C++ Template

한양대학교 컴퓨터소프트웨어학부

2017년 2학기

Introduction

- C++ templates
  - Allow very "general" definitions for functions and classes
  - Type names are "parameters" instead of actual types

Polymorphism

- One of four polymorphisms in C++:
  - Subtype polymorphism
    - Runtime polymorphism
  - Parametric polymorphism (C++ template !!)
    - Compile-time polymorphism
  - Ad-hoc polymorphism
    - Overloading
  - Coercion polymorphism
    - (Implicit or explicit) casting

Function Templates

- Recall function swapValues:

```
void swapValues(int& var1, int& var2)
{
    int temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

- Applies only to variables of type int
- But code would work for any types!

Function Templates vs. Overloading

- Could overload function for char's:

```
void swapValues(char& var1, char& var2)
{
    char temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

- But notice: code is nearly identical!
  - Only difference is type used in 3 places

Function Template Syntax

- Allow "swap values" of any type variables:

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

- First line called "template prefix"
  - Tells compiler what's coming is "template"
  - And that T is a type parameter

Template Prefix

- Recall:
  template<class T>
- In this usage, "class" means "type", or "classification"
- Can be confused with other "known" use of word "class"!
  - C++ allows keyword "typename" in place of keyword "class" here
  - But most use "class" anyway

Template Prefix 2

- Again:
  template<class T>
- T can be replaced by any type
  - Predefined or user-defined (like a C++ class type)
- In function definition body:
  - T used like any other type
- Note: can use other than "T", but T is "traditional" usage

Function Template Definition

- swapValues() function template is actually large "collection" of definitions!
  - A definition for each possible type!
- Compiler only generates definitions when required
  - But it's "as if" you'd defined for all types
- Write one definition → works for all types that might be needed

Calling a Function Template

- Consider following call:
  swapValues(int1, int2);
  - C++ compiler "generates" function definition for two int parameters using template
- Likewise for all other types
- Needn’t do anything "special" in call
  - Required definition automatically generated

Generic Programming

Generic programming is a style of computer programming in which algorithms are written in terms of **to-be-specified-later** types that are then instantiated when needed for specific types provided as parameters.[wikipedia]

- C++ Standard Template Library (STL).
- Data containers such as matrix, vector, array, image, etc.
- Algorithms such as sorting, searching, hashing, etc.
- ...

Generic Programming

```
// Suppose we want to sort an integer array.

void SelectionSort(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

Generic Programming

```cpp
// Suppose we want to sort an integer array.

void SelectionSort(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```cpp
// We also want to sort a double array.

void SelectionSort(double* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    double tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

Generic Programming

```
// Suppose we want to sort an integer array.

void SelectionSort(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    int tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
// We also want to sort a double array.

void SelectionSort(double* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    double tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}

// And also a string array.

void SelectionSort(string* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    string tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

C++ Template

- C++ template allows us to avoid this repeated codes.

```
// Suppose we want to sort an array of type T.

template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    // Swap array[i] and array[min_idx].
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

C++ Template

```
// Suppose we want to sort an integer array.

template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    T tmp = array[i];
    array[i] = array[min_idx];
    array[min_idx] = tmp;
  }
}
```

```
int main() {
  int array[] = { 2, 5, 3, 1, 4 };
  const int size = sizeof(array) / sizeof(int);
  SelectionSort<int>(array, size);    // You may use SelectionSort(array, size);
  for (int i = 0; i < size; ++i) cout << " " << array[i];
  cout << endl;
  return 0;
}
```

C++ Template

```
template <typename T>
void Swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}

template <typename T>
void SelectionSort(T* array, int size) {
  for (int i = 0; i < size; ++i) {
    int min_idx = i;
    for (int j = i + 1; j < size; ++j) {
      if (array[min_idx] > array[j])
        min_idx = j;
    }
    Swap(array[i], array[min_idx]);  // Clearly states the meaning of operation.
  }
}
```

