An arithmetic assignment operator combines the effect of Assignment (=) and arithmetic (+,\*,-,/).
10. A function body is defined by **braces { }.**
11. The loop body, which consists of braces delimiting several statements, is called a block of code. One important aspect of a block is that a variable defined inside the block is not visible outside it. Visible means that program statements can access or “see” the variable.

You can’t access this variable outside the block; it’s only visible within the braces. Thus if you placed the statement cube = 10; after the loop body, the compiler would signal an error because the variable cube would be undefined outside the loop.

1. Main() function is important as it is the first function executed when the program starts.
2. Two exceptions to the rule that the compiler ignores whitespace are the string constant and pre-processor directives.
3. If the access specifier is not specified in the class the default access specifier is private
4. The actual code for library functions is contained in a library file.
5. All member functions are to be declared as public if not they are not accessible outside the class.
6. There are 2 types of scope. First there is **Local Scope** which is local to the function of the program and the other one is **Global Scope** which has visibility to all functions of the program. **Scope Resolution Operator is “::”**
7. **Inline Functions 🡪** An inline function is a function that is expanded in line when it is invoked. Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. It is defined by using key word “**inline**”.

**Why Inline Functions?**

1. One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times.
2. Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.
3. When a function is small, a substantial percentage of execution time may be spent in such overheads. One solution to this problem is to use macro definitions, known as macros. Pre-processor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.
4. C++ has different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function.

**Why not Inline Functions?**

1. A function that is returning value, if it contains switch, loop or both then it is treated as normal function.
2. if a function is not returning any value and it contains a return statement then it is treated as normal function
3. if a function is not returning any value and it contains a return statement then it is treated as normal function
4. If the inline function is declared as recursive function, then it is executed as normal function.
5. **Memory Allocation for Objects:** Memory for objects is allocated when they are declared but not when class is defined. All objects in a given class uses same member functions. The member functions are created and placed in memory only once when they are defined in class definition
6. **Static Class Members 🡪 a) Static Data Members b) Static Member Functions**

**Static Data Members** 🡪 A data member of a class can be qualified as static. A static member variable has certain special characteristics:

1. It is initialized to zero when the first object of its class is created. No other initialization is permitted.
2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
3. It is visible only within the class, but its lifetime is the entire program.
4. Static data member is defined by keyword **static**

**Static Member Functions 🡪** Like static member variable, we can also have static member functions. A member function that is declared static has the following properties: A static function can have access to only other static members (functions or variables) declared in the same class. A static member function is to be called using the class name (instead of its objects) as follows: **class-name :: function-name;**

1. **Arrays of Objects 🡪** Arrays of variables of type "class" is known as "Array of objects". An array of objects is stored inside the memory in the same way as in an ordinary array.
2. **Friend Class:** A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class. This requires the condition that the friend becoming class must be first declared or defined (forward declaration).
3. **Constructors 🡪** A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same name as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

**Characteristics of Constructors 🡪**

1. They should be declared in the public section.
2. They are invoked automatically when the objects are created.
3. They do not have return type, not even void.
4. They cannot be inherited, though a derived class can call the base class constructor.
5. Like other C++ functions, they can have default arguments.
6. Constructors cannot be virtual.
7. We cannot refer to their addresses.
8. They make **implicit calls** to the operators new and delete when memory allocation is required.

**Types of Constructors 🡪**

1. Default Constructor 🡪 A constructor that accepts no parameters is called the default constructor.
2. Parameterized Constructor 🡪 The constructors that take parameters are called parameterized constructors.
3. Copy Constructor 🡪 A copy constructor is used to declare and initialize an object from another object.
4. **Multiple Constructors 🡪** Multiple constructors can be declared in a class. There can be any number of constructors in a class.
5. **Destructors 🡪** A destructor, is used to destroy the objects that have been created by a constructor.   
    Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a **tilde**.

**Example 🡪 ~item() { }**

1. A destructor never takes any argument nor does it return any value.

2. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.

3. It is a good practice to declare destructors in a program since it releases memory space for future use.

1. **Inheritance 🡪** The mechanism of deriving a new class from an old one is called inheritance or derivation. The old class is referred to as the base class and the new one is called the derived class or sub class. The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. Reusability is an important feature of OOP. A derived class can be defined by specifying its relationship with the base class in addition to its own details.

