Week 1: Data Structures and Abstraction

Tuesday: 2D Arrays

Definition. A **2D array** is a collection of identical arrays (both in type and size) grouped together. It is used for storing sets of data in a grid-like structure.

To declare a 2D array:

Declaring a 2D Array

```
<type> name_of_array[i][j];
```

Here, `<type>` is the data type, `i` is the number of rows, and `j` is the number of columns.

Example. A 2D integer array with 2 rows and 3 columns can be declared as:

2D Array Declaration

```
int times[2][3];
```

The indices for this array would look like this:

```
0,0    0,1    0,2
1,0    1,1    1,2
```

The first index represents the row, and the second index represents the column. For example:

Accessing a 2D Array

```
myArr[0][2] = 11;  // Sets the value in row 0, column 2 to 11.
```

Note. 2D arrays are stored as sequential blocks of memory. This is why pointer arithmetic works:

Pointer Arithmetic in 2D Arrays

```
cout << *(myarray + 1);          // 1D array
cout << *(*(my2Darray + 2) + 3); // 2D array
```

You can initialize a 2D array during declaration:

Initializing a 2D Array

```
int times[2][3] = {{1, 2}, {4, 5, 6}};
```

This initializes the array as:

```
1    2    (compiler-determined)
4    5    6
```

When passing a 2D array to a function, you must specify the number of columns but not the number of rows:

Passing a 2D Array to a Function

```
void CheckTimes(int myArray[][3], int rows) {
    // Function implementation
}
```

Exercises

Exercise. Declare a 2D array `grades` with 30 rows and 10 columns.

Declaring a 2D Array

```cpp
int grades[30][10];
```

Exercise. What is the size of the array `sales[6][4]`?

Calculating Array Size

```
The array has 6 rows and 4 columns, so its size is 6 * 4 = 24 elements.
```

Exercise. Write a function to display a 2D array with 7 columns.

Displaying a 2D Array

```cpp
void DisplayArray7(int arr[][7], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 7; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

Thursday: Structures and Abstraction

Definition. An **abstract data type (ADT)** is a general model of something, composed of one or more primitive types or previously defined ADTs. The programmer determines what values are acceptable and what operations are allowed.

Definition. A **structure** is a user-defined data type that groups related variables together. It is defined using the `struct` keyword.

To define a structure:

Defining a Structure

```cpp
struct Student {
    string Name;
    int GradYear;
    float GPA;
};
```

To declare an instance of the structure:

Declaring a Structure

```cpp
Student s1;
```

You can access structure members using the dot operator:

Accessing Structure Members

```cpp
s1.Name = "Mary";
s1.GradYear = 2027;
s1.GPA = 3.9478;
```

You can initialize a structure during declaration:

Initializing a Structure

```cpp
Student s2 = {"Bob", 2028, 3.874};
```

Note. The order of initialization must match the order of the structure definition.

You cannot compare two structures directly. The comparison will compare memory addresses, not the data. To compare structures, you must overload the `==` operator.

You can use structures with arrays:

Using Structures with Arrays

```cpp
Student myClass[100];
cout << myClass[17].Name;  // Outputs the 18th student's name.
```

You can also initialize an array of structures:

Initializing an Array of Structures

```cpp
Student myClass[120] = {
    {"Ann", 2028, 3.987},
    {"Bob", 2025},
    {"Jim"},
    {"Mary", 2025, 4.0}
};
```

Definition. A **nested structure** is a structure that contains another structure as one of its members.

To define a nested structure:

Defining a Nested Structure

```cpp
struct Address {
    string Street;
    string City;
    string State;
};

struct Student {
    string First;
    string Last;
    Address Home;
    Address School;
};
```

To access nested structure members:

Accessing Nested Structure Members

```cpp
Student s1;
s1.Home.City = "Cincinnati";
s1.Home.State = "OH";
```

Exercises

Exercise. Create a nested structure for a `Car` that includes an `Engine` structure with members `horsepower` and `fuelType`.

Nested Structure for a Car

```cpp
struct Engine {
    int horsepower;
    string fuelType;
};

struct Car {
    string make;
    string model;
    Engine engine;
};
```

Week 2: Files and File Handling

Tuesday: Files

Definition. Files in C++ are handled using the `<fstream>` library. There are three main data types for file handling:

- `ifstream` – Input file stream (read-only).
- `ofstream` – Output file stream (write-only).
- `fstream` – General file stream (read and write).

To create a variable for a file:

Creating a File Stream

```
fstream data; // Creates a file stream for reading and/or writing.
```

To open a file:

Opening a File

```
data.open("info.txt", ios::app); // Opens the file in append mode.
```

You can use multiple flags at the same time:

```
data.open("info.xlsx", ios::in | ios::out); // Opens the file for reading and writing.
```

Note. Unless you fully qualify the path, the file is assumed to be in the same location as the executable.

To write to a file, use the `<<` operator, which works the same as `cout`:

Writing to a File

```
data << "It is cold outside"; // Writes to the file.
cout << "It is cold outside"; // Writes to the console.
```

Before using a file, you should check if it was successfully opened:

Checking if a File is Open

