42676-02 ADVANCED PROGRAMMING PRACTICE

Spring Term 2025

Robotic Engineering Dept.

Prof. Essaid Meryam

--- ---

Lecture Notes on C++ Templates

1. Introduction to Templates

Templates in C++ are a powerful feature that allows functions and classes to operate with generic types. This means you can write a single function or class that works with many different data types, reducing code duplication and increasing code reusability.

2. Function Templates

Function templates allow you to create functions that can operate with different data types.

Syntax:

```
template <typename T>
return_type function_name(parameters);
```

Example:

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

template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    int i1 = 10, i2 = 20;
    cout << "Max of int: " << max(i1, i2) << endl;

    double d1 = 10.5, d2 = 20.5;
    cout << "Max of double: " << max(d1, d2) << endl;

    return 0;
}
```

In this example, the `max` function template works with both `int` and `double` types.

3. Class Templates

Class templates allow you to create classes that can operate with different data types.

Syntax:

```
template <typename T>
class class_name {
```

```
    // class members
};
```

Example:

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

template <typename T>
class Pair {
    T first;
    T second;
public:
    Pair(T f, T s) : first(f), second(s) {}
    T getFirst() { return first; }
    T getSecond() { return second; }
};

int main() {
    Pair<int> intPair(10, 20);
    cout << "First: " << intPair.getFirst() << ", Second: " <<
intPair.getSecond() << endl;

    Pair<double> doublePair(10.5, 20.5);
    cout << "First: " << doublePair.getFirst() << ", Second: " <<
doublePair.getSecond() << endl;

    return 0;
}
```

In this example, the `Pair` class template works with both `int` and `double` types.

4. Template Parameters

Template parameters can be of two types:

- **Typed Parameters:** These are placeholders for data types.
- **Non-Typed Parameters:** These are placeholders for values.

Example:

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

template <typename T, int size>
class Array {
    T arr[size];
public:
    void set(int index, T value) {
        if (index >= 0 && index < size)
            arr[index] = value;
    }
    T get(int index) {
        if (index >= 0 && index < size)
            return arr[index];
        return T(); // return default value of T
    }
```

```
};

int main() {
    Array<int, 5> intArray;
    intArray.set(0, 10);
    cout << "Value at index 0: " << intArray.get(0) << endl;

    return 0;
}
```

In this example, `T` is a typed parameter and `size` is a non-typed parameter.

5. Template Specialization

Template specialization allows you to provide a special implementation for a specific data type.

Example:

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

template <typename T>
class myIncrement {
    T value;
public:
    myIncrement(T arg) : value(arg) {}
    T toIncrement() { return ++value; }
};

template <>
class myIncrement<char> {
    char value;
public:
    myIncrement(char arg) : value(arg) {}
    char uppercase() {
        if (value >= 'a' && value <= 'z')
            value += 'A' - 'a';
        return value;
    }
};

int main() {
    myIncrement<int> myint(7);
    myIncrement<char> mychar('s');
    myIncrement<double> mydouble(11.0);

    cout << "Incremented int value: " << myint.toIncrement() << endl;
    cout << "Uppercase value: " << mychar.uppercase() << endl;
    cout << "Incremented double value: " << mydouble.toIncrement() <<
endl;

    return 0;
}
```

In this example, the `myIncrement` class template has a special implementation for the `char` type.

6. Default Template Arguments

You can provide default values for template parameters.