**Types of Inheritance:**

1. Single Inheritance
2. Multi-level Inheritance
3. Multiple Inheritance
4. Hybrid inheritance
5. Hierarchical Inheritance.
6. **DYNAMIC MEMORY ALLOCATION & DEALLOCATION (new & delete)**

C uses malloc() and calloc() functions to allocate memory dynamically at run time. It uses the function free() to deallocated dynamically allocated memory.

C++ supports these functions; it defines two unary operators new and delete that perform the task of allocating and deallocating the memory in a better and easier way.

A object can be created by using new, and destroyed by using delete.

A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete.

1. **New operator** can be used to create objects of any type. Hence new operator allocates sufficient memory to hold data of objects and it returns address of the allocated memory.

**pointer-variable = new data-type;** Ex: int \*p = new int;

pointer-variable = new data-type[size]; Ex: int \*p = new int[10];

1. **Delete Operator 🡪** If the variable or object is no longer required or needed is destroyed by “delete” operator, there by some amount of memory is released for future purpose.

Syntax: delete pointer-variable; Ex: delete p;

If we want to free a dynamically allocated array: delete [size] pointer-variable

1. **Instances 🡪** An instance is an occurrence of a class. Different instances can have their own set of values in their fields.
2. **OBJECTS AS FUNCTION ARGUMENTS**

Like any other data type, an object may be used as A function argument. This can come in two ways

1. A copy of the entire object is passed to the function.

2. Only the address of the object is transferred to the function

The first method is called pass-by-value. Since a copy of the object is passed to the function, any change made to the object inside the function do not affect the object used to call the function. The second method is called pass-by-reference. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the functions will reflect in the actual object. The pass by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

1. **Friend Function**

A friend function processes certain special characteristics:

* 1. It is not in the scope of the class to which it has been declared as friend.
  2. Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a member function without the help of any object.
  3. Unlike member functions

-----------------------------------------------

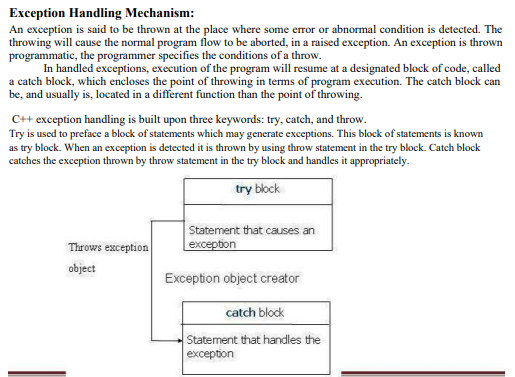
-------------------------------------------

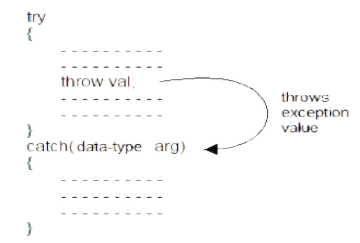
1. **Destructor** 🡪 A destructor, us the name implies is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.
2. The **private and protected members** of a class can be accessed by: # INHERITANCE
   1. A function i.e., friend of a class.
   2. A member function of a class that is the friend of the class.
   3. A member function of a derived class.
3. **Generic Programming 🡪** Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

**Features 🡪**

* + 1. It eliminates redundant code
    2. It enhances the reusability of the code.
    3. It provides great flexibility to language

1. **Exception Handling 🡪** Exceptions: Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing .Anomalies might include conditions such ass division by zero, accessing an array outside of its bounds or running out of memory or disk space. When a program encounters an exception condition, it must be identified and handled.

****



1. **Templates 🡪** Template supports generic programming, which allows developing reusable software components such as functions, classes, etc supporting different data types in a single frame work. A template in C++ allows the construction of a family of template functions and classes to perform the same operation o different data types. The templates declared for functions are called class templates. They perform appropriate operations depending on the data type of the parameters passed to them.
   * 1. C++ supports a mechanism known as template to implement the concept of generic programming.
     2. Template allows us to generate a family of classes or a family of functions to handle different data types.
     3. Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable.
     4. We can use multiple parameters in both the class templates and function templates.
     5. A specific class created from a class template is called a template class and the process of creating a template class is known as instantiation.
     6. Like other functions, template functions can be overloaded.
     7. Member function of a class template must be defined as function templates using the parameters of the class template.
     8. We may also use non type parameters such basic or derived data types as arguments templates.
2. **SYSTEMS 🡪**

There exist a number of software development paradigms, each using a different set of methods and tools.

