Module 39: Programming in C++

Template (Class Template): Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

- Understand Templates in C++

Module Outline

- What is a Template?
- Function Template
  - Function Template Definition
  - Instantiation
  - Template Argument Deduction
  - Example
- `typename`
- Class Template
  - Class Template Definition
  - Instantiation
  - Partial Template Instantiation & Default Template Parameters
  - Inheritance

What is a Template?: RECAP (Module 38)

- Templates are specifications of a collection of functions or classes which are parameterized by types
- Examples:
  - Function search, min etc.
    - The basic algorithms in these functions are the same independent of types
    - Yet, we need to write different versions of these functions for strong type checking in C++
  - Classes list, queue etc.
    - The data members and the methods are almost the same for list of numbers, list of objects
    - Yet, we need to define different classes

Function Template: Code reuse in Algorithms: RECAP (Module 38)

- We need to compute the maximum of two values that can be of:
  - `int`
  - `double`
  - `char *` (C-String)
  - `Complex` (user-defined class for complex numbers)
  - ...
- We can do this with overloaded `Max` functions:

```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

  With every new type, we need to add an overloaded function in the library!

- Issues in `Max` function
  - **Same algorithm** (compare two value using the appropriate operator of the type and return the larger value)
  - **Different code versions** of these functions for strong type checking in C++

Class Template: Code Reuse in Data Structure

- Solution of several problems needs stack (LIFO)
  - Reverse string (`char`)
  - Convert infix expression to postfix (`char`)
  - Evaluate postfix expression (`int` / `double` / `Complex` ...)
  - Depth-first traversal (`Node *`)
  - ...
- Solution of several problems needs queue (FIFO)
  - Task Scheduling (`Task *`)
  - Process Scheduling (`Process *`)
  - ...
- Solution of several problems needs list (ordered)
  - Implementing stack, queue (`int` / `char` / ...)
  - Implementing object collections (UDT)
  - ...
- Solution of several problems needs ...
- Issues in Data Structure
  - Data Structures are **generic - same interface, same algorithms**
  - C++ **implementations are different** due to element type

Stack of char and int

```
class Stack {
    char data_[100];           // Has type
    int top_;
public:
    Stack() :top_(-1) {}
    ~Stack() {}

    void push(const char& item) // Has type
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const char& top() const     // Has type
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

- Stack of char

```
class Stack {
    int data_[100];            // Has type
    int top_;
public:
    Stack() :top_(-1) {}
    ~Stack() {}

    void push(const int& item)  // Has type
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const int& top() const      // Has type
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

- Stack of int

- Can we combine these Stack codes using a type variable T?

Class Template

- A class template
  - describes how a class should be built
  - Supplies the class description and the definition of the member functions using some arbitrary type name, (as a place holder)
  - is a:
    - parameterized type with
    - parameterized member functions
  - can be considered the definition for a unbounded set of class types
  - is identified by the keyword `template`
    - followed by comma-separated list of parameter identifiers (each preceded by keyword `class` or keyword `typename`)
    - enclosed between $<$ and $>$ delimiters
    - followed by the definition of the class
  - is often used for container classes
  - Note that every template parameter is a built-in type or class – type parameters

Stack as a Class Template: Stack.h

```
template<class T>
class Stack {
    T data_[100];
    int top_;
public:
    Stack() :top_(-1) {}
    ~Stack() {}

    void push(const T& item)
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const T& top() const
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

- Stack of type variable `T`
- **The traits of type variable `T` include**
  copy assignment operator (`T operator=(const T&)`)
- **We do not call our template class as `stack` because `std` namespace has a `class stack`**

Reverse String: Using Stack template

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() {
    char str[10] = "ABCDE";

    Stack<char> s;        // Instantiated for char

    for (unsigned int i = 0; i < strlen(str); ++i)
        s.push(str[i]);

    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }

    return 0;
}
```

- Stack **of type** char

Postfix Expression Evaluation: Using Stack template

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() {
    // Postfix expression: 1 2 3 * + 9 -
    unsigned int postfix[] = { '1', '2', '3', '*', '+', '9', '-' }, ch;

    Stack<int> s;        // Instantiated for int

    for (unsigned int i = 0; i < sizeof(postfix) / sizeof(unsigned int); ++i) {
        ch = postfix[i];
        if (isdigit(ch)) { s.push(ch - '0'); }
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch (ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\nEvaluation " << s.top();

    return 0;
}
```

- `Stack` **of type** `int`

Template Parameter Traits

- Parameter Types
  - may be of any type (including user defined types)
  - may be parameterized types, (that is, templates)
  - MUST support the methods used by the template functions:
    - What are the required constructors?
    - The required operator functions?
    - What are the necessary defining operations?