```
if (data.fail()) {
    // No file opened
} else {
    // File opened, do work
}
```

To read from a file, use the `>>` operator:

Reading from a File

```
int i;
data >> i; // Reads an integer from the file.
```

Note. Anything you can do with `cout` or `cin`, you can do with a file (e.g., formatting, escape sequences like `n`).

Checkpoint

Exercise. What is the difference between `ios::app` and `ios::ate`?

Answer:

`ios::app` appends to the end of the file, while `ios::ate` opens the file and moves the pointer to the end.

Exercise. How do you open a file for both reading and writing?

Answer:

Use the pipe operator: `ios::in | ios::out`.

Exercise. Open a file named `name.data` in append mode.

Solution:

```
diskInfo.open("name.data", ios::app);
```

File Stream Member Functions

File streams have several member functions for reading and writing:

- `get()` – Fetches a single character.
- `put()` – Writes a single character.
- `getline()` – Reads a full line of text until the end-of-line character.

Exercise. What is the output of the following code?

```
fstream file;
file.open("data.txt", ios::in);
string line;
while (file >> line) {
    cout << line << endl;
}
```

Answer:

The code reads one word at a time from the file and prints it to the console.

Working with Multiple Files

You can create multiple stream objects to work with multiple files simultaneously:

Working with Multiple Files

```
fstream data;
data.open("info.txt", ios::in);

fstream data2;
data2.open("mydata.txt", ios::app);

string val1;
getline(data, val1); // Reads a line from info.txt
data2 << val1 << endl; // Writes the line to mydata.txt
```

Binary Files

To work with binary files, use the `ios::binary` flag. Binary files use the `.read()` and `.write()` member functions instead of `<<` and `>>`.

Reading and Writing Binary Data

```cpp
int x = 12;
fstream data;
data.open("myfile.txt", ios::in | ios::out | ios::binary);

data.read(reinterpret_cast<char *>(&x), sizeof(x)); // Reads an int from the file.
data.write(reinterpret_cast<char *>(&x), sizeof(x)); // Writes an int to the file.
```

Note. Use `reinterpret_cast<char *>` to treat the address as a character pointer, even if it isn't. This is necessary for binary file operations.

Structures and Files

Structures work the same as any other data type with files (as long as they are fixed-width):

Writing a Structure to a File

```cpp
struct Test {
    int I1;
    char C1;
    double D1;
};

Test myTest;
myTest.I1 = 123;
myTest.C1 = 'A';
myTest.D1 = 123.456;

data.write(reinterpret_cast<char *>(&myTest), sizeof(myTest));
```

Note. Binary files prevent problems with non-character data and data errors. Avoid storing pointers in files, as addresses may be incorrect the next time the program runs.

Random Access Files

Random access files allow you to move to any spot in the file:

- `seekp()` – Moves the write pointer.
- `seekg()` – Moves the read pointer.

Using seekp and seekg

```cpp
data.seekp(10, ios::beg); // Moves the write pointer to the 10th byte from the beginning.
data.seekg(-5, ios::cur); // Moves the read pointer 5 bytes backward from the current position.
```

Note. Use `tellp()` and `tellg()` to get the current position of the write and read pointers, respectively.

Thursday: Object-Oriented Programming

Definition. **Procedural Programming** separates data and functions, while **Object-Oriented Programming (OOP)** combines data and functions into objects. Objects contain:

- **Attributes**: Data inside an object.
- **Member Functions**: Procedures that operate on the data.

Definition. Key concepts in OOP:

- **Encapsulation**: Combining data and code into a single object.
- **Data Hiding**: Restricting access to certain attributes.

- **Object Reusability**: Packaging code into objects for easier reuse.

Classes

A **class** is a blueprint for creating objects. It specifies both attributes and member functions.

Class Syntax

```
class Student {
    string name; // Attribute
    int id;      // Attribute
    void Enroll(); // Member function
};
```

By default, all members of a class are **private**. To make them accessible outside the class, use access specifiers:

- `public`: Accessible from outside the class.
- `private`: Accessible only within the class.
- `protected`: Accessible within the class and derived classes.

Example. A class with public and private members:

```
class Student {
public:
    string name; // Public attribute
    void Enroll(); // Public member function
private:
    int id; // Private attribute
};
```

Note. It is best practice to group `public` and `private` sections together for readability and to always define access specifiers.

Getters and Setters

Frequently, attributes are made private, and public functions (called **getters** and **setters**) are used to access or modify them.

Getters and Setters

```
class Student {
private:
    int id;
public:
    int getId() const { return id; } // Getter (constant function)
    void setId(int newId) { id = newId; } // Setter
};
```

Note. The `const` keyword in a member function indicates that the function does not modify any class attributes.

Defining Member Functions Outside the Class

Member functions can be defined outside the class using the scope resolution operator (`::`).

Defining Member Functions Outside the Class

