

Unit –V
Exception Handling and Templates

1. Exception Handling

1.1 Fundamentals of Exceptional Handling

Exception handling is a mechanism used in C++ to deal with runtime errors (exceptions) that may occur during program execution. Without exception handling, when an error occurs, the program terminates abruptly, which can lead to data loss or an inconsistent state. Exception handling helps manage such errors by allowing the program to handle them gracefully and recover without terminating immediately.

Common Examples of Exceptions in C++

C++ exceptions are triggered when an error or unexpected condition occurs during program execution. Below are four common examples of exceptions:

1. Division by Zero

- **Cause:** Occurs when a program attempts to divide a number by zero, which is mathematically undefined.
- **Explanation:** This typically leads to a runtime error or undefined behavior.
- **Handling:** Use a conditional check or throw an exception to handle the error.

2. Array Out-of-Bounds Access

- **Cause:** Happens when a program tries to access an array index that is outside its valid range.
- **Explanation:** This can corrupt memory or cause segmentation faults.
- **Handling:** Use bounds checking or throw an exception if an invalid index is accessed.

3. File Not Found

- **Cause:** Occurs when a program tries to open or read a file that does not exist or is inaccessible.

- **Explanation:** The file handling functions fail, and the program cannot proceed.
- **Handling:** Check if the file exists before accessing it, or use exception handling to manage the error.

4. Invalid Type Conversion

- **Cause:** Happens when a program attempts to convert between incompatible types, such as converting a string to an integer.
- **Explanation:** May result in undefined behavior or runtime errors.
- **Handling:** Validate the input or throw an exception for invalid conversions.

For example:

```
int c = a / b;  // if b = 0, this will generate a runtime error
```

Need for Exception Handling

Without exception handling, errors cause the program to terminate, potentially leading to:

- **Data loss**: If the program crashes before saving user input.
- **Inconsistent state**: Data might be left in an invalid state, especially in commercial applications.
- **Financial loss**: For programs involving transactions or sensitive data, an error can cause significant problems.

How to Handle Exceptions

C++ provides three common ways to handle exceptions:

1. **Remedial Action**: The program tries to fix the issue and continues execution.
2. **Retry**: The program displays the error and prompts the user to correct the mistake.
3. **Terminate**: The program shows an error message and stops.

C++ Exception Handling Keywords

C++ uses three keywords for exception handling:

1. **`try`**: Defines a block of code that may generate an exception.

2. **`catch`**: Catches and handles the exception thrown by the try block.
3. **`throw`**: Used to throw an exception when an error occurs.

Structure of try-catch

A basic try-catch block looks like this:

```
try {
    // Code that may generate an exception
}
catch (Type variable) {
    // Code to handle the exception
}
```

- The **`try`** block contains code that may throw an exception.
- The **`catch`** block catches the thrown exception and handles it.
- **Type** is the data type of the exception being thrown (e.g., `int`, `string`).

Key Points:

- A program can have multiple catch blocks to handle different types of exceptions.
- If no exception occurs, the catch block is skipped.
- If an exception is thrown, the control jumps directly to the catch block, and the rest of the try block is skipped.
- The data type of the thrown exception must match the type in the catch block to ensure proper handling.

Example: Program to handle exception using class type exception.

```
#include <iostream>
using namespace std;

// Define a custom exception class
class MyException {
public:
    // Constructor to display an error message
```



```
    MyException(const char* msg) {
        errorMessage = msg;
    }

    // Function to get the error message
    const char* getErrorMessage() {
        return errorMessage;
    }

private:
    const char* errorMessage;
};

int main() {
    try {
        // Simulate an error by throwing a custom exception
        throw MyException("Custom exception occurred!");

    } catch (MyException& e) {
        // Catch the custom exception and display the error message
        cout << "Caught exception: " << e.getErrorMessage() << endl;
    }

    return 0;
}
```

1.2 Simple Exception Handling - Divide by Zero

Example of Exception Handling

```
#include <iostream>
using namespace std;

int divide(int a, int b) {
```



```
    if (b == 0) {
        throw "Division by zero error";  // Throw an exception if
        // divisor is zero
    }
    return a / b;
}

int main() {
    int numerator, denominator;

    cout << "Enter numerator: ";
    cin >> numerator;
    cout << "Enter denominator: ";
    cin >> denominator;

    try {
        int result = divide(numerator, denominator);
        // Attempt division
        cout << "Result: " << result << endl;
    }
    catch (const char* msg) {  // Catch and handle the exception
        cout << "Exception caught: " << msg << endl;
    }
    return 0;
}
```

Sample Output:

Case 1: Valid division

```
Enter numerator: 10
Enter denominator: 2
Result: 5
```

Case 2: Division by zero

```
Enter numerator: 10
Enter denominator: 0
Exception caught: Division by zero error
```


1.3 Multiple Catching

In C++, exception handling is managed using the keywords `try`, `catch`, and `throw`. This allows a program to detect and handle errors (exceptions) gracefully during execution.

