**Aromal S Kunnel**

**22BAI10288**

**ASSIGNMENT-I**

|  |  |  |  |
| --- | --- | --- | --- |
| **Branch/Semester** | **B.Tech/Fall semester** | **Session** | **2024-2025** |
| **Name of Faculty** | **Dr. Jitendra P S Mathur** | **Subject** | **Object Oriented Programming With C++** |
| **Module** | **1** | **Sub Code** | **CSE-2001** |
| **Last date of Submission** | **11.0.2024** |  | **Through Google classroom** |

|  |  |  |
| --- | --- | --- |
| **S.No** | **Questions** | **CO Attainment** |
| 1 | Explain the concept of polymorphism by an example in C++. | CO.1 |
| 2 | Define Object-oriented programming and Explain feature of Object oriented programming. How it is different than procedure oriented programming. | CO.1 |
| 3 | Classify the different statements available in C++ with examples. | CO.1 |
| 4 | Write a program in C++ that display entered string into reverse order. | CO.1 |
| 5 | Discuss the data types in C++ programming. | CO.1 |

A1. Polymorphism is a fundamental concept in C++ that enables the creation of flexible, reusable, and extensible code.

**Inheritance**: Polymorphism leverages inheritance hierarchies. Base classes define interfaces (pure virtual functions) that derived classes must implement. These functions serve as the common entry point for polymorphic behaviour.

**Dynamic Binding**: During runtime, the actual implementation of the virtual function called depends on the object's type. This dynamic selection ensures the appropriate behaviour for each derived class object.

#include <iostream>

class Animal {

public:

  virtual void speak() const {

    std::cout << "Animal sound..." << std::endl;

  }

};

class Dog : public Animal {

public:

  void speak() const override {

    std::cout << "Woof!" << std::endl;

  }

};

class Cat : public Animal {

public:

  void speak() const override {

    std::cout << "Meow!" << std::endl;

  }

};

void makeAnimalSpeak(const Animal& animal) {

  animal.speak();

}

int main() {

  Animal animal;

  Dog dog;

  Cat cat;

  animal.speak();

  makeAnimalSpeak(dog);

  makeAnimalSpeak(cat);

  return 0;

}

A screenshot of a computer

Description automatically generated

A2. OOP is a programming paradigm that organizes software design around "objects" that encapsulate data (properties) and the code that operates on that data (methods or functions). Objects interact with each other through well-defined interfaces, promoting modularity and reusability. Here are the core features of OOP:

Objects: Objects are fundamental building blocks in OOP. They represent real-world entities or concepts with attributes (data) and behaviors (methods). An object's state is defined by its attributes, and its behavior is defined by its methods.

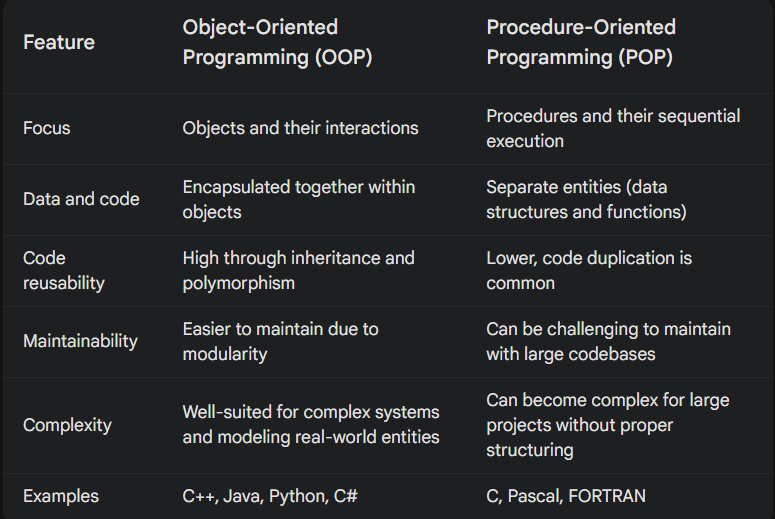
Classes: Classes act as blueprints for creating objects. They define the properties and methods that objects of that class will share. Classes provide a way to create multiple objects with similar characteristics and functionalities.

Inheritance: Inheritance allows new classes (derived classes) to inherit properties and methods from existing classes (base classes). This promotes code reuse and enables the creation of class hierarchies with increasing specialization.

