2075-chaitra

Chapter-1.
Introduction to Object Oriented Programming

1.1 Issues with procedure Oriented Programming (structured programming)

In procedure-oriented approach, the complete problem is viewed as a sequence of tasks such as reading input from the user, doing necessary calculations and displaying output.
In procedure oriented programming, complete task is broken down into smaller manageable parts called procedures, subroutines or functions.

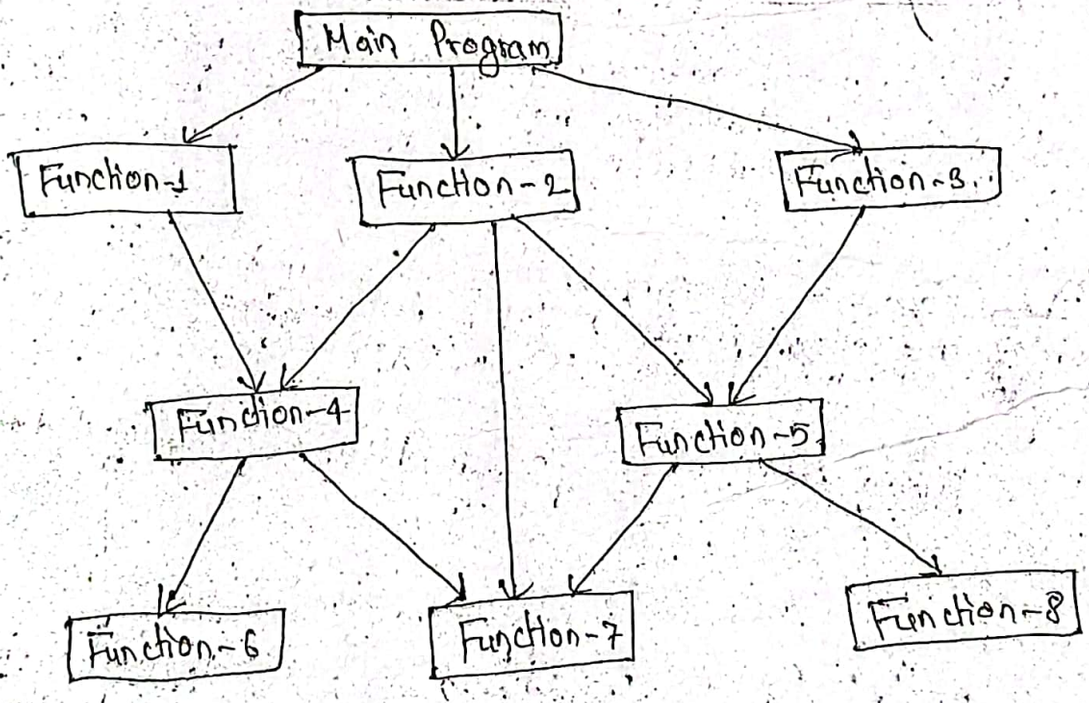


Fig: Typical structure of POP.

While we concentrate on the development of functions very little focus is given to the data that are being used by various functions.

Types of data used in POP:

1. Global data.
2. Local data.

① Global data: Data which are declared globally, outside the functions are called global data.

② local data: Data which are declared inside functions are called local data. These data are accessible to the only function where it is declared.

In a POP program, each function may access its local data as well as global data. The local data of particular function cannot be accessed by other functions in a program. If any data is to be accessed by two or more functions it should be made global.

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Due to this, important data can be changed unknowingly.

Also,
The separate arrangement of data and functions does a poor job of modeling things in real world. That's why POP approach does not model real world system perfectly.

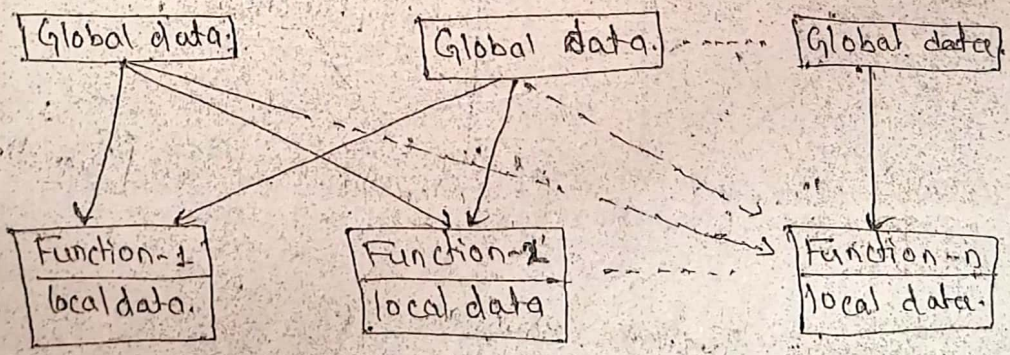


Fig: Organization of data and function in OOP.

Limitations of POP.

① Focus on functions rather than data.
② The use of global data is error prone and it could be an obstacle in code maintenance and enhancements.
③ It is difficult to hide information to unauthorized user.
④ In a large program, it is difficult to identify belongings of global data.

⑤. In a large program it is very difficult to identify what data is used by which function. So datas can be changed unknowningly.

1.2 Basics of Object Oriented Programming (OOP).

The errors faced in the procedure oriented programming approach are the motivating factor in the development of object oriented approach. In OOP, data are treated as a critical element in the program and restricts freely transformation of data around the system.

OOP approach permits decomposition of a problem into entities called objects.

Data of an object are accessible only by the function belonging with the object. But function of one object may access the function of another object.

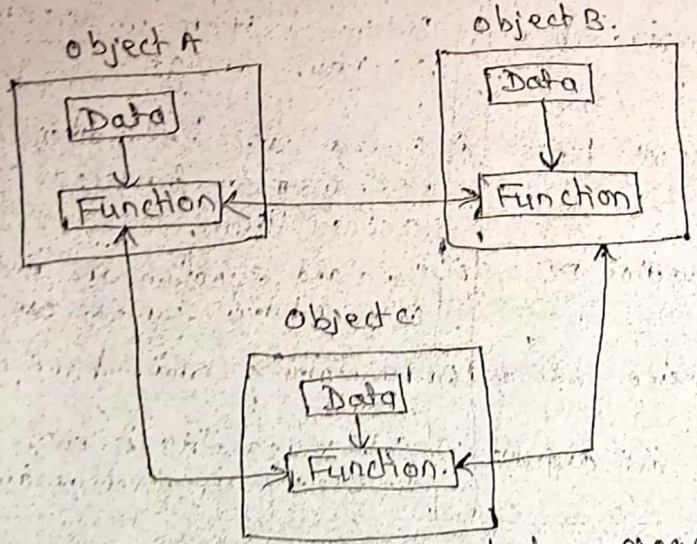


Fig: Object Oriented approach.

In other words, OOP is a method of implementation in which programs are organized as co-operative collection of objects each of which represents an instance of some class.

Characteristics of OOP.

① Emphasis is on data rather than procedures.
② Programs are divided into objects.
③ Functions and data are tied together in a single unit.
④ Data can be hidden to prevent from accidental modificatio

(5) Objects may communicate with each other through functions.
(6) It supports reusability of code. Also supports inheritance polymorphism, data abstraction and encapsulation.
(7) Follows bottom-up approach in program design.
(8) Functions and datas are tied together in a single unit.

3 Procedure Oriented Versus Object Oriented Programming

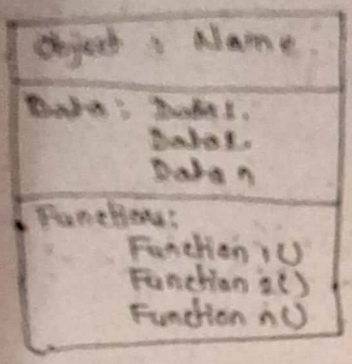
	Procedure Oriented Programming	Object Oriented Programming
1)	Emphasis is given to procedures	Emphasis is given on data.
2)	Programs are divided into functions.	Programs are divided into objects.
3)	Follow top-down approach of program design.	Follow bottom-up approach of program design.
4)	Generally data cannot be hidden.	Data can be hidden, so that non-member function cannot access them.
5)	Data move from function to function.	Data and function are tied together. Only related function can access them.
6)	Maintaining and enhancing code is still difficult.	Maintaining and enhancing code is easy.
7)	Code reusability is still difficult.	Code reusability is easy in compare to procedure oriented approach.

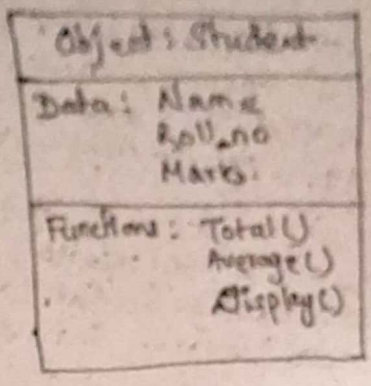
4 Concept of Object Oriented Programming

Features of OOP

Objects.

Objects are defined as the physical instance of a class. An object can have attributes (properties) and behaviors. When the object is created, space for that object is allocated in primary memory. [Note: Objects are variables of type class]
Eg: Televisions are objects as they have size, weight, color etc as attributes (i.e data) and turning on the tv, turning off tv, increasing volume as operation or function.





classes:

Classes are templates /model/ blue-print that specifies data and their operations.

Once a class is defined we can create any number of object of its type. Each object are associated with the data of type class with which they are created. For eg: Project manager, developer, QA etc. are the object of class employee. Similarly orange, apple and mango are the objects of class fruits.

classes are user defined data types and behave like the built in types of programming language.

Ex:

```
class Student
{
char name[30];
int reg_no;
int marks[7];

tot_marks();
percentage_marks();
};
Student ram;
```

// this will create object ram belonging to the class Student.

Abstraction: Abstraction feature of OOP hides the internal details of how any object does its works. It only provides the interface to use the service that the object provide.
We can manage complexity through abstraction.

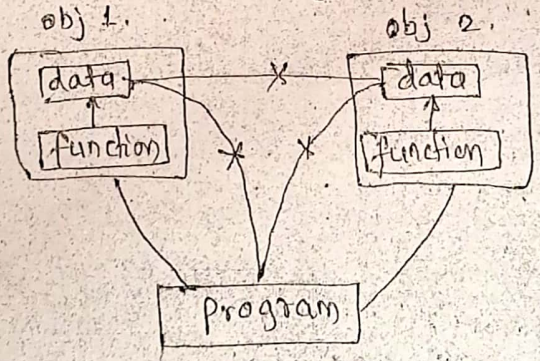
For eg: a class 'car' would be made up of an Engine, Gearbox, steering objects. The abstraction allows the driver of the car to drive without having detail knowledge of the complexity of the parts. The driver can drive the whole car treating like a single object.

Encapsulation:

The wrapping up of data and function into a single unit is called encapsulation.
Data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.

Data hiding:

The data and function in OOP are always together within an object that provides data hiding. Thus, the insulation of data from direct access by the program is called data hiding or information hiding.



Inheritance.

Inheritance is the process of creating new classes based on the existing class. The new classes acquires features from the existing class and adds more features in it. The existing class is called 'base class' and newely created class is called derived class.