- **`try` block**: This contains the code that might throw an exception.
- **`catch` block**: This handles the exception. Multiple catch blocks can be used to handle different types of exceptions separately.
- **`throw` keyword**: This is used to throw an exception from the try block.

Multiple catch Blocks

When an exception occurs in the try block, the control is transferred to the corresponding catch block that matches the type of the thrown exception. If no exception occurs, none of the catch blocks are executed. Multiple catch blocks allow the programmer to handle different types of exceptions separately, based on the type of exception thrown.

Each catch block handles a specific type of exception, and only the first matching catch block is executed. After the first matching block executes, the remaining blocks are skipped.

Syntax of Multiple Catch Blocks

```
try {
    // Code that may generate exceptions
}
catch (Type1 variable) {
    // Code to handle exception of type Type1
}
catch (Type2 variable) {
    // Code to handle exception of type Type2
}
catch (Type_n variable) {
    // Code to handle exception of type Type_n
}
```

Example: Consider a program where we perform division and subtraction, and exceptions are thrown in specific conditions.


```
#include <iostream>
using namespace std;

int main() {
    int e1, sub;
    float a, b, div, e;

    try {
        cout << "Enter two numbers: ";
        cin >> a >> b;

        // Throw exception if second number is zero (divide by zero)
        if (b == 0) throw e;

        div = a / b;
        cout << "\nDivision is: " << div;

        // Throw exception if first number is smaller than the second number
        if (a < b) throw e1;

        sub = a - b;
        cout << "\nSubtraction is: " << sub;
        cout << "\nTry block ends here" << endl;
    }
    catch (float e) {
        cout << "\nException - Division by zero" << endl;
    }
    catch (int e1) {
        cout << "\nException - First number smaller than second" <<
endl;
    }
```



```
    cout << "Program ends\n";
    return 0;
}
```

Explanation of Example

- **Case 1**: Both numbers are valid (second is not zero, and first is not smaller than the second), so the program executes normally without exceptions.
- **Case 2**: The second number is zero, causing a division by zero exception (float type). The corresponding catch block for the float exception handles it.
- **Case 3**: The first number is smaller than the second, causing an int exception (`e1`). The corresponding catch block for the int type handles it.

1.4 Catching All Types of Exceptions

In C++, exception handling is commonly done using the try, catch, and throw keywords. Typically, specific exceptions are handled by defining multiple catch blocks, each handling a different type of exception. However, there might be situations where you want to handle all types of exceptions with a single catch block, without needing to define a separate handler for each type.

To achieve this, C++ provides a special type of catch block that handles all exceptions, regardless of their type. This is done using `catch(...)`.

Syntax of Catch-All Block

```
try {
    // Code that may throw exceptions
}
catch(...) {
    // Code to handle any type of exception
}
```

The three dots `(...)` in the catch block indicate that it is capable of catching **all types of exceptions**. This block acts as a general error handler when you are unsure of the specific types of exceptions that might be thrown or if you want to handle all exceptions in the same way.

When to Use Catch-All Block

You should use the catch-all block when:

- You don't know all the possible types of exceptions that could be thrown in your program.
- You don't want to write separate catch blocks for each type of exception.
- You need a fallback handler for any unexpected exceptions.

However, it is important to note that the `catch(...)` block should be placed **last** after all specific catch blocks, as C++ will try to match exceptions with the first available catch block. If `catch(...)` appears before specific handlers, it will catch all exceptions, preventing any other handlers from being executed.

Example: Catching All Types of Exceptions

Consider the following example where we use a catch-all block to handle different types of exceptions:

```
#include <iostream>
using namespace std;

int main() {
    int e1, sub;
    float a, b, div, e;

    try {
        cout << "Enter two numbers: ";
        cin >> a >> b;

        // Throw exception if second number is zero (divide by zero)
        if (b == 0)
            throw e;

        div = a / b;
        cout << "\nDivision is: " << div;
```



```
        // Throw exception if second number is greater than the first
        if (a < b)
            throw e1;

        sub = a - b;
        cout << "\nSubtraction is: " << sub;
        cout << "\nTry block ends here" << endl;
    }
    catch(...) {
        cout << "\nException raised" << endl;
    }

    cout << "Program ends\n";
    return 0;
}
```

Explanation of Example

- **Case 1**: If both numbers are valid (second is not zero, and the first number is not greater than the second), the program executes normally without any exceptions.
- **Case 2**: If the second number is zero, a division by zero exception is thrown. Since we use a catch(...) block, it handles the exception and prints "Exception raised".
- **Case 3**: If the first number is greater than the second, an exception (e1) is thrown. Again, the catch(...) block handles this exception, printing "Exception raised".