```
void Student::Enroll() {
    // Function implementation
}
```

Note. Best practice is to:

- Declare the class in a header file (`className.h`).
- Define member functions in a source file (`className.cpp`).

Creating and Using Objects

To use a class, you must create an instance of it.

Creating an Instance of a Class

```
Student s1; // Creates an instance of the Student class
s1.name = "Sue"; // Accessing a public attribute
```

Note. For member functions that use member data, it is best to re-fetch the data every time the function is called to ensure it is up-to-date.

Pointers to Classes

Pointers to classes work similarly to pointers to structs. Use the `->` operator to access members.

Using Pointers with Classes

```
Student *currStudent = &s1;
currStudent->name = "Bill"; // Accessing a member using a pointer
(*currStudent).name = "Bill Smith"; // Alternative syntax
```

Note. Use parentheses when dereferencing a pointer to a class or accessing the address of a class instance or its members.

Dynamic Memory Allocation

The `new` and `delete` operators work the same with classes as they do with structs and primitives.

Dynamic Memory Allocation

```
Student *s2 = new Student; // Allocate memory for a Student object
// Use s2...
delete s2; // Free memory to prevent memory leaks
s2 = nullptr; // Set pointer to null to avoid dangling pointers
```

Why Private Members?

Private members:

- Prevent unauthorized access.
- Allow for error checking (e.g., in setters).

Class Files

Classes are typically defined in separate files:

- **Header File** (`.h`): Contains the class declaration.
- **Source File** (`.cpp`): Contains the class implementation.

Include Guards

```cpp
#ifndef STUDENT_H
#define STUDENT_H

class Student {
    // Class declaration
};

#endif // STUDENT_H
```

Note. Use `#pragma once` in Visual Studio as an alternative to include guards.

Exception for Short Functions

If a member function is very short (1-3 lines), it is common to define it in the header file.

Short Function in Header File

```cpp
class Student {
public:
    int getId() const { return id; } // Short function in header
private:
    int id;
};
```

Week 3: Constructors, Destructors, and Arrays of Objects

Tuesday: Constructors, Destructors, and Arrays of Objects

Checkpoint

Exercise. Why should accessors and mutators (getters and setters) be used for private member variables?

Answer:

Accessors and mutators ensure that private member variables are accessed and modified in a controlled manner, allowing for validation and error checking.

Exercise. What is the purpose of a class specification file and a class implementation file?

Answer:

The class specification file (`.h`) declares the class and its members, while the implementation file (`.cpp`) defines the member functions. This separation improves code organization and reusability.

Exercise. What is the purpose of include guards?

Answer:

Include guards prevent a header file from being included multiple times, avoiding redefinition errors.

Exercise. What is the difference between a class declaration and a class implementation?

Answer:

The class declaration (in the `.h` file) specifies the class members, while the implementation (in the `.cpp` file) defines the member functions.

Exercise. When should you implement a member function directly in the class declaration?

Answer:

When the function is very short (1-3 lines), it is common to implement it directly in the class declaration.

Constructors

A **constructor** is a special member function that initializes the values in a class when an instance is created. It has the same name as the class and is called automatically.

Constructor Example

```cpp
class Student {
public:
    Student(); // Default constructor
    Student(string name, double GPA); // Parameterized constructor
private:
    string FullName;
    double GPA;
};
```

Constructor Implementation

```cpp
// Default constructor
Student::Student() {
    FullName = "Unknown";
    GPA = 0.0;
}

```

```cpp
// Parameterized constructor
Student::Student(string name, double gpa) {
    FullName = name;
    GPA = gpa;
}
```

Note. Constructors are called automatically when:

- An instance of the class is declared.
- The `new` operator is used to allocate memory for an object.

Destructors

A **destructor** is a special member function that is called automatically when an object is destroyed. It is used for cleanup tasks like freeing memory or closing files.

Destructor Example

```cpp
class Student {
public:
    Student(); // Constructor
    ~Student(); // Destructor
    // Other members...
};
```

Destructor Implementation

```cpp
Student::~Student() {
    // Cleanup code (e.g., freeing memory, closing files)
}
```

Note. You can only have one destructor per class. If you don't define one, the compiler will generate a default destructor.

Private Member Functions

Private member functions can only be called from other member functions within the same class. They can access both public and private members.

Private Member Function Example

```cpp
class Student {
public:
    void Enroll();
private:
    void ValidateGPA(double gpa); // Private member function
};
```

Arrays of Objects

You can create arrays of class objects. The constructor is called for each element in the array.

Array of Objects Example

```cpp
Student CS2028C[125]; // Calls the default constructor 125 times
Student myFav[3] = {Student("Mary", 3.6), Student("Bob", 3.5)}; // Calls parameterized constructors
```

Note. If the array is initialized with fewer elements than its size, the remaining elements are initialized using the default constructor.

Checkpoint

Exercise. What is the output of the following code?

```cpp
class Test {
public:
    Test() { cout << "10 "; }
    Test(int x) { cout << x << " "; }
};

int main() {
    Test t1;
    Test t2(20);
    Test t3 = 50;
}
```

Answer:

The output is: `10 20 50`

Exercise. What is the output of the following code?