Example:

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

template <typename T, int size = 10>
class Buffer {
    T buf[size];
public:
    void set(int index, T value) {
        if (index >= 0 && index < size)
            buf[index] = value;
    }
    T get(int index) {
        if (index >= 0 && index < size)
            return buf[index];
        return T(); // return default value of T
    }
};

int main() {
    Buffer<int> intBuffer;
    intBuffer.set(0, 10);
    cout << "Value at index 0: " << intBuffer.get(0) << endl;

    Buffer<int, 5> intBuffer5;
    intBuffer5.set(0, 20);
    cout << "Value at index 0: " << intBuffer5.get(0) << endl;

    return 0;
}
```

In this example, the `Buffer` class template has a default size of 10.

7. Template Definitions in Header Files

Template definitions often need to go in header files because the compiler needs the source code to instantiate objects.

Example:

```cpp
// myclass.h
#ifndef MYCLASS_H
#define MYCLASS_H

template <typename T>
class MyClass {
    T value;
public:
    MyClass(T v) : value(v) {}
    void set(T v) { value = v; }
    T get() { return value; }
};
```

```
#endif // MYCLASS_H

// main.cpp
#include "myclass.h"
#include <iostream>
using namespace std;

int main() {
    MyClass<int> intobj(0);
    MyClass<double> floatobj(1.2);
    cout << "intobj value = " << intobj.get() << " and floatobj value =
" << floatobj.get() << endl;
    intobj.set(1000);
    floatobj.set(0.12345);
    cout << "intobj value = " << intobj.get() << " and floatobj value =
" << floatobj.get() << endl;
    return 0;
}
```

In this example, the `MyClass` template definition is in the header file `myclass.h`.

Conclusion

Templates in C++ provide a powerful way to write generic code that can work with different data types. They help reduce code duplication and increase code reusability. Understanding how to use function templates, class templates, template parameters, template specialization, and default template arguments is essential for effective use of templates in C++.

Practice Examples with Solutions

Example 1: Generic Swap Function

Write a function template to swap the values of two variables of any type.

Solution:

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

template <typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int intA = 10, intB = 20;
    cout << "Before swap: intA = " << intA << ", intB = " << intB <<
endl;
    swap(intA, intB);
```

```
    cout << "After swap: intA = " << intA << ", intB = " << intB <<
endl;

    double doubleA = 1.5, doubleB = 2.5;
    cout << "Before swap: doubleA = " << doubleA << ", doubleB = " <<
doubleB << endl;
    swap(doubleA, doubleB);
    cout << "After swap: doubleA = " << doubleA << ", doubleB = " <<
doubleB << endl;

    return 0;
}
```

Output:

```
    Before swap: intA = 10, intB = 20
    After swap: intA = 20, intB = 10
    Before swap: doubleA = 1.5, doubleB = 2.5
    After swap: doubleA = 2.5, doubleB = 1.5
```

Example 2: Generic Stack Class

Implement a generic stack class using templates.

Solution:

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

template <typename T>
class Stack {
    vector<T> elements;
public:
    void push(T value) {
        elements.push_back(value);
    }

    void pop() {
        if (!elements.empty())
            elements.pop_back();
    }

    T top() {
        if (!elements.empty())
            return elements.back();
        throw runtime_error("Stack is empty");
    }

    bool isEmpty() {
        return elements.empty();
    }
};

int main() {
    Stack<int> intStack;
```

```
    intStack.push(10);
    intStack.push(20);
    cout << "Top element: " << intStack.top() << endl;

    Stack<double> doubleStack;
    doubleStack.push(1.5);
    doubleStack.push(2.5);
    cout << "Top element: " << doubleStack.top() << endl;

    return 0;
}
```

Output:

```
Top element: 20
Top element: 2.5
```

Example 3: Template Specialization for a Specific Type

Create a function template that prints the square of a number, but provide a specialized version for the `char` type that prints the character itself.

Solution:

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

template <typename T>
void printSquare(T value) {
    cout << "Square: " << value * value << endl;
}

template <>
void printSquare<char>(char value) {
    cout << "Character: " << value << endl;
}

int main() {
    printSquare(5); // int
    printSquare(3.5); // double
    printSquare('a'); // char

    return 0;
}
```

Output:

```
Square: 25
Square: 12.25
Character: a
```

Example 4: Class Template with Multiple Template Parameters

Create a class template for a simple 2D point that can handle different data types for the x and y coordinates.

