Advanced Programming Lab
CS6150

08-September-2025

Templates

(Slides Courtesy : Rupesh Nasre)

Recap : Classes and Objects

- Class, Objects
- Inheritance
- Polymorphism

Code Reusability

- Inheritance and Polymorphism gives some ways of improving code reusability.

- Are there other ways?
    - Counting the number of elements in an array does not depend on whether its an int /float / objects
        - Can we write one function to count the length of an array regardless of what the array is storing?
    - Stack supports push, pop and peek irrespective of what the stack is storing
        - Can we write one Stack class that implements Push/Pop/Peek that works for anything that the stack can store?

- Can be done using the notion of templates

Templates

- Allows code to be written once and used with many type

- Avoids code duplication

- Templates can be defined for functions, classes, or variables with types supplied as parameters.

Student Stack

```cpp
class student {
    string name;
    int age;

    public:

    student() {
        name = "";
        age = 0;
    }

    student(string n, int a){
        name = n;
        age = a;
    }

    ~student() {};

    void getDetails() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }

};
```

```cpp
class Stack {
private:
    int top;
    student arr[100];

public:
    Stack() {
        top = -1;
    }

    ~Stack() { }

    bool safeToPush() {
        return (top < 99);
    }
    void push(student x) {
        arr[++top] = x;
    }

    bool safeToPop() {
        return (top >= 0);
    }
    student pop() {
        return arr[top--];
    }

    bool safeToPeek() {
        return (top >= 0);
    }

    student peek() {
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

```cpp
int main() {

    Stack studentStack;

    student s1("Alice", 24);

    if(studentStack.safeToPush()) {

        studentStack.push(s1);
        cout << "Pushed to studentStack: student object" << endl;
    } else {
        cout << "studentStack Overflow!" << endl;
    }

    if(studentStack.safeToPeek()) {
        student topStudent = studentStack.peek();
        cout << "Top student details: ";
        topStudent.getDetails();
    } else {
        cout << "studentStack is empty!" << endl;
    }

    return 0;
}
```

Stack as a Template

```cpp
template <typename T>
class Stack {
private:
    int top;
    T arr[100];

public:
    Stack() {
        top = -1;
    }

    ~Stack() { }

    bool safeToPush() {
        return (top < 99);
    }
    void push(T x) {
        arr[++top] = x;
    }

    bool safeToPop() {
        return (top >= 0);
    }
    T pop() {
        return arr[top--];
    }

    bool safeToPeek() {
        return (top >= 0);
    }

    T peek() {
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

```cpp
int main() {
    Stack<int> stack(5);

    // Push elements onto the stack
    for (int i = 1; i <= 5; ++i) {
        if (stack.safeToPush()) {
            stack.push(i);
            cout << "Pushed: " << i << endl;
        } else {
            cout << "Stack Overflow! Cannot push " << i << endl;
        }
    }

    // Peek at the top element
    if (stack.safeToPeek()) {
        cout << "Top element is: " << stack.peek() << endl;
    } else {
        cout << "Stack is empty, cannot peek." << endl;
    }

    // Pop elements from the stack
    while (!stack.isEmpty()) {
        if (stack.safeToPop()) {
            cout << "Popped: " << stack.pop() << endl;
        } else {
            cout << "Stack Underflow! Cannot pop." << endl;
        }
    }

    // Attempt to pop from an empty stack
    if (stack.safeToPop()) {
        cout << "Popped: " << stack.pop() << endl;
    } else {
        cout << "Stack Underflow! Cannot pop." << endl;
    }

    return 0;
}
```

Stack as a Template

```cpp
template <typename T>
class Stack {
private:
    int top;
    T arr[100];

public:
    Stack() {
        top = -1;
    }

    ~Stack() { }

    bool safeToPush() {
        return (top < 99);
    }
    void push(T x) {
        arr[++top] = x;
    }

    bool safeToPop() {
        return (top >= 0);
    }
    T pop() {
        return arr[top--];
    }

    bool safeToPeek() {
        return (top >= 0);
    }

    T peek() {
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