1.5 Rethrowing an exception

In C++, an exception can be rethrown from within a catch block using the throw statement without any arguments. This is useful for propagating the exception to an outer try-catch block, allowing the same exception to be handled differently by multiple handlers.

Key Points

- `throw;` inside a `catch` block rethrows the caught exception to the next matching handler in an outer `try-catch` block.

- Useful when an inner handler performs partial handling and delegates further processing to an outer handler.
- If no matching handler exists in the outer block, the program terminates.

Example

```
#include <iostream>
using namespace std;

int main() {
    int a;
    try { // Outer try block
        try { // Inner try block
            cout << "Enter a number: ";
            cin >> a;
            if (a == 1)
                throw 5; // Throw an integer exception
            cout << "Try ends, a = " << a << endl;
        }
        catch (int ex) { // Inner catch block
            cout << "Inner catch, exception = " << ex << endl;
            throw; // Rethrow the exception
        }
    }
    catch (int ex) { // Outer catch block
        cout << "Outer catch, exception = " << ex << endl;
    }
    cout << "Program ends\n";
    return 0;
}
```

Explanation

Case 1: If `a == 1`:

- An exception is thrown in the inner block.
- The inner `catch` block handles it partially and rethrows it using `throw;`.
- The outer `catch` block then handles the rethrown exception.

Case 2: If `a != 1`:

- No exception is thrown, so the inner block executes successfully, and no catch blocks are triggered.

1.6 Exception Specification

Exception specification in C++ is a mechanism that limits the types of exceptions a function can throw. It serves as a guarantee to the caller of the function, specifying which exceptions might be thrown.

Syntax

```
void function_name() throw (type1, type2, ...);
```

- **`type1, type2, ...`**: List of exception types the function is allowed to throw.
- **`throw()`**: Indicates that the function will not throw any exceptions.

Key Points

1. If a function throws an exception that is not in its exception specification, the runtime calls `unexpected()`.
2. A function with no exception specification (`throw()`) guarantees it will not throw any exceptions.
3. A function without any exception specification can throw exceptions of any type.
4. Exception specification is enforced **only at runtime**, meaning the compiler does not prevent exceptions outside the specified list.

Rules for Nested Function Calls

- If a function calls another function with a broader exception specification (i.e., one that allows more exception types), any exceptions not in the outer function's specification must be handled by the outer function.

- Failing to handle such exceptions results in a call to `unexpected()`.

Example

```
#include <iostream>
using namespace std;

void subFunction() throw (int) {
    throw 10; // Throws an int exception
}

void outerFunction() throw (int) {
    try {
        subFunction(); // Calls a function with the same specification
    } catch (int e) {
        cout << "Caught exception: " << e << endl;
    }
}

int main() {
    try {
        outerFunction();
    } catch (...) {
        cout << "Unhandled exception!" << endl;
    }
    return 0;
}
```

Explanation

1. The `subFunction` and `outerFunction` both specify that they can throw int exceptions.
2. The `subFunction` throws an exception, which is caught by the `outerFunction`.

3. If an exception outside the specified types (e.g., `float`) is thrown, `unexpected()` would be called.

1.7 User Defined Exceptions

In C++, we can create custom exception classes to handle specific error conditions in a program. These are called **user-defined exceptions** and are typically derived from the standard exception class.

Key Features

1. **Custom Class**: Define a class for the exception, usually derived from the built-in exception class.
2. **Members**: The class can include data members, constructors, and member functions for custom error handling or displaying error messages.
3. **Throwing an Exception**: When an error condition occurs, an object of the user-defined exception class is created and thrown using the throw keyword.
4. **Catching the Exception**: To handle the exception, define a catch block that matches the type of the user-defined class.

Example: The following program throws and handles a user-defined exception if the second number is smaller than the first:

```
#include <iostream>
using namespace std;

// User-defined exception class derived from 'exception'
class DemoExpt : public exception {
public:
    void showErr() {
        cout << "Demo Error Occurred: Second number is not smaller than the first.\n";
    }
};

int main() {
```



```
    int a, b;
    cout << "Enter two numbers: ";
    cin >> a >> b;

    try {
        if (a < b) {
            // Create and throw user-defined exception object
            DemoExpt obj;
            throw obj;
        } else {
            cout << "Ans = " << (a - b) << endl;
        }
    } catch (DemoExpt ex) { // Handle user-defined exception
        ex.showErr();
    }

    cout << "Program ends\n";
    return 0;
}
```

Explanation

1. **Custom Exception Class**:
   - The `DemoExpt` class is derived from `exception`.
   - It includes a function `showErr` to display a custom error message.
2. **Throwing an Exception**:
   - If `a < b`, an object of `DemoExpt` is created and thrown.
3. **Catching the Exception**:
   - The catch block matches the `DemoExpt` type and calls the `showErr` function to display the error message.

