C++ Templates

IT Academy

Účel šablón

Generické programovanie

Spôsob programovania, pri ktorom píšeme algoritmy, ktoré nechávajú špecifikáciu typov parametrov až na moment, keď algoritmus použijeme - napr vytvorenie inštancie triedy.

Vznik tohoto programovacieho postupu - zhruba rok 1970

Prvky generického programovania sa nachádzajú vo viacerých jazykoch:

- C++ Templates
- C# Generics - http://msdn.microsoft.com/en-us/library/ms379564(v=vs.80).aspx
- Java - 2004 - http://en.wikipedia.org/wiki/Generics_in_Java

Účel šablón

- umožniť programátorom písať obecné riešenia pre problémy, ktorých vstupné parametre sú rôzne
- umožniť implementáciu všeobecných štruktúr, ktoré sa dajú použiť pre viaceré typy (int stack, double stack)

Fakty

- šablóna funguje pre tie typy, pre ktoré je možné vykonať operácie, ktoré sa v šablóne vykonávajú => ak predáme šablóne argument, s ktorým nevie pracovať, program sa neskompiluje
- ak chceme overovať typ vo vnútri šablóny, musíme overenie implementovať vnútri šablóny
- šablóna je priamy nástroj zabraňujúci duplicite kódu
- šablóna má význam iba v typových jazykoch
- šablóny sa pri použití s určitým typom preložia a začlenia sa do programu tak, ako keby boli algoritmy v šablóne implementované viackrát pre rôzne typy
- šablóny sa zvyčajne kombinujú s preťažením operátorov
- šablónami je možné implementovať alternatívu ku dedičnosti - http://www.hackcraft.net/cpp/templateInheritance/

Šablóny funkcií

Deklarujeme takto:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

v tomto prípade class == typename

Takto by sa dala napríklad implementovať šablóna pre funkciu max() z STL:

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

Šablóny funkcií

- typ predaného parametra určuje, pre aký typ sa šablóna preloží
- ak šablóne predáme typ manuálne, musíme tento typ dodržať!

```
#include <iostream>

int main()
{
  // This will call max <int> (by argument deduction)
  std::cout << max(3, 7) << std::endl;
  // This will call max<double> (by argument deduction)
  std::cout << max(3.0, 7.0) << std::endl;
  // This type is ambiguous, so explicitly instantiate max<double>
  std::cout << max<double>(3, 7.0) << std::endl;
  return 0;
}
```

Šablóny funkcií a referencie

Modifikujeme predanú premennú:

```
template<class T>
void TwiceIt(T& tData)
{
    tData *= 2;
    // tData = tData + tData;
}
```

Vraciame predanú premennú

```
template<class T>
T& GetMax(T& t1, T& t2)
{
    if (t1 > t2)
    {
        return t1;
    }
    // else
    return t2;
}
```

```
template<class TYPE>
void PrintTwice(const TYPE& data)
{
    cout<<"Twice: " << data * 2 << endl;
}
```

Šablóny funkcií

Vytvorenie šablóny s kompatibilným typom:

```
template<class Type>
void Show(Type tData) {}
```

```
// This will produce (instantiate) 'Show(int)'
Show ( 1234 );

// But you want it to produce 'Show(double)'
```

```
Show<double> ( 1234 );
```

vytvorí =>

```
void Show(double);
```

Šablóny funkcií

Syntax:

Show<>()

hovorí, že kompilátor si má domýšľať typ explicitne podľa predanej premennej, nie podľa typu argumentu funkcie!

Šablóny funkcií

Pre kompletnosť ujasnenie pojmov:

Šablóna funkcie (function template):

```
template<class T>
void Show(T data)
{ }
```

=> Funkcia (zo) šablóny (template function)

```
void Show(double data){}
```

```
void Show<double>(double x){}
```

Šablóny funkcií

Explicitné definovanie argumentov šablóny

- nám pomôže nedovolit' kompilátoru inteligentne určit' typ vstupného parametra
- môže ušetrit' počet funkcií zo šablóny:

```
PrintNumbers<double, double>(10, 100);    // int, int
PrintNumbers<double, double>(14, 14.5);   // int, double
PrintNumbers<double, double>(59.66, 150); // double, int
```

=>

```
void PrintNumbers<double, double>(const double& t1Data, const T2& t2Data)
{}
```

Šablóny funkcií

2 vstupné argumenty a iba jeden typ:

```
template<class T>
T max(T t1, T t2)
{
    if (t1 > t2)
        return t1;
    return t2;
}
```

```
max<double>(120, 14.55); // Instantiates max<double>(double,double);
```

Typ definujem explicitne!