```cpp
class Test {
public:
    Test() { cout << "4 "; }
    Test(int x) { cout << x << " "; }
    ~Test() { cout << "2 "; }
};

int main() {
    Test t1;
    Test t2(7);
}
```

Answer:

The output is: `4 7 2 2`

Exercise. How do you declare an array of objects?

Answer:

```cpp
InventoryItem items[3]; // Declares an array of 3 InventoryItem objects
```

Exercise. How do you dynamically allocate an array of objects?

Answer:

```cpp
int yardCount;
cin >> yardCount;
Yard *yards = new Yard[yardCount]; // Dynamically allocate an array of Yard objects
delete[] yards; // Free the allocated memory
```

Thursday: More Class Stuff (Static Members, Friends, Copy Constructors, and Operator Overloading)

Static Members

Static members are shared across all instances of a class. They do not require an instance of the class to be accessed.

Static Member Example

```cpp
class Student {
public:
    string Name;
    double GPA;
    static string School; // Static member variable
};
```

Accessing Static Members

```
Student s1;
s1.Name = "Sue";
s1.GPA = 3.6;
s1.School = "UC"; // Accessing static member through an instance
cout << Student::School; // Accessing static member directly
```

Note. Static member variables are stored separately from instance members. Static member functions cannot access non-static members because they do not operate on a specific instance.

Friends

A **friend** of a class is a function or class that is not a member of the class but has access to its private and protected members.

Friend Function Example

```
class Student {
    friend void MyFunction(); // Friend function declaration
private:
    string Name;
    double GPA;
};

void MyFunction() {
    Student s;
    s.Name = "Bob"; // Accessing private member
    s.GPA = 3.8;    // Accessing private member
}
```

Note. Friend functions or classes should be used with care, as they break encapsulation by exposing private members.

Copy Constructors

A **copy constructor** is used to initialize an object using another object of the same class. It is called during member-wise assignment.

Copy Constructor Example

```
class Student {
public:
    Student(const Student &right) { // Copy constructor
        Name = right.Name;
        GPA = right.GPA;
    }
private:
    string Name;
    double GPA;
};
```

Note. The copy constructor is called when:

- An object is initialized using another object.
- An object is passed by value to a function.
- An object is returned by value from a function.

Operator Overloading

Operator overloading allows you to define custom behavior for operators like +, -, +=, etc.

Operator Overloading Example

```cpp
class Math {
public:
    int a;
    void operator+=(int right) { // Overload += operator
        a += right;
    }
    void operator+=(Math right) { // Overload += operator for Math objects
        a += right.a;
    }
};
```

Note. Operator overloading cannot change the number of parameters or the precedence of the operator.

The `this` Pointer

The `this` pointer refers to the current instance of the class. It is useful for accessing members within the class.

Using the `this` Pointer

```cpp
class Math {
public:
    int a;
    void operator+(int right) {
        this->a += right; // Accessing member using this pointer
    }
};
```

Type Conversion

You can define custom type conversion for classes using conversion operators.

Type Conversion Example

```cpp
class MyTime {
public:
    int hours;
    int minutes;
    operator int() { // Conversion operator
        return hours * 60 + minutes;
    }
};

MyTime rightNow;
rightNow.hours = 2;
rightNow.minutes = 30;
int x = rightNow; // Calls the conversion operator
```

Note. Conversion operators do not have a return type or parameters.

Checkpoint

Exercise. What is the difference between instance members and static members?

Answer:

Instance members are unique to each instance of a class, while static members are shared across all instances.

Exercise. Where should static member variables be defined?

Answer:

Static member variables should be defined in the source file (`.cpp`) using the scope resolution operator (`::`).

Exercise. Can static member functions access non-static members?

Answer:

No, static member functions cannot access non-static members because they do not operate on a specific instance.

Exercise. What is the purpose of the `this` pointer?

Answer:

The `this` pointer refers to the current instance of the class and is used to access its members.

Exercise. What is the output of the following code?

```cpp
class Pet {
public:
    Pet(const Pet &right) { cout << "Copy Constructor"; }
};

int main() {
    Pet cat;
    Pet dog = cat;
}
```

Answer:

The output is: `Copy Constructor`

Week 4: Inheritance, Polymorphism, and Templates

Tuesday: Inheritance

Definition. **Inheritance** allows a new class (derived class) to be based on an existing class (base class). The derived class inherits all members from the base class except for constructors and destructors.

Inheritance implements the "is a" relationship:

- A smartphone **is a** computer.
- A dog **is an** animal.
- A student **is a** person.

Note. Not all "is a" relationships indicate inheritance. For example:

- Mary **is a** student (Mary is an instance of the `Student` class).
- Professor Chuck **is a** jerk (an attribute of the `Professor` class).

Inheritance Syntax

To inherit in C++, use the following syntax:

Inheritance Syntax