Solution:

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

template <typename T1, typename T2>
class Point {
    T1 x;
    T2 y;
public:
    Point(T1 x, T2 y) : x(x), y(y) {}

    void print() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point<int, int> intPoint(10, 20);
    intPoint.print();

    Point<double, int> mixedPoint(10.5, 20);
    mixedPoint.print();

    return 0;
}
```

Output:

```
(10, 20)
(10.5, 20)
```

Exercises on C++ Templates

Exercise 1: Simple Function Template

Objective: Write a function template that finds the minimum of two values.

Instructions:

1. Create a function template `min` that takes two parameters of the same type and returns the smaller of the two.
2. Test your function with different data types (e.g., `int`, `double`, `char`).

Exercise 2: Class Template for a Generic Array

Objective: Implement a class template for a generic array that supports basic operations like setting and getting elements.

Instructions:

1. Create a class template `Array` with a fixed size.
2. Implement methods to set and get elements.
3. Test your class with different data types (e.g., `int`, `double`).

Exercise 3: Template Specialization for a Specific Type

Objective: Create a function template that prints the square of a number, but provide a specialized version for the `char` type that prints the character itself.

Instructions:

1. Write a function template `printSquare` that prints the square of a number.
2. Provide a specialized version for the `char` type that prints the character itself.
3. Test your function with different data types.

Exercise 4: Class Template with Multiple Template Parameters

Objective: Create a class template for a simple 2D point that can handle different data types for the x and y coordinates.

Instructions:

1. Create a class template `Point` with two template parameters for the x and y coordinates.
2. Implement a method to print the coordinates.
3. Test your class with different combinations of data types.

Exercise 5: Advanced Template Metaprogramming

Objective: Implement a template metaprogram to calculate the factorial of a number at compile time.

Instructions:

1. Create a template structure `Factorial` that calculates the factorial of a number using template specialization.
2. Test your template with different integer values.

Mini Project: Simple Text Editor

Project Description

Develop a simple text editor application that supports basic text manipulation and formatting. The application should demonstrate the use of multiple design patterns, including Singleton, Factory, Observer, Strategy, and Composite.

Features

1. **Text Editing**: Basic text input and manipulation (insert, delete, copy, paste).
2. **Text Formatting**: Apply different text formats (bold, italic, underline).
3. **File Operations**: Open, save, and close text files.
4. **Undo/Redo**: Support for undoing and redoing text operations.

5. **Search and Replace**: Basic search and replace functionality.

Design Patterns to Implement

1. **Singleton Pattern**: For managing the application's configuration settings.
2. **Factory Pattern**: For creating different text formatting strategies.
3. **Observer Pattern**: For updating the UI when text changes.
4. **Strategy Pattern**: For implementing different text formatting strategies.
5. **Composite Pattern**: For managing the document structure (e.g., paragraphs, lines).

Project Structure

1. **Configuration Manager**: Use the Singleton pattern to manage application settings.
2. **Text Formatting**: Use the Factory and Strategy patterns to create and apply different text formatting strategies.
3. **Document Structure**: Use the Composite pattern to manage the hierarchical structure of the document.
4. **UI Updates**: Use the Observer pattern to update the UI when the text changes.

Implementation Steps

Step 1: Singleton Pattern - Configuration Manager

- Create a `ConfigManager` class to manage application settings.

Step 2: Factory Pattern - Text Formatting

- Create a `TextFormatter` interface and concrete classes for different formatting strategies.

Step 3: Observer Pattern - UI Updates

- Create an `Observer` interface and a `Subject` class to notify observers of text changes.

Step 4: Strategy Pattern - Text Formatting

- Use the Strategy pattern to apply different formatting strategies to the text.

Step 5: Composite Pattern - Document Structure

- Create a `DocumentItem` interface and concrete classes for different document components.

Putting It All Together

- Combine all the components to create a simple text editor.

Expected Output

Formatted Text: *BOLD*Hello, World!