UNIT I – OBJECT ORIENTED PROGRAMMING FUNDAMENTALS

C++ Programming features - Data Abstraction - Encapsulation - class - object - constructors – static members – constant members – member functions – pointers – references - Role of this pointer – Storage classes – function as arguments.

Briefly explain the features of OOP or C++

- OOP is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions. The characteristics are:
  - Data structures characterize real time objects
  - Data and functions that operate on data are tied together
  - Data is hidden from external access, hence secure.
  - Bottom-up approach in program design

OOP Features

- *Classes and Objects*—Data and functions are put together in a single unit called class exhibiting data hiding. Objects are runtime entities and instance of a class.
- *Encapsulation*—mechanism that binds together function and the data it manipulates. Data is kept safe from external interference and misuse, since it is accessible only to functions of that class. This is known as data hiding.
- *Abstraction*—refers to act of representing essential features and hiding the implementation details. Classes use the concept of abstraction, therefore it is also known as abstract data type. Abstraction is useful for the implementation purpose. Actually the end user need not worry about how the particular operation is implemented. They should be facilitated only with the operations and not with the implementation.
- *Inheritance*—Process by which objects of one class acquire the properties of another class. Enables reusability with the possibility of adding new features or redefining existing features. It is important because it supports the concept of classification. C++ supports different types of inheritance such as single inheritance, multiple inheritance, multilevel inheritance and hierarchical inheritance.
- *Polymorphism*—is characterized by the phrase "one interface, multiple methods". An operation exhibiting different behaviors in different context. Polymorphism reduces complexity by allowing the same interface to access a general class of actions. In C++, both compile-time (function overloading/operator overloading) and run-time polymorphism (virtual function) are supported.

Key Benefits

- Through inheritance, we can eliminate redundant code and the use of existing classes
- Principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is easy to partition the work in a project based on the objects.
- Object oriented systems can be easily upgraded from small to large systems.
- Software complexity can be easily managed.
- Message passing techniques for communication between objects make the interface description with external systems much simpler.

List some key applications of OOP

- Development of Operating Systems
- Creation of DLL (Dynamic Linking Library)
- Computer Graphics

- Network Programming (Routers, firewalls)
- Voice Communication (Voice over IP)
- Web-servers (search engines)

Give the structure of a C++ program

- C++ was developed by Bjarne Stroustrup at Bell laboratories in 1980s.
- C++ is a superset of C, i.e., all C programs are also C++ programs.
- C++ programs have file extension `.cpp` The structure of a C++ program looks like

Comments
Include files
Class declaration
Member function definitions
Main function

List some operators in C++

::	Scope resolution operator
::*	Pointer-to-member declarator
->*	Pointer-to-member operator
.*	Pointer-to-member operator
new	Memory allocation operator
delete	Memory release operator
endl	Line feed operator
setw	Field width operator

List some keywords in C++

`catch`	`new`	`template`	`class`
`operator`	`this`	`delete`	`private`
`throw`	`friend`	`protected`	`try`
`inline`	`public`	`virtual`	

Mention the data types in C++.

- C++ adds *class*, *bool* and *reference* to C data types.
- The qualifiers short, long, signed and unsigned are also applicable.

Built-in	Derived	User-defined
int	array	structure
char	function	union
void	pointer	enum
float	*reference*	*class*
double		
bool		

How are comments specified in C++

- C++ introduces a single line comment symbol //.
- Any text after // till the end of line is treated as a comment.

Define constant in C++. Give an example.

- The qualifier **const** is used to create symbolic constants in C++.

  const *datatype vaname* = *value*;

```
const float pi = 3.14;
```

Briefly explain the standard input / output statements in C++

- The standard output stream is represented by predefined object **cout** and contents are streamed using the *insertion* operator <<.

```
cout << "Sum is " << sum << "\n";
```

- The input stream is represented by object **cin** and input is obtained by using >> *extraction* operator. The objects cin and cout are defined in header file **iostream**

```
cin >> a >> b;
```

What is Type casting? Give an example.

- Explicit conversion of variables or expression from one built-in data type to another is known as type casting.

datatype (*expression*)

```
average = float(a + b)/2;
```

Give the structure of C++ main() function

- The return type of main function in C++ is `int`.

```
int main()
{
      …
      return 0;
}
```

What is function prototyping?

- Function prototyping was introduced in C++ and is mandatory.
- Prototyping describes the function interface to the compiler.

returntype functionname (*arguments*)

```
{


}
```

What is an inline function.