* + - Satisfy the user requirements
    - Be easy to understand by the users and operators
    - Be easy to operate
    - Have a good user interface
    - Be easy to modify
    - Be expandable
    - Have adequate security controls against misuse of data
    - Handle the errors and exceptions satisfactorily, and
    - Be delivered on schedule within the budget.

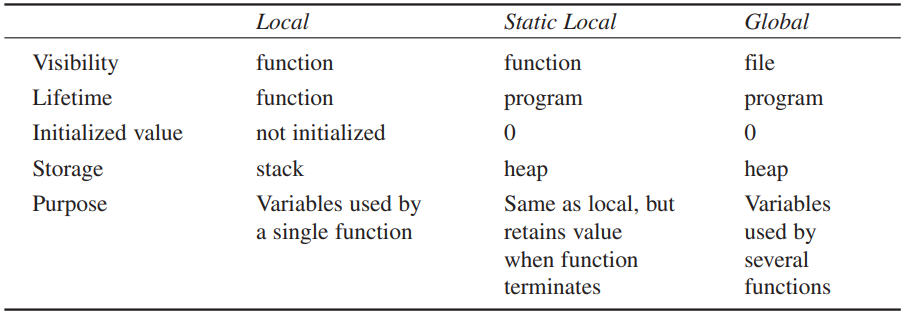
1. **Exception Handling 🡪** C++ exception handling mechanism is basically built upon three keywords namely try, throw and catch. The keyword try is used to preface a block of statements which may generate exceptions. This block of statement is called try block. When an exception is detected, it is thrown using throw statement in the try block. A catch block defined by the keyword catch ‘catches’ the exception thrown by the throw statement in the try block and handles it appropriately. The catch block that catches an exception must immediately follow the try block that throws the exception.

* Exceptions are peculiar problems that a program may encounter at run time.
* ANSI C++ has built in language function for trapping the errors and controlling the exceptions. All C ++ compilers support this newly added facility.
* An exception is an object. It is send from the part of the program where an error occurs to the part of program which is going to control the error.
* C++ exception method provides three keywords, try, throw and catch. The keyword try is used at the starting of exception. The entire exception statement are enclosed in the curly braces. It is known as try block.
* The catch block receives the exception send by the throw block in the try block.
* Multiple catch blocks can be used in a program .
* It is also possible to define single or default catch () block from one or more exception of different type. In such situation a single catch block is used for catch exceptions thrown by the multiple throw statement.
* It is also possible to again pass the exception received to another exception handler i.e. an exception is thrown from catch () block and this is known as rethrowing the exception.
* The specified exception are used when we want to bind the function to throw only specified exceptions. Using a throw list condition can also do this.