The derived class inherits all the features except the private members of base class. Thus, inheritance supports program reusability.

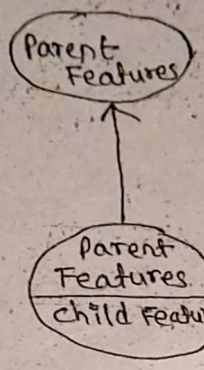


Fig: Inheritance.

Polymorphism :

The word polymorphism is derived from two Greek words 'poly' and 'morphe'. The word poly means many and morphe means forms. Thus, polymorphism means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation.

Example: operator overloading
Function overloading
Dynamic binding.

1.5. Example of some Object Oriented languages.

(a) Java.
(b) C++
(c) C#
(d) Smalltalk
(e) Eiffel.

1.6.1 Advantages of OOP

(1) Redundant code is eliminated by various techniques like inheritance and templates.
(2) Through data hiding, programmer can build secure programs.

The derived class inherits all the features except the private members of base class. Thus, inheritance supports program reusability.

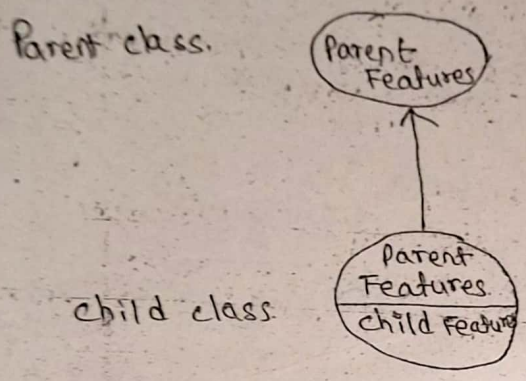


Fig: Inheritance.

Polymorphism :

The word polymorphism is derived from two Greek words 'poly' and 'morphe'. The word poly means many and morphe means forms. Thus, polymorphism means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation.

Example: operator overloading
Function overloading
Dynamic binding.

1.5 Example of some Object Oriented languages.

(a) Java.
(b) C++
(c) C#
(d) Smalltalk
(e) Eiffel.

1.6.1 Advantages of OOP

(1) Redundant code is eliminated by various techniques like inheritance and templates.
(2) Through data hiding, programmer can build secure programs.

(c) Existing classes can serve as library class for further enhancements.
(d) Division of program into objects makes software development easy.
(e) Code reusability is much easier.
(f) Message passing techniques makes communication easier among various objects.
Function
parameter ⇒ information
(g) Upgrading and maintenance of software is easily manageable.
(h) Models real world system perfectly.
(i) Software complexity can be managed easily.
(j) Data can be hidden from outside world using encapsulation & data hiding.
(k) User can create/define new data type or user-defined type by making class.

Disadvantages of OOP.

1. Compiler and runtime overhead is high because object oriented program requires more time during compilation.
2. For dynamic and runtime support it requires more resource and processing time.
3. The message passing between many objects of complex application may be difficult to trace and debug.
4. Benefits only in long run while managing large software projects.
5. Requires the mastery in software engineering and programming methodology.

Chapter-2
Introduction to c++

2.1 The Need of c++:

The software needed by big organization like banking, insurance, civil aviation military etc are more complex problems. So, in order to solve the complex problems and to model the real world perfectly, c++ was developed.

Features of c++:

1) Namespace.
2) classes.
3) Derived classes.
4) Access controllers
5) Constructor and destructor.
6) Friend function and classes.
7) Reference variable
8) Default arguments.
9) Function overloading.
10) Operator overloading.
11) Inline function.
12) Run time polymorphism
13) Generic programming.
14) Exception handling.
15) Template
16) Run time type information (RTTI)

C		c++
This language was introduced in 1972 by Dennis Ritchie at Bell Lab.	1)	This language was introduced in 1985 by Bjarne Strowstup at Bell.
It has been implemented by gcc, Borland C, Microsoft and many more.	2)	C++ has been implemented by GCC, MS visual c++, c++ Builder.
This language is influenced by B, BCPL, ALGOL 68 etc.	3)	This language is influenced by C, Simula 67, Ada 83 etc.
Languages such as c++, perl, php, javascript are influenced by C.	4)	Language such as Java, C# are influenced by c++.

5.	It follows procedural approach of program development.	5.	It follows object oriented approach of program development.
6.	C applications are faster to compile than c++ applications.	6.	c++ applications are comparatively slower to compile than C.
7.	c has comparatively weaker type checking than c++.	7.	c++ provide stronger type checking than c.
8.	C does not support extension in programming.	8.	c++ supports extension in programming like operator overloading, inheritance etc.
9.	C has comparatively lesser implementation than c++.	9.	c++ has higher implementation in area like GUI and graphics as c++ supports robust and powerful GUI.
10.	C has fewer libraries than c++.	10.	c++ has improved and extended form of libraries that of C.
11.	c has fewer keywords than c++	11.	c++ has more keywords than c.

2.4 History of C++.

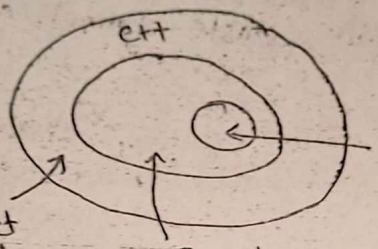
C++ Author – Bjarne Stroustrop. (Software engineer at AT & Bell lab.)
Launch on → 1985

Originally called "C with classes"
↑ simula.67

"superset of C, extension of C, advanced C"

C++ → class.
1 increment

Chapter-3
C++ language Constructs.



Features that are not used in C++.

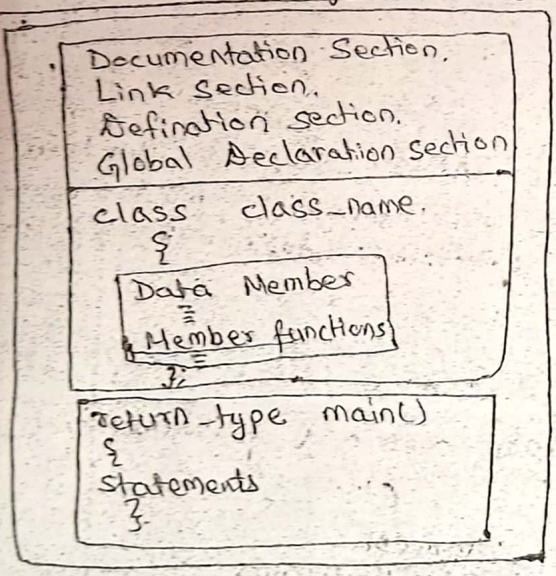
{ double sin(double x);
  float sinf(float x)

In C++
double sin(double x);
float sin(float x);

Other in C:
~~Goto~~
~~Call by ref.~~
~~& unitia reference~~
Compound literals;
Designated initializers

∴ Fig: Relation of features in C and C++.

Basic structure of C++ Program.



class defination section.

[Documentation section]
Header files
[Global variables]
[class declaration & definition]
[function declaration & definition]
main()

main() function section.

Our first C++ Program.

```
//C++ program to display "Hello world"          } documentation
// This is our first programme.
#include <iostream.h>                            } link section.
using namespace std;
int main()
  {
  cout<<"Hello world";                           } Main function section.
  return 0;
}.
```

Lexical elements of C++.

Comments in C++

// (double slash) is used to for comments in C++.

Comments start with a double slash symbol and terminate at the end of the line.

Multiline comment

/* ..... */

#include "filename"
#include "c:\tc\bin\myfile"
//filename

The line #include <iostream.h> causes the compiler to include file iostream in the program.

The file iostream is a system supplied file which has definitions required to use stream input or output. It contains declarations for the identifier 'cout' with operator << and 'cin' with operator >>.

Using namespace std:

American national standards institute
International organization for standardization

Std is the namespace where ANSI/ISO C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global space.

Comparision programme between C & C++.

Write a programme to calculate area and perimeter of circle.

C

```
/* area.c */
#include <stdio.h>
#define pi 3.14159
int main(void)
{
float r,a,c;
printf("Enter radius : ");
scanf("%d",&r)
a= pi *r *r;
c= 2 * pi * r;
printf("\n Area of Circle=%f",a);
printf("\n Circumference =%f",c);
return 0;
}
```

C++

```
//area.cpp
#include <iostream>
#define pi 3.14159
using namespace 3.14159
int main()
{
float r,a,c;
cout<<" Enter radius of circle";
cin>>r;
a= pi *r *r;
c = 2* pi *r;
cout<<"\n area of circle="<<a;
cout<<"\n perimeter of circle="<<c;
return 0;
}
```

Instead of printf() function, c++ uses 'cout' object to display output on the screen. Similarly, instead of scanf() function, c++ uses 'cin' object to read input from the keyboard.

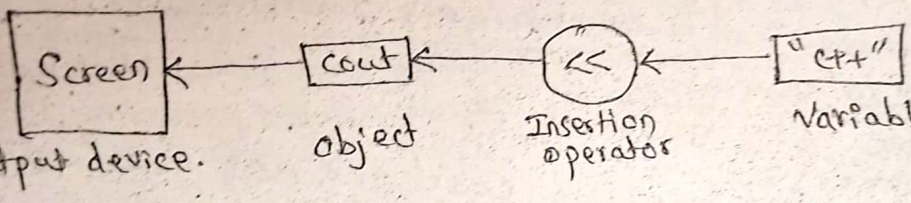
Output operator.

```
cout << " I love Nepal";
```

This statement causes the string in quotation marks to be displayed on the screen.

The identifier cout (pronounced as 'c out') is predefined object that represents the standard output stream in c++.

The operator << is called the 'insertion' or 'put to' operator. It inserts (or sends) the contents of the variable on its right to the object on its left.



Input operator.

```
Cin >> input1;
```

This statement causes the program to wait for the user to provide input from keyboard.

The identifier cin (pronounced as 'c in') is a predefined object in c++ that corresponds to the standard input stream.

The operator >> is known as 'extraction' or 'get from' operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right.

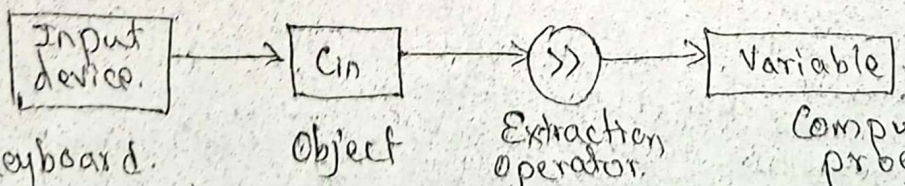


Fig: Input using extraction operator.

Q: WAP to calculate sum & average of two numbers provided by the user and display on the screen.

```
#include <iostream.h>
using namespace std;
int main()
    {
    float number1, number2 sum average;
    Cout << "Enter two numbers: ";
    Cin >> number 1;
    cin >> number 2;
    sum = number1 + number 2;
    average = sum/2;
    Cout << "sum =" << sum << "\n";
    cout << "Average =" << average << "\n";
    return 0;
    }
```

Q. calculate area of rectangle.
Q. calculate simple interest

d. Cascading of I/O operators:

In above example, the statement

```
Cin >> number1;
Cin >> number 2;
```

can be cascaded as;

```
Cin >> number1 >> number 2;
```

The values are assigned from left to right.