C++ Template

- Functions and classes can be templated.
- Template parameters can be typenames (= classes) or integers.

```
template <class First, class Second>  // Same as <typename First, typename Second>.
struct Pair {
  First first;
  Second second;
};

template <typename T, int d>   // d must be a constant integer.
void Reverse(T array[d]) {     // Same as (T* array) - array size is not checked.
  for (int i = 0; i < d / 2; ++i) Swap(array[i], array[d - i - 1]);
}

int main() {
  int array[10] = { ... };
  int size = 10;
  Reverse<int, 10>(array);    // OK.
  Reverse<int, size>(array);  // Error.
  return 0;
}
```

C++ Template

```
template <class First, class Second>
struct Pair {
  First first;
  Second second;

  Pair(const First& f, const Second& s) : first(f), second(s) {}
};

template <class First, class Second>
Pair<First, Second> MakePair(const First& first, const Second& second) {
  return Pair<First, Second>(first, second);
}

int main() {
  Pair<int, int> p = MakePair(10, 10);  // Equivalently MakePair<int, int>(10, 10);
  Pair<int, int> q = Pair<int, int>(20, 20);
  return 0;
}
```

Another Function Template

- **Declaration/prototype:**

```
Template<class T>
void showStuff(int stuff1, T stuff2, T stuff3);
```

- **Definition:**

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
	cout << stuff1 << endl
	          << stuff2 << endl
	          << stuff3 << endl;
}
```

showStuff Call

- Consider function call:
  showStuff(2, 3.3, 4.4);
- Compiler generates function definition
  - Replaces T with double
    - Since second parameter is type double
- Displays:
  2
  3.3
  4.4

Compiler Complications

- Function declarations and definitions
  - Typically we have them separate
  - For templates → not supported on most compilers!
- Safest to place template function definition in file where invoked
  - Many compilers require it appear 1st
  - Often we #include all template definitions

Multiple Type Parameters

- Can have:
  template<class T1, class T2>
- Not typical
  - Usually only need one "replaceable" type
  - Cannot have "unused" template parameters
    - Each must be "used" in definition
    - Error otherwise!

Algorithm Abstraction

- Refers to implementing templates
- Express algorithms in "general" way:
  - Algorithm applies to variables of any type
  - Ignore incidental detail
  - Concentrate on substantive parts of algorithm
- Function templates are one way C++ supports algorithm abstraction

Defining Templates Strategies

- Develop function normally
  - Using actual data types
- Completely debug "ordinary" function
- Then convert to template
  - Replace type names with type parameter as needed
- Advantages:
  - Easier to solve "concrete" case
  - Deal with algorithm, not template syntax

Inappropriate Types in Templates

- Can use any type in template for which code makes "sense"
  - Code must behave in appropriate way
- e.g., swapValues() template function
  - Cannot use type for which assignment operator isn't defined
  - Example: an array:
    ```
    int a[10], b[10];
    swapValues(a, b);
    ```
    - Arrays cannot be "assigned"!

C++ Template Classes

```
template <typename T, int d>
class Vector {
 public:
  typedef T DataType;    // Access as Vector<T, d>::DataType.

  Vector() { for (int i = 0; i < d; ++i) vec_[i] = T(); }
  Vector(const Vector& v) { for (int i = 0; i < d; ++i) vec_[i] = v.vec_[i]; }
  const int size() const { return d; }

  const T& operator[](int i) const { return vec_[i]; }
  T& operator[](int i) { return vec_[i]; }

  Vector operator+() const { return *this; }
  Vector operator-() const;

  T Sum() const;
  T Dot(const Vector& v) const;

 private:
  T vec_[d];
};
```

C++ Template Classes

```
template <typename T, int d>
Vector<T, d> Vector<T, d>::operator-() const {
  Vector<T, d> ret;
  for (int i = 0; i < d; ++i) ret.vec_[i] = -vec_[i];
  return ret;
}

template <typename T, int d>
T Vector<T, d>::Sum() const {
  T ret = T();
  for (int i = 0; i < d; ++i) ret += vec_[i];
  return ret;
}

template <typename T, int d>
T Vector<T, d>::Dot(const Vector& v) const {
  T ret = T();
  for (int i = 0; i < d; ++i) ret += vec_[i] * v.vec_[i];
  return ret;
}
```

C++ Template Classes

```
template <typename T, int d>
class Vector {
 public:
  // ....

  template <typename S>
  Vector<S, d> cast() const {
    Vector<S, d> ret;
    for (int i = 0; i < d; ++i) ret[i] = static_cast<S>(vec_[i]);
    return ret;
  }

 private:
  T vec_[d];
};
```

```
int main() {
  Vector<int, 3> v, w;
  Vector<int, 3>::DataType dot = v.Dot(-w);
  Vector<double, 3> x = v.cast<double>();
  cout << x.Sum();
  return 0;
}
```

C++ Template Classes

```
int main() {
  Vector<int, 3> v;
  return 0;
}
```

```
template <>
class Vector<int, 3> {
 public:
  typedef int DataType;    // Access as Vector<T, d>::DataType.