1.8 Processing Unexpected Exceptions

In C++, exceptions are usually handled using a try-catch block. However, if an exception is thrown and no matching catch block is found, the exception handling system performs the following actions:

Default Process for Uncaught Exceptions

1. **Call to `unexpected()`**: If no catch block matches the exception type, the system calls the `unexpected()` function. By default, this function calls `terminate()`.
2. **Call to `terminate()`**: The `terminate()` function is responsible for defining actions during process termination. By default, it calls the `abort()` function to stop program execution.
3. **Process Abortion**: The program terminates without resolving the exception.

Customizing the Termination Process

You can change the default termination behavior by using the `set_terminate()` function, defined in the `<exception>` header. This function allows you to set a custom termination handler that will execute instead of the default `terminate()` function.

Example: Custom Termination Handler

The following program demonstrates how to set a custom handler for unexpected exceptions:

```
#include <exception>
#include <iostream>
using namespace std;

// Custom termination handler
void myhandler() {
    cout << "Inside custom terminate handler\n";
    abort(); // Aborts the program
}

int main() {
    // Set the custom termination handler
    set_terminate(myhandler);
```



```
    try {
        cout << "Inside try block\n";
        throw 100; // Throws an integer exception
    }
    catch (char a) { // This block doesn't catch the int exception
        cout << "Inside catch block\n";
    }
    return 0;
}
```

Explanation

1. **Custom Handler**:
   - The `myhandler` function is defined to replace the default termination behavior.
   - It displays a message and calls `abort()`.
2. **Uncaught Exception**:
   - The `try` block throws an integer exception (`throw 100`).
   - The `catch` block is designed to handle a `char` exception, so it doesn't match the thrown exception.
3. **Result**:
   - The program calls `terminate()`, which invokes the custom handler (`myhandler`).
   - The program outputs a message from `myhandler` and aborts.

1.9 Constructor and Exception Handling

In C++, constructors can be used in exception handling to initialize and store data related to the error condition. When creating a **user-defined exception class**, constructors are often used to pass and store error-specific data that can later be displayed or used in the error-handling process.

Key Points

1. **User-Defined Exception Class**:

- Typically derived from the built-in exception class.
- Contains data members, constructors, and functions for error handling.

2. **Parameterized Constructor**:
   - Used to pass and store invalid values or details about the error condition when the exception object is created.
3. **Throwing the Exception**:
   - An object of the user-defined exception class is created with the error-specific data and then thrown using the `throw` keyword.
4. **Catching the Exception**:
   - The `catch` block defines an object of the user-defined class to handle the exception and can access the stored error data.

Example

```
#include <exception>
#include <iostream>
using namespace std;

// User-defined exception class
class DemoExpt : public exception {
    int v1, v2; // Data members to store invalid values

public:
    // Constructor to collect error-specific data
    DemoExpt(int a, int b) {
        v1 = a;
        v2 = b;
    }

    // Function to display error details
    void ShowErr() {
```



```
        cout << "Error: Invalid values - " << v1 << " and " << v2 <<
endl;
    }
};

int main() {
    int a, b;
    cout << "Enter two numbers: ";
    cin >> a >> b;

    try {
        if (a < b) {
            // Create exception object with invalid values
            DemoExpt obj(a, b);
            throw obj; // Throw the object
        } else {
            cout << "Ans = " << (a - b) << endl;
        }
    }
    // Catch block to handle the exception
    catch (DemoExpt ex) {
        ex.ShowErr();
    }

    cout << "Program ends\n";
    return 0;
}
```

Explanation

1. **User-Defined Class**:
   - `DemoExpt` is derived from the exception class.
   - Contains:

- Data members (`v1, v2`) to store error-related data.
- A parameterized constructor to initialize these values.
- A function `ShowErr()` to display the error.

2. **Throwing the Exception**:
   - If the first number (`a`) is smaller than the second number (`b`), the program creates a `DemoExpt` object with a and b as arguments and throws it.
3. **Catching the Exception**:
   - The catch block handles the exception by creating a `DemoExpt object (ex)` and calling its `ShowErr()` method to display the error details.

1.10 Destructor and Exception Handling

In C++, when an exception is thrown and the program control passes from a `try` block to a catch block, the **destructors** of all automatic (local non-static) objects created since the start of the `try` block are called automatically. This process is known as **stack unwinding**.

Key Points

1. **Stack Unwinding**:
   - It is the process where destructors of all automatic objects are called in reverse order of their creation during exception handling.
   - Automatic objects are local non-static objects that are destroyed when the block or function in which they are declared terminates.
2. **Destructor Behavior**:
   - Destructors clean up resources, release memory, or perform any necessary final actions for objects.
   - This ensures that resources are not leaked even when an exception is thrown.
3. **Exception in Destructors**:
   - If a destructor throws an exception during stack unwinding and the exception is not caught, the **`terminate()`** function is called, which halts the program execution.