Polymorphism: Polymorphism allows objects of different classes to respond differently to the same message (method call). This can be achieved through virtual functions (runtime polymorphism) or function overloading (compile-time polymorphism). Polymorphism fosters flexibility and adaptability in code.

Encapsulation: Encapsulation bundles data (attributes) and methods (functions) together within an object, restricting direct access to the data and controlling its manipulation through methods. This promotes data integrity and security.

Abstraction: Abstraction focuses on the essential aspects of an object, hiding its implementation details from the outside world. Users interact with objects through their public methods, without needing to know the internal workings. This simplifies code and promotes maintainability.



A3.

1. Declaration Statements

These statements introduce names (variables, functions, classes) into your program. They don't directly perform any actions but allocate memory or establish definitions.

2. Expression Statements

These statements evaluate an expression (a combination of variables, operators, function calls) and might produce a result, but don't explicitly control flow.

3. Selection Statements

These statements make decisions and control program flow based on conditions.

4. Iteration Statements

These statements create loops that execute a block of code repeatedly until a certain condition is met.

5. Jump Statements

These statements abruptly transfer control to another part of the code.

* break Statement: Used to exit a loop prematurely.
* continue Statement: Used to skip the remaining part of the current loop iteration and move to the next iteration.
* goto Statement: Considered outdated in modern C++ due to potential for spaghetti code. Use with caution.

6. Compound Statements

These statements group multiple statements together into a single block, often enclosed in curly braces {}.

#include <iostream>

int age;

double pi = 3.14159;

void greet(std::string name);

int calculateArea(int length, int width);

void greet(std::string name) {

  std::cout << "Hello, " << name << "!" << std::endl;

}

int calculateArea(int length, int width) {

  return length \* width;

}

int main() {

  age = 30;

  if (age >= 18) {

    greet("User");

  } else {

    std::cout << "You are not an adult yet." << std::endl;

  }

  for (int i = 1; i <= 5; i++)

A4.

#include <iostream>

#include <string>

using namespace std;

int main() {

  string inputString;

  cout << "Enter a string: ";

  getline(cin, inputString);

  string reversedString;

  for (int i = inputString.length() - 1; i >= 0; i--) {

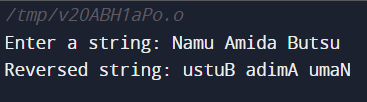
    reversedString += inputString[i];

  }

  cout << "Reversed string: " << reversedString << endl;

  return 0;

}



A5. Fundamental Data Types:

Integer Types:

int: Most common integer type, typically 4 bytes on modern systems. Stores whole numbers (positive, negative, or zero). Example: int age = 25;

short: Smaller integer type, often 2 bytes. Useful for memory optimization when whole numbers within a limited range are needed. Example: short temperature = -10;

long: Larger integer type, often 4 or 8 bytes. Used for storing larger whole numbers. Example: long population = 1000000;

long long: Even larger integer type, often 8 bytes. Used for very large whole numbers. Example: long long distanceInLightYears = 9461e12;

Floating-Point Types:

float: Stores single-precision floating-point numbers (numbers with decimal points). Less precise but smaller memory footprint. Example: float pi = 3.14159f; (adding 'f' suffix indicates a float literal)

double: Stores double-precision floating-point numbers, more precise than float but uses more memory. Example: double avogadrosNumber = 6.02214e23;

Character Type:

char: Stores a single character (e.g., 'a', 'Z', '$'). Example: char initial = 'J';

Boolean Type:

bool: Represents logical values: true or false. Example: bool isLoggedIn = true;

Void Type:

void: Represents the absence of a value. Often used for functions that don't return anything. Example: void greetUser() { std::cout << "Hello!" << std::endl; }

Derived Data Types:

Arrays:

Collections of elements of the same data type, accessed using an index (position). Example: int numbers[5] = {10, 20, 30, 40, 50};

Pointers:

Variables that store memory addresses of other variables. They enable dynamic memory allocation and efficient data manipulation. Example: int\* ptr = &age; // ptr points to the memory address of age

References:

Aliases for existing variables, providing another way to access the same data. Example: int& refToAge = age; // refToAge refers to the same memory location as age

Structures (Structs):

User-defined data types that group variables of different data types under a single name.

Unions:

Similar to structs but can hold only one value at a time, with the data type of the value changing depending on what's assigned. Example (less common than structs):