The statement,

```
Cout << "Sum =" << sum << "\n";
Cout << "Average =" << average << "\n";
```

can be cascaded as;

```
Cout << "Sum =" << sum << "\n" << "Average =" <<
average << "\n";
```

Character Set and Tokens:

Character set:

A character set denotes any alphabet, digit or special symbol used to represent information. Compiler collects these character sets to make sensible tokens.

Alphabets :- Uppercase (i.e A, B, ... Y, Z) or
Lowercase (i.e a, b, ... y, z)

Digits :- 0, 1, 2 ..... 8, 9

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

white space characters :- blank, new line, tab etc.

whitespace characters are used to separate tokens.

Tokens:

Tokens are valid sets (collection) of different characters symbols, operator or punctuators. Compiler collects the valid character-sets to make sensible tokens. Thus, tokens are the smallest meaningful individual units in program.

Different types of tokens used on identifiers are:-

a) Keywords
b) Identifiers
c) Constants
d) Operators
e) Punctuators.

Keywords:

Keywords are predefined or reserved words for. All keywords have fixed meaning and these meaning can't be changed. Keywords used on c++ are:

auto	double	new	switch
break	else	operator	template
asm	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

Identifiers:

The name given to variables, functions, arrays, classes etc. given by the programmers is called identifiers. These are tokens which are sequence of letters, digits and underscore (_). Identifiers are used to give unique names to the element in the program. c++ does not provide any restriction on number of characters in

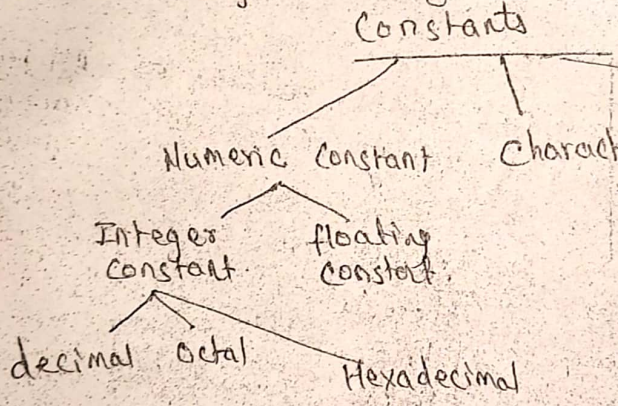
the identifier like in C (32 character) but it depends on implementation.

Rules to define a identifiers.

(a) Only alphabetic characters, digits and underscore are permitted.
(b) First character should be alphabet or underscore.
(c) Underscore is permitted between two words, but white space is not allowed.
(d) keywords cannot be used as identifiers.

Constants:

Constants also called 'literals' are the fixed values that cannot be changed during execution of the program.



Different ways of defining constant.

(1) Defined constant (using define keyword)
```
#define constant_name value.
#define PI 3.14
```

(2) Declared constant (const)
```
const int size = 30;
const float salary = 3.14;
```

Operators:

Operators are special symbols which are used for arithmetic or logical operation.
&&, ||, !

Types:

(i) Arithmetic operator (+, −, *, /, %)
(ii) Assignment (=)
(iii) Self assignment (++, −−)
(iv) Arithmetic assignment (+=, −=, *=, /=, %=)
(v) Relational operator (>, <, <=, >=)
(vi) Equality operator (==, !=)
(vii) Logical operator (&&, ||, !)
(viii) Bitwise operator (>>, <<, ~, ^, |)
(x) Bitwise assignment operator (>>=, <<=, &=, ^=, |=)
(x) Other operators (?:, ,, (), [], ->, ...
sizeof(), &, * (deference))

++a or a++ ⇒ a = a+1
--a or a-- ⇒ a = a-1

Eg: b = ++a;
b = a++;

In the first case the value of a is increased by 1, and increased value is assigned to b.
In second case the previous value of a is assigned to b and value of a is increased by 1.

new
delete
endl
ends
:: → scope resolution operator
*. } pointer to member
*:: } access operator.

Punctuators:

(), {}, ,, ;

Variable declaration and Expression.

Variables are the containers used to store certain values.
Each and every variable must be declared before its use.
In C++ variables can be declared anywhere but varables declared within the blocks are invisible outside that block.
Values stored on a variables can be changed during execution.

Syntax:

data_type variable_name1, variable_2, ... variable_n;

Eg:

```
int length, breadth, height;
float area;
char section;
```

After variable declaration we can assign values later.

Eg:
```
length = 15;
breadth = 8;
height = 15.2;
```

Also variables can be initialized during declaration.

```
int length = 15;
int breadth = 8;
float height = 15.2;
char section = 'A';
```

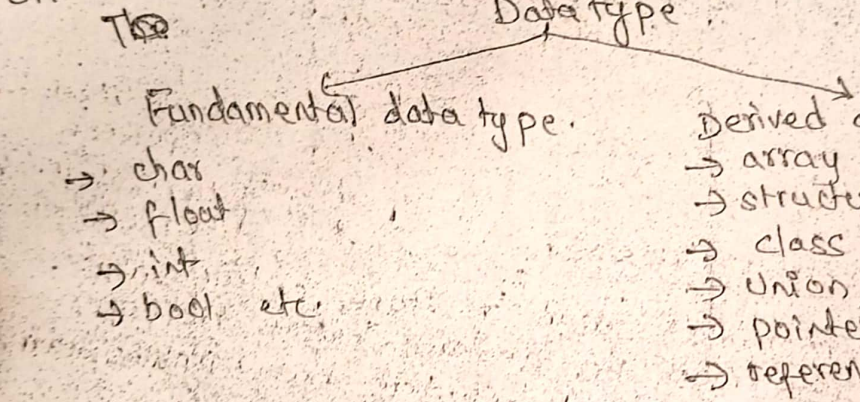
Expressions

Expressions are the combination of variables, constants, function calls, operators and punctuators.

Eg: x = p+q;
i++;
15 % 5;
(a+b) / (p+q);
sqrt(16);

Data types:

Data types define what type of values that a variable can store along with a set of operations that can be performed on that variable.

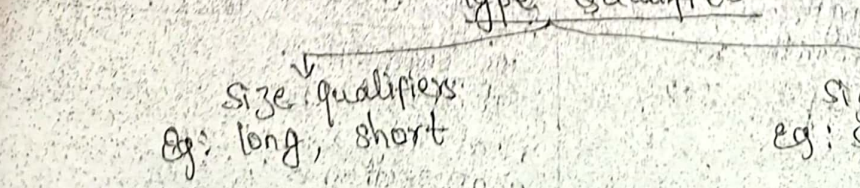


Data type and memory management.

Data type	Memory requirement
char	1 byte.
int	2 bytes in 16 bit system 4 bytes in 32 bit system
float	4 bytes
double	8 bytes.

Type Qualifiers.

They are the keywords which are used to modify or extend data types.



Data types in c++ can be used in their long form and short form.

long form	short form	Memory Used (bytes)
char	char	1
signed char (-128 to +127)	signed char	1
unsigned char (0 to 255)	unsigned char	1
signed int	int	2 in 16 bit system
unsigned int	unsigned	4 in 32 bit system
signed short int	short	2
short int	short	
unsigned short int	unsigned short	2
signed long int	long	4
signed long	long	
unsigned long int	unsigned long	4
float	float	4
double	double	8
long double	long double	10

Type Conversions:

The process of converting one basic type into another is called type conversion (data conversion).
There are two types of data conversions. They are:-
(i) Automatic (implicit) type conversion.
(ii) Type cast (explicit) type conversion.

(i) Automatic / implicit type conversion.

Conversions done by compiler itself is called automatic / implicit conversions.
Implicit conversion is also known as promotion or widening or arithmetic.

Eg;
```
float result;
int a=7;
float b = 2.02;
result = a+b;
```
= 9.02

Rule for automatic type conversion.

lower type
char
short
int
long
float
double
long double
Higher type.

2. Explicit type conversion (Typecasting)

Conversions of one data type into another performed explicitely by a programmer as per need is called explicit conversion.

Syntax:

```
type_name (expression);
```

to

or

```
(type_name) expression;
```

For eg:

```
int x;
(float) x;
```

Result = 3.0

or (By using static_cast keyword).

Syntax:

```
static_cast <data_type> variable
```

Eg:

```
float result;
float a=5.2; int a = 7, b = 9;
cout << a;      result = a/b;
cout << static_ cout << result;;   [result = 0, without casting]
  cast<int>(a); // After_casting;
    result = static_cast <float> a/b;
    cout << result;
```

Preprocessor Directives

⇒ The lines beginning with (#) sign are called preprocessor directives. They are not program statements as they are not terminated by a semicolon.

Eg: #include
#define

```
#include<iostream>
// this includes preprocessor to insert a file iostream in our program.
Our program looks this file in include directory not in our current working directory.
#include "filename"
#include "C:\\tc\\bin\\ mydir\\ filename"
```

Control structure.

The structures those regulate the order in which program statements are executed are called control structures.

Types:

(a) Sequential structure
(b) Selective structure
(c) Repetitive structure

(a) Sequential structure:

It consist of a sequence of program statements that are executed one after another in order.

Eg:

```
{
int val1 = 5;
int val2 = 55;
cout << val1 << endl << val2 << endl;
{
    int val2 = 555; val3 = 5555;   → These variables are local to internal braces.
    cout << val1 << endl;   //5 is displayed.
    cout << val2 << endl;   // 555
    cout << val3 << endl;   // 5555
}.
cout << val1 << endl;   //5
cout << val2 << endl;   //55
cout << val3 << endl;   //undefined. so error.
}.
```

(b) Selective structure:

Selective statements are also called conditional branching statements. It alters the execution of normal executions.

Types:

(a) If statement.
(b) If-else statement.
(c) switch statement.
(d) ?: (conditional operator).

