NAME	
ROLL NUMBER	
SEMESTER	**2nd**
COURSE CODE	**DCA1203**
COURSE NAME	**Object Oriented Programming – C++**

SET - I

Q.1) Describe the various datatypes available in C++?

Answer .:-

C++ provides a set of basic building blocks for storing different kinds of data:

- **Integer Types:**
  - int: Represents whole numbers (typically 4 bytes on most systems).
  - short: Stores smaller integers (often 2 bytes).
  - long: Holds larger integers (usually 4 bytes, but can be 8 bytes on some architectures).
  - long long: Can store very large integers (typically 8 bytes).

  Choosing the appropriate integer type depends on the range of values you need to represent and memory efficiency considerations.
- **Floating-Point Types:**
  - float: Stores single-precision floating-point numbers (decimal numbers with a fractional part, typically 4 bytes).
  - double: Provides double-precision floating-point numbers (more accurate than float, usually 8 bytes).
  - long double: Offers extended-precision floating-point numbers (highest precision, size can vary).

  Use float for most calculations unless you require very high precision or a wider range, in which case double or long double might be necessary.
- **Character Types:**
  - char: Represents a single character (usually 1 byte).
  - wchar_t: Stores a wide character for Unicode support (often 2 bytes).
  - char16_t and char32_t: Support 16-bit and 32-bit Unicode characters, respectively.

  char is suitable for storing basic ASCII characters, while wchar_t and the Unicode character types are used for representing characters from a wider range of languages.
- **Boolean Type:**
  - bool: Represents logical values (true or false, typically 1 byte).

  Use bool to store Boolean values for conditional statements and other logical operations.

- **Void Type:**
    - void: Indicates the absence of a specific data type.

  void is used in function declarations to specify that the function doesn't return a value, and for function parameters that don't require any arguments.

Derived Data Types:

C++ extends the functionality of fundamental datatypes by providing mechanisms to create more complex data structures:

- **Arrays:**
    - A fixed-size collection of elements of the same data type.
    - Example: int numbers[10] declares an array of 10 integers.
- **Pointers:**
    - Variables that store memory addresses of other variables.
    - Allow for dynamic memory allocation and efficient manipulation of data structures.
    - Example: int* ptr = &number; creates a pointer ptr that points to the memory location of the integer variable number.
- **References:**
    - Aliases for existing variables, providing another way to access the same memory location.
    - Useful for passing arguments to functions by reference, avoiding unnecessary copying of data.
    - Example: int x = 10; int& ref = x; creates a reference ref that refers to the same memory location as x.
- **Enumerated Types (enums):**
    - User-defined data types consisting of named integer constants.
    - Improve code readability by using meaningful names instead of raw integer values.
    - Example: enum Color { RED, GREEN, BLUE }; defines an enum Color with constants RED, GREEN, and BLUE.
- **Structures (structs):**
    - Composite data types that group variables of different data types under a single name.
    - Useful for representing real-world entities with various attributes.

- Example: struct Point { int x; int y; }; defines a struct Point with two integer members x and y.

- **Unions:**
  - Similar to structs, but all members share the same memory location.
  - Useful when you know only one member will have a value at a time.
  - Example: union Data { int num; float value; }; defines a union Data where either num or value can be used, but not both simultaneously.
- **Classes:**
  - Fundamental building blocks of object-oriented programming in C++.
  - Encapsulate data (member variables) and behavior (member functions) within a blueprint.
  - Objects are instances of classes that provide data storage and operations specific to that object type.

Q.2) What is the difference between the do-while and the while statements?

Answer .:- The key difference between do-while and while loops in C++ lies in when the loop's condition is checked:

do-while loop:

- Executes the loop body **at least once**, regardless of the initial condition.
- Checks the condition **after** the body has executed.
- Continues looping as long as the condition is true.

while loop:

- Checks the condition **before** executing the loop body.
- Executes the body only if the condition is true initially.
- Continues looping as long as the condition remains true.

Here's a table summarizing the key points:

Feature	do-while loop	while loop
Condition check	After the body execution	Before the body execution
Minimum runs	At least once	Zero times (if condition is false initially)

Example:

C++

```
int count = 10;
```

```
do {
  // Code to be executed at least once
  std::cout << count << " ";
  count--;
} while (count > 0);

while (count > 0) {
  // This code might not execute at all
  std::cout << count << " ";
  count--;
}
```

In this example:

- The do-while loop will always print the numbers from 10 down to 1, because it executes the body at least once (even though count is initially 10).
- The while loop might not print anything at all if count happens to be 0 before the loop begins.

When to use which:

- Use do-while when you **guarantee** the loop body needs to execute at least once, regardless of the initial condition.
    - Common scenarios include initialization steps or user input validation.
- Use while when the loop body might not need to execute at all, depending on the initial condition.
    - This is suitable for traditional iterative tasks where the loop continues as long as a certain condition holds.

Q.3) Brief about class and objects.

Answer .:- C++, being an object-oriented programming (OOP) language, revolves around the concepts of classes and objects. These concepts provide a powerful way to model real-world entities and their interactions within your programs.