Function Template Instantiation: RECAP (Module 38)

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);
template<> char *Max<char *>(char *x, char *y);
template <class T, int size> Type Max(T x[size]);

int a, b; Max(a, b); // Binds to Max<int>(int, int);
double c, d; Max(c, d); // Binds to Max<double>(double, double);
char *s1, *s2; Max(s1, s2); // Binds to Max<char*>(char*, char*);

int pval[9]; Max(pval); //Error!
```

- Three kinds of conversions are allowed
  - L-value transformation (for example, Array-to-pointer conversion)
  - Qualification conversion
  - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same

Class Template Instantiation

- Class Template is instantiated only when it is required:
  - `template<class T> class Stack;` is a forward declaration
  - `Stack<char> s;` is an error
  - `Stack<char> *ps;` is okay
  - `void ReverseString(Stack<char>& s, char *str);` is okay
- Class template is instantiated before
  - An object is defined with class template instantiation
  - If a pointer or a reference is dereferenced (for example, a method is invoked)
- A template definition can refer to a class template or its instances but a non-template can only refer to template instances

Class Template Instantiation Example

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

template<class T> class Stack;                      // Forward declaration

void ReverseString(Stack<char>& s, char *str);  // Stack template definition is not needed

template<class T>                                   // Definition
class Stack { T data_[100]; int top_;
public: Stack() :top_(-1) {} ~Stack() {}

    void push(const T& item) { data_[++top_] = item; }
    void pop() { --top_; }
    const T& top() const { return data_[top_]; }
    bool empty() const { return top_ == -1; }
};
int main() {
    char str[10] = "ABCDE";
    Stack<char> s;                                  // Stack template definition is needed

    ReverseString(s, str);

    return 0;
}
void ReverseString(Stack<char>& s, char *str) { // Stack template definition is needed
    for (unsigned int i = 0; i < strlen(str); ++i) s.push(str[i]);

    cout << "Reversed String: ";
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
```

Partial Template Instantiation and Default Template Parameters

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

template<class T1 = int, class T2 = string> // Version 1 with default parameters
class Student { T1 roll_; T2 name_;
public: Student(T1 r, T2 n) : roll_(r), name_(n) {}
    void Print() const { cout << "Version 1: (" << name_ << ", " << roll_ << ")" << endl; }
};
template<class T1> // Version 2: Partial Template Specialization
class Student<T1, char *> { T1 roll_; char *name_;
public: Student(T1 r, char *n) : roll_(r), name_(strcpy(new char[strlen(n) + 1], n)) {}
    void Print() const { cout << "Version 2: (" << name_ << ", " << roll_ << ")" << endl; }
};
int main() {
    Student<int, string> s1(2, "Ramesh");    // Version 1: T1 = int, T2 = string
    Student<int>         s2(11, "Shampa");   // Version 1: T1 = int, defa T2 = string
    Student<>            s3(7, "Gagan");     // Version 1: defa T1 = int, defa T2 = string
    Student<string>      s4("X9", "Lalita"); // Version 1: T1 = string, defa T2 = string
    Student<int, char*>  s5(3, "Gouri");     // Version 2: T1 = int, T2 = char*

    s1.Print(); s2.Print(); s3.Print(); s4.Print(); s5.Print();

    return 0;
}
-----
Version 1: (Ramesh, 2)
Version 1: (Shampa, 11)
Version 1: (Gagan, 7)
Version 1: (Lalita, X9)
Version 2: (Gouri, 3)
```

Templates and Inheritance: Example (List.h)

```
#ifndef __LIST_H
#define __LIST_H

#include <vector>
using namespace std;

template<class T>
class List {
public:
    void put(const T &val) { items.push_back(val); }
    int length() { return items.size(); }
    bool find(const T &val) {
        for (unsigned int i = 0; i < items.size(); ++i)
            if (items[i] == val) return true;
        return false;
    }
private:
    vector<T> items;
};

#endif // __LIST_H
```

- List is basic container class

Templates and Inheritance: Example (Set.h)

```
#ifndef __SET_H
#define __SET_H

#include "List.h"

template<class T>
class Set {
public:
    Set()   { };
    virtual void add(const T &val);
    int length();
    bool find(const T &val);
private:
    List<T> items;
};

template<class T>
void Set<T> :: add(const T &val)
{
    if (items.find(val)) return;
    items.put(val);
}
template<class T> int Set<T> :: length() { return items.length(); }
template<class T> bool Set<T> ::find(const T &val) { return items.find(val); }
#endif // __SET_H
```

- Set is a base class for a set
- Set uses List for container

Templates and Inheritance: Example (BoundSet.h)

```
#ifndef __BOUND_SET_H
#define __BOUND_SET_H

#include "Set.h"

template<class T>
class BoundSet : public Set<T> {
    public:
        BoundSet(const T &lower, const T &upper);
        void add(const T &val);
    private:
        T min;
        T max;
};

template<class T> BoundSet<T>::BoundSet(const T &lower, const T &upper)
                                   : min(lower), max(upper) { }
template<class T> void BoundSet<T>::add(const T &val) {
    if (find(val)) return;
    if ((val <= max) && (val >= min))
        Set<T>:: add(val);
}
#endif // __BOUND_SET_H
```

- BoundSet is a specialization of Set
- BoundSet is a set of bounded items

Templates and Inheritance: Example (Bounded Set Application)

```
#include <iostream>
using namespace std;
#include "BoundSet.h"

int main() {
    int i;
    BoundSet<int> bsi(3, 21);
    Set<int> *setptr = &bsi;

    for (i = 0; i < 25; i++) setptr->add(i);

    if (bsi.find(4))
        cout << "We found an expected value\n";

    if (bsi.find(0) || bsi.find(25))  {
        cout << "We found an Unexpected value\n";
        return -1;
    }
    else
        cout << "We found NO unexpected value\n";

    return 0;
}
-----
We found an expected value
We found NO unexpected value
```

- Uses BoundSet to maintain and search elements

Module Summary

- Introduced the templates in C++
- Discussed class templates as generic solution for data structure reuse
- Explained partial template instantiation and default template parameters
- Demonstrated templates on inheritance hierarchy
- Illustrated with examples

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655