- inline functions eliminate the cost of function calls for small routines by substitution.
- An inline function is a function preceded by keyword **inline** and is expanded inline when invoked.
- It is preferred over macro, since macro is not compiled.
- The keyword inline is a request and would be ignored if the function contains a loop, goto or switch statement or if it is recursive.

```
#include <iostream.h>

inline int square(int a)
{
      return (a*a);
}
```

```
int main()
{
     cout << "Square = " << square(5);
     return 0;
}
```

State the advantages of default arguments with an example program.

- C++ allows calling a function without specifying all its arguments by mentioning the default value for arguments in the prototype.
- When a function call is made with insufficient parameters, the compiler checks the prototype and uses the default value.
- Default value for parameters should be assigned from right to left in sequence.

```
#include <iostream.h>

void repchar(char='*', int=45);

int main()
{
     repchar();
     repchar('=');
     repchar('+', 30);
     return 0;
}

void repchar(char ch, int n)
{
     for(int j=0; j<n; j++)
        cout << ch;
     cout << endl;
}
```

Define class and object. Give syntax of the class. Explain the ways in which a member function can be defined with an example.

- Classes are extension to C structures.
- A class is a user-defined type that binds the data and its associated functions together.
- Data members are generally declared under private section and member functions under public section.
- Class members are private by default.
- Class supports OOP concepts such as polymorphism & inheritance

```
class classname
{
private:
     datamembers;
public:
     memberfunctions;
};
```

- Wrapping data and functions into a single unit is known as **encapsulation**.
- The private class members can be accessed only from within the class whereas public members can be accessed from outside the class.

- The data members under private section can be accessed only by member functions of that class and not externally is known as data **hiding**.
- Member functions can be defined outside the class using scope resolution **::** operator.

  returntype classname **::** *functionname* (*arguments*)
  {

  }

- Objects are instance of a class and are runtime entities.
- The dot **.** operator is used by an object to access public members of its class.

  classname objectname;

  objectname.publicmember

- Memory for data members are allocated separately for each object of that class and only one copy of member function resides in memory.

```
#include <iostream.h>

class rectangle
{
     int length;
     int breadth;
  public:
     void getdata();
     int area()
     {
         return (length * breadth);
     }
};

void rectangle::getdata()
{
     cin >> length >> breadth;
}

int main()
{
     rectangle r1;
     r1.getdata();
     cout << "Area: " << r1.area();
     return 0;
}
```

Distinguish between classes and structures.

Class	Structures
Members are private by default	Members are public. No private members
Contains data members and member	Contains variables only. No functions

functions	
Provides data abstraction and data hiding	No data hiding
Supports polymorphism and inheritance	Not designed to support OOP concepts.

What is the use of scope resolution operator?

- Scope resolution operator (`::`) defines the scope of variables and functions.
- Used to define member functions outside the class.
- Used to access global variables in a block, when there is a local variable of the same name.
- Static members are accessed using scope resolution operator

Define Constructor. List the different types of constructors. Write a program to illustrate the use of different types of constructors.

- A constructor is a member function whose name is same as class name and is used to initialize / allocate memory dynamically to data members.
- A constructor is automatically executed whenever objects are created.
- Some special characteristics of constructors are:
  - Declared under public section
  - Has no return type, not even `void`
  - Can be overloaded
  - Cannot be inherited / virtual
  - Makes implicit call using `new` for memory allocation
- Constructor can be classified into the following types:
  - *Default constructor*—Constructor that has no arguments. Used to initialize data members with default values.
  - *Parameterized constructor*—Constructor that takes arguments. Initializes data members with arguments passed.
  - *Copy constructor*—Constructor that takes object reference as argument. Used to initialize data members with the data member of another object
  - *Dynamic constructor*—Construct that allocates memory dynamically to data members using `new` operator.

```
// Constructor types

#include <iostream.h>

class distance
{
     int feet,
     int inch;
public:
     distance()                        // Default constructor
     {
          feet =  0;
          inch  =  0;
     }
     distance (int n, int d = 0)    //Parameterized constructor
     {
          feet =  n;
          inch  =  d;
     }
```

```
     distance (Distance &a)        // Copy constructor
     {
          feet =  a.feet;
          inch  =  a.inch;
     }
     void print()
     {
          cout << feet  << inch ;
     }
};

int main()
{
     distance d1, d2(4),  d3(22,7);
     distance d4(d2);
     d1.print();
     d3.print();
     d4.print();
}
```

What is a Destructor? State its purpose.

- A destructor is used to destroy objects when they go out of scope.
- Like constructor, destructor has the same name as class, but preceded by a tilde **~**.
- A destructor never takes any arguments nor does it return.

```
class mystring
{
public:
     ~mystring()          // Destructor
     {
          cout << "\n Memory released";
     }
};
```

- Destructor cannot be overloaded.
- Memory allocated dynamically in a constructor should be released using `delete` in the destructor.