```cpp
int main() {
    Stack<int> intStack(5);
    Stack<string> strStack(3);

    if (intStack.safeToPush()) {
        intStack.push(10);
        cout << "Pushed to intStack: 10" << endl;
    } else {
        cout << "intStack Overflow!" << endl;
    }

    if(strStack.safeToPush()) {
        strStack.push("Hello");
        cout << "Pushed to strStack: Hello" << endl;
    } else {
        cout << "strStack Overflow!" << endl;
    }

    if(intStack.safeToPeek()) {
        cout << "Top of intStack: " << intStack.peek() << endl;
    } else {
        cout << "intStack is empty!" << endl;
    }

    if(strStack.safeToPeek()) {
        cout << "Top of strStack: " << strStack.peek() << endl;
    } else {
        cout << "strStack is empty!" << endl;
    }

}
```

Stack as a Template

```cpp
class student {
    string name;
    int age;

    public:

    student() {
        name = "";
        age = 0;
    }

    student(string n, int a){
        name = n;
        age = a;
    }

    ~student() {};

    void getDetails() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }

};
```

```cpp
template <typename T>
class Stack {
private:
    int top;
    T arr[100];

public:
    Stack() {
        top = -1;
    }

    ~Stack() { }

    bool safeToPush() {
        return (top < 99);
    }
    void push(T x) {
        arr[++top] = x;
    }

    bool safeToPop() {
        return (top >= 0);
    }
    T pop() {
        return arr[top--];
    }

    bool safeToPeek() {
        return (top >= 0);
    }

    T peek() {
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

```cpp
int main() {
    Stack<int> intStack;
    Stack<student> studentStack;

    if (intStack.safeToPush()) {
        intStack.push(10);
        cout << "Pushed to intStack: 10" << endl;
    } else {
        cout << "intStack Overflow!" << endl;
    }

    if(studentStack.safeToPush()) {
        student s1("Alice", 24);
        studentStack.push(s1);
        cout << "Pushed to studentStack: student object" << endl;
    } else {
        cout << "studentStack Overflow!" << endl;
    }

    if(studentStack.safeToPeek()) {
        student topStudent = studentStack.peek();
        cout << "Top student details: ";
        topStudent.getDetails();
    } else {
        cout << "studentStack is empty!" << endl;
    }

    return 0;
}
```

Multiple Template Arguments

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


template <class T1, class T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) : first(a), second(b) {}

    void show() {
        cout << first << " and " << second << endl;
    }
};

int main() {
    Pair<float, int> pair1(3.14, 42);
    pair1.show();

    Pair<string, float> pair2("Pi", 3.14);
    pair2.show();
    return 0;
}
```

Function Templates

- Functions can also be implemented based on Templates

- Can be called for any datatype / class as long as the compiler knows how to evaluate > for T

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

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

bool operator>(string a, string b) {
    return a.length() > b.length();
}


int main() {
    cout << findMax(10, 20) << endl;
    cout << findMax("Anantha", "Padmanabha") << endl;
    return 0;
}
```

Default Arguments

- In a stack template, suppose we want to say
    - If no data type is specified then build a stack for integers

```
template <typename T>
class Stack {
private:
    int top;
    int capacity;
    T* arr;

public:
    Stack(int size) {
        capacity = size;
        arr = new T[capacity];
        top = -1;
    }

    ~Stack() {
        delete[] arr;
    }

    bool safeToPush() {
        return (top < capacity - 1);
    }
    void push(T x) {
        arr[++top] = x;
    }

    bool safeToPop() {
        return (top >= 0);
    }
    T pop() {
        return arr[top--];
    }

    bool safeToPeek() {
        return (top >= 0);
    }

    T peek() {
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

Default Arguments

- In a stack template, suppose we want to say
  - If no data type is specified then build a stack for integers

```cpp
template <typename T = int>
class Stack {
private:
    int top;
    int capacity;
    T* arr;

public:
    Stack(int size) {
        capacity = size;
        arr = new T[capacity];
        top = -1;
    }