```
class DerivedClass : access_specifier BaseClass {
    // Derived class members
};
```

The `access_specifier` determines how the members of the base class are treated in the derived class. If omitted, it defaults to `private`.

Access Specifiers

The access specifier affects the visibility of base class members in the derived class:

Access Specifier	Public Members	Private Members	Protected Members
public	public	private	protected
private	private	private	private
protected	protected	private	protected

Note. The access specifier can increase security but cannot decrease it. For example, a `public` member in the base class cannot become `private` in the derived class.

Protected Members

Protected members are like private members outside the class but are accessible to derived classes.

Protected Members Example

```
class Base {
protected:
    int protectedVar;
};

class Derived : public Base {
public:
    void AccessProtected() {
        protectedVar = 10; // Accessing protected member
    }
};
```

Constructors and Destructors

Constructors are called from the base class up to the derived class. Destructors are called in the reverse order (derived class first, then base class).

Constructor and Destructor Order

```
class Base {
public:
    Base() { cout << "Base Constructor\n"; }
    ~Base() { cout << "Base Destructor\n"; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived Constructor\n"; }
    ~Derived() { cout << "Derived Destructor\n"; }
};
```

Note. If the base class constructor requires parameters, you must explicitly call it in the derived class constructor:

```
Derived::Derived(int x) : Base(x) { ... }
```

Redefining Base Class Functions

A derived class can redefine (override) a base class function. Use the `virtual` keyword in the base class to enable dynamic binding.

Redefining Base Class Functions

```
class Base {
public:
    virtual void Print() { cout << "Base Print\n"; }
};

class Derived : public Base {
public:
    void Print() override { cout << "Derived Print\n"; }
};
```

Note. With `virtual` and `override`, the derived class version of the function is called, even if the object is accessed through a base class pointer:

```
Base* bp = new Derived;
bp->Print(); // Calls Derived::Print
```

The `final` Keyword

The `final` keyword prevents a derived class from overriding a base class function.

Using the `final` Keyword

```
class Base {
public:
    virtual void Print() final { cout << "Base Print\n"; }
};

class Derived : public Base {
public:
    void Print() override { cout << "Derived Print\n"; } // Error: Cannot override final function
};
```

Checkpoint

Exercise. What is the difference between private and protected members?

Answer:

Private members can only be accessed within the class, while protected members can also be accessed by derived classes.

Exercise. What is the order of constructor and destructor calls in inheritance?

Answer:

Constructors are called from the base class up to the derived class. Destructors are called in the reverse order (derived class first, then base class).

Exercise. What is the purpose of the `virtual` keyword?

Answer:

The `virtual` keyword enables dynamic binding, allowing the derived class version of a function to be called even when accessed through a base class pointer.

Exercise. What happens if a derived class tries to override a `final` function?

Answer:

The compiler will generate an error because a `final` function cannot be overridden.

Thursday: Polymorphism and Templates

Polymorphism

Definition. **Polymorphism** (many forms) allows an object or object pointer to reference objects of different types. This enables calling the correct member function based on the actual object type, not the declared type.

Definition. **Static Binding**: The process of matching a function call with the function code at compile time.

Definition. **Dynamic Binding**: The process of determining at runtime which function to call based on the actual object type.

Note. To enable dynamic binding, use the `virtual` keyword in the base class for the function:

```cpp
virtual returnType MethodName(parameters);
```

Example. Polymorphism with dynamic binding:

```cpp
class Animal {
public:
    virtual void Move() { cout << "Animal moves\n"; }
};

class Dog : public Animal {
public:
    void Move() override { cout << "Dog runs\n"; }
};

Animal* a = new Dog;
a->Move(); // Output: "Dog runs" (dynamic binding)
```

Abstract Classes

An **abstract class** defines a set of members that must be implemented by derived classes. It cannot be instantiated directly.

Abstract Class Example

```cpp
class Animal {
public:
    virtual void Move() = 0; // Pure virtual function
};

class Dog : public Animal {
public:
    void Move() override { cout << "Dog runs\n"; }
};
```

Note. A class containing a pure virtual function (`= 0`) is abstract. Derived classes must implement all pure virtual functions.

Multiple Inheritance

C++ supports multiple inheritance, where a class can inherit from multiple base classes.

Multiple Inheritance Example

```cpp
class Base1 {
public:
    void Func1() { cout << "Base1 Func1\n"; }
};

class Base2 {
public:
    void Func2() { cout << "Base2 Func2\n"; }
};

class Derived : public Base1, public Base2 {
public:
    void Func3() { cout << "Derived Func3\n"; }
};
```

Note. If base classes have functions with the same signature, the derived class must resolve the ambiguity.

Templates

Templates allow you to create generic functions and classes that work with multiple data types.

Function Template Example

```cpp
template <class T>
T Add(T a, T b) {
    return a + b;
}

int x = Add(3, 5); // int version
double d = Add(3.5, 4.61); // double version
string s = Add("a", "z"); // string version
```

Note. The compiler generates a version of the template function for each data type used.

Class Template Example

```cpp
template <class T>
class Shelf {
private:
    T items[10];
    int count = 0;
public:
    void AddItem(T item) { items[count++] = item; }
    T RemoveItem() { return items[--count]; }
};