Šablóny funkcií

Čo ak sa typ nedá zistiť zo vstupných argumentov???

```
template<class T>
void PrintSize()
{
   cout << "Size of this type:" << sizeof(T);
}
```

Nie =>

```
PrintSize();
```

Ano =>

```
PrintSize<float>();
```

Šablóny funkcií

Preddefinované argumenty

```
template<class T>
void PrintNumbers(T array[], int array_size, T filter = T())
{
    for(int nIndex = 0; nIndex < array_size; ++nIndex)
    {
        if ( array[nIndex] != filter) // Print if not filtered
            cout << array[nIndex];
    }
}
```

```
template<class T>
void PrintNumbers(T array[], int array_size, T filter = T(60))
```

Šablóny funkcií

Pre úplnosť - predefinovanie výstupného typu:

```
template<class T>
T SumOfNumbers(int a, int b)
{
   T t = T(); // Call default CTOR for T

   t = T(a)+b;


   return t;
}
```

```
double nSum;
nSum = SumOfNumbers<double>(120,200);
```

Šablóny tried

- generujú triedy podľa typov vstupných parametrov
- riešia situáciu, kde potrebujeme deklarovať veľké množstvo podobných tried pre rôzne typy
- často sa využívajú pre implementáciu kontajnerov (tých istých pre rôzne typy)
- štandardné typy kontajnerov C++ využívajú šablóny (STL, napr vector - http://www.cplusplus.com/reference/vector/vector/)

Šablóny tried

```
template<class T>
class Item
{
    T Data;
public:
    Item() : Data( T() )
    {}

    void SetData(T nValue)
    {
        Data = nValue;
    }

    T GetData() const
    {
        return Data;
    }

    void PrintData()
    {
        cout << Data;
    }
};
```

- typ sa musí predávať explicitne!

```
Item<float> item2;
float n = item2.GetData();
```

Šablóny tried

Akceptujú aj **netypové** parametre:

```
template<class T, int SIZE>
class Array{};
```

```
Array<int, 10> my_array;
```

```
template<class T, int SIZE>
class Array
{
 static const int Elements_2x = SIZE * 2;
};
```

Šablóny tried

Dokonca je možné aj niečo takéto:

```
void DoSomething(int arg = SIZE);
// Non-const can also appear as default-argument...
```

Default method argument

```
private:
    T TheArray[SIZE];
```

Array Size

```
// Initialize with default (i.e. 0 for int)
void Initialize()
{
    for(int nIndex = 0; nIndex < SIZE; ++nIndex)
        TheArray[nIndex] = T();
}
```

vyskúšať!

Šablóny tried

Zvyšok kódu pre doplnenie implementácie triedy Array:

```
T operator[](int nIndex) const
{
  if (nIndex>0 && nIndex<SIZE)
  {
    return TheArray[nIndex];
  }
  return T();
}

T& operator[](int nIndex)
{
   return TheArray[nIndex];
}
```

```
T Accumulate() const
{
  T sum = T();
  for(int nIndex = 0; nIndex < SIZE; ++nIndex)
  {
     sum += TheArray[nIndex];
  }
  return sum;
}
```

Šablóny tried

Šablónová trieda ako argument pre triedu zo šablóny

```
Array< Pair<int, double>, 40> ArrayOfPair;
```

???

```
Pair<int, Array<double, 50>> PairOfArray;
```

Šablóny tried

Defaultné argumenty pre šablónu triedy

Ak

```
template<class T, int SIZE=100>
class Array
{
private:
    T TheArray[SIZE];
    ...
};
```

potom `Array<int> IntArray;` == `Array<int, 100> IntArray;`

(kompilátor)

Šablóny tried

Aj toto sa dá:

```
const int _size = 120;
// #define _size 150
template<class T, int SIZE=_size>
class Array
```

Šablóny tried

Aj toto sa dá - nastavenie defaultnej hodnoty parametra šablóny konštantou:

```
const int _size = 120;
// #define _size 150
template<class T, int SIZE=_size>
class Array
```

Šablóny tried

Aj toto sa dá - preddefinovaný typ argumentu v šablóne:

```
template<class T = int>
class Array100
{
    T TheArray[100];
};
```

Potom:

```
Array100<float> FloatArray;
Array100<> IntArray;
```

A nie:

```
Array100 IntArray;
```

Šablóny tried

Čo znamená toto???

```
Array<>             IntArray1;
Array<int>          IntArray2;
Array<float, 40>    FlaotArray3;
```

Šablóny tried

Navyše je možné:

```
template<class Type1, class Type2 = Type1>
class Pair
{
    Type1 first;
    Type2 second;
};
```

Lebo potom:

```
Pair<int,int> IntPair;
```

==

```
Pair<int> IntPair;
```

A:

```
Pair<> IntPair;
```

=>

```
class Pair<int,int>{};
```

Šablóny tried

Navyše je možné:

```
template<class Type1, class Type2 = Type1>
class Pair
{
    Type1 first;
    Type2 second;
};
```

Lebo potom:

```
Pair<int,int> IntPair;
```

==

```
Pair<int> IntPair;
```

A:

```
Pair<> IntPair;
```

=>

```
class Pair<int,int>{};
```

Šablóny tried

Možné využitie:

```
template<class T, int ROWS = 8, int COLUMNS = ROWS>
class Matrix
{
    T TheMatrix[ROWS][COLUMNS];
};
```

Šablóny tried

Toto však nebude fungovať:

```
template<class Type1=Type2, class Type2 = int>
class Pair{};

template<class T, int ROWS = COLUMNS, int COLUMNS = 8>
class Matrix
```

Prečo?

Šablóny tried

Špecializácia šablón:

- **šablóny funkcií môžu byť špecializované iba plne**

http://stackoverflow.com/a/1416382

http://www.gotw.ca/publications/mill17.htm

- **template forward declaration:**

http://stackoverflow.com/a/13848492

Šablóny tried

Zdroj:

http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part-1#PtrWithTempl