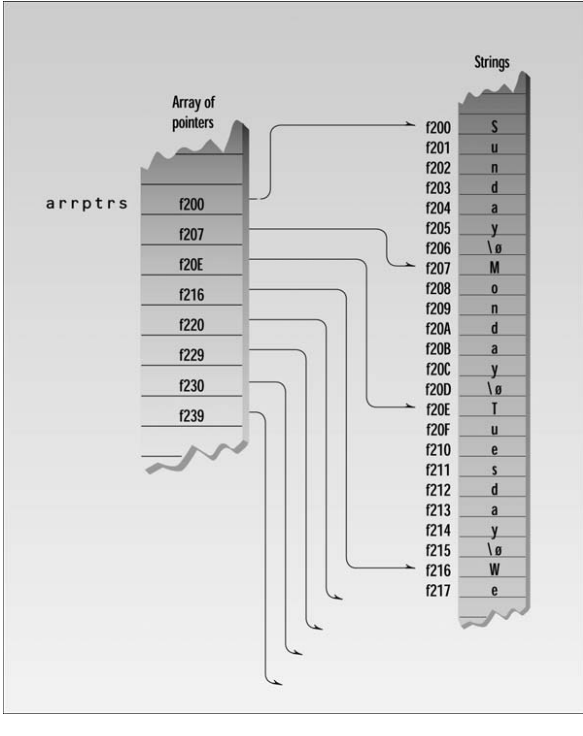
1. Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called **reusability**. It is similar to the way a library of functions in a procedural language can be incorporated into different programs.
2. 2 major components of objects are 🡪 data and functions that operate on that data.
3. A widget is to the blueprint for a widget as an object is to a class. Protecting data from access by unauthorized functions is called **data hiding.**
4. **Objects** model entities in the real world more closely than do functions.
5. When a language has the capability to produce new data types , it is said to be **Extensible.**
6. The ability of a function or operator to act in different ways on different data types is called **polymorphism**.
7. A normal C++ operator that acts in special ways on newly defined data types is said to be, **Overloaded.**
8. The unified Modelling language is, **helpful in developing software systems.**
9. **Whitespace** is defined as spaces, carriage returns, linefeeds, tabs, vertical tabs, and form feeds. These characters are invisible to the compiler. You can put several statements on one line, separated by any number of spaces or tabs, or you can run a statement over two or more lines.
10. Library Function exit() When PRIME discovers that a number is not prime, it exits immediately, since there’s no use proving more than once that a number isn’t prime. This is accomplished with the library function exit(). This function causes the program to terminate, no matter where it is in the listing. It has no return value. Its single argument, 0 in our example, is returned to the operating system when the program exits. (This value is useful in batch files, where you can use the ERRORLEVEL value to query the return value provided by exit(). The value 0 is normally used for a successful termination; other numbers indicate errors.)
11. A relational operator yields a Boolean result and compares two operands
12. A block of code is delimited by braces(curly braces).
13. A variable defined within a block is visible from the point of definition onward in the block.
14. The goto statement causes control to go to a label.
15. Executing the continue operator from within a loop causes control to go to the top of the loop.
16. What is the character obtained from cin when the user presses the Enter key? 🡪 ‘\r’
17. An else always matches the preceding if, unless the if is surrounded by braces.
18. The else...if construction is obtained from a nested if...else by reformatting.
19. The first three enumerators of an enum type are normally represented by the values 0, 1 and 2.
20. Global Variables are also called external variables. Global variables create organizational problems because they can be accessed by any function. The wrong functions may access them, or functions may access them incorrectly. In an object-oriented program, there is much less necessity for global variables.

Global variables have storage class static, which means they exist for the life of the program. Memory space is set aside for them when the program begins, and continues to exist until the program ends. You don’t need to use the keyword static when declaring global variables; they are given this storage class automatically.

1. A static local variable has the visibility of an automatic local variable (that is, inside the function containing it). However, its lifetime is the same as that of a global variable, except that it doesn’t come into existence until the first call to the function containing it. Thereafter it remains in existence for the life of the program. Static local variables are used when it’s necessary for a function to remember a value when it is not being executed; that is, between calls to the function. A function, getavg(), calculates a running average. It remembers the total of the numbers it has averaged before, and how many there were. Each time it receives a new number, sent as an argument from the calling program, it adds this number to the total, adds 1 to a count, and returns the new average by dividing the total by the count.