    ~Stack() {
        delete[] arr;
    }

    bool safeToPush() {
        return (top < capacity - 1);
    }
    void push(T x) {
        arr[++top] = x;
    }

    bool safeToPop() {
        return (top >= 0);
    }
    T pop() {
        return arr[top--];
    }

    bool safeToPeek() {
        return (top >= 0);
    }

    T peek() {
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

```cpp
int main() {
    Stack<> intStack(5);
    Stack<string> strStack(3);

    if (intStack.safeToPush()) {
        intStack.push(10);
        cout << "Pushed to intStack: 10" << endl;
    } else {
        cout << "intStack Overflow!" << endl;
    }

    if(strStack.safeToPush()) {
        strStack.push("Hello");
        cout << "Pushed to strStack: Hello" << endl;
    } else {
        cout << "strStack Overflow!" << endl;
    }

    if(intStack.safeToPeek()) {
        cout << "Top of intStack: " << intStack.peek() << endl;
    } else {
        cout << "intStack is empty!" << endl;
    }

    if(strStack.safeToPeek()) {
        cout << "Top of strStack: " << strStack.peek() << endl;
    } else {
        cout << "strStack is empty!" << endl;
    }

}
```

Default Arguments

- Default arguments can only be at the end.

```cpp
template <typename T = int>
class Stack {
private:
    int top;
    int capacity;
    T* arr;

public:
    Stack(int size) {
        capacity = size;
        arr = new T[capacity];
        top = -1;
    }

    ~Stack() {
        delete[] arr;
    }

    bool safeToPush() {
        return (top < capacity - 1);
    }
    void push(T x) {
        arr[++top] = x;
    }

    bool safeToPop() {
        return (top >= 0);
    }
    T pop() {
        return arr[top--];
    }

    bool safeToPeek() {
        return (top >= 0);
    }

    T peek() {
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};
```

```cpp
int main() {
    Stack<> intStack(5);
    Stack<string> strStack(3);

    if (intStack.safeToPush()) {
        intStack.push(10);
        cout << "Pushed to intStack: 10" << endl;
    } else {
        cout << "intStack Overflow!" << endl;
    }

    if(strStack.safeToPush()) {
        strStack.push("Hello");
        cout << "Pushed to strStack: Hello" << endl;
    } else {
        cout << "strStack Overflow!" << endl;
    }

    if(intStack.safeToPeek()) {
        cout << "Top of intStack: " << intStack.peek() << endl;
    } else {
        cout << "intStack is empty!" << endl;
    }

    if(strStack.safeToPeek()) {
        cout << "Top of strStack: " << strStack.peek() << endl;
    } else {
        cout << "strStack is empty!" << endl;
    }

}
```

Template Specialization

- We can define how a function should behave for some specific Input Types.

- A generic Stack can have some methods

- But Student Stack can have some different methods

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

template <typename T>
void print(T value) {
    cout << "Input is neither an integer not a string. The value is " << value << endl;
}

// Specialization for strings
template <>
void print<string>(string s) {
    cout << "Input String is: " << s << endl;
}

template <>
void print<int>(int n) {
    cout << "Input Integer is: " << n << endl;
}

int main() {
    print(42);          // Uses int version
    print(3.14);        // Uses generic version
    print("Hello");     // Uses string version
    return 0;
}
```

Template Specialization

- If there are multiple template inputs, we can also specify what should happen if one of the template is specified.
    - Partial specialization

- Can be done only for class templates
    - Cannot be done for function templates

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


template <typename T1, class T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) : first(a), second(b) {}

    void show() {
        cout << "Second entry is not int. Values are" << first << " and " << second << endl;
    }
};

// Partial specialization for when U is int
template <typename T>
class Pair<T, int> {
    private:
    T first;
    int second;
public:
    Pair(T a, int b) : first(a), second(b) {}
    void show() { cout << "Second entry is int. Values are " << first << second<< endl; }
};


int main() {
    Pair<float, int> pair1(3.14, 42);
    pair1.show();

    Pair<string, float> pair2("Pi", 3.14);
    pair2.show();
    return 0;
}
```

Type Generality

- Code is generic
  - Not tied to a specific data type
  - Can operate on multiple types
    - Even types not known when the code is first written.

- Many Standard C++ libraries are written in this way so that it can be used across multiple data types / classes
  - Sorting, vectors, …

See you in the lab on Friday

Try out examples

Practise problems will be available by tomorrow