Explain with an example the use of Static members

- When a data member is qualified `static`, it is initialized (`0` by default) when the first object is created.
- Only one copy of static data member exists and is shared by all objects.
- It's visible only within the class, but lifetime is the entire program.
- Static members are also called as class variables. Hence can be accessed using scope resolution operator.
- Static member functions can access only static data members.

```
#include <iostream.h>

class book
{
     static int count;        // static data member
```

```
public:
     book()
     {
          count++;
     }
     static void display()     // static member function
     {
          cout << "No. of copies : " << count;
     }
};

int book::count;     // define static variable

int main()
{
     book b1, b2, b3;
     b1.display();
     book::display();
     b3.display();
     return 0;
}
```

Explain the usage of const qualifier using example. What is the difference between pointer to a constant and constant pointers.

- The const qualifier to a variable is used to declare a symbolic constant.

  ```
  const float pi = 3.14;
  ```

- Argument to a function can be qualified as const. Functions cannot modify value of a const argument. Compiler error will be generated in case of violation.

  ```
  int length(const char* str);
  ```

- Argument for a copy constructor is usually qualified as const.
- Member functions can also be made const by appending function declaration with the qualifier const. `const` member functions do not alter the value of any data members. Compiler error will be generated in case of violation.

  ```
  void display() const
  {
       cout << length << breadth;
  }
  ```

- `const` qualifier can be used along with pointers. It has two different interpretation based on where it is placed.
- If the const qualifier precedes the declaration, then it is a pointer to a constant. A pointer to a constant cannot alter the value it points to.

  ```
  const char* src;     //src is a pointer to constant char
  ```

  The string contents pointed to by *src* cannot be altered, but address of *src* can be changed to point another string.
- A constant pointer is one, whose address cannot be changed but can alter the contents pointed to.

  ```
  int* const ptr;     //ptr is a constant pointer to int
  ```

- `const` can be used in both positions to make the pointer and what it points to, a constant.

```
#include <iostream.h>
int main()
{
     void copystr(char*, const char*);
     char* str1 = "Self-conquest is the greatest victory";
     char str2[80];
     copystr(str2, str1);
     cout << str2;
     return 0;
}

void copystr(char* dest, const char* src)
{
     while( *src )
          *dest++ = *src++;
     *dest = '\0';
}
```

Explain pointer variable with an example.

- Pointer is a derived data type.
- Pointer variables refer to another variable by storing the address of that variable.
- A pointer variable can point to another pointer (double pointer).
- Pointer variables are preceded by an `*` in the declaration.

  *datatype *pointer_variable;*
- Pointer variable is assigned to the address of another variable using `&` (address-of) operator.

```
int *ptr, a;          // Pointer variable declaration
ptr = &a;             // Address assignment
```

- Both the pointer and pointee should be of the same type.
- Pointers of type void are known as *generic* pointers. A generic pointer can hold the address of any variable.
- Pointers are used to:
  - Access array elements
  - Passing arguments when the function needs to modify the actual argument
  - Passing arrays and strings to functions
  - Obtaining memory from the system
  - Creating data structures such as linked lists
- Data pointed to by a pointer variable can be manipulated using the indirection operator `*` (dereference operator).
- Pointer variables can be subjected to arithmetic expression such as increment, decrement, addition or subtraction.

```
#include <iostream.h>

int main()
{
```

```
     int var1 = 11;
     int var2 = 22;
     int* ptr;                //pointer variable
     ptr = &var1;             //pointer points to var1
     cout << *ptr;             //prints 11
     ptr = &var2;             //pointer points to var2
     cout << *ptr;             //prints 22
     return 0;
}
```

Pointers and Arrays

- Pointers can be used to access and manipulate array elements efficiently.
- Base address (starting address) of an array is assigned to a pointer variable.
- Pointer variable is incremented to access the subsequent array elements.
- An array of pointers can be created

```
#include <iostream.h>

int main()
{
     int arr[]={31, 54, 77, 52, 93};
     int *ptr, sum=0;
     ptr = arr;                          //points to array
     for(int j=0; j<5; j++)
     {
          sum = sum + *ptr;
          ptr++;
     }
     cout << sum;
     return 0;
}
```

Pointers and Functions

- When a function is required to modify actual arguments, the corresponding formal parameters are of pointer type.

```
#include <iostream.h>

void swapptr(int*, int*);

int main()
{
     int a=10, b=20;
     swapptr (&a, &b);   // Pass using pointers
     cout << a << b ;
}

void swapptr(int *x, int *y)
{
     int t = *x;
     *x = *y;
     *y = t;
}
```

Explain reference variable with an example. Compare reference variables and pointers.