(a). **if statement :**

Syntax:

```
if (test_expression)
    statement;
```

Eg:

```
#include <iostream.h>
using namespace std;
int main()
{
int marks;
cout << "Enter your marks :";
cin >> marks;
if (marks >= 40)
    cout << "You are passed";
  return 0;
}
```

(b) **if-else statement :**

Syntax:

```
if (test_expression)
  statement1;
else
  statement2;
```

Eg:

```
#include <iostream.h>
using namespace std;
int main()
{
int marks;
cout << "Enter your Marks";
cin >> marks;
if (marks >= 40)
  cout << "You passed the exam";
else
  cout << "You failed the exam";
return 0;
}
```

(c) Nested if-else statement:

Syntax:

```
if (exp1)
    if (exp2)
        statement 1;
    else
        statement 2;
else
    if (exp3)
        statement 3;
    else
        statement 4;
```

Eg:

```
#include <iostream>
using namespace std;
int main()
{
  int num1, num2, num3;
  cout << "Enter three integer numbers:";
  cin >> num1 >> num2 >> num3;
  if (num1 > num2)
      if (num1 > num3)
          cout << num1 << " is largest ";
      else
          cout << num3 << "is largest";
  else
      if (num2 > num3)
          cout << num2 << " is largest";
      else
          cout << num3 << "is largest";
  return 0;
}
```

(d) Multiway conditional statement:

Syntax:

```
if (expr_1)
    statement_1;
else if (exp_2)
    statement_2;
else if (exp_3)
    statement_3;
    .
    .
else if (exp_n)
    statement_n;
else
    default_statement;
```

Eg1.

```
#include <iostream>
using namespace std;
int main()
{
int marks;
cout << "enter marks :";
cin >> marks;
if (marks >= 80)
    cout << " Grade: Distinction";
else if (marks >= 60)
    cout << "Grade: First Division";
else if (marks >= 50)
    cout << "Grade: Second Division";
else if (marks >= 40)
    cout << "Grade: Third Division";
else
    cout << " Grade : Fail";
return 0;
}
```

Switch statement:

```
switch (expr)
  {
  case value 1:
       statement 1;
       break;
  case value 2:
       statement 2;
       break;
    :
    case value_n:
       statement n;
       break;
  default:
       default statement
}
```

Eg:

```
#include <iostream>
using namespace std;
int main()
{
int fruit;
cout<<" Enter the fruit code: ";
cin>>fruit;
cout<<" The fruit is ";
switch (fruit)
{
case 1:
    cout<<"Orange";
    break;
case 2:
    cout<<"Apple";
    break;
case 3:
    cout<<"Grape";
    break;
case 4:
    cout<<"Banana";
    break;
default:
    cout<<"Fruit not in the list";
}
return 0;
}
```

Conditional operator.

expr 1 ? expr 2 : expr 3;

Eg:

```
c = a>b ? a : b;
```

⟹

Equivalent code.

```
if(a>b)
    return a;
else
    return b;
```

Repetitive Structure:

(i) while loop.
(ii) do-while loop.
(iii) for-loop.

(i) while loop:

```
while (test_expression)
    statement;
```

Eg:

```
#include <iostream>
using namespace std;
int main()
{
int i;
i=1;
while (i<=40)
{
cout << "\t" <<i;
    i++;
}
return 0;
}
```

(ii) do-while loop:

The loop body is executed at least once even though the condition to be tested is false because the test expression is at bottom of the loop.

```
do
{
  statement
} while (test expression);
```

Eg:
```
#include <iostream.h>
using namespace std;
int main(void)
{
 int i=1;
 do
 {
  cout << "\t" << i ;
  i++;
 }
 while (i<=40);
 return 0;
}
```

(ii) for-loop:

Syntax: for (initialization statement; test exp^{n}; increment expression)

Eg:
```
#include <iostream>
using namespace std;
int main()
{
 int i;
 for (i=1; i<=40; i++)
  cout << "\t" << i;
 return 0;
}
```

Eg:
```
#include <iostream>
using namespace std;
int main()
{
 int i, sum;
 for (sum=0, i=1; i<=40; sum+=i
                  i++) {
  cout << "sum = " << sum; }
 return 0;
}
```

Nested loop:
```
for ( )
{
    for ( )
    {
        statement;
    }
}
```

loop ~~Iteration~~ Interruption.

① break statement.
② continue statement.

① break statement:

```
for (i=0; i<5; i++)
  {
  if(i==2)
  break;
 Cout<<"\t"<<i;
}.
```

// Break the loop. (while or do-while or for).

② continue statement

```
for (i=0; i<5; i++)
  {
  if(i==2)
    continue;
  cout<<"\t"<<i;
  }.
```

//skips current iteration.

Namespace

The namespace mechanism is used for the logical grouping of variables, classes and functions in C++. Also, namespace is a container for variables, functions, classes and other identifiers that avoids conflicts residing in different scopes.

Syntax:

```
namespace namespace_name
{
 //declaration of variables, classes, functions etc.
} xNo semicolon at the end.
```

Eg:

```
#include <iostream.h>
namespace number
{
int num = 10;
}
int main()
{
int num = 20;
cout << "Iside main: num =" << num;
cout << "Num within namespace :" << number :: num;
return 0;
}
```

scope resolution operator.

Nested Namespace:

Syntax:

```
namespace Outer_namespace.
{
    namespace Inner_namespace.
    {

    }
}.
```

Eg:

```
#include <iostream.h>
namespace OuterNamespace
{ int num = 10;
  namespace InnerNamespace.
  {
    void display()
    {
    std :: cout << "The num is" << num;
    }
  }
}.
```

→ Using namespace —

```
int main()
{
OuterNamespace::
InnerNamespace::
    Display();
return 0;
```

Anonymous Namespace (Unnamed namespace).

⇒ We can define namespace without names. They occupy global scope.

Syntax:

```
namespace
{
.....
}.
```

For each unnamed namespace, the compiler generates a unique name, which differs from every other name in the program.

```
# ~~namespa~~ include <iostream.h>
namespace
{
int i = 5;
int m;
}
int main()
{
cout << "i = " << i;
m = 100;
cout << "m = " << m;
return 0;
}.
```

Accessing identifiers defined within Namespace.

(i) Use of using keyword.

(ii) Use of scope resolution operator.

⇒ We can use 'using' Keyword to access all members through 'using' keyword.

```
#include <iostream>
# using namespace std;
namespace First
{
float f = 5.5;
int i = 100;
```

```
int main()
  {
  using namespace First;
    cout<< "f =\t" << f <<"\n";
    cout<< "i =\t" <<i<< "\n";
    return 0;
  }
```

Note: ✱ To select only selected member.

```
int main()
  {
  using First::f;
  cout <<"f = " << f;
   return 0;
   }
```

✱ Using first::f
(can access only selected member, i.e, f).

User defined Constant const

We can define symbolic constants using the keyword keyword 'const'. This indicates that the value does not change during program execution. Since we cannot change the value of the constant we must initialized it.

Eg: const int max = 1000;
const float a = 4.5;
const int arr[] = {1,2,3,4};
After declaring constant they cannot be changed within the scope.

Manipulators:

Manipulators are special variables & functions used to modify the output as user's requirement. Some of the commonly used manipulators are endl, setw(), setfill(), setprecision() etc.

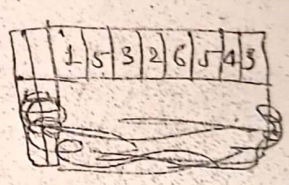
The headerfile <iomanip.h> must be included to use the standard manipulators.

✓ endl → Insert new line character.

✓ setw() → Used to set width

Eg:

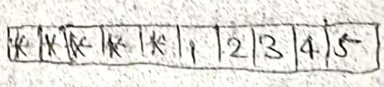
```
#include <iostream.h>
#include <iomanip.h>
int main()
{
long num = 1532654;
cout << setw (10) << num;
return 0;
}
```



✓ Setfill() → Used to fill blank fields set by setw()

Eg:

```
#include <iostream.h>
#include <iomanip.h>
Void main()
{
int num = 12345;
cout << setw(10) << setfill('*') << num;
return 0;
}
```



✓ setprecision() → It is used to specify the number of digits to be displayed after decimal point.

Eg:

```
void main()
{
float num = 50.1234567;
cout << " num = " << setprecision (3) << num << endl;
}
```

// 50.123

Dynamic memory allocation:

The process of allocating and deallocating memory at run-time is known as dynamic memory allocations.

C++ provides 'new' operator for dynamic memory allocation and 'delete' operator for dynamic memory de-allocation.

* **Note:-** The 'new' operator obtains memory at runtime from the memory heap from the operating system and returns the address of obtained memory.

* The memory reserved by 'new' in any scope remains as it is even after the program control goes out of scope.

Syntax:

```
data_type  * data_type_ptr;
data_type_ptr = new data_type;        //allocates single variable memory.
data_type_ptr = new data_type[size];  //allocates an array.
```

Eg:

```
int *P;
P=new int
```

```
int *P;
P= new int[10];
```

⇒ First element pointed by P can be accessed either with the expression P[0] or *P.

⇒ Second element can be accessed either with P[1] or *(P+1).

* Note: when the reserved memory is no longer needed, it should be freed so that the memory becomes available again for other requests of dynamic memory. We use 'delete' operator to free allocated memory.

Syntax:

```
delete data_type_ptr;      //release single dynamic variable.
delete [] data_type_ptr;   // release dynamically created array;
```

Eg:

```
int *p;
p=new int;
!
delete p;
```

```
int *p
p=new int[5];
!
!
delete [] p;
```

Q. WAP to find the sum and average of the number by using new and delete operator.

```
#include <iostream.h>
int main()
{
int n, i, *p, tot=0;
float avg;
cout<<"How many numbers? ";
cin>>n;
p=new int[n];    // allocate memory for array.
cout<<"Enter elements";
for(i=0; i<n; i++)
   cin>>p[i];
for(i=0; i<n, i++)
   tot=tot+p[i];
avg=static_cast<float>(tot)/n;
cout<<"Total = "<<tot<<endl;
cout<<"Average = "<<avg;
delete []p;
return 0;
}
```

Functions:

A function is a self-contained subprogram that is meant to do some specific, well-defined task.

Types:

1. Built-in functions.
2. User-defined functions.

User-defined functions:

Users can create their own functions for performing any specific task of the program. These types of functions are called user-defined functions.

Function contains 3 parts:-

(i) Function defination.
(ii) Function declaration.
(iii) Function call.

```
#include <iostream.h>
Void find(int n);                    // Function declaration.
main()
{
 int num;
 cout << " Enter a number";
 cin >> num;
 find(num);                          // Function call.
}
Void find(int n)
{ if (n%2 ==0)
    Cout << " It is ~~prime~~ odd";   // function definition
  else
    Cout << " It is even";
}
```

~~Function overloading~~
~~It is convenient to give them same name~~

Function overloading.

Some functions conceptually perform the same task on objects of different types and numbers. In such case it is convenient to give them same name.

When the same name is used for different operation it is called function overloading.

When an overloaded function is called, the function with matching arguments is invoked.

Example:

```
Void display ();
Void display (int);
Void display (float);
Void display (int, float);
```

→ When a function is called the compiler must figure out which of the functions is to be invoked. This is done by the compiler by comparing the types and numbers of the actual arguments to type and numbers of formal arguments.

→ Therefore, the idea is to invoke the function that is best match on the arguments and give a compile time error if no function is the best match.

* WAP to find the volume of cube, cylinder and rectangular box by using function overloading concept.

```
#include <iostream.h>
using namespace std;
// Declarations (prototypes)
        int volume (int);
    double volume (double, int);
    long volume (long, int, int);
```

```
int main()
    {
    cout << volume(10) << endl;
    cout << volume (2.5,8) << endl;
    cout << volume (100l, 75, 5) << endl;
    return 0;
}

int volume (int s)          //cube.
  {
  return (s*s*s)
  }
  double volume (double r, int h)  //cylinder.
    {
    return (3.14519 * r*r * h)
    }
    long volume (long l, int b , int h)   //rectangular box.
    {
      return (l * b*h)
    }.
```

Types of function Overloading.

