TEMPLATES AND GENERIC PROGRAMMING

Bùi Tiến Lên

2022

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Introduction

- Frequently, we have to implement the same functions or classes for arguments on different data types.
- The templates enable us to implement the function only once to be used for different argument data types.

The Different Types of Template Declarations

A template declaration can be

- A declaration or definition of a function
- A declaration or definition of a class
- A definition of a member function or a member class of a class template
- A definition of a static data member of a class template
- A definition of a static data member of a class nested within a class template
- A definition of a member template of a class or class template

Function Templates

An example

- Overloaded functions make programming convenient because only one function name must be remembered for a set of functions that perform similar operations.. Each of the functions, however, must still be written individually, even if they perform the same operation

```
int Max(int a, int b) {
  return (a>b) ? a : b;
}
```

```
double Max(double a, double b) {
  return (a>b) ? a : b;
}
```

- Templates enable us to write the function once

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

Function Templates

Concept 1

A **function template** is a "generic" function that can work with any data type. The programmer writes the specifications of the function, but substitutes parameters for data types. When the compiler encounters a call to the function, it generates code to handle the specific data type(s) used in the call

Syntax

The format of this declaration is

```
template <parameter list>
function declaration
```

Note

The entire template code is *usually* located in *a header file*.

Example

```
template <class T>
void swapVars(T &var1, T &var2) {
  T temp;
  temp = var1;
  var1 = var2;
  var2 = temp;
}
```

Note

All type parameters defined in a function template must appear at least once in the function parameter list.

Class templates

Class templates

Concept 2

Templates may also be used to create generic classes and abstract data types. **Class templates** allow us to create one general version of a class without having to duplicate code to handle multiple data types.

Syntax

The format of this declaration is

```
template <parameter list>
class declaration
```

Listing 1

```
template <typename T>
class CustomizableHuman {
public:
    void SetAge (const T& newValue) { Age = newValue; }
    const T& GetAge() const {return Age;}
private:
    T Age;   // T is type we choose to customize this template for!
};
void main() {
  // instantiate for type int
  CustomizableHuman<int> NormalLifeSpan;
  NormalLifeSpan.SetAge(80);
  // instantiate for type long long
  CustomizableHuman<long long> LongLifeSpan;
  LongLifeSpan.SetAge(3147483647);
  // instantiate for type short
  CustomizableHuman<short> ShortLifeSpan;
  ShortLifeSpan.SetAge(40);
}
```

Template Instantiation and Specialization

- The word *instantiation* normally refers to *objects* as instances of *classes*.
- In case of templates, *instantiation* is the act *or process of* creating a specific type from a template declaration and one or more template arguments.

```
CustomizableHuman<int> NormalLifeSpan;
```

- The specific type created as a result of this instantiation is called a *specialization*.

Declaring Templates with Multiple Parameters

- The template parameter list can be expanded to declare multiple parameters separated by a comma.

```
template <typename T1, typename T2>
class HoldsPair {
private:
    T1 Value1;
    T2 Value2;
public:
    // Constructor that initializes member variables
    HoldsPair (const T1& value1, const T2& value2) {
        Value1 = value1;
        Value2 = value2;
    };
};
void main() {
  // A template instantiation that pairs an int with a double
  HoldsPair <int, double> pairIntDouble (6, 1.99);
  // A template instantiation that pairs an int with an int
  HoldsPair <int, int> pairIntDouble (6, 500);
}
```

Declaring Templates with Default Parameters

- We could modify the previous version of `HoldsPair <...>` to declare `int` as the default template parameter type.

```
template <typename T1=int, typename T2=int>
class HoldsPair {
  // ... method declarations
};
void main() {
  // A template instantiation that pairs an int
  // with an int (default type)
  HoldsPair <> pairIntDouble (6, 500);
}
```

Template params are not data type

- Besides the data type params, class can have non-type arguments.

```
template<class T, int size>
class Myfilebuf {
private:
  T* filepos;
  static int array[size];
public:
  Myfilebuf() { /* ... */ }
  ~Myfilebuf() { }
  // ... Other function declarations
};
```