Shelf<Book> bookShelf;
Shelf<Game> gameShelf;
```

Note. Class templates allow you to define a generic class that can work with any data type.

Specialized Templates

Specialized templates allow you to create custom implementations for specific types.

Specialized Template Example

```cpp
template <>
class Shelf<string> {
private:
    string items[10];
    int count = 0;
public:
    void AddItem(string item) { items[count++] = item; }
    string RemoveItem() { return items[--count]; }
};
```

STL (Standard Template Library)

The STL provides commonly used data structures and algorithms.

- **Sequences**:
  - `vector`: Dynamic array.
  - `deque`: Double-ended queue.
  - `list`: Doubly linked list.
  - `forward_list`: Singly linked list.
  - `array`: Fixed-size array.
- **Associative Collections**:
  - `set`: Unique keys.
  - `multiset`: Non-unique keys.
  - `map`: Key-value pairs with unique keys.
  - `multimap`: Key-value pairs with non-unique keys.
- **Algorithms**:
  - `max_element`: Finds the maximum element.
  - `min_element`: Finds the minimum element.
  - `for_each`: Applies a function to each element.
  - `find_if`: Finds an element based on a condition.
  - `sort`: Sorts elements.
  - `random_shuffle`: Randomly shuffles elements.

Example. Using STL algorithms:

```cpp
#include <algorithm>
#include <vector>

vector<int> nums = {3, 1, 4, 1, 5, 9};
sort(nums.begin(), nums.end()); // Sorts the vector
auto maxIt = max_element(nums.begin(), nums.end()); // Finds the maximum element
```

Checkpoint

Exercise. What is the difference between static and dynamic binding?

Answer:

Static binding resolves function calls at compile time, while dynamic binding resolves them at runtime based on the actual object type.

Exercise. What is a pure virtual function?

Answer:

A pure virtual function (`virtual void Func() = 0;`) is a function that must be implemented by derived classes, making the class abstract.

Exercise. What is the purpose of templates?

Answer:

Templates allow you to write generic code that works with multiple data types, reducing code duplication.

Exercise. What is the difference between a `set` and a `multiset`?

Answer:

A `set` contains unique keys, while a `multiset` allows duplicate keys.

Week 5: Exceptions

Tuesday: Exceptions

Definition. An **exception** is a signal used to indicate that something unexpected has occurred while your program is running. Examples include:

- Division by zero.
- Incorrect data type.
- File I/O errors.
- Out-of-memory errors (from the `new` operator).

Handling Exceptions

Exceptions are handled using `try` and `catch` blocks. The `try` block contains code that may throw an exception, and the `catch` block contains code to handle the exception.

Basic Exception Handling

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

Note. You can have multiple `catch` blocks for a single `try` block to handle different types of exceptions.

Multiple Catch Blocks

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e) {
    // Handle ExceptionType1
} catch (ExceptionType2 e) {
    // Handle ExceptionType2
}
```

Note. When an exception occurs, the program stops execution in the `try` block and jumps to the first matching `catch` block. If no matching `catch` block is found, the program terminates.

Throwing Exceptions

You can throw your own exceptions using the `throw` keyword.

Throwing an Exception

```
throw "An error occurred!"; // Throws a string as an exception
```

Note. When a `throw` is executed, control is passed to the exception handler. If the exception is not caught, the program will terminate.

Custom Exceptions

You can create custom exception classes to provide more detailed error information.

Custom Exception Class

```
class MyDataException {
public:
    string Message;
    int ErrorNumber;
    MyDataException(string m, int e) : Message(m), ErrorNumber(e) {}
};

throw MyDataException("My Message here", 123); // Throwing a custom exception
```

Note. Custom exception classes should end with the suffix `Exception` (e.g., `MyDataException`).

Rethrowing Exceptions

You can rethrow an exception to pass it up the call stack for handling by a higher-level function.

Rethrowing an Exception

```
try {
    // Code that may throw an exception
} catch (MyDataException& e) {
    // Handle the exception or rethrow it
    throw; // Rethrow the exception
}
```

Best Practices for Exception Handling

- **Don't swallow exceptions**: If you can't handle an exception, rethrow it.
- **Check conditions first**: Avoid using exceptions for conditions that can be checked beforehand.
- **Exceptions should be exceptional**: Don't use exceptions for frequently occurring events.
- **Keep `try` blocks short**: Minimize the amount of code in `try` blocks.
- **Order `catch` blocks**: Place more specific exceptions before more general ones.
- **Throw exceptions, not error codes**: Use exceptions instead of returning error codes.
- **Use meaningful exception names**: Custom exception classes should end with `Exception`.

Checkpoint

Exercise. What is the purpose of a `try` block?

Answer:

A `try` block contains code that may throw an exception.

Exercise. What happens if an exception is not caught?

Answer:

If an exception is not caught, the program will terminate.

Exercise. What is the purpose of the `catch` block?

Answer:

A `catch` block contains code to handle exceptions thrown in the corresponding `try` block.