1. Variables possess a characteristic called the storage class. The most common storage class is automatic. Local variables have the automatic storage class: they exist only while the function in which they are defined is executing. They are also visible only within that function. Global variables have static storage class: they exist for the life of a program. They are also visible throughout an entire file. Static local variables exist for the life of a program but are visible only in their own function.
2. An overloaded function is actually a group of functions with the same name. Which of them is executed when the function is called depends on the type and number of arguments supplied in the call. An inline function looks like a normal function in the source file but inserts the function’s code directly into the calling program. Inline functions execute faster but may require more memory than normal functions unless they are very small.
3. A function itself is called the function **definition**
4. A static local variable is used to make a variable visible to only one function, retain a value when a function is not executing.
5. Overloaded Functions are a group of functions with the same name and they make life simple!
6. The first line of a function definition is referred to as the declarator.
7. When an argument is passed by reference the function accesses the argument’s original value in the calling program
8. In general, an inline function executes faster than a normal function, but requires more memory
9. In general, an inline function executes visibility than a normal function, but requires lifetime memory.
10. A one-statement description of a function is referred to as a function declarator or a prototype.
11. A program that invokes the function is function call and statements that carry out the work is function body
12. When a function returns a value, the entire function call can appear on the right side of the equal sign and be assigned to another variable.
13. A **class** has the same relation to an **object** that a basic data type has to a variable of that type
14. For the object for which it was called, a const member function can modify neither const nor non-const member data.
15. If you declare a const object, it can only be used with const member functions.
16. In a class you can have more than one constructor with the same name.
17. Sending a message to an object is the same as calling one of its member functions.
18. The only technical difference between structures and classes in C++ is that member functions and data are, by default, public in structures but private in classes.
19. A string in C++ is an array of type character.
20. In a stack, the data item placed on the stack first is the last data item to be removed.
21. An array name, used in the source file, represents the memory address of the array.
22. Objects of the string class can be copied with the assignment operator and do not require memory management.
23. The extraction operator (>>) stops reading a string when it encounters a space.
24. Uses of Pointers 🡪
    * + Accessing array elements
      + Passing arguments to a function when the function needs to modify the original argument
      + Passing arrays and strings to functions
      + Obtaining memory from the system
      + Creating data structures such as linked lists
25. Every byte in the computer’s memory has an address. Addresses are numbers, just as they are for houses on a street. The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575. (Of course you have much more.)
26. The addresses appear in descending order because local variables are stored on the stack, which grows downward in memory. If we had used global variables, they would have ascending addresses, since global variables are stored on the heap, which grows upward. Again, you don’t need to worry too much about these considerations, since the compiler keeps track of the details for you.
27. When an asterisk is used in front of a variable name, as it is in the \*ptr expression, it is called the dereference operator (or sometimes the indirection operator). It means the value of the variable pointed to by.



1. Advantages of inheritance include facilitating class libraries, avoiding the rewriting of code and providing a useful conceptual framework.
2. The scope-resolution operator usually specifies a particular class and resolves ambiguities.
3. It is sometimes useful to specify a class from which no objects will ever be created.
4. Composition is a stronger form of aggregation.
5. A class hierarchy describes “is a kind of” relationships.
6. A class D can be derived from a class C, which is derived from a class B, which is derived from a class A.
7. Aggregation is a “has a” relationship.
8. In the UML, inheritance is called **generalization**.

**QUESTIONS AND ANSWERS**

**Q- What are the major issues facing the software industry today?**

A- Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession. This has forced the software engineers and industry to continuously look for new approaches to software design and development, and they are becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. These rapid advances appear to have created a situation of crisis within the industry. The following issued need to be addressed to face the crisis:

• How to represent real-life entities of problems in system design?

• How to design system with open interfaces?

• How to ensure reusability and extensibility of modules?

• How to develop modules that are tolerant of any changes in future?

• How to improve software productivity and decrease software cost?

• How to improve the quality of software?

• How to manage time schedules?

**Q- Describe how data are shared by functions in procedure-oriented programs?**

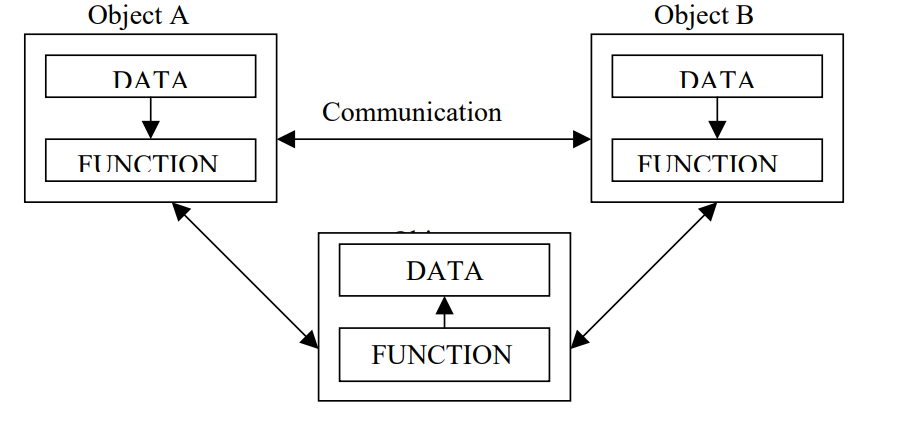
A- Procedure oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions. We normally use flowcharts to organize these actions and represent the flow of control from one action to another. In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