Example

```
#include <iostream>
using namespace std;

class Test {
    int id;
public:
    Test(int id) : id(id) {
        cout << "Constructor of object " << id << endl;
    }
    ~Test() {
        cout << "Destructor of object " << id << endl;
        // Uncommenting the below line will call terminate()
        // throw runtime_error("Exception in destructor");
    }
};

int main() {
    try {
        Test t1(1); // Automatic object 1
        Test t2(2); // Automatic object 2
        cout << "Throwing an exception...\n";
        throw runtime_error("Error occurred!");
    }
    catch (runtime_error &e) {
        cout << "Caught exception: " << e.what() << endl;
    }

    cout << "Program ends\n";
    return 0;
}
```


Explanation

1. **Constructor Calls**:
   - Objects `t1` and `t2` are created in the try block, and their constructors are called.
2. **Stack Unwinding**:
   - When an exception is thrown, control moves to the catch block, and the destructors of `t2` and `t1` are called in reverse order of their creation.
3. **Exception in Destructor**:
   - If a destructor throws an exception and it is not handled, the program will terminate by calling `terminate()`.

1.11 Exception And Inheritance

In C++, it is possible to throw and catch **class objects** as exceptions. When dealing with **inheritance** in exceptions, special rules apply due to the relationship between base and derived classes.

Key Points

1. **Catching Base and Derived Classes**:
   - If a derived class is thrown as an exception, it can be caught by a catch block for either the **derived class** or its **base class**.
   - A catch block for the derived class takes precedence over a base class block.
2. **Order of Catch Blocks**:
   - **Rule**: The catch block for the **derived class** must appear before the block for the **base class**.
   - If the base class catch block is placed first, the derived class catch block will never be executed because the base class block matches all derived types.
3. **Compiler Behavior**:
   - The C++ compiler might issue a **warning** if the base class catch block precedes the derived class block, but the code will still compile.
   - However, this can lead to unexpected behavior.

Example

```
#include <iostream>
using namespace std;

class Base {};
class Derived : public Base {};

int main() {
    Derived d; // Object of derived class
    try {
        throw d; // Throwing an object of derived class
    }
    catch (Derived &d) { // Catch derived class first
        cout << "Caught Derived Exception" << endl;
    }
    catch (Base &b) { // Catch base class
        cout << "Caught Base Exception" << endl;
    }

    return 0;
}
```

Explanation

1. **Throwing Derived Class**:
   - The object `d` of the Derived class is thrown.
2. **Order of Catch Blocks**:
   - The `catch` block for `Derived` is matched and executed first.
   - If the `catch` block for `Base` was placed first, it would catch all derived class objects, and the `Derived` block would never be executed.

2. Templates

Templates are a fundamental feature in C++ that allow the creation of **generic code**. They enable the definition of functions and classes that can operate with different data types without requiring separate implementations for each type.

What Are Templates?

- When defining a function or a class, we usually specify the **data types** of the variables to match the type of data and operations required.
- However, sometimes the same logic can be applied to different types of data.
  - **Example**: Swapping two values, whether `int`, `float`, or `char`, involves the same steps.
- Using **Generic Programming**, a single function or class can be designed to work with any type of data.
- Templates provide a way to achieve **"Same logic, different data types."**

Generic Programming with Templates

Definition: A programming style where one algorithm is designed to perform specific tasks, independent of the data type it operates on.

- The data type is specified later, during the function call or class instantiation.

Advantages of Templates

1. **Code Reusability**: Write one code block and use it for various data types.
2. **Efficiency**: Avoid code duplication for similar operations on different data types.
3. **Flexibility**: Handle primitive types (int, float, etc.), arrays, pointers, or even user-defined objects.
4. **Scalability**: Easily extend functionality to new data types by reusing the same logic.

The Power of Templates

- Templates enable programmers to write **generalized and reusable code**.
- They are particularly useful for data structures and algorithms, such as sorting, searching, and stacks, which can work with different types of data.

2.1 Generic Programming

Generic Programming is a programming paradigm where the algorithm is designed to perform a specific task without depending on a particular data type. Instead, the same algorithm can operate on **different data types**, with the type being specified **at runtime**.

In simpler terms, it's about writing algorithms and functions that can handle any data type without changing the underlying code for each data type. This allows the code to be more **flexible**, **reusable**, and **maintainable**.

Core Concept of Generic Programming

- The **same algorithm** works for different data types.
- The **data type** is **not hardcoded** into the algorithm or function but is defined during function or class invocation (at **runtime**).
- **Types are generic**, meaning we use placeholders for types (usually `T` or `typename`), and the actual type is specified when the function or class is called.

Generic Programming in C++ with Templates

In C++, **templates** provide the mechanism for implementing **generic programming**. They allow you to create **generic functions** and **classes** that can work with any data type.

- **Template Function**: Allows you to write a function that can work with any data type.
- **Template Class**: Allows you to create a class that can handle any data type.

The **template type** is used to define variables within the function or class, and the specific data type is provided when you call the function or instantiate the class.