  Vector() { for (int i = 0; i < 3; ++i) vec_[i] = int(); }
  Vector(const Vector& v) { for (int i = 0; i < 3; ++i) vec_[i] = v.vec_[i]; }
  const int size() const { return d; }

  const int& operator[](int i) const { return vec_[i]; }
  int& operator[](int i) { return vec_[i]; }

  Vector operator+() const { return *this; }
  Vector operator-() const;

  int Sum() const;
  int Dot(const Vector& v) const;

 private:
  int vec_[3];
};
```

Class Templates

- Can also "generalize" classes
  template<class T>
  - Can also apply to class definition
  - All instances of "T" in class definition replaced by type parameter
  - Just like for function templates!
- Once template defined, can declare objects of the class

Class Template Definition

- ```
  template<class T>
  class Pair
  {
  public:
      Pair();
      Pair(T firstVal, T secondVal);
      void setFirst(T newVal);
      void setSecond(T newVal);
      T getFirst() const;
      T getSecond() const;
  private:
      T first; T second;
  };
  ```
  ```

Template Class Pair Members

```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal)
{
    first = firstVal;
    second = secondVal;
}
template<class T>
void Pair<T>::setFirst(T newVal)
{
    first = newVal;
}
```

Template Class Pair

- Objects of class have "pair" of values of type T
- Can then declare objects:
  Pair<int> score;
  Pair<char> seats;
  - Objects then used like any other objects
- Example uses:
  score.setFirst(3);
  score.setSecond(0);

Pair Member Function Definitions

- Notice in member function definitions:
  - Each definition is itself a "template"
  - Requires template prefix before each definition
  - Class name before :: is "Pair<T>"
    - Not just "Pair"
  - But constructor name is just "Pair"
  - Destructor name is also just "~Pair"

Class Templates as Parameters

- Consider:
  int addUP(const Pair<int>& the Pair);
  - The type (int) is supplied to be used for T in defining this class type parameter
  - It "happens" to be call-by-reference here
- Again: template types can be used anywhere standard types can

Class Templates Within Function Templates

- Rather than defining new overload:

```
template<class T>
T addUp(const Pair<T>& the Pair);
//Precondition: Operator + is defined for values
                of type T
//Returns sum of two values in thePair
```

- Function now applies to all kinds of numbers

Predefined Template Classes

- Recall vector class
  - It's a template class!
- Another: basic_string template class
  - Deals with strings of "any-type" elements
  - e.g.,

basic_string<char> works for char's
basic_string<double> works for doubles
basic_string<YourClass> works for YourClass objects

`vector` - a resizable array

- Iterator : access the elements in the container iteratively in order.
  - Const and non-const types : `const_iterator` and `iterator`.
  - In many cases, it can considered as a pointer to an element.

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

int main(void) {
// vector(sz)
vector<int> v(10);
for (int i = 0; i < v.size(); ++i) v[i] = i;

// begin(), end()
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
  cout << " " << *it;
}
// Output:  0 1 2 3 4 5 6 7 8 9

// rbegin(), rend()
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
  cout << " " << *it;
}
// Output:  9 8 7 6 5 4 3 2 1 0
}
```

basic_string Template Class

- Already used it!
- Recall "string"
  - It’s an alternate name for basic_string<char>
  - All member functions behave similarly for basic_string<T>
- basic_string defined in library <string>
  - Definition is in std namespace

Inline Function

- Request the compiler to insert the function body in the place that the function is called.
  - The function body will not be included in the object file.
- Compilers are not obligated to respect this request.
- Member functions defined in the class definition are inlined.
- Inline function definitions are placed in header files.
- Pros/cons: eliminate function call overhead / code bloat.

```
inline void Swap(int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```


Q&A