Template Classes and static Members

- A static member is shared across all instances of a template class with the same *specialization*.
- Each *specialization* of the template class effectively gets its *own static variable*.
- **Note**: do not forget *static member initialization*

Listing 2

```
#include <iostream>
using namespace std;
template <typename T>
class TestStatic {
private:
  static int staticValue;
public:
  void setValue(int value) { staticValue = value; }
  int getValue() { return staticValue; }
};
// static member initialization
template<typename T> int TestStatic<T>::staticValue;
int main() {
  TestStatic<int> Int_Year;
  TestStatic<int> Int_2;
  TestStatic<double> Double_1;
  TestStatic<double> Double_2;
  cout << "Setting staticValue for Int_Year to 2011" << endl;
  Int_Year.setValue(2011);
  cout << "Setting staticValue for Double_2 to 1011" << endl;
  Double_2.setValue(1011);
  cout << "Int_2.staticValue = " << Int_2.getValue() << endl;
  cout << "Double_1.staticValue = " << Double_1.getValue() << endl;
  return 0;
}
```

Template inheritance

- It is possible to inherit from a template class. All the usual rules for inheritance and polymorphism apply.

```
template<typename T>
class Base {
private:
  T data;
public:
  void set(T val) { data = val; }
};

template<typename T>
class Derived1 : public Base<T> {
public:
  void set(T val) { Base<T>::set(val); }
};
```

```
class Derived2 : public Base<int> {
public:
  void set(int val) { Base<int>::set(val); }
};

int main() {
  Derived1<double> obj1;
  Derived2 obj2;
  obj1.set(4.0);
  obj2.set(1);
  return 0;
}
```

Variadic template

- Variadic templates are supported by C++ (since the C++11 standard)

Syntax

The format of this declaration is

```
template <typename First, typename... Rest>
function/class declaration
```

Listing 3

```cpp
// base case
void print() {
  cout << "I am empty function and called at last";
}

// recursive
template <typename T, typename... Ts>
void print(T var1, Ts... var2) {
  cout << var1 << endl;
  print(var2...);
}
```

Generic programming

Generic programming

Concept 3

Generic programming means writing code that can be reused for objects of many different types.

Three primary tasks:

- Categorize abstractions in a domain into concepts
- Implement generic algorithms based on concepts
- Build concrete models for the concepts

Characteristics of Generic Libraries

- **Reusable**: able to operate on user-defined types
- **Composable**: able to operate on data types defined in another library
- **Efficient**: performance on par with non-generic, hand-coded implementations

Generic Programming Process

1. The **Generic Programming process** focuses on finding commonality among similar implementations of the same algorithm, then providing suitable abstractions in the form of concepts so that a single, generic algorithm can realize many concrete implementations.
2. This process, called **lifting**, is repeated until the generic algorithm has reached a suitable level of abstraction, where it provides maximal reusability without sacrificing performance.
3. Dual to the lifting process is **specialization**, which synthesizes efficient concrete implementations for particular uses of a *generic algorithm*. Only by balancing the lifting and specialization processes can we ensure that the resulting generic algorithms are both reusable and efficient.

Lifting

```
int sum(int* array, int n) {
  int result = 0;
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

↓

```
template<typename T>
T sum(T* array, int n) {
  T result = 0;
  for (int i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Workshop

Quiz

1. Why is it more convenient to write a function template than a series of overloaded functions?

   ....

   ....

   ....

2. Why must we be careful when writing a function template that uses operators such as `[]` with its parameters?

   ....

   ....

   ....

Quiz (cont.)

3. Do we need to specify template arguments when invoking a template function?

   ....

   ....

   ....

4. How many instances of static variables exist for a given template class?

   ....

   ....

   ....

Exercises

1. Implement template function for search operation
2. Implement template function for sort operation

References

Deitel, P. (2016).
C++: How to program.
Pearson.

Gaddis, T. (2014).
Starting Out with C++ from Control Structures to Objects.
Addison-Wesley Professional, 8th edition.

Jones, B. (2014).
Sams teach yourself C++ in one hour a day.
Sams.