2.2 Function Template

A **function template** allows you to define a generic function that can work with any data type. Instead of specifying a particular data type for function parameters and return types, you can define a **template** that allows the function to accept different types of data, such as `int`, `float`, `double`, or even user-defined types.

What is a Function Template?

A **function template** is a function that can operate with any data type. It is defined using the template keyword and a placeholder for the data type, which is typically represented by `T` or another generic name.

Syntax for Defining a Function Template:

```
template <class TypeName>
ReturnType FunctionName(Parameters) {
    // Function logic
}
```

- **`template`**: The keyword used to define a template function.
- **`class TypeName`**: This is the placeholder for the data type that will be used in the function. You can also use `typename` instead of class. The **type name** will be specified when the function is called.
- **`ReturnType`**: The return type of the function, which will be replaced with the template type during function call.
- **`FunctionName`**: The name of the function being defined.
- **Parameters**: The function's parameters, which can be of the template type.

When you call a template function, you provide the **actual data type** for `TypeName` in the form of `FunctionName<DataType>(arguments)`.

Benefits of Function Templates:

1. **Code Reusability**: You can write one function that works with multiple data types, avoiding code duplication.
2. **Flexibility**: The same function can work with different types of data, such as integers, floats, and user-defined types, with no additional code changes.
3. **Type Safety**: The compiler ensures that the function is used with compatible data types.
4. **Maintainability**: Having a single function template makes the code easier to maintain and extend.

Example 1: Function Template for Sum of Two Values

```
#include <iostream>
using namespace std;

template <class TempType>
```



```
TempType Add(TempType a, TempType b) {
    TempType c;
    c = a + b;
    return c;
}

int main() {
    int p = 4, q = 3, r;
    float x = 3.4, y = 1.2, z;

    // Call function with int type
    r = Add<int>(p, q);
    cout << "Sum of ints = " << r << endl;

    // Call function with float type
    z = Add<float>(x, y);
    cout << "Sum of floats = " << z << endl;

    return 0;
}
```

- **Explanation**:
  - The function `Add` is a template function that can add two values of any data type (`TempType`).
  - When the function is called in the `main()` function, the data type (int or float) is specified, and the compiler generates the appropriate function.
  - `Add<int>(p, q)` adds two integers, and `Add<float>(x, y)` adds two floating-point numbers.

Example 2: Function Template for Finding the Minimum Value in an Array

```
#include <iostream>
using namespace std;
```



```
// Function template to find the minimum value in an array
template <typename T>
T findMin(T arr[], int size) {
    T minVal = arr[0]; // Assume the first element is the minimum
    for (int i = 1; i < size; ++i) {
        if (arr[i] < minVal) {
            minVal = arr[i];
        }
    }
    return minVal;
}

int main() {
    // Example with integers
    int intArr[] = {10, 20, 5, 8, 15};
    int size = sizeof(intArr) / sizeof(intArr[0]);
    cout << "Minimum in integer array: " << findMin(intArr, size) <<
endl;

    // Example with floating-point numbers
    float floatArr[] = {2.5, 1.2, 3.8, 0.9, 4.5};
    size = sizeof(floatArr) / sizeof(floatArr[0]);
    cout << "Minimum in float array: " << findMin(floatArr, size) <<
endl;

    return 0;
}
```

- **Explanation**:
    - The `Min` function is a template function that finds the smallest element in an array.
    - It works with both `int` and `float` types because the data type is specified at the time of function call (`Min<int>` for integers, `Min<float>` for floats).

- Depending on the user's choice (opt), the function either works with an array of integers or floats to find the smallest number.

2.3 Overloading Function Templates

Function overloading is a form of **polymorphism** in C++, where multiple functions can have the same name but differ in their **parameter types** or the **number of parameters**. This allows the programmer to define several functions that perform similar operations but on different types of data.

When using **function templates**, overloading can also be applied to define a function template that works with different data types.

How Does Overloading Work with Templates?

If you have a **normal function** (non-template) and a **template function** with the same name, the compiler will use the normal function for **exact matches** of data types. If there is no exact match, the template function will be used.

For example:

```
#include <iostream>
using namespace std;

void add(int a, int b) {
    cout << "Sum of int = " << (a + b) << endl;
}

template <class TempType>
void add(TempType a, TempType b) {
    TempType c;
    c = a + b;
    cout << "Sum = " << c << endl;
}

int main() {
    int x = 5, y = 10;
```



```
    float p = 2.5f, q = 3.5f;

    add(x, y);        // Calls the normal function (add(int, int))
    add(p, q);        // Calls the template function (add<float>)

    return 0;
}
```

- **Explanation**:
  - The function `add(int, int)` is a regular function, so it will be used when the arguments are of type int.
  - For other types like float, the template function `add<TempType>` is used.