1. Different number of arguments.
2. Different type of arguments.

1. Different number of arguments:

```
#include <iostream>
using namespace std;
Void display (INT);
Void display (int, int);
int main()
{
```

```
int a=5, b=6;
cout << "One argument function:";
display(a);
cout<<" Two argument function:";
display(a,b);
return 0;
}
void display(int a){
  {
  cout << a << endl;
  }
void display(int a, int b)
 {
cout << a <<" and " << b << endl;
}.
```

2. Different type of argument:

```
#include <iostream>
using namespace std;
void display(char);
void display(int);
void display(float);
int main()
 {
 char ch = 'a';
 int inum = 25;
 float fnum = 50.5;
 cout <<" character function ";
 display(ch);
 cout <<"Integer function";
 display(inum);
 cout << "Float function :";
 display(fnum);
  return 0; }
```

```
void display (char character)
  {
  cout << character << endl;
  }
void display (int integer)
  {
  cout << integer << endl;
  }
void display (float floatnum)
  {
  cout << floatnum << endl;
  }
```

Inline Functions.

Function is used to save memory space and reduce size of the codes.

Use of function adds extra overheads. The calling of a function requires activities like jumping to the function definition, saving registers (i.e memory values), pushing arguments into the stack and returning result to the calling function. These instructions take time for execution.

When the size of function is large, this time can be compromised as it reduces size of code much. But when the size of function is small, the use of function seems useless as the advantage of function in reduction of code is less than disadvantage of function due to extra time taken by instructions during calling function.

Thus, the use of function is disadvantageous in small sized function. This is because when a function is small, a substantial percentage of execution time may be spent in such overheads.

To eliminate this problem, c++ proposes a new feature called inline function.

When an inline function is called, the compiler will replace the function call with the function code. So in reality there will be no function call in that place, only the code of the function is copied. No jumping is needed to different address such that it doesn't add extra time for jumping to function definition and returning the result to calling function.

A function is made inline by adding prefix 'inline' to the function definition.

The syntax is:

```
inline return_type function_name (argument_list)
{
function body
}
```

Eg:
```
# include <iostream.h>
inline float interest (float p, float t, float r)
{
return ((p*t*r)/100);
}
void main()
{
float result;
result = interest (10,000, 5, 10);
cout << "The interest is" << result;
}
```

Advantage:

1. Increases performance of program.
2. It doesnot require function calling overhead.
3. It saves overhead of variables push/pop on the stack, while function calling.
4. It saves overhead of return of result from called function to calling function.

Disadvantages:

Inline function makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called.

Situations where inline expansion may not work:

1. For large function.
2. For functions returning values, if a loop, a 'switch' or a 'goto' exists.
3. If inline functions are recursive.

Default arguments:

In C++, there is a provision of supplying less number of arguments than the actual number of parameters. This mechanism is supported by the default arguments. So, with the help of default arguments we can call a function without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call.

* Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alert the program for possible default values.

```
#include <iostream>
using namespace std;
void marks_tot(int m1=40, int m2=40, int m3=40);
int main()
{
  marks_tot();
  marks_tot(55);
  marks_tot(66,77)
  marks_tot(75,85,92);
  return 0;
}
void marks_tot(int m1, int m2, int m3)
{
cout<<"total:"<<(m1+m2+m3)<<endl;
}
```

For character:

```
char name[50] = "kathmandu"
char *myname = name;
```

pointer

Types of function call:

The arguments in function can be passed in two ways.
1. pass by value.
2. Pass by reference (or address).

Function call by value (Pass by value)

→ when values of actual arguments are passed to a funct as arguments, it is know as function call by value. In this call, the value of each actual argument is copied into corresponding formal argument of the function definition. The content of the arguments in the calling function are not alterned, even if they are changed in the called function.

Eg:
```
#include <iostream.h>
Void swap (int, int);
Void main ()
{
int a,b;
a=99 ; b=89;
Cout << "Before function calling, a & b are: "<<a<<
Swap (a,b);
Cout << "After function calling, a & b are: "<<a<<b
}
Void swap (int x, int y)
{
int temp; temp=x; x=y; y=temp;
Cout <<" The values within functions are: "<<x<<y;
}
```

Output:
Before function calling, a & b are : 99 89
The values within functions are : 89 99
After function calling, a & b are : 99 89

Function call by Reference (Pass by reference)

→ In this type of function call, the address of variable or argument is passed to the function as argument instead of actual value of variable. So the variable passed as arguments during the function call are changed by the called function.

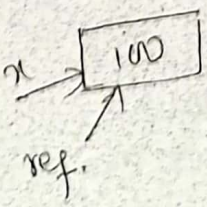
```
# include <iostream.h>
  Void swap (int &, int &)
int main ()
{
int a=5,b=6;
cout <<"Before swapping: a =" <<a <<" and b =
      "<<b<<endl;
 swap (a,b);
 cout <<" After swapping: a = "<<a <<" and b ="<<b<<;
 return 0;
 }
void swap (int &x, int &y)
 {
  int temp;
  temp= x;
   x=y;
   y= temp;
  }
```

Reference Variable:

→ A reference variable is another name (i.e alias) for a variable.

→ It is declared using & operator.

Syntax
datatype & reference_variable = original value;



Here ref and x references same memory.

```
#include <iostream.h>
Void main()
{
int x = 100;
int & ref = x;
cout << "x = " << x " ref" << "ref = " << ref;
}
```

Output:

x = 100
ref = 100

Array:

An array is a series of homogeneous pieces of data tha are all identical in type.

Declaration of array:

- Syntax:

type identifier [number_of_elements]

Eg:

int marks [5];

Return by reference:

In C++, There is a provision of return by reference. Here the true value is shown as reference but false value is shown as it is. That means the reference value is shown in true case and the true value is shown.

Syntax:

function_call = reference_value;

Eg:
```
#include <iostream.h>
#include <conio.h>
using namespace std;
```

```
int &max (int &a , int &b).
  {
   if (a>b)
     {
     return (a);
     }
    else
     {
      return (b);
      }
     }
int main()
 {
  int a1 = 10, b1 = 15, q;
  Max (a1, b1) = q;
  cout << "a1 = " << q;
  getch();
  return 0;
 }
```

cout<<a1<<endl;
cout << b1;

32765 to 32765

Array

Grouping of similar types of variables with a common tag is known as array. Arrays decrease the number of variable names used in the programme.

Syntax: (declaration of array)

data_type array_name [size];

Eg: int a[10];

Arrays are index based starting from 0 index as a[0], a[1] ....

Direct initialization of array:

int a[] = {1,2,3,4,5,6}

8. Pointer

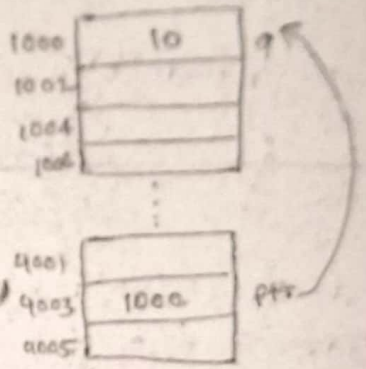
Pointers are the special type of variables that can hold or store the memory location of other variable.

Syntax:

```
type *variable_name;
```

eg:

```
int a;
a=5
int *ptr;
ptr = &a;
```



Advantage:

We can perform pointer arithmetic.

String:

Array of characters are called strings.

Syntax:

```
char string_variable [size];
```

Example:

```
char str1[] = "kathmandu";
char str2[] = {'K', 'a', 't', 'h', 'm', 'a', 'n', 'd', 'u', '\0'};
```

Structure:

user defined complex data type.

college bag → contains books of different subjects.

Structure ⇒ To store several value of different data types.
(heterogeneous)

Primitive data types + derived data type.
(int, float) (array)

Bill –

S.N	Name	quantity	Cost	Total

Both are user defined data type.

	C – structure.		C++ structure.
1.	Collection of variables or data only.	1.	Collection of data/variables and functions.
2.	Members are public.	2.	Members are public/private/protected.

Differences & Similarities

	class		C++ structure.
①	collection of variables or data & function.	①	Collection of data/variables and functions.
②	~~public~~ Members are public/private/protected.	②	Members are public/private/protected.
③	By default members are private	③	By default members are public
④	Takes part in inheritance.	④	Never takes part in inheritance
⑤	We can create complex programmes.	⑤	Simple programmes.

C++ structure example:

Syntax:

```
struct [<tag_name>]
{
structure_members;
} [structure variables];
```

Eg:

```
#include <iostream.h>
#include <conio.h>
struct stu
{
int id;
char name[30];

void getdata()
{
cout<<" Enter student id, name";
cin>>id>>name;
cout<<" Id =" <<id <<endl;
cout<<"Name =" <<name;
}

} s;
```

↑ structure variable.

```
Void main()
{
//struct stu s;
s.getdata();
getch();
}
```

→ dot operator, member operation.

Union

→ Same as structure but share to memory address.
→ Total size of memory is the largest size of data used.

Enumeration.

Enumeration (or Enum) is a user defined data type. It is mainly used to assign names to integral constants, that makes a program easy to read & maintain.

```
#include <iostream>
using namespace std;
enum day{sun, mon, tue, wed, thu, fri, sat};
int main()
{
enum day d, x;
d = wed; x = thu;
cout << "The day no. for wed is" <<d;
cout << "The day no. for thu is" <<x;
return 0;
}
```

Chapter-4
Objects and classes

A class is a way to bind the data and its associated function together.

Class has two parts.

① class declaration.
② class function definition.

Syntax:

```
class class_name
  {
  private:
     variable declarations;
     function declarations;
  public:
     variable declarations;
     function declarations;
  protected:
     variable declarations;
     function declarations;
  };
```

① class members

Access specifiers:

The access specifiers controls the visibility of member ~~variability~~ variables and functions.

→ The members which are declared as private can be accessed only from within the class. Only member function can access the data.
→ Public members can be accessed from outside the class.
→ A member declared as protected is accessible within its class and any class immediately derived from it.
Protected visibility modifier is used in inheritance.

(By default data members and member functions are private).

Thus, the class consists of data members along with member functions, enclosed within braces and terminated by a semicolon. The binding of data and functions together into a single unit is called encapsulation.

Creating Objects.

The instance of a class or variables of type class are known as objects. After the class has been created, we can create any number of objects associated with that class.

Syntax:

```
Student s;   // Student is assumed to be class name.
```

Multiple objects:

```
Student s, s1, s2, s3, s4;
```

Complete syntax:

```
class Student
  {
  private:
    ---
  public:
    ---
  }
  :
Student s1, s2, s3
```

or

```
class Student
  {
  private:
    ---
  public:
    ---
  } s1, s2;
```

Accessing class members.

The public members variables and public member function can be accessed using object_name and dot (.) operator.

Syntax:

```
object-name.data_members;
object-name.function-name (actual-argument);
```

For eg:

```
class Demo
{
private:
    int data1;
    int data2;
public:
    void setdata(int d1, int d2)
    {
    data1 = d1;
    data2 = d2;
    }
    void showdata()
    {
    cout << "data1 = " << data1 << endl;
    cout << "data2 = " << data2 << endl;
    }
};
int main()
{
Demo d;
d.setdata(10,20);
s.showdata();
return 0;
}
```

private data members

Public member function

s.data1 ✗ because private member

Defining Member Function.

⇒ The member function can be defined in two ways.

1) Inside the class
2) Outside the class.