Exercise. What is the difference between throwing and rethrowing an exception?

Answer:

Throwing an exception creates a new exception, while rethrowing an exception passes the existing exception up the call stack.

Week 6: Stacks and Queues

Tuesday: Stacks

Definition. A **stack** is an abstract data type (ADT) where elements are added and removed in a **Last In, First Out (LIFO)** manner. This means the last element added to the stack is the first one to be removed.

Example. Real-world examples of stacks:

- A stack of books.
- A Pringles can.
- A clown car.
- Bullets in a gun magazine.

Stack Operations

A stack typically supports the following operations:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes and returns the element at the top of the stack.
- **IsEmpty**: Checks if the stack is empty.
- **IsFull**: Checks if the stack is full (for fixed-size stacks).
- **Top (or Peek)**: Returns the element at the top of the stack without removing it.

Note. Stacks are widely used in computer science, such as:

- Operating systems use stacks to manage function calls (the call stack).
- Stacks are used in algorithms like depth-first search (DFS).
- They are also used in parsing expressions and syntax checking.

Stack Errors

Two common errors can occur when working with stacks:

- **Stack Overflow**: Occurs when you try to add an element to a full stack.
- **Stack Underflow**: Occurs when you try to remove an element from an empty stack.

C++ Implementation of a Stack

Below is a C++ implementation of a stack using a fixed-size array.

Stack Class Implementation

```cpp
#include <iostream>
#include <stdexcept> // For std::overflow_error and std::underflow_error

class Stack {
private:
    static const int MAX_SIZE = 100; // Maximum size of the stack
    int data[MAX_SIZE];              // Array to store stack elements
    int topIndex;                    // Index of the top element

public:
    // Constructor
    Stack() : topIndex(-1) {}

    // Push an element onto the stack
    void Push(int value) {
        if (IsFull()) {
            throw std::overflow_error("Stack Overflow: Cannot push to a full stack.");
        }
        data[++topIndex] = value;
    }

    // Pop an element from the stack
    int Pop() {
        if (IsEmpty()) {
            throw std::underflow_error("Stack Underflow: Cannot pop from an empty stack.");
        }
        return data[topIndex--];
    }

    // Check if the stack is empty
    bool IsEmpty() const {
        return topIndex == -1;
    }

    // Check if the stack is full
    bool IsFull() const {
        return topIndex == MAX_SIZE - 1;
    }

    // Get the top element without removing it
    int Top() const {
        if (IsEmpty()) {
            throw std::underflow_error("Stack Underflow: Cannot get top of an empty stack.");
        }
        return data[topIndex];
    }
};

int main() {
    Stack stack;

    // Push elements onto the stack
    stack.Push(10);
    stack.Push(20);
    stack.Push(30);

    // Print the top element
    std::cout << "Top element: " << stack.Top() << std::endl;

    // Pop elements from the stack
    std::cout << "Popped element: " << stack.Pop() << std::endl;
    std::cout << "Popped element: " << stack.Pop() << std::endl;

    // Check if the stack is empty
    if (stack.IsEmpty()) {
        std::cout << "Stack is empty." << std::endl;
    } else {
        std::cout << "Stack is not empty." << std::endl;
    }

    return 0;
}
```

Explanation of the Code

- **MAX_SIZE**: The maximum size of the stack is defined as a constant.
- **data[]**: An array is used to store the stack elements.
- **topIndex**: Tracks the index of the top element in the stack. It is initialized to `-1` to indicate an empty stack.
- **Push**: Adds an element to the top of the stack. Throws an exception if the stack is full.
- **Pop**: Removes and returns the top element from the stack. Throws an exception if the stack is empty.
- **IsEmpty**: Returns `true` if the stack is empty, otherwise `false`.
- **IsFull**: Returns `true` if the stack is full, otherwise `false`.
- **Top**: Returns the top element without removing it. Throws an exception if the stack is empty.

Checkpoint

Exercise. What is the LIFO principle in stacks?

Answer:

The LIFO (Last In, First Out) principle means that the last element added to the stack is the first one to be removed.

Exercise. What is the difference between a stack overflow and a stack underflow?

Answer:

A **stack overflow** occurs when you try to add an element to a full stack. A **stack underflow** occurs when you try to remove an element from an empty stack.

Exercise. What is the purpose of the `Top` function in a stack?

Answer:

The `Top` function returns the element at the top of the stack without removing it, allowing you to inspect the top element.

Exercise. What happens if you call `Pop` on an empty stack?

Answer:

Calling `Pop` on an empty stack results in a **stack underflow** error, typically throwing an exception.

Thursday: Queues

Definition. A **queue** is an abstract data type (ADT) where elements are added and removed in a **First In, First Out (FIFO)** manner. This means the first element added to the queue is the first one to be removed.

Example. Real-world examples of queues:

- A line of people waiting for a bus.
- A print queue for a printer.
- A queue of tasks in a task scheduler.