2.3 Class Template and Non-Type Parameters

In C++, **class templates** are used when we need to create a class that can operate on a variety of data types. A class template allows us to define a class where the data type is specified at the time of object creation (instantiation), making the class flexible and reusable for different types of data. Class templates are ideal for generic programming, as they allow the same class to work with different data types without duplicating the code.

Class Template Syntax

The basic syntax for defining a class template is:

```
template <class TypeName>
class ClassName {
    // Data members
    // Member functions
};
```

- **template <class TypeName>**: This part defines a template class. The `TypeName` is a placeholder for a data type that will be specified later when an object of the class is created.
- **class `ClassName`**: This defines the name of the class template. The class can now use the `TypeName` in its data members and member functions to allow flexibility in the types of data it can work with.

Object Creation for Class Template

When creating an object of a class template, we specify the **actual data type** to be used. The compiler then replaces the template with the specified data type, generating a new class that operates on that data type.

The syntax for creating an object of a class template is:

```
ClassName<DataType> object_name;
```

- **DataType**: This is the type of data (such as int, float, char, etc.) that the class template will use.
- **object_name**: This is the name of the object created from the class template.

Class Template using multiple parameters

In C++, a class template can accept multiple template parameters, allowing you to create a class that works with more than one data type. This feature enables you to design classes that can operate on various types of data and support multiple operations that involve these types.

Syntax of Class Template with Multiple Parameters:

```
template <class Type1, class Type2>
class ClassName {
    // Class definition
};
```

- `Type1` and `Type2` are placeholders for data types that will be defined when creating objects of the class.
- You can have more than two template parameters as needed.

Example Program Using Class Template with Multiple Parameters

```
#include <iostream>
using namespace std;

// Class template accepting two parameters
template <class Type1, class Type2>
class Adder {
```



```
private:
    Type1 first;   // First number
    Type2 second;  // Second number

public:
    // Constructor to initialize the numbers
    Adder(Type1 f, Type2 s) : first(f), second(s) {}

    // Method to add the two numbers and return the result
    Type1 add() {
        return first + second;
    }
};

int main() {
    // Creating objects of Adder class with int and float types
    Adder<int, float> adder1(5, 3.5);
    cout << "Sum (int + float): " << adder1.add() << endl;

    // Creating objects of Adder class with double and int types
    Adder<double, int> adder2(5.5, 7);
    cout << "Sum (double + int): " << adder2.add() << endl;

    // Creating objects of Adder class with float and double types
    Adder<float, double> adder3(2.5f, 4.5);
    cout << "Sum (float + double): " << adder3.add() << endl;

    return 0;
}
```

This C++ program demonstrates a **class template** with **multiple parameters**. The `Adder` class takes two data types as template parameters and adds two numbers of those types.

- The `Adder` class has a constructor to initialize the two numbers and a method `add()` to perform the addition.
- In `main()`, three Adder objects are created with different types (e.g., `int` and `float`, `double` and `int`).
- The result of the addition is printed for each pair of types.

Non-Type Parameters in Class Templates

In addition to type parameters, **non-type parameters** can also be used in class templates. These are values that are passed as arguments to the template. Non-type parameters are typically constant expressions, such as integers, pointers, or other compile-time constant values.

Syntax for non-type parameters:

```
template <class TypeName, int Size>
class ClassName {
    // Data members and member functions
};
```

- `int` Size is a non-type parameter. It represents a constant value that must be specified when the class is instantiated.
- Non-type parameters are often used when you need to define sizes, limits, or other constant values for a class template.

2.4 Template and Friend Generic Function

A **template friend function** combines the concepts of templates and friend functions. You can declare a **friend function** as a **template function** to allow it to work with any data type while still having access to the private members of a class.

To do this:

1. The class defines the `friend` function.
2. The `friend` function is also defined as a `template`, allowing it to work with various data types.

Syntax for a Template Friend Function:

```
template <typename T>
```



```
class ClassName {
    // Declare the template friend function
    friend ReturnType FriendFunctionName(ClassName<T>&);
};

// Define the friend function outside the class
template <typename T>
ReturnType FriendFunctionName(ClassName<T>& obj) {
    // Function implementation
}
```

Example of Template and Friend Function:

```
#include <iostream>
using namespace std;

template <typename T>
class Box {
    private:
        T length;

    public:
        Box(T l) : length(l) {}

        // Friend function declaration as template
        friend T getArea(Box<T>& b);  // Friend template function
};

// Friend template function definition
template <typename T>
T getArea(Box<T>& b) {
    return b.length * b.length;  // Access private member 'length' of Box
}
```



```
int main() {
    Box<int> intBox(5);
    Box<double> doubleBox(3.5);

    cout << "Area of int box: " << getArea(intBox) << endl;
    cout << "Area of double box: " << getArea(doubleBox) << endl;

    return 0;
}
```

In this example:

- The `Box` class is a template class that holds a data member length of type T.
- The `getArea` function is a friend function template that can access the private member length and calculate the area (assuming the box is square in this case).
- The `getArea` function is a template, meaning it can work with any data type (`int`, `double`, etc.) passed to the Box class.