- A reference variable is an alias for another variable.
- Reference variables are preceded by `&` in the declaration.
- It must be initialized at the time of declaration itself.

datatype `&`*refname* `=` *varname;*

```
float total = 3.7;
float &sum = total;
cout << sum;
```

- Both reference variable and actual variable point to the same memory location.
- Changes made to the reference variable are reflected in the original variable.
- Parameters can be passed to a function by reference instead of using pointers. In pass-by-reference, reference to the original variable is passed, i.e., the function accesses the actual arguments.

```
#include <iostream.h>

void swapref(int&, int&);

int main()
{
     int a=10, b=20;
     swapref (a, b);      // Pass by reference
     cout << a << b ;
}

void swapref(int &x, int &y)
{
     int t = x;
     x = y;
     y = t;
}
```

Explain the role of this pointer with a suitable program.

- When a member function is called, `this` pointer is automatically passed an implicit argument, i.e., a pointer to the invoking object.
- It is used by the member function to identify the object that invoked the function. Thus any member function can find out the address of the object.
- It acts as an implicit argument to all member function. `this` pointer like any other pointer to an object can be used to access the data in the object it points to.
- *this* pointer is used to return value from member functions and overloaded operators.
- When overloading a binary operator using member function, one of the operands is implicitly passed using `this` pointer.
- Objects can be returned by reference using `this` pointer rather than creating and returning a temporary object.

```
#include <iostream.h>
#include <string.h>
```

```
class person
{
     char name[25];
     float exp;
public:
     person (char *s, float e)     {
          strcpy(name, s);
          exp = e;
     }
     person& greater(person &p)    // return by reference
     {
          if (p.exp > this->exp)
               return p;
          else
               return *this;
     }
     void display()
     {
          cout <<  name <<  exp;
     }
};

int main()
{
     person P1("John", 5);
     person P2("Ram",7);
     person P3 = P1.greater(P2);
     P3.display();
     return 0;
}
```

What is scope of a variable? (or) Distinguish between local and global variables.

- Variables have two scope, namely local and global (file).
- Variables declared within a block `{}` are *local* variables and visible only to that block.
- Variables that are not declared within any block are *global* or *file* variables. It is visible throughout the file or program.

```
int g;          // global variable
int main()
{
    int x;      // local variable
}
```

Briefly explain the various storage classes specifiers.

- C++ supports storage specifier *mutable* in addition to storage classes *extern*, *static*, *register* and auto supported by C.
- Storage specifiers tell the compiler how to store a variable.

storage_specifier data_type var_name;

extern

- Global variables declared in one file can be used in another file using extern specifier.

- The *extern* specifier tells the compiler about global variable types and names defined elsewhere, without creating storage for them once again.

  `extern` *`global_variables;`*
- Allows global variables compiled in one module to be linked and used in other modules of a large program.

static

- static variables are permanent variables in the file in which it is declared.
- Unlike global variables, it is not known outside the function or file, but maintain their values between calls.
- Static can be applied to both local variables and global variables.

register

- register storage specifier is applied only to variables of type int, char, or pointer types.
- Requests the compiler to store variable's value in CPU register, rather than in memory. Thus operations on a register variable are much faster.
- register specifier can be applied to local variables only.

auto

- auto specifier is the default storage class.
- Variables are automatically created when a function is called and automatically destroyed when it returns.
- Automatic variables defined within a block, lose their meaning when control leaves the block.

mutable

- A mutable data member allows a const member function to alter its value.
- A const member function cannot alter a variable value, except if it is a mutable variable.

```
#include <stdio.h>

extern int g;       // Global variable

void recall()
{
     static int s;       // static variable
     cout << "Static variable : " << s;
     g++;
     s++;
}

int main()
{
     register int r;     // register variable
     {
          int y;         // automatic variable
          cout << "Local variable : " << y;
     }
     cout << "Global variable : " << g;
     recall();
     recall();
}
```

What is function pointer? Write a program that passes function as an argument.

- Pointers to functions or function pointers refer to a function. It is also known as callback function.
- Functions have a physical location in memory that can be assigned to a pointer.
- Once a pointer points to a function, the function can be called through that pointer.
- Function pointers also allow functions to be passed as arguments to other functions.
- Function pointers allow a function to be selected dynamically at runtime. The set of functions must have the same return type and same set of argument types.

  datatype `(*` *functionpointer*`)(`*arguments*`);`
- It is also used in event-based applications such as GUI.
- The function pointed to can be either static or non-static.

```
#include <iostream.h>

typedef void (*foo)(int, int);      // function pointer

void add(int i, int j)
{
     cout << i+j;
}

void sub(int i, int j)
{
     cout << i-j;
}

int main()
{
     foo fptr;
     fptr = add;
     fptr(1,2);
     fptr = sub;
     fptr(5, 4);
     return 0;
}
```