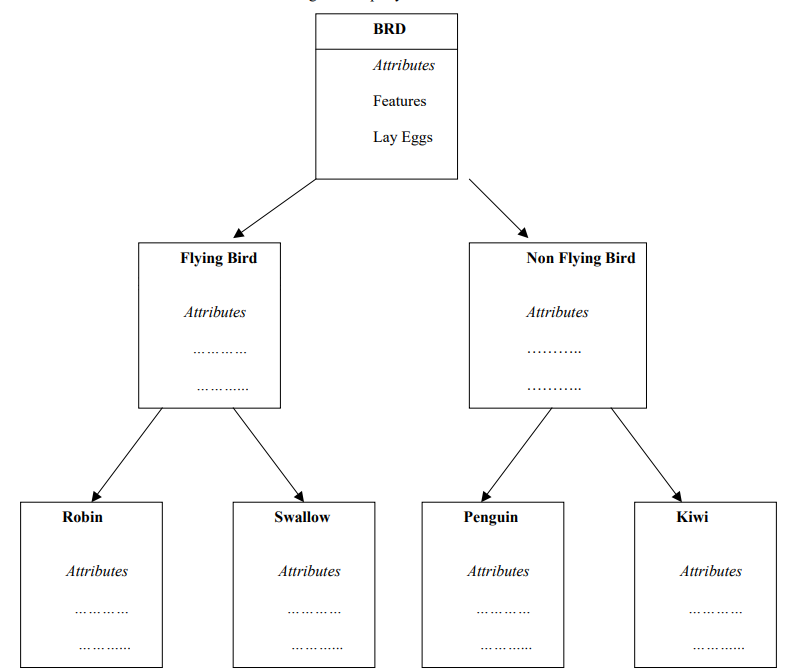
**Q- What are the difference between POP and OOP?**

| **Sr. No.** | **Key** | **OOP** | **POP** |
| --- | --- | --- | --- |
| 1 | Definition | OOP stands for Object Oriented Programing. | POP stands for Procedural Oriented Programming. |
| 2 | Approach | OOP follows bottom up approach. | POP follows top down approach. |
| 3 | Division | A program is divided to objects and their interactions. | A program is divided into funtions and they interacts. |
| 4 | Inheritance supported | Inheritance is supported. | Inheritance is not supported. |
| 5 | Access control | Access control is supported via access modifiers. | No access modifiers are supported. |
| 6 | Data Hiding | Encapsulation is used to hide data. | No data hiding present. Data is globally accessible. |
| 7 | Example | C++, Java | C, Pascal |

A-

**Q- How are data and functions organized in an object-oriented program?**

A- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.



**Q- Describe inheritance as applied to OOP.**

A- Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of hierarchical classification. For example, the bird, ‘robin’ is a part of class ‘flying bird’ which is again a part of the class ‘bird’. In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes. The real appeal and power of the inheritance mechanism is that it

**Q- What is the use of pre-processor directive #include?**

A- The header iostream contains the functions required for input/output data streaming like cout, cin, etc. As our programs grow larger or the functionality becomes complex, we might want to divide our program into various files or import functionality from the other files. In this case, we make use of user-defined files. To include user-defined files in our program we can make use of the following syntax of #include directive.

**Q- How does a main () function in c++ differ from main () in c?**

In *C* **main()**function *can be****called through other functions.***

while in*C++***main()** function ***cannot be called through other functions.***

1. In C++ you should always use **int main()**.
2. Using **void main()**results in an error for all compilers except ancient boreland compilers, and is not standard.

* It's also worth noting that in**C++**, int main() can be**left without a return value** at which point it *defaults to returning 0*. This is also true with a C99 program. Whether return 0 should be omitted or not is open to debate. The range of valid C program main signatures is much greater.
* Also, efficiency is not an issue with the main function. It can only be entered and left once (marking program start and termination) according to the C++ standard. For C, the case is different and re-entering main() is allowed, but should probably be avoided.

**Q- What do you mean by dynamic binding? How it is useful in OOP?**

A- Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

**Q- Distinguish between the following terms:**

**(a) Object and classes (b) Data abstraction and data encapsulation (c) Inheritance and polymorphism (d) Dynamic binding and message passing**

**Q – What are the key concepts of OOP?**

A – Objects and Classes

**Q-** **Why OOP?**

A - Object-oriented programming was developed because limitations were discovered in earlier approaches to programming. C, Pascal, FORTRAN, and similar languages are procedural languages. That is, each statement in the language tells the computer to do something: Get some input, add these numbers, divide by six, display that output. A program in a procedural language is a list of instructions.