Inside the class.

```
#include <iostream>
using namespace std;
class Student
 {
 int roll;
 char name[20];
 float marks;
 char address[30];
 public:
 void getdata()
 {
 cout<<" Enter roll, name, marks and
       address: "<<endl;
 cin>>name>>roll>>marks>>address;
 }
 void showdata()
 {
 cout<<" Details are "<<endl;
 cout<<name<<endl<<roll<<endl<<
  marks<<endl<<address<<endl;
 }
 };
 int main()
 {
 student s;
 s.getdata();
 s.showdata();
 return 0;
 }
```

Outside the class.

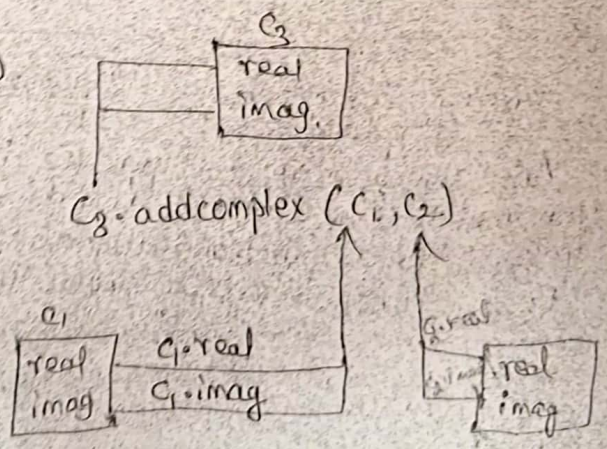
```
#include <iostream>
using namespace std;
class Student
 {
 int roll;
 int name[20];
 float marks;
 char address[30];
 public:
 void getdata();
 void showdata();
 };
Void student::getdata()
 {
cout<<" Enter roll, name, marks and
 address: " endl;
cin>> name>>roll>> marks>>address;
 }
Void student::showdata()
 {
cout<<" Details are "; <<endl;
cout<<name<<endl<<roll<<endl<<
 marks<<endl<<address<<endl;
 }
int main()
 {
 Student s;
 s.getdata();
 s.showdata();
 return 0;
 }
```

Passing Object as a argument:

An object can be passed to a function as an argument or parameter like normal variables. When an object of a class is passed to a function, all data members of the class are passed to a function as a single unit.

Eg: Addition of two complex numbers using object as arguments

```
#include <iostream.h>
class complex
    {
    private:
      float real,imag;
    public:
      void getvalue()
        {
        cout << "Enter real part" << endl;
        cin >> real;
        cout << "Enter imaginary part" << endl;
        cin >> imag;
        }
      void showvalue()
        {
        cout << "The sum is " << real << "+j" << imag;
        }
      void addcomplex (complex c1, complex c2)
            {
            real = c1.real + c2.real;
            imag = c1.imag + c2.imag;
            }
int main()
  {
  complex c1, c2, c3;
  cout << "Enter first complex number" << endl;
  c1.getvalue();
  cout << "Enter second complex number" << endl;
  c2.getvalue();
  c3.addcomplex (c1, c2)
  c3.showvalue();
  return 0;
  }
```



Q. WAP to perform the addition of time in hour and minute format.

```
#include <iostream>
using namespace std;
class time
 {
 private:
     int hours, minutes;
 public:
    void gettime (int h, int m)
     {
     hours = h;
     minutes = m;
     }
     void puttime (void)
      {
      cout << hours << ":" << minutes << endl;
      }
     void sum (time, time );   // declaration with objects
                                  as arguments.
      };
Void time::sum (time t1, time t2)
     {
     minutes = t1.minutes + t2.minutes;
     hours = minutes/60;
     minutes = minutes % 60;
     hours = hours + t1.hours + t2.hours;
      }
int main()
 {
 time T1, T2, T3;
 T1.getTime (2, 45);
 T2.getTime (3, 30);
 T3.sum (T1, T2);
 cout << "T1 = " << endl;
  T1.putTime ();
 cout << "T2 = " << endl;
  T2.putTime();
 cout << "T3 = " << endl;
  T3.puttime ();
  return 0;
  }
```

Returning objects from function:

```
#include <iostream>
using namespace std;
class complex
{
private:
    int real, imag;
public:
    void GetComplex ();
    Complex AddComplex (complex);
    void display ();
};
void Complex :: GetComplex ()
{
cout << "Real part : ";
cin >> real;
cout << " Imaginary part : ";
cin >> imag;
}
void Complex :: Display ()
{
cout << " The sum is : " << real << "+j" << imag;
}
```

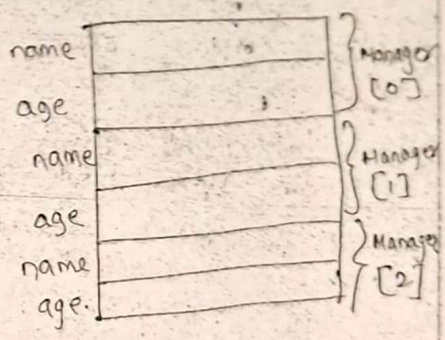
Return type class "type" → `Complex Complex :: AddComplex(complex c)`

```
{
Complex ob;
ob.real = real + c.real;
ob.imag = imag + c.imag;
return ob;
}
void main()
{
Complex c1, c2, c3;
cout << " Enter First Complex number";
c1. GetComplex ();
cout << " Enter Second Complex number";
c2. GetComplex ();
c3 = c1. Add Complex (c2);
c3. Display (); }
```

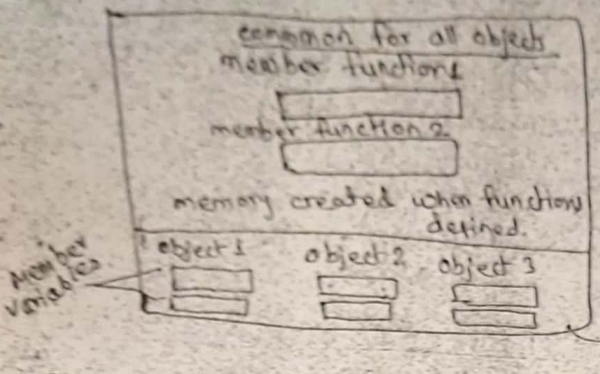
Array of objects:

Collection of similar types of object is known as array of objects.

```
#include <iostream>
class employee
    {
    char name[30];
    int age;
    public:
     void getdata (void);
     void putdata (void);
     };
void employee::getdata(void)
     {
     cout <<"Entername";
     cin >> name;
     cout <<"Enter age";
     cin >> age;
     }
void employee::putdata(void)
     {
     cout<<"Name:" << name << endl;
     cout<<" Age :" << age << endl;
     }
const int size = 3;
int main()
   {
    employee manager[size];
    for(int i=0; i<size; i++)
      {
      cout<<" Details of manager" << i+1 << endl;
      manager[i].getdata();
      }
      cout <<"\n";
      for (i=0; i<size ; i++)
       {
        cout <<"\n Manager " << i+1 <<"\n";
        manager[i].putdata();
       }
       return 0;
    }
```



Memory allocation for objects:



Static Data Members

→ Each object contains its own separate data. But if a data item in a class is defined as 'static', then only one such item is created for entire class, no matter how many objects there are.

→ A static data member is useful when all objects of the same class must share a common item of information.

characteristics:

(a) When we precede a member variable's declaration with static, we are telling the compiler that only one copy of that variable will exist and all objects of the class will share that variable. Hence static variables are called class variables.

Note: Normal data members are called object variable but static data members are called class variables.

(b) Unlike regular data members, individual copies of a static member variables are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use same variable.

(c) All static variables are initialized to zero when the first object is created.

(d) The type and the scope of each static member variable must be defined outside class.

Syntax:

```
data_type class_name :: variable_name;
```

Eg: `int complex1::count;`

```
#include <iostream>
using namespace std;
class item
  {
  static int count;
  int number;
public:
  void getdata (int a)
     {
      number = a;
      count ++;
     }
  void getcount (void)
   {
    cout <<"count : ";
    cout <<count <<endl;
   }
  };
int item :: count;   // definition of static data member
int main()
  {
  item a,b,c;        // count is initialized to zero
  a.getcount();      // count is initialized to zero
  a.getcount();
  b.getcount();
  c.getcount();

  a.getdata(100);
  b.getdata(200);
  c.getdata(300);

  cout <<"After reading data";
  a.getcount();      // display count
  b.getcount();
  c.getcount();
  return 0;
  }
```

or

```
int item :: count = 10;
```

Output

```
3
3
3
```

Static member functions.

→ A class may also have static methods as its member. Static is defined by using the keyword 'static' before the member that is to be declared as static function.

Syntax:

```
static return_type function_name (argument_list)
{
body
}
```

A normal member function of a class is accessed using object of that class and dot operator. The functions declared with the keyword 'static' is accessed using classname and scope resolution operator.

For eg:

```
class_name :: function_name (argument_list)
```

characteristics of static member functions.

(a) A static function can have access to only other static members (functions or variables) declared in the same class. It can't access non-static members.

(b) A static member function can be called using the class name (instead of its objects).

(c) A static member function do not have access to the 'this' pointer of the class.

(d) A static member function cannot be declared as virtual.

```
#include <iostream>
using namespace std;
class test
{
int code;
static int count; //static member variable
public:
void setcode (void)
{
code = ++count;
}
void showcode (void)
{
cout << "object number:" << code << endl;
}
```

```
a=6
b=a++   } b=6
        } a=7

b=++a   } b=7
        } a=7
```

```
    static void showcount (void)   //static member function
{ cout << "count:" << count << endl;
  }
  };
  int test:: count;
  int main()
   {
   test t1, t2;
    t1.setcode();    1
    t2.setcode();    2
    test::showcount();  // accessing static function.
    test t3;
    t3.setcode();
    test::showcount();
    [redacted];
    [redacted];
    [redacted];
    return 0;
    }.
```

output
1
2
3

Friend function:

Private members cannot be accessed from outside the class. i.e a non-member function cannot have an access to the private data of a class.

But there comes a situations, c++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes.

Syntax:

```
class class_name
    {
    friend return_type function_name(type1, type2...typen);
    };
```

~~Advantages~~ characteristics

1) Friend function acts as bridge between two classes by operating on their private datas.
2) Can be declared either in the public or private part of a class.
3) Friend function is not in the scope of the class to which it has been declared as friend.
4) It cannot be called using the object of that class.
5) It can be called like a normal function without using any objects.
6) It cannot directly access the data members like other member function and it can access the data members by using object through dot operator.
7) It has objects as arguments.

```
#include <iostream.h>
using namespace std;
class circle
  {
  private:
     float rad;
  public:
     void Getdata()
     {
     cout <<" Enter radius ";
     cin >> rad;
     }
    friend void Area(circle);
     };
void Area (circle cir)
   {
    float a;
    a=3.14 * cir.rad * cir.rad;
    cout << " Area of circle is:" <<a;
    }
```

```
void main()
  {
  circle c;
  c.Getdata();
   Area(c);
  }.
```

Bridging classes with friend function.

```
# include <iostream.h>
// class second;
  class first
    {
    private:
      int data1;
    public:
      void getdata1 (int x)
        {
        data1 = x;
        }
      friend int sum (first, second); // friend function.
  };
  class second
      {
      private:
        int data2;
      public:
        void getdata2 (int y)
          {
          data2 = y;
          }
      friend int sum (first, second); //friend function.
        };
int sum (first a, second b)
      {return (a.data1 + b.data2);
      }
```

```
int main()
{
first a;
a.getdata1 (100);
second b;
b.getdata2 (200);
cout << " Sum of first and second is : " << sum(a,b);
return 0;
}.
```

objects

Friend class:

Syntax:

```
class B;
class A
{
//class B is friend class of class A.
friend class B;
}.
class B
{
:::
}.
```

When a class is made a friend class, all the member functions of that class becomes friend function.