Classes: Blueprints for Objects

- A class acts as a blueprint or template that defines the properties (data) and behaviors (functions) of similar objects.
- It encapsulates data members (variables) and member functions (methods) that operate on that data.

- Think of a class like a cookie cutter – it defines the shape and basic characteristics of the cookies you'll create, but the individual cookies themselves are the objects.

Creating a Class:

C++

```
class Car {
public:
  // Data members (properties)
  int wheels;
  std::string color;

private:
  // Member functions (behaviors)
  void accelerate() {
    // Code to increase speed
  }

  void brake() {
    // Code to slow down
  }
};
```

- In this example, the Car class defines two data members: wheels (an integer) and color (a string).
- It also has two private member functions, accelerate() and brake(), which presumably control the speed of the car but their implementation details are hidden within the class.

Objects: Instances of a Class

- An object is a concrete instance of a class. It represents a particular entity with its own set of properties and behaviors defined by the class.
- Just like baking multiple cookies from a single cookie cutter, you can create many Car objects from the Car class.

Creating Objects:

C++

```
Car myCar; // Create an object named myCar of class Car
myCar.wheels = 4; // Set the wheels property of myCar
myCar.color = "red"; // Set the color property of myCar
myCar.accelerate(); // Call the accelerate() function on myCar
```

- Here, we create an object named myCar of type Car.
- We can then access and modify the data members (wheels and color) of myCar using the dot notation.
- We can also call the member functions (accelerate()) on the object to invoke its behavior.

Benefits of Using Classes and Objects:

- **Encapsulation:** Classes promote data hiding by restricting direct access to data members. Member functions control how data is accessed and modified, ensuring data integrity.
- **Code Reusability:** Classes allow you to create reusable code templates. You can define a class once and create multiple objects with the same functionality.
- **Modular Design:** Classes help break down complex programs into smaller, manageable units, making code easier to understand, maintain, and modify.
- **Real-World Modeling:** Classes and objects provide a natural way to represent real-world entities and their interactions in your program, leading to more intuitive and maintainable code.

SET - II

Q.4) Define exception. Explain exception handling mechanism.

Answer .:- **Exceptions and Exception Handling in C++**

Exceptions:

- In C++, exceptions represent unexpected or erroneous conditions that arise during program execution. These are typically runtime errors that the program can potentially recover from, unlike traditional program crashes.
- Examples of exceptions include:
    - Division by zero
    - Attempting to access an array element out of bounds
    - Opening a file that doesn't exist
    - Memory allocation failures

Exception Handling Mechanism:

C++ provides a structured approach to handling exceptions using three keywords: try, catch, and throw. This mechanism allows you to isolate and deal with exceptional circumstances gracefully, preventing program crashes and improving program robustness.

1. The try Block:

- The try block identifies the code section where an exception might potentially occur.
- Any code within the try block is monitored for exceptions.

2. The throw Statement:

- When an exceptional condition is encountered within the try block, a throw statement is used to signal the error.
- The throw statement can optionally throw an object (an exception object) that carries additional information about the error.

3. The catch Block:

- One or more catch blocks follow the try block.
- Each catch block specifies the type of exception it can handle.
- When an exception is thrown, the program searches for the first catch block that can handle the type of exception thrown.
- If a matching catch block is found, the code within that block is executed to deal with the exception.

Example:

C++

```
#include <iostream>

int main() {
  try {
    int num1 = 10;
    int num2 = 0;
    int result = num1 / num2; // Potential division by zero exception
    std::cout << "Result: " << result << std::endl;
  } catch (const std::exception& e) {
    std::cerr << "Error: " << e.what() << std::endl; // Handle the exception
  }

  return 0;
}
```

In this example:

- The try block contains the code that might cause an exception (division by zero).
- The catch block is designed to catch any std::exception (or a derived class of it).
- If a division by zero exception occurs, the catch block will be executed, printing an error message to the standard error stream (std::cerr).

Benefits of Exception Handling:

- **Improved Program Robustness:** Exception handling allows programs to recover from errors gracefully, preventing crashes and maintaining program execution.
- **Error Isolation:** Exceptions help localize error handling code, making the program easier to understand and maintain.
- **Cleaner Code:** By separating normal program flow from error handling, exceptions lead to more readable and maintainable code.

Q.5) List and explain the STL components.

Answer .:- The C++ Standard Template Library (STL) provides a powerful collection of components that simplify data storage, manipulation, and algorithmic operations.

1. Containers:

- Containers are objects that hold collections of elements of the same or similar data types.
- The STL offers a variety of containers, each with its own strengths and use cases:

- **Vectors:** Dynamic arrays that allow random access and efficient insertions/deletions at the end (good for frequent appends/removals).
- **Lists:** Doubly-linked lists that enable efficient insertions/deletions at any position (good for frequent insertions/removals anywhere).
- **Deques (Double-ended queues):** Similar to vectors but allow insertions/deletions at both ends efficiently (good for FIFO/LIFO operations).
- **Sets:** Unordered collections of unique elements (useful for storing unique values).
- **Multisets:** Unordered collections that can store duplicate elements (useful for keeping track of element frequency).
- **Maps:** Associative containers that store key-value pairs, where each key is unique and maps to a value (efficient for lookups based on keys).
- **Multimaps:** Similar to maps but allow duplicate keys (useful for storing multiple values associated with a single key).
- **Unordered sets and maps:** Hash table-based versions of sets and maps, offering faster average-case performance for lookups and insertions (but with slightly less predictable behavior).

