C++ Templates

Computer Vision 2018

P. Zanuttigh
Some slides Wei Du,S. Ghidoni, M. Carraro
Some material cplusplus.com

C++ Templates

- Method for generalizing
  - Classes
  - Functions & methods
- Parameters for type names

C++ Function Templates

- Function templates are special functions that can operate with *generic types*
- Their functionalities can be adapted to more than one type or class without repeating the entire code for each type
- This can be achieved using *template parameters*
  - A template parameter is a special kind of parameter that can be used to *pass a type as argument*
  - Just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function

Same Task for Different Data Types

- Approaches for functions that implement identical tasks for different data types
  - Naïve Approach
  - Function Overloading
  - Function Template

Approach 1: Naïve Approach

Create unique functions with unique names for each combination of data types

- difficult to keeping track of multiple function names
- lead to programming errors

Example

```
void PrintInt( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void PrintChar( char ch )
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void PrintFloat( float x )
{
  …
}
void PrintDouble( double d )
{
  …
}
```

To output the traced values, we insert:

```
PrintInt(sum);

PrintChar(initial);

PrintFloat(angle);
```

Approach 2: Function Overloading

The use of the same name for different C++ functions, distinguished from each other by their parameter lists

- Eliminates need to come up with many different names for identical tasks
- Reduces the chance of unexpected results caused by using the wrong function name
- But we need to write multiple copies of the function

Example of Function Overloading

```
void Print( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void Print( char ch )
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void Print( float x )
{

}
```

To output the traced values, we insert:

```
Print(someInt);
Print(someChar);
Print(someFloat);
```

Approach 3: Function Template

- *A C++ language construct that allows the compiler to generate multiple versions of a function by allowing parameterized data types.*

FunctionTemplate

template < TemplateParamList >
FunctionDefinition

TemplateParamDeclaration: placeholder

class typeIdentifier
typename variableIdentifier

```
template<typename SomeType>
```

Example of a Function Template

```
template<typename SomeType>

void Print( SomeType val )
{
     cout << "***Debug" << endl;
     cout << "Value is " << val << endl;
}
```

Template parameter

(class, user defined type, built-in types)

Template argument

To output the traced values, we insert:

```
Print<int>(sum);
Print<char>(initial);
Print<float>(angle);
```

Summary of The Three Approaches

Naïve Approach
Different Function Definitions
Different Function Names

Function Overloading
Different Function Definitions
Same Function Name

Template Functions
One Function Definition (a function template)
Compiler Generates Individual Functions

Example: The SwapValues Function

- An example: function for swapping values

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

- The algorithm is general – specification of the argument type is not needed to implement the algorithm
  - Other versions for char, float, double, … would differ *only in the argument type*

SwapValues: Use a Function Template

```
// Defining a function template
template <typename T>
void SwapValues(T& var1, T& var2)
{
    T temp;

    temp = var1;
    var1 = var2;
    var2 = temp;
}
// Calling a function template
int i = 2, j = 9;
SwapValues(i, j); // or SwapValues<int>(i, j);

double a = 6.4, b = 4.9;
SwapValues(a, b); // or SwapValues<double>(i, j);
```

Template prefix Introduces a type parameter (T)

The Mat::at function needs the template specification

Example of a Function Template

```
// function template 1
#include <iostream>
using namespace std;

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

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
// function template 2
#include <iostream>
using namespace std;

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


int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Function Templates

- A template function is a large collection of function definitions
- Cause the compiler to generate a version for each type used in the code
- Works for built-in types and for user-defined classes
- Declarations and definitions work as for normal functions

Example: Sorting a Vector

```
template<typename T> void sort(vector<T>& v)
{
   const size_t n = v.size();

   for (int gap=n/2; 0<gap; gap/=2)
       for (int i=gap; i<n; i++)
             for (int j=i-gap; 0<j; j-=gap)
                   if (v[j+gap]<v[j]) {
                         T temp = v[j];
                         v[j] = v[j+gap];
                         v[j+gap] = temp;
                   }
}
```

Class Template

A C++ language construct that allows the compiler to generate multiple versions of a class by allowing parameterized data types.

Class Template

template < TemplateParamList >
ClassDefinition

TemplateParamDeclaration: placeholder

class typeIdentifier
typename variableIdentifier

Class Templates

- The same concepts applies to classes
- E.g., a class with members of generic type

```
template <typename T>
class Pair
{
public:
    Pair();
    Pair(T first_value, T second_value);
    void SetFirst(T value);
    T GetFirst(void) const;
    //…

private:
    T first, second;
}
```

Defined in your .h header

Class Templates: Implementation

```
// Defining a function template
template <typename T>
Pair<T>::Pair(T first_value, T second_value)
{
    first = first_value;
    second = second_value;
}

Template <typename T>
Void Pair<T>::SetFirst(T new_value)
{
    first = new_value;
}

Template <typename T>
T Pair<T>::Getfirst(void)
{
    return first;
}
```

Use a Class Template

```
// .cpp main file

#include “pair.hpp”

int main() {
  Pair<int> p;
}
```

Another Example…

```
template<typename ItemType>
class GList
{
 public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
    void SelSort();
    void Print() const;
    GList();                          // Constructor
 private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```

Template parameter

Instantiating a Class Template

To create lists of different data types

```
// Client code

GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

template argument

Compiler generates 3 distinct class types

```
GList_int list1;
GList_float list2;
GList_string list3;
```

Substitution Example

```
class GList_int
{
public:

void Insert( /* in */ ItemType item );

    void Delete( /* in */ ItemType item );

    bool IsPresent( /* in */ ItemType item ) const;

private:
    int       length;
    ItemType  data[MAX_LENGTH];
};
```

int

int

int

int

Substitution performed by the compiler !

Function Definitions for Members of a Template Class

```
template<typename ItemType>
void GList<ItemType>::Insert( /* in */ ItemType item )
{
    data[length] = item;
    length++;
}
```

```
//after substitution of float
void GList<float>::Insert( /* in */ float item )
{
    data[length] = item;
    length++;
}
```

Mix of Template and Standard Params

```
template <class T, int size>
  class Stack {...
        T buf[size];
  };
Stack<int,128> mystack;
```

non-type parameter

Recap

- Templates are mechanisms for generating functions and classes on type parameters
- We can design a single class or function that operates on data of many types
  - function templates
  - class templates