OBJECT ORIENTED PROGRAMMING

# AGENDA:

OOPS – CLASSES - STRUCTURES - UNION– OBJECTS – ENCAPSULATION – ABSTRACTION – POLYMORPHISM – INHERITANCE

# OBJECT ORIENTED PROGRAMMING:

* It refers to the process that uses objects in programming.
* Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming.
* The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
* Classes and Structs are the constructs whereby you define your own types. They can both contain data members and member functions, which enable to describe the type’s status and behaviour.
* The three class types are:
  + CLASS
  + STRUCTURE
  + UNION

return 0;

}

PS D:\Zoho Corporation\May 03\OOPS> g++ prog1.cpp

PS D:\Zoho Corporation\May 03\OOPS> ./a

Sri 15 4 1

# STRUCT:

The **struct** keyword declares a structure type and/or a variable of a structure type.

[template-spec] struct [ms-decl-spec] [tag [: base-list ]]

{

member-list

} [declarators];

[struct] tag declarators;

* **template-spec:** Optional Template Specification
* **struct:** Keyword
* **ms-decl-spec:** Optional Storage-Class Specification
* **tag:** Type Name of Class, if omitted **Anonymous Structure is used.**
* **base-list:** The list of Base Classes
* **member-list:** The list of Data Members and Member Functions
* **declarators:** The Class Instances or Objects.

## Example:

#include <iostream>

using namespace std;

struct PERSON

{ // Declare PERSON struct type

int age; // Declare member types

long ss;

float weight;

char name[25];

} family\_member;

// Define object of type PERSON

struct CELL

{ // Declare CELL bit field

unsigned short character : 8; // 00000000 ????????

unsigned short foreground : 3; // 00000??? 00000000

unsigned short intensity : 1; // 0000?000 00000000

unsigned short background : 3; // 0???0000 00000000

unsigned short blink : 1; // ?0000000 00000000

} screen[25][80];

// Array of bit fields

int main()

{

struct PERSON sister; // C style structure declaration

PERSON brother; // C++ style structure declaration

sister.age = 13; // assign values to members

brother.age = 7;

cout << "sister.age = " << sister.age << '\n';

cout << "brother.age = " << brother.age << '\n';

CELL my\_cell;

my\_cell.character = 1;

cout << "my\_cell.character = " << my\_cell.character;

}

PS D:\Zoho Corporation\May 03\OOPS> g++ prog2.cpp

PS D:\Zoho Corporation\May 03\OOPS> ./a

sister.age = 13

brother.age = 7

my\_cell.character = 1

# CLASS:

The **class** keyword declares a class type or defines an object of a class type.

[template-spec]

class [ms-decl-spec] [tag [: base-list ]]

{

member-list

} [declarators];

[ class ] tag declarators;

* **template-spec:** Optional Template Specification
* **ms-decl-spec:** Optional Storage-Class Specification
* **tag:** Type Name of Class
* **base-list:** The list of Base Classes
* **member-list:** The list of Data Members and Member Functions
* **declarators:** The Class Instances or Objects.

## Example:

#include <iostream>

#include <string>

using namespace std;

class dog

{

private:

string name;

int size, legs;

bool bark;

public:

dog()

{

legs = 4;

bark = true;

}

void set (string name, int size)

{

this->name = name;

this->size = size;

}

void get ()

{

cout << name << '\t' << size << '\t' << legs << '\t' << bark;

}

};

int main ()

{

dog d;

d.set ("Sri", 15);

d.get();

return 0;

}

PS D:\Zoho Corporation\May 03\OOPS> g++ prog1.cpp

PS D:\Zoho Corporation\May 03\OOPS> ./a

Sri 15 4 1

# CLASS MEMBER OVERVIEW:

A class or struct consists of its members.

The work that a class does is performed by its member functions. The state that it maintains is stored in its data members.

Initialization of members is done by constructors, and clean up work (freeing of memory and resource release) is done by destructors.

## Kind of Class Members:

* Special member functions.
* Overview of member functions.
* Data members including built-in types and other user defined types.
* Operators
* Nested class declarations and.)
* Unions
* Enumerations.
* Bit fields.
* Friends. (They are not true class members as they are not within the scope of the class.)
* Aliases and typedefs.

## Member Accessibility:

The members of a class are declared in the member list, which may be divided into any number of **private, protected and public** sections.

It designates the access of the members up until the next access specifier or the closing brace.

## Special Member Functions:

They are provided by the compiler automatically, if they are not specified explicitly.

* Default Constructor
* Copy Constructor
* Move Constructor
* Copy Assignment Operator
* Move Assignment Operator
* Destructor

## Member-Wise Initialization:

Non-static members can be initialized.

If a member is assigned a value in the constructor, that value overwrites the initialized values by the constructor parameters.

There is only one shared copy of the static data members for all objects of a given class type. Static data members must be defined and initialized at the file scope.

