Advanced Programming

Templates

Zeinab Zali

References: (1) "C++ How to program" Deitel&Deitel, (2) "A Tour of C++" Bjarne Stroustrup, (3) Other useful learning pages such as geeksforgeeks and tutorialpoints

ECE Department, Isfahan University of Technology

List of any Type

Emagin

- You need a List of Student objects
- Also, you need a List of int numbers
- Also, you need a List of strings

What is your solution?

- Implement a List that has a data member of class Student
- again, implement a List that has an int data member
- again, implement a List that has a string data member

Are you satisfied with this way?

List Data Structure

Emagin

- You want to provide a library of data structures and many algorithms for manipulating them

What is your solution?

Generic Programming

It's possible to understand the concept of a List independent of the type of the items being placed in the List using **generic programming**

- We define a list generically, then use type-specific versions of this generic list class when creating an object of a list.
- A template is a blueprint or formula for creating a generic class or a function.

Templates

A **template** is a class or a function that we parameterize with a set of types or values

- Class templates are called parameterized types, because they require one or more type parameters to specify how to customize a generic class template to form a class-template specialization.
- To produce many specializations you write only one class-template definition
- When a particular specialization is needed, you use a concise, simple notation, and the compiler writes the specialization source code

Template Definition

- All templates begin with keyword template followed by a list of template parameters enclosed in angle brackets (< and >) template<typename T>
- Each template parameter that represents a type must be preceded by either of the interchangeable keywords typename or class (though type-name is preferred).
- The type parameter T (or any valid identifier) acts as a placeholder for the list's element type

Function Template Implementation

```
template <typename T>
int search(T* array, int len, T a){
    for(int i=0; i< len; i++)
        if (array[i]==a)
            return i;
    return -1;
}
```

Using a Function Template

It is required to specify the generic types, when calling a function template

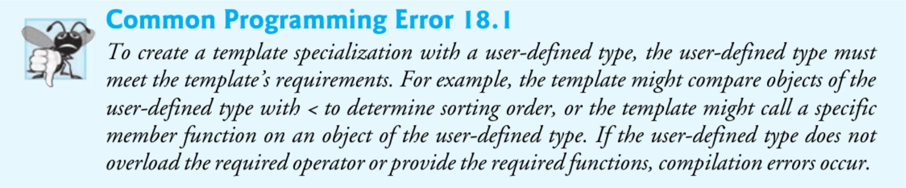
```
    int a[]={2,10,30,20,1};
    cout<<search<int>(a,5,10)<<endl;

    string crs[] ={"AP", "Math", "OS","Probability", "Database"};
    cout<<search<string>(crs,5,"OS")<<endl;
```

Template's Requirements

```cpp
    Student std[3];
    std[0].id = 1; std[0].name = "Mina";
    std[1].id = 2; std[1].name = "Hamid";
    std[2].id = 3; std[2].name = "Arya";
    Student s;
    s.name = "Arya"; s.id = 3;
    //error, no match for operator== (operand types are Student and Student)
    int indx = search<Student>(std,3,s);
```

Template's Requirements



Common Programming Error 18.1

To create a template specialization with a user-defined type, the user-defined type must meet the template's requirements. For example, the template might compare objects of the user-defined type with < to determine sorting order, or the template might call a specific member function on an object of the user-defined type. If the user-defined type does not overload the required operator or provide the required functions, compilation errors occur.

Template’s Requirements

```
bool operator==(Student s1, Student s2){
    if(s1.name==s2.name && s1.id == s2.id)
        return true;
    return false;
}
```

```
    Student std[3];
    std[0].id = 1; std[0].name = "Mina";
    std[1].id = 2; std[1].name = "Hamid";
    std[2].id = 3; std[2].name = "Arya";
    Student s;
    s.name = "Arya"; s.id = 3;
    //ok, because we implemented operator== for Students
    cout<<search<Student>(std,3,s)<<endl;
```

Non-Type Parameters

It's also possible to use nontype template parameters, which can have default arguments and are treated as constants

```
template <typename T, size_t num>
int find(T* array, T a){
    for(int i=0; i< num; i++)
        if (array[i]==a)
            return i;
    return -1;
}
```

```
    int nums[]={2,10,30,20,1};
    cout<<find<int,5>(nums,10)<<endl;
```

Template with Multiple Types

It's also possible to use more than one Type name.

```
template<typename T1, typename T2>
T1 min(T1 a, T2 b){
    if(a<b)
        return a;
    return (T2)b;
}
```

Default Type Arguments and Type Deduction

In order to instantiate a function template, every template argument must be known, but not every template argument has to be specified, but sometimes, we can use an implicit instantiation of a template function

- A type parameter can specify a default type argument.

```
template <typename T=int>
int search(T* array, int len, T a){
```

- When possible, the compiler will deduce the missing template arguments from the function arguments.

```
    //ok, because Student type is deducted from std and
    s types
    cout<<search(std,3,s)<<endl;
```

Class Template Implementation

It is sufficient to place template <typename T> before the class definition

```
template <typename T>
class Node{
    friend ostream& operator<<<>(ostream &out,const List<T>);
    friend class List<T>;
    T data;
    Node *next;
public:
    Node();
    explicit Node(T);
};
```

Class Template Implementation

It is sufficient to place template <typename T> before the class definition

```cpp
template <typename T>
class List{
    friend ostream& operator<<<>(ostream &out,const List<T>);
public:
    explicit List(); // constructor
    void addTail(T);
    void addHead(T);
    int rmTail();
    int rmHead();
    T operator[](int) const;
    T & operator[](int);
private:
    Node<T> *head;
};
```

Declaring Methods of Class Template

If we want to declare the methods of a class template outside the class definition, each method begin with the template keyword followed by the same set of template parameters as the class template.

```
template <typename T>
void List<T>::addTail(T d){
    Node<T> *node = new Node<T>(d);
    Node<T> * p = head;
    while(p->next!=nullptr)
        p = p->next;
    p->next = node;
}
```

Instantiating from a Class Template

```
List<int> num_list;
num_list.addTail(3);
num_list.addTail(4);
num_list.addTail(10);

List<string> str_list;
str_list.addTail("Welcome");
str_list.addTail("C++");
str_list.addTail("Good luck");
```

Friend Function or Class in Templates

Forward declaration of class List and non-member operator functions are required to announce that class or functions exist so they can be used in the friend declaration.

```
template <typename T> class List;
template <typename T>  ostream& operator<<(ostream &out, const List<T>);
template <typename T>
class Node{
    friend ostream& operator<<<>(ostream &out,const List<T>);
    friend class List<T>;
    T data;
    Node *next;
public:
    Node();
    explicit Node(T);
};
```

Pay attention to <> in the operator freindship declaration instead of specifing <T>