2. Iterators:

- Iterators act as pointers to elements within containers.
- They provide a way to traverse (visit) elements in a container and access their values.
- Different iterator categories exist, such as input iterators (read-only access), output iterators (write-only access), forward iterators (one-directional traversal), and bidirectional iterators (two-directional traversal).
- Iterators work seamlessly with algorithms, allowing them to operate on elements within containers without knowing the specific container type.

3. Algorithms:

- The STL provides a rich set of generic algorithms that operate on containers and iterators.
- These algorithms perform common operations like sorting, searching, finding minimum/maximum elements, copying, transforming elements, and more.
- Algorithms are designed to be generic and work with various container and iterator types, promoting code reusability.

4. Functors (Function Objects):

- Functors are objects that can be called like functions.
- They are used to customize the behavior of algorithms by providing custom logic for comparisons, transformations, or other operations.
- Functors are particularly useful when you need to define a specific behavior for an algorithm without writing a separate function.

Benefits of Using STL:

- **Reduced Coding Effort:** STL provides pre-built, efficient data structures and algorithms, saving you time and effort compared to implementing them from scratch.
- **Improved Code Readability:** The use of generic algorithms and iterators leads to cleaner and more concise code.
- **Type Safety:** STL components enforce type safety, reducing the risk of runtime errors.
- **Flexibility and Reusability:** STL components are designed to be generic and work with various data types, promoting code reusability.

Q.6) Explain the types of methods to open a file.

Answer .:- In C++, there are several methods to open a file for different purposes, each offering specific control over how you interact with the file. Here's a breakdown of the common methods using the fstream class (which combines functionalities of ifstream for reading and ofstream for writing):

1. Opening for Reading (ios::in):

- This method opens the file for reading existing data.
- Any attempt to write to the file will result in an error.
- Example:

C++

```
#include <fstream>
#include <iostream>

int main() {
  std::ifstream file("data.txt", std::ios::in); // Open "data.txt" for reading

  if (file.is_open()) {
    std::string line;
    while (std::getline(file, line)) {
      std::cout << line << std::endl; // Read and print each line
    }
```

```
  file.close(); // Close the file
 } else {
  std::cerr << "Error: Could not open file." << std::endl;
 }

 return 0;
}
```

2. Opening for Writing (ios::out):

- This method opens the file for writing.
- If the file doesn't exist, it will be created.
- Existing content in the file will be overwritten.
- Example:

C++

```
#include <fstream>
#include <iostream>

int main() {
 std::ofstream file("output.txt", std::ios::out); // Open "output.txt" for writing

 if (file.is_open()) {
  file << "This is some text to write to the file." << std::endl;
  file.close(); // Close the file
 } else {
  std::cerr << "Error: Could not open file." << std::endl;
 }

 return 0;
}
```

3. Opening for Appending (ios::app):

- This method opens the file for appending data to the end of its existing content.
- If the file doesn't exist, it will be created.
- New data will be written at the end of the file, preserving existing content.
- Example:

C++

```
#include <fstream>
#include <iostream>

int main() {
 std::ofstream file("log.txt", std::ios::app); // Open "log.txt" for appending

 if (file.is_open()) {
  file << "Adding a new log entry." << std::endl;
  file.close(); // Close the file
 } else {
```

```
  std::cerr << "Error: Could not open file." << std::endl;
 }

 return 0;
}
```

4. Opening in Binary Mode (ios::binary):

- This mode is used for opening binary files (like images or executables).
- It ensures that the file contents are read or written byte-for-byte without any character interpretation.
- Often used in combination with ios::in or ios::out for binary data handling.
- Example (reading a binary image file):

```
C++
#include <fstream>
#include <iostream>

int main() {
 std::ifstream file("image.bmp", std::ios::binary | std::ios::in); // Open "image.bmp" in binary mode for reading

 // ... (read image data in binary format)

 file.close(); // Close the file
}
```

5. Opening with Truncation (ios::trunc):

- This mode, often used with ios::out or ios::app, discards any existing content in the file before opening it.
- Similar to creating a new file, but the existing file path is used.
- Example:

```
C++
#include <fstream>
#include <iostream>

int main() {
 std::ofstream file("data.txt", std::ios::out | std::ios::trunc); // Open "data.txt" for writing, truncating existing content

 if (file.is_open()) {
  file << "This will overwrite any existing data." << std::endl;
  file.close(); // Close the file
 } else {
  std::cerr << "Error: Could not open file." << std::endl;
 }

 return 0;
```

}

6. Opening with Error Handling:

- It's crucial to check if the file was successfully opened using the is_open() member function.
- If `is