For very small programs, no other organizing principle (often called a paradigm) is needed. The programmer creates the list of instructions, and the computer carries them out.

As programs grow ever larger and more complex, even the structured programming approach begins to show signs of strain. You may have heard about, or been involved in, horror stories of program development. The project is too complex, the schedule slips, more programmers are added, complexity increases, costs skyrocket, the schedule slips further, and disaster ensues.

In a large program, there are many functions and many global data items. The problem with the procedural paradigm is that this leads to an even larger number of potential connections between functions and data

**Q – What is the fundamental idea behind OOP?**

A- The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object.

An object’s functions, called member functions in C++, typically provide the only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will access the data and return the value to you. You can’t access the data directly. The data is hidden, so it is safe from accidental alteration. Data and its functions are said to be encapsulated into a single entity. Data encapsulation and data hiding are key terms in the description of object-oriented languages.’

**Q – Why use comments?**

A - Use comments to explain to the person looking at the listing what you’re trying to do. The details are in the program statements themselves, so the comments should concentrate on the big picture, clarifying your reasons for using a certain statement or group of statements.

**Q – What are identifiers?**

A - The names given to variables (and other program features) are called identifiers. You can use upper- and lowercase letters, and the digits from 1 to 9. You can also use the underscore (\_). The first character must be a letter or underscore. Identifiers can be as long as you like, but most compilers will only recognize the first few hundred characters. The compiler distinguishes between upper- and lowercase letters, so Var is not the same as var or VAR. You can’t use a C++ keyword as a variable name. A keyword is a predefined word with a special meaning. int, class, if, and while are examples of keywords.

**Q – What are integer constants?**

A - The number 20 is an integer constant. Constants don’t change during the course of the program. An integer constant consists of numerical digits. There must be no decimal point in an integer constant, and it must lie within the range of integers.

**Q – What is Cascading >> ?**

**Q – What are expressions?**

A - Any arrangement of variables, constants, and operators that specifies a computation is called an expression. Thus, alpha+12 and (alpha-37)\*beta/2 are expressions. When the computations specified in the expression are performed, the result is usually a value. Thus if alpha is 7, the first expression shown has the value 19. Parts of expressions may also be expressions. In the second example, alpha-37 and beta/2 are expressions. Even single variables and constants, like alpha and 37, are considered to be expressions. Note that expressions aren’t the same as statements. Statements tell the compiler to do something and terminate with a semicolon, while expressions specify a computation. There can be several expressions in a statement.

**Q – Difference between double and long double?**

A - The larger floating point types, double and long double, are similar to float except that they require more memory space and provide a wider range of values and more precision. Type double requires 8 bytes of storage and handles numbers in the range from 1.7x10–308 to 1.7x10308 with a precision of 15 digits. Type long double is compiler-dependent but is often the same as double.

**Q – What is a const Qualifier?**

A - Besides demonstrating variables of type float, the CIRCAREA example also introduces the qualifier const. It’s used in the statement const float PI = 3.14159F; //type const float The keyword const (for constant) precedes the data type of a variable. It specifies that the value of a variable will not change throughout the program. Any attempt to alter the value of a variable defined with this qualifier will elicit an error message from the compiler. The qualifier const ensures that your program does not inadvertently alter a variable that you intended to be a constant, such as the value of PI in CIRCAREA. It also reminds anyone reading the listing that the variable is not intended to change. The const modifier can apply to other entities besides simple variables

**Q – What are casts?**

**Q- What are arithmetic Assignment Operators?**

A - There are arithmetic assignment operators corresponding to all the arithmetic operations: +=, -=, \*=, /=, and %= (and some other operators as well)

**Q – What are arguments?**

A - An argument is the input to the function; it is placed inside the parentheses following the function name. The function then processes the argument and returns a value; this is the output from the function.

**Q – What are the differences between header files and library files?**

**Q - Write a statement that displays the variable Arjun in a field ten characters wide.**

A - Cout<<setw(10)<<ARJUN;

**Q – What are Header Files used for?**

A - Header files are used for declaration, data for library, objects and over loaded operators.