Queue Operations

A queue typically supports the following operations:

- **Enqueue**: Adds an element to the back of the queue.
- **Dequeue**: Removes and returns the element at the front of the queue.
- **Peek (or Front)**: Returns the element at the front of the queue without removing it.
- **Length**: Returns the number of elements in the queue.
- **IsEmpty**: Checks if the queue is empty.
- **IsFull**: Checks if the queue is full (for fixed-size queues).
- **MakeEmpty**: Clears all elements from the queue.

Note. Queues are widely used in computer science, such as:

- Task scheduling in operating systems.

- Handling requests in web servers.
- Breadth-first search (BFS) in graph algorithms.

Ring Queue

A **ring queue** (or circular queue) solves the issue of needing to shift all elements when enqueueing or dequeueing. It uses a circular buffer to efficiently manage the queue.

Note. To handle the "start=end" issue (where the queue could be either full or empty), two common solutions are:

- Include an empty spot as a buffer between the start and end.
- Keep track of a count to determine if the queue is full or empty.

C++ Implementation of a Queue

Below is a C++ implementation of a queue using a ring buffer.

Queue Class Implementation

```cpp
#include <iostream>
#include <stdexcept> // For std::overflow_error and std::underflow_error

class Queue {
private:
    static const int MAX_SIZE = 100; // Maximum size of the queue
    int data[MAX_SIZE];              // Array to store queue elements
    int frontIndex;                  // Index of the front element
    int rearIndex;                   // Index of the rear element
    int count;                       // Number of elements in the queue

public:
    // Constructor
    Queue() : frontIndex(0), rearIndex(0), count(0) {}

    // Enqueue an element
    void Enqueue(int value) {
        if (IsFull()) {
            throw std::overflow_error("Queue Overflow: Cannot enqueue to a full queue.");
        }
        data[rearIndex] = value;
        rearIndex = (rearIndex + 1) % MAX_SIZE; // Wrap around using modulo
        count++;
    }

    // Dequeue an element
    int Dequeue() {
        if (IsEmpty()) {
            throw std::underflow_error("Queue Underflow: Cannot dequeue from an empty queue.");
        }
        int value = data[frontIndex];
        frontIndex = (frontIndex + 1) % MAX_SIZE; // Wrap around using modulo
        count--;
        return value;
    }

    // Peek at the front element
    int Peek() const {
        if (IsEmpty()) {
            throw std::underflow_error("Queue Underflow: Cannot peek at an empty queue.");
        }
        return data[frontIndex];
    }

    // Get the number of elements in the queue
    int Length() const {
        return count;
    }

    // Check if the queue is empty
    bool IsEmpty() const {
        return count == 0;
    }

    // Check if the queue is full
    bool IsFull() const {
        return count == MAX_SIZE;
    }

    // Clear the queue
    void MakeEmpty() {
        frontIndex = 0;
        rearIndex = 0;
        count = 0;
    }
};

int main() {
    Queue queue;

    // Enqueue elements
    queue.Enqueue(10);
    queue.Enqueue(20);
    queue.Enqueue(30);

    // Peek at the front element
    std::cout << "Front element: " << queue.Peek() << std::endl;

    // Dequeue elements
    std::cout << "Dequeued element: " << queue.Dequeue() << std::endl;
    std::cout << "Dequeued element: " << queue.Dequeue() << std::endl;

    // Check if the queue is empty
    if (queue.IsEmpty()) {
        std::cout << "Queue is empty." << std::endl;
    } else {
        std::cout << "Queue is not empty." << std::endl;
    }

    return 0;
}
```

Explanation of the Code

- **MAX_SIZE**: The maximum size of the queue is defined as a constant.
- **data[]**: An array is used to store the queue elements.
- **frontIndex**: Tracks the index of the front element in the queue.
- **rearIndex**: Tracks the index of the rear element in the queue.
- **count**: Tracks the number of elements in the queue.
- **Enqueue**: Adds an element to the rear of the queue. Throws an exception if the queue is full.
- **Dequeue**: Removes and returns the front element from the queue. Throws an exception if the queue is empty.
- **Peek**: Returns the front element without removing it. Throws an exception if the queue is empty.
- **Length**: Returns the number of elements in the queue.
- **IsEmpty**: Returns `true` if the queue is empty, otherwise `false`.
- **IsFull**: Returns `true` if the queue is full, otherwise `false`.
- **MakeEmpty**: Clears all elements from the queue.

Checkpoint

Exercise. What is the FIFO principle in queues?

Answer:

The FIFO (First In, First Out) principle means that the first element added to the queue is the first one to be removed.

Exercise. What is the purpose of a ring queue?

Answer:

A ring queue (or circular queue) efficiently manages the queue by using a circular buffer, avoiding the need to shift elements when enqueueing or dequeueing.

Exercise. What is the difference between `Enqueue` and `Dequeue`?

Answer:

`Enqueue` adds an element to the rear of the queue, while `Dequeue` removes and returns the element at the front of the queue.

Exercise. What happens if you call `Dequeue` on an empty queue?

Answer:

Calling `Dequeue` on an empty queue results in a **queue underflow** error, typically throwing an exception.