## Friends:

In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all members in a separate class. Only the class implementer can declare who its friends are. A function or class cannot declare itself as a friend of any class.

If you declare a friend function that was not previously declared, that function is exported to the enclosing nonclass scope.

Functions declared in a friend declaration are treated as if they had been declared using the extern keyword.

Although functions with global scope can be declared as friends prior to their prototypes, member functions cannot be declared as friends before the appearance of their complete class declaration.

friend class F; 🡪 It introduces a new class F if no existing class by that name was found in the innermost namespaces.

friend F; 🡪 it does not introduce a new class. It can be used when the class has already been declared, and it must be used when declaring a template type parameter.

## Friend Functions:

A friend function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges.

Friends are not in the class's scope, and they are not called using the member-selection operators (. and ->) unless they are members of another class. A friend function is declared by the class that is granting access. The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

#include <iostream>

using namespace std;

class Point

{

friend void ChangePrivate(Point &);

public:

Point(void) : m\_i(0) {}

void PrintPrivate(void)

{

cout << m\_i << endl;

}

private:

int m\_i;

};

void ChangePrivate(Point &i)

{

i.m\_i++;

}

int main()

{

Point sPoint;

sPoint.PrintPrivate();

ChangePrivate(sPoint);

sPoint.PrintPrivate();

}

PS D:\Zoho Corporation\May 03\OOPS> g++ prog4.cpp

PS D:\Zoho Corporation\May 03\OOPS> ./a

0

1

# OBJECT LIFETIME AND RESOURCE MANAGEMENT:

Unlike managed languages, C++ doesn't have automatic garbage collection. That's an internal process that releases heap memory and other resources as a program runs. A C++ program is responsible for returning all acquired resources to the operating system. Failure to release an unused resource is called a leak.

Leaked resources are unavailable to other programs until the process exits. Memory leaks in particular are a common cause of bugs in C-style programming.

Modern C++ avoids using heap memory as much as possible by declaring objects on the stack. When a resource is too large for the stack, then it should be owned by an object. As the object gets initialized, it acquires the resource it owns. The object is then responsible for releasing the resource in its destructor. The owning object itself is declared on the stack.

The principle that objects own resources is also known as **"resource acquisition is initialization," or RAII.**

When a resource-owning stack object goes out of scope, its destructor is automatically invoked. In this way, garbage collection in C++ is closely related to object lifetime, and is deterministic. A resource is always released at a known point in the program, which you can control. Only deterministic destructors like those in C++ can handle memory and non-memory resources equally.

The following example shows a simple object w . It's declared on the stack at function scope, and is destroyed at the end of the function block. The object w owns no resources (such as heap-allocated memory). Its only member g is itself declared on the stack, and simply goes out of scope along with w . No special code is needed in the widget destructor.

# COMPILE-TIME ENCAPSULATION:

The pimpl idiom is a modern C++ technique to hide implementation, to minimize coupling, and to separate interfaces.

Pimpl is short for "pointer to implementation." You may already be familiar with the concept but know it by other names like Cheshire Cat or Compiler Firewall idiom.

* Minimization of Compilation Dependencies
* Seperation of Interface and Implementation
* Portability

# ENCAPSULATION:

Encapsulation is defined as binding together the data and the functions that manipulates them.

Encapsulation also leads to data abstraction or hiding. As using encapsulation also hides the data.

#include <iostream>

using namespace std;

class encap

{

private:

int x;

public:

void set (int a) { x = a; }

int get () { return x; }

};

int main ()

{

encap a;

a.set(25);

cout << a.get();

return 0;

}

PS D:\Zoho Corporation\May 03\OOPS> g++ prog5.cpp

PS D:\Zoho Corporation\May 03\OOPS> ./a

25

The process of implementing encapsulation can be sub-divided into two steps:

* The data members should be labeled as private using the private access specifiers
* The member function which manipulates the data members should be labeled as public using the public access specifier.

# ABSTRACTION:

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details.

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

* Members declared as public in a class, can be accessed from anywhere in the program.
* Members declared as private in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

#include <iostream>

using namespace std;

class implementAbstraction

{

private:

int a, b;

public:

void set(int x, int y)

{

a = x;

b = y;

}

void display()

{

cout<<"a = " <<a << endl;

cout<<"b = " << b << endl;

}

};

int main()

{

implementAbstraction obj;

obj.set(10, 20); obj.display();

return 0;

}

# POLYMORPHISM:

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type.

Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. In your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

## Compile-Time Polymorphism:

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

* Function Overloading
* Operator Overloading

## Run-Time Polymorphism:

Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

# INHERITANCE:

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

## SINGLE INHERITANCE:

 It is a common form of inheritance. There exists only one base class and only one derived class.

The base class from which each class is derived is declared before the declaration of the derived class. It is not sufficient to provide a forward-referencing declaration for a base class; it must be a complete declaration.