In the above syntax, all member functions of class B will be friend function of class A.

Thus, any member function of class B can access the private and protected data of class A.

[∴But member functions of class A cannot access the data of class B.]

```
# include <iostream>
class xyz
  {
  float a,b;
  public:
    Void Read()
     {
     cout << " Enter two numbers : ".
     cin >> a >> b;
     }
     friend class ABC;
     };
class ABC
     {
     public:
      Void Display (xyz x)
       {
       cout << " First member is " << x.a;
       cout << " The second member is " << x.b;
       }
     float sum (xyz x)
        {
         float s;
         s = x.a + x.b
         return s;
        }
};
Void main ()
   {
     xyz x;
     ABC a;
     x. Read();
     a. Display(x).
     Cout << " The Sum is " << a.sum (x);
        }
```

Note: Friend function and friend class breaches the wall of OOP.

Object as Data Member / containership / nested classes.

→ If any class has object of another class then it is said to be nested class.

```
#include <iostream>
class A
 {
 int data1;
 public:
 void getdata()
  {
  cout << "Enter data 1";
  cin >> data1;
  }
 Void showdata()
  {
  cout << "Data 1 is" << data1;
  }
};
class B
 {
 int data2;
 A a;  // containership
 public:
  Void getdata()
  {
   a.getdata();
   Cout << "Enter data 2";
   Cin >> data 2;
  }
 void showdata()
  {
  a.showdata();
  cout << "Data2 is" << data2;
  }
 };
```

```
int main()
{
B b;
b.getdata();
b.showdata();
return 0;
}.
```

Constructors and Destructors

Constructor:

When an object of a class is created, separate copy of the data members of the class for each object are created. Initially, data member of the created object have no values assigned. To initialize value of data members of the class, we used normal member function. But, we can use constructor to assign some initial value to data members of the object instead of normal member function.

so, A constructor is a special member function of a class which initializes member of its object.

Characteristics:

1. The name of constructor must be same as that of its class name.
2. It is invoked/accessed automatically when the objects is created.
3. It has no return type (not even void).
4. It should be defined in public section.
5. It can't be inherited, though a derived class can call the base class constructer.
6. It can't be virtual.
7. The address of constructor can't be referred in program.
8. The constructor makes implicit calls to the operator 'new' and 'delete' when dynamic memory allocation is required.
9. The constructor can be overloaded.

Eg:
```
class integer
{
int m,n;
public:
integer(void); //constructor declared.
... };
```

```
# integer :: integer(). //Constructor defined.
{
m=0; n=0;
}
```

```
#include <iostream.h>
class Sample
{
public:
  Sample()
  {
  cout << "Constructor called"; }
  };
void main()
{
Sample S1, S2, S3;
}
```

Output:

Constructor called
Constructor called
Constructor called

WAP to calculate volume of a box with data members l, b and h.

without using Constructor.

```
# include <iostream.h>
class Box
{
private:
  int l,b,h;
public:
void ReadData(int len, int br, int ht)
  {
  l = len;
  b = br;
  h = ht;
  }
float getvolume()
  {
  return (l*b*h)
  }
};
int main()
{
int vol;
Box ob;
ob.ReadData(10,5,2);
vol = ob.getvolume();
cout << "Volume = " << vol;
return 0;
}
```

with using Constructor.

```
#include <iostream.h>
class Box
{
private:
  int l,b,h;
public:
Box(int len, int br, int ht)
  {
  l = len;
  b = br;
  h = ht;
  }
float getvolume()
  {
  return (l*b*h)
  }
};
int main()
{
int vol;
Box ob(10,5,2);
vol = ob.getVolume();
cout << "volume = " << vol;
return 0;
}
```

Types of constructor:

1. Default constructor.
2. Parameterized constructor.
3. Copy constructor.

Default Constructor.

The constructor which has no arguments is known as default constructor. A default constructor is automatically called when no argument are supplied while creating objects as:

```
class_name object_name;
```

```
#include <iostream.h>
using namespace std;
class Add
  {
  int x,y,z;
  public:
  Add();      //default constructor.
  void calculate();
  void display();
  };
Add :: Add()
  {
  x=6;
  y=5;
  }
void Add :: calculate()
  {
  z=x+y;
  }
void Add :: display()
  {
  cout<<z;
  }
```

```
void main()
  {
  Add a;
  a.calculate();
  a.display();
  }
```

Parametrized Constructor.

The constructor which takes some argument is known as parametrized constructor. In this type of constructor we should supply/pass arguments while defining object of the class.

```
Eg: #include <iostream>
    using namespace std;
    class Add
      {
      int x,y,z;
    public:
      Add (int, int);
      Void calculate();
      Void display();
      }
    Add :: Add (int a, int b)
      {
      x=a;
      y=b;
      }
    Void Add :: calculate()
      {
      z=x+y;
      }
    Void Add :: display()
      {
      cout<<z;
      }
```

```
Void main()
  {
  Add a(5,6)
  a.calculate();
  a.display();
  }
```

Parametrized constructor can be called in two ways.

```
Eg: class integer
    {
    int m,n;
    public:
    integer(int x, int y);
    };
    integer::integer(int x, int y)
    {
    m=x; n=y;}
```

```
void main()
{
integer int = integer (10,20);
//explicit call
integer int(10,20);
// implicit call
}
```

Copy constructor:

The constructor that initializes data members of the object by copying the value of another object initialized by either default constructor or parametrized constructor is called copy constructor.

Syntax:

```
class name (class_name & object)
  {
  // body
  }.
```

```
# include <iostream.h>
using namespace std;
   class Add
        {
        int x,y,z;
        public:
        Add()
          {
          }                    // default constructor.
        Add (int a, int b)
          {
          x=a;
          y=b;
          }                    // Parametrized constructor.
        Add (Add &);           // copy constructor.
        void calculate();
        void display();
        };

Add::Add (Add &p)
   {
   x = p.x;
   y = p.y;
   cout<<x<<y;
   }
void Add :: calculate()
   {
   z=x+y;
   }
void Add::display()
   {
   cout<<z;
   }

int main()
{
Add a(5,6);
Add b(a);   // copy constructor
b.calculate();
b.display();
};
```

~~Parameterized constructor can be called in two ways:~~

Q. WAP to initialize an object of class with parametrized constructor and copy this object into another object using copy constructor.

```
#include<iostream.h>
class Box
{
private:
    float l,b,h;
public:
Box(float len, float br, float ht)      // parameterized constructor
  {
   l = len;
   b = br;
   h = ht;
  }
Box( Box &p)                            // copy constructor.
    {
    l = p.l;
    b = p.b;
    h = p.h;
    }
void display()
  {
  cout<<"Length: "<<l;
  cout<<" Breadth: "<<b;
  cout<<" Height: "<<h;
  }
  float getvolume()
    {
    return (l*b*h);
    }
};
void main()
 {
 float vol;
 Box b1(10.5,5.5,6.5)
 Box b2(b1);
 cout<<"For first object";
 b1.display();
 vol =b1.getvolume();
 cout<<vol;
 cout<<"For second object";
 b2.display();
 vol = b2.getvolume();
 cout<<" The volume is" <<vol;
 }
```

Constructor Overloading:

If a program contains more than one constructor with different no of arguments or different types of arguments then it is said constructor overloading.

```
#include <iostream.h>
class Account
  {
  private:
    int accno;
    float balance;
  public:
    Account ()          // constructor with no argument.
     {
     accno = 12345;
     balance = 2555.100;
     }
    Account (int acc);  // constructor with one argument.
     {
     accno = acc;
     balance = 0.0;     // constructor with
     }
    Account (int acc, float bal)  // constructor with two arguments.
     {
     accno = acc;
     balance = bal;
     }
   void display ()
      { cout << accno;
        cout << balance;
      }
  };
```

```
Void main()
{
Account acc1 ; // first constructor called.
Account acc2(100); 2nd constructor called.
Account acc3 (1432, 10000.50); // 3rd constructor called.
cout << "Account Info";
acc1.display();
acc2.display();
acc3.display();
  }
```

Destructors:

Destructors are used for deallocation of memory, initialized by constructors. In a simple way we can say destructors are used to destroy the objects that has been created by constructors.

Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde (~).

Thus, a destructor is special member function of a class that is executed when an object of that class is destroyed. It is the counterpart of constructor.

When a variable or object goes out of scope, or a dynamically allocated variable or object is explicitly deleted using the 'delete' keyword, the ~~cons~~ destructor is called to clean up the class before it is removed from memory.

For simple classes, a destructor is not needed because C++ will automatically clean up the memory. However, if we have dynamically allocated memory, or if we need to do some kind of maintenance/operations before the class is destroyed (eg. closing a file, closing a database etc.), the destructor is the perfect place to do so.

Like constructors, destructors have specific characteristics:

1) The destructor must have same name as class, preceded by tilde (~).
2) The destructor can't take arguments.
3) The destructor has no return type.
4) The destructor is called automatically whenever an instance of the class to which it belongs goes out of existence.
5) The destructor can't be declared static.
6) The destructor is defined within public section.
7) There is only one destructor in a class.
8) Destructor can't be overloaded.

Eg:
```
#include <iostream>
using namespace std;
class Box
  {
  public:
    Box()
     {
      cout <<"\n Constructor is called.";
     }
   ~ Box()
     {
      cout <<"\n Destructor is called.";
     }
  };
void main()
  {
    Box b1;
    int i= 2;
     if (i>0)
       {
        Box b2, b3;
       }
  }
```

Output:

Constructor is called.
Constructor is called.
constructor is called
destructor is called.
destructor is called.

The object b1 is destroyed after termination of main() block. The final call to destructor is not seen in normal run of the program. But we can see the previous output by pressing Alt + F5 in Turbo C++.

Eg:
```
#include <iostream>
int count = 0;
class alpha
  {
  public:
    alpha()
     {
     count ++;
     cout <<"\n No of object created" <<count;
     }
   ~ alpha()
     {cout <<"\n No of object destroyed" <<count;
      count --;
     } };
```

```
int main()
{
cout <<"\n\n  Enter MAIN \n";
alpha A1, A2, A3, A4;
{
cout <<"\n\n Enter Block I\n";
alpha A5;
}
{
cout <<"\n\n Enter Block 2 \n";
alpha A6;
}
cout <<"\n\n Re-enter main \n";
return 0;
}
```

OUTPUT

Enter MAIN.

No. of object created 1
No. of object created 2
No of object created 3.
No. of object created 4.

Enter Block I.

No. of object created 5
No. of object destroyed 5.

Enter Block 2

No of object created 5.
No of object destroyed 5.

Re-enter main.