2.5 The `typename` and `export` Keywords

In C++, two important keywords related to templates are **`typename`** and **`export`**. These keywords have specific roles that enhance the functionality and reusability of templates in C++. Below is an explanation of each keyword:

1. typename Keyword

The **typename** keyword in C++ serves two primary purposes, both related to templates.

First Use: Substituting class in Template Declaration

In C++, when defining a template, you can use either the class or typename keyword to declare a template parameter. Both keywords work identically in this context and specify that a given type is generic, allowing the template to be used with any data type.

Syntax:

```
template <typename TypeName>
```



```
FunctionOrClassDefinition;
```

Example:

```
template <typename tempType>
void Swap(tempType& a, tempType& b) {
    tempType t;
    t = a;
    a = b;
    b = t;
}
```

Here:

- **`typename`** declares `tempType` as a generic type.
- You can also use the **class** keyword instead of **`typename`**. There is **no functional difference** between using class and `typename` in this context. Both are used to specify the type parameter for the template.

Second Use: Indicating a Type Name in Nested Classes

The **`typename`** keyword is also used to inform the compiler that a name is a **type name** rather than an object or variable. This is particularly important in cases where a name is part of a template class or when accessing types nested within templates.

Example:

```
template <typename tempType>
class MyClass {
public:
    typename tempType::Name someObject;
};
```

Here:

- **`typename tempType::Name`** tells the compiler that `tempType::Name` is a type (not an object) and should be treated as such.
- This is necessary because the compiler might not automatically recognize it as a type due to its dependency on the template parameter.

2. `export` Keyword

The **`export`** keyword was introduced in earlier versions of C++ to help with separating the declaration and definition of template classes or functions. It allows you to declare a template in one file and then define it in another file, promoting reusability and cleaner code.

Purpose:

- The **`export`** keyword enables the template to be used across multiple translation units (source files) without requiring the full definition in each file.
- It is used to make templates reusable by declaring them in a header file while keeping their definitions in a source file.

However, the **`export`** keyword is not widely supported in modern C++ compilers and has been **deprecated**. Most modern C++ compilers do not implement it due to its complexity and lack of widespread use. Today, templates are typically declared and defined in header files, and **export** is rarely used in practice.

Example (conceptual, as `export` is not commonly supported):

```
export template <typename T>
void Swap(T& a, T& b) {  // Define the template function with `export`
    T t;
    t = a;
    a = b;
    b = t;
}
```

In this example:

- **export** allows the template definition in one file while still being usable in other files that include the header with the declaration.
- However, this syntax is mostly outdated and not widely supported.

2.6 Difference between Overloaded Functions and Function Templates

Aspect	Overloaded Function	Function Template
Definition	A feature in C++ where multiple functions with the same name but different parameter lists are defined.	A generic function that can work with any data type, defined using templates.
Flexibility	Works with specific data types defined in individual function definitions.	Works with multiple data types using a single generic definition.
Number of Functions	Requires separate function definitions for each data type.	Requires only one generic function definition for multiple data types.
Code Reusability	Limited, as each data type requires a separate function.	High, as a single function can handle multiple data types.
Implementation	Handled by the programmer, with manual definitions for each type.	Handled by the compiler, which generates type-specific functions from the template at compile time.
Performance	Slightly faster, as the function is specifically designed for a single data type.	May introduce minor overhead during compilation, as the compiler generates code for each used data type.
Usage	Best for situations where functions need specific logic for different data types.	Best for scenarios where the same logic can be applied to multiple data types.
Syntax	No use of `template` keyword.	Defined with the `template<class T>` or `template<typename T>` keyword.

2.7 Difference between Class Template and Function Template

Aspect	Class Template	Function Template
Definition	A template used to define a class that can work with multiple data types.	A template used to define a function that can work with multiple data types.
Purpose	Enables the creation of classes that handle generic types, such as data structures or utility classes.	Enables the creation of functions that can perform the same operation on different data types.
Scope of Use	Provides a blueprint for creating objects that operate on different data types.	Provides a mechanism for creating type-independent functions.
Complexity	Usually more complex as it deals with multiple member functions and variables.	Simpler as it focuses on a single operation or logic.
Usage	Used when multiple member functions or data members depend on the generic type.	Used for single operations like sorting, searching, or arithmetic where only the logic depends on the type.
Syntax	Defined using `template<class T>` or `template<typename T>` before the class definition.	Defined using `template<class T>` or `template<typename T>` before the function definition.
Example Usage	Often used to implement data structures (e.g., vector, stack, queue).	Often used for algorithms or utilities (e.g., addition, swapping values).
Object Creation	Requires creating objects of the class by specifying the data type at runtime.	Does not require an object; directly calls the function with the required data type.