**Q – Throw some light on “default” keyword.**

**Q – What are structures?**

A - A structure is a collection of simple variables. The variables in a structure can be of different types: Some can be int, some can be float, and so on. (This is unlike array in which all the variables must be the same type.) The data items in a structure are called the members of the structure. structures are one of the two important building blocks in the understanding of objects and classes. In fact, the syntax of a structure is almost identical to that of a class. A structure (as typically used) is a collection of data, while a class is a collection of both data and functions.

The structure definition tells how the structure is organized: It specifies what members the structure will have. (Give Syntax of Structure)(Give example of Structure)

Accessing Structure Members 🡪 Once a structure variable has been defined, its members can be accessed using something called the dot operator.

**Q - Write a statement that declares an enumeration called speeds with the enumerators obsolete, single, and album. Give these three names the integer values 78, 45, and 33.**

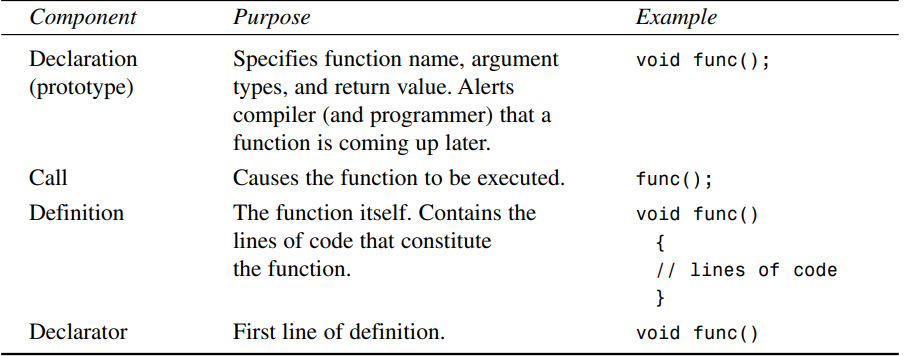
A - enum speeds { obsolete=78, single=45, album=33 };

**Q – What and why FUNCTIONS?**

A – (Include Definition of Functions)

Functions provide a couple of benefits: Functions allow the same piece of code to run multiple times. Functions break long programs up into smaller components. Functions can be shared and used by other programmers.

3 components of Functions 🡪 The function declaration, the calls to the function, and the function definition



**Q – Difference between Library Functions and User-Defined Functions?**

|  |  |
| --- | --- |
| Library / built-in functions | User-defined functions |
| These functions are predefined in a header file or preprocessor directive. | These functions are not predefined rather it is defined by user according to the requirements. |
| These functions can be simply used by including respective header file. | These functions should be declared, defined and called in order to use. |
| Since, these functions are predefined programs are short. | Since, these functions should be defined by the user programs are lengthy. |
| Program execution time is faster. | Program execution time is slower. |
| It simplifies the program. | Using more user-functions increase complexity. |
| Eg, strlen( ), strcmp( ), strcpy( ), strcat( ) | Eg, fact( ), average( ), greatest( ), anyname( ) |

**Q – What is Storage Class?**

A - A local variable is not created until the function in which it is defined is called. (More accurately, we can say that variables defined within any block of code are not created until the block is executed. Thus variables defined within a loop body only exist while the loop is executing.) In the program fragment just given, the variables somevar and othervar don’t exist until the somefunc() function is called. That is, there is no place in memory where their values can be stored; they are undefined. When control is transferred to somefunc(), the variables are created and memory space is set aside for them. Later, when somefunc() returns and control is passed back to the calling program, the variables are destroyed and their values are lost. The name automatic is used because the variables are automatically created when a function is called and automatically destroyed when it returns. The time period between the creation and destruction of a variable is called its lifetime (or sometimes its duration). The lifetime of a local variable coincides with the time when the function in which it is defined is executing. The idea behind limiting the lifetime of variables is to save memory space. If a function is not executing, the variables it uses during execution are presumably not needed. Removing them frees up memory that can then be used by other functions.

**Q - What is the purpose of using argument names in a function declaration?**

A - To clarify the purpose of the arguments

**Q - Where is a function’s return type specified?**

A - At the beginning of the declaration and declarator

**Q - What is the purpose of a class definition?**

A - A class declaration describes how objects of a class will look when they are created.

**Q - Write a declaration (not a definition) for a const void function called aFunc() that takes one const argument called jerry of type float.**

A - void aFunc(const float jerry) const;

**Q - If three objects of a class are defined, how many copies of that class’s data items are stored in memory? How many copies of its member functions**

A - three, one