No of object destroyed 4
No of object destroyed 3
No of object destroyed 2
No of object destroyed 1

Constructor.	Destructor.
1. It has the same name as the class name.	1. It has the same name as the class name preceded by tilde (~).
2. It is automatically called when an object is created.	2. It is called automatically when an object goes out of scope (or is destroyed).
3. It can have argument but no return type.	3. It neither has argument nor return type.
4. It can be overloaded.	4. It cannot be overloaded.
5. It cannot be virtual.	5. It can be virtual.
6. Constructor invocation order is same as object creation order.	6. Destructor invocation order is reverse of object creation order.
7. In inheritence, base class constructor is invoked first and derived class constructor is invoked next.	7. In inheritance, derived class destructor is invoked first and base class destructor is invoked next.

8. For eg:
class test
{
private:
//....
public:
test()
{
....
}
test(int n)
{
....
}
};
	8. For eg:
	class test
	{
	private
	//....
	public:
	//....
	~test()
	{
	....
	}.

Pointer to objects:

We can create pointer variable that will hold the address of the object. By using (&) address operator, the starting address of an object can be achieved similar to standard data type.

Thus, if a variable represents an objects, then & variable represents the starting address of that variable.

We can ~~create~~ declare a pointer to object by writing

```
class_name * pointer_to_object;
class_name * object_name;
```

Now we can assign the address of an object to pointer variable as:

```
pointer_to_object = & object_name;
```

We can access members by making use of arrow operator (->)

```
pointer_to_object -> member;
```

or

```
(*pointer_to_object).member;
```

```
#include <iostream.h>
class employee
    {
    private:
        char name[25];
        float salary;
    public:
        void getdata()
            {
            cout <<"\n Enter name:";
            cin >>name;
            cout <<"\n Enter salary :";
            cin>>salary;
            }
        void showdata()
            {
            cout <<"\n Name : " << name <<endl;
            cout <<"\n Salary : " <<salary <<endl;
            }
```

```
int main()
  {
  employee *eptr;
   employee e;
   eptr = &e;
   eptr -> getdata ();
   eptr -> showdata ();
   return 0;
   }.
```

Dynamic Memory Allocation for objects and Object Array.

→ It is the mechanism of allocating memory dynamically for objects.

Syntax:

```
class_name *object_ptr;
object_ptr = new class_name ; // allocates memory for single object.

object_ptr = new class_name[size]; // allocates memory for an array of objects.
```

The new operator can also be used to initialize the allocated memory location by calling the constructor.

```
class_name *object_ptr;
 object_ptr = new class_name (args....);
```

or

```
class_name * obj_ptr = new class_name (args....);
```

For eg:

```
class test
    {
    private:
       int data;
       //....
    public:
    test (){
        }
    test (int n)
        {
        data=n;
        }
    //...
};
```

```
test *test_ptr,*test_arr;
test_ptr = new test ; //allocates space for object of type test.
test_arr = new test[n]; // allocates space for an array of objects of type test with n elements.
```

Construction initialization:

```
test *test_ptr = new test(10);
```

Constant Member functions and constant Objects.

C++ allows us to define function members that guarantees to change the data members value (eventually object's value) and prevents accidental alternation. These types of functions that guarantee not to change object's value (or data member value) are called constant functions.

The constant member functions are useful for constant objects because they guarantee not to change objects value.

As we know that constant objects should not change their value after their declaration and constant member function guarantees this.

So, constant objects can call only constant functions.

constant member functions are declared by placing 'const' after the parameter list and before function body.

Eg:
```
int second () const
    {
    return s;
    }
```

⇒ The constant member function can be declared inside or outside its class.

⇒ For a constant function, if the function tries to change objects members or value then compiler indicates error.

Eg:
```
int second() const
    {
    return s++;        //error.
    }
```

⇒ Constant function can be called by constant as well as non constant objects.

But

A non constant function cannot be called by constant object because it can change the object's value or data members value.

Eg:

```
class Test
    {
    private:
      int data;
    public:
      Test (int n=0)
       {
        data=n;
       }
      void setdata (int n)
       {
       data=n;
       }
     int getdata() const
       {
       return data;
       }
       };
int main ()
    {
     Test t1(1);
    const Test t2(2);
     t1.setdata(5);
     cout<< t1.getdata();
```

✓ non constant object can call non constant function.

✓ non constant object can call constant function.

```
t2.setdata(7)
```

✗ constant object cannot call non constant function.

```
cout<< t2.getdata
```

✓ constant object can call constant function.

this pointer.

C++ uses a unique keyword called 'this' to represent an object that invokes a member function. this is a pointer that points to the object for which this function was called.

Eg:

The function call c.getdata() will set the pointer this to the address of the object c. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer 'this' acts as an implicit argument to all the member functions.

```
#include <iostream.h>
  class Box
    {
     private:
        float l, b, h;
        public:
           Void getAddress ()
              {
               cout<<" Address of object using this pointer:"
                                                  << this;
                }
              };
      Void main()
          {
           Box b;
           b.getAddress();
           Cout <<"Address of object directly." << & b;
           }
```

```
#include <iostream.h>
  class employee
     {
      private :
         int salary;
      public:
```

```
void getsalary (int s)
  {
  this -> salary = s;
  }
 void display()
  {
  cout << "The salary is " << salary;
  }
 };
int main()
  {
  employee e;
  e.getsalary (5000);
  e.display();
  return 0;
  }
```

```
# include <iostream>
using namespace std;
class complex
  {
  float real;
  float imag;
  public :
   complex()
   {
   this->real = 0;
   this->imag = 0;
   }
complex (float real, float imag)
  {
  this -> real = real;
  this->imag = imag;
  }
void showdata()
 {cout << this -> real << "+i" << this -> imag << endl;
 }
};
```

```
int main()
  {
complex c1, c2 (1.1, 2.4);
c1.showdata();
c2.showdata();
  return 0;
  }
```

Assignments – I
chapter 1, 2, 3, 4.

Krishna
Date
Page

Object Oriented Programming (CT 501).

(1) Explain why OOP is better than POP? Point out issues with POP.

(2) What are the advantages and disadvantages of OOP. Explain in brief history of c++.

(3) Differenciate between:
   (1) OOP and POP.
   (2) C and c++ (In points).
   (3) c and c++ (By program).

(4) Explain the features of OOP. Describe in brief the lexical components of C++.

(5) What is type conversion in c++? Explain explicit type conversion with program. (use static_cast).

(6) What is namespace? Why do we need this? Write a program to demonstrate namespace using "using" keyword and scope resolution operator.

(7) How do we define user defined constant? Show by program

(8) What are manipulators? How can we use them? (show by program

(9) What ~~are~~ is dynamic memory allocation (DMA)? How can we implement this? Write a program to show ~~do~~ DMA.

(10) What is function overloading? Overload a function with
   (a) Different number of arguments.
   (b) Different type of arguments.

(11) What is inline function? WAP to implement inlinefunction. what is the advantage of inline function? Where we cannot use inline function?

(12) What is default arguments? What is the rules to define default arguments? show with program.

(13) What is pass by value and pass by reference in function call? Show with program.

Krishna
Date
Page

(14) What is reference variable ? Show return by reference by the help of example.

(15) Differentiate between:
① C structure and C++ structure.
② C++ structure and class.
③ C++ macros and inline function.
④ C++ structure and union.

(16) What are enumeration (enum) in C++? WAP for enum.

(17) WAP to calculate simple interest by using concept of default argument with default value of rate = 15%.

(18) WAP to calculate area of circle, rectangle and triangle using concept of function overloading.

(19) WAP to calculate the volume of cube, cuboid and cylinder using the concept of function overloading.

(20) WAP to display N number of characters by using default arguments for both parameters. Assume that the function takes two arguments one character to be printed and other number of characters to be printed.

(21) WAP to relate default arguments and function overloading on same program.

(22) WAP creating function that passes two temperatures by reference and sets the larger of the two numbers to 100 by using return by reference.

(23) Write syntax for class and object. Explain briefly about access specifiers.

(24) Write a program ~~to define~~ showing member function definition: ⓐ inside the class ⓑ outside the class.

(25) Create a class called student with data member name, ID and faculty. Use necessary member function to read the data member from user and display the details.

Krishna

Date
Page

(26) NAP to find TSA, CSA and volume of cylinder using concept of class and object.

$CSA = 2\pi rh$

$TSA = 2\pi r(r+h)$

$Vol = \pi r^2 h$.

(27) WAP to find the equivalent resistance in series and parallel for given two resistance r_1 & r_2.

$r_s = r_1 + r_2$

$r_p = \frac{r_1 \times r_2}{r_1 + r_2}$.

(28) Discuss briefly about array of an object. Create a class Student with data member name, id and faculty. Use necessary member function and write a program to scan and display the details of 5 students.

(29) How can you pass object as an argument? WAP to add two complex numbers ~~using~~ passing object as an argument.

(30) Create a class distance with data member feet and inch. Use necessary member function to add two distance and display result. (1ft → 12 inch).

(31) Create a class called distance with data member cm, m and km. Use member function to add and display result.

(32) Create a class called time with data member second, minute and hour. Use necessary member function to add two time.

(33) WAP to show "Returning objects from function".

(34) What are static data members? List out its properties.

(35) What are static member functions? List out its properties.

(36) WAP to demonstrate static data member and static member function.

(37) Why do we need friend function? List its properties. WAP to add one data member of two different class using the concept of friend function.

Krishna
Date
Page

(38) What are friend classes? WAP to ~~dem~~ show use of friend class.

(39) WAP to find a greatest among two data member using the concept of friend function.

(40) WAP to add data member of two different class using the concept of friend function. (Bridging classes).

(41) What is containership? show example.

(42) What is constructor? List out its properties. WAP to calculate volume of box. ~~using~~ Use constructor to initialize ~~l,b,h~~

(43) Why do we need constructor? Explain its types with program.

(44) What is copy constructor? WAP to show use of copy constructor.

(45) WAP to initialize an object of class with parametrized constructor and copy this object into another object using copy constructor.

(46) What is constructor overloading? Show with program.

(47) What is destructor? Why do we need it? List its characteristics. WAP to show destructor.

(48) Create a class called Complex with data member real and imaginary. Initialize all the data member using constructor and use necessary member function to add two complex numbers.

(49) Create a class called distance with data member cm, m and km. Initialize all the data member using constructor and use necessary member function to add two distance and display result.

(50) Create a class called time with data member hour, minute, second and day. Use constructor to initialize all the data member and use necessary member function to add two time and display result in main function.

(51) Differenciate between constructor and destructor.

(52) WAP to show pointer to object.

Krishna

Date
Page

Write a syntax for DMA for objects and object array. WAP to show DMA for objects and object array.

List the properties for constant member function and constant object. Write their syntax. Also write a program to show constant member functions and constant object.

What is this pointer? What is its importance? WAP showing use of this pointer.

Justify with example:

(a) In case of friend classes, friendship is not mutual.

(b) Friend function and classes breaches the wall of OOP.

(c) A friend function is not a member of any classes but has full access to the members of class where it is declared as friend.

WAP designing a class called midpoint to find mid-point between two points by returning object from member function using this pointer.

Explain about invocation order of constructor and destructor with the help of examples.