C++ Program Design -- Template

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

Function templates

The need for function templates

```
int max(int nX, int nY)
{
    return (nX > nY) ? nX : nY;
}

double max(double dX, double dY)
{
    return (dX > dY) ? dX : dY;
}
```

Creating function templates in C++

```
int max(int nX, int nY){
    return (nX > nY) ? nX : nY;
}

Type max(Type tX, Type tY){
    return (tX > tY) ? tX : tY;
}

template <typename Type> // this is the template parameter declaration
Type max(Type tX, Type tY)
{
    return (tX > tY) ? tX : tY;
}
```

however, it won't compile

Using function templates

- `int nValue = max(3, 7); // returns 7`
- `double dValue = max(6.34, 18.523); // returns 18.523`
- `char chValue = max('a', '6'); // returns 'a '`

- Template functions reduce code maintenance, because duplicate code i s reduced significantly.
- template functions can be safer, because there is no need to copy funct ions and change types by hand whenever you need the function to wor k with a new type!

- template functions produce crazy-looking error messages that are muc h harder to decipher than those of regular functions

- If the template function uses multiple template type parameter, they can be separated by commas:
- `template <typename T1, typename T2>`
- `// template function here`

Function template instances

- The function with actual types is called a **function template instance**.
  - Compiler will create the template instance automatically
  - We also can write them ourselves

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

```
int max(int nX, int nY){
    return (nX > nY) ? nX : nY;
}
```

Operators, function calls, and function templates

```
class Cents{
private:
    int m_nCents;
public:
    Cents(int nCents) : m_nCents(nCents) { }
};

Cents cNickle(5);
Cents cDime(10);

Cents cBigger = max(cNickle, cDime);
```

- C++ will create a template instance for max() that looks like this:

```
Cents max(Cents tX, Cents tY)
{
    return (tX > tY) ? tX : tY;
}
```

- C++ has no idea how to evaluate tX > tY! Consequently, this will produce a compile error.

Solution

```
class Cents{
private:
    int m_nCents;
public:
    Cents(int nCents) : m_nCents(nCents){}

    friend bool operator>(Cents &c1, Cents&c2){
        return (c1.m_nCents > c2.m_nCents);
    }
};
```

Function classes

Templates and container classes

```
template <typename T> class Array{
private:
    int m_nLength;    T *m_ptData;
public:
    Array(int nLength){
          m_ptData= new T[nLength]; m_nLength = nLength;
}

    T& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_ptData[nIndex];
    }
```

IntArray, DoubleArray, … => Array<T>

Splitting up template classes

```cpp
#include "Array.h"//Array.cpp

template <typename T>
int Array<T>::GetLength() { return m_nLength; }
```

```cpp
#include "Array.h""//main.cpp
int main(){
Array<int> anArray(12);
Array<double> adArray(12);
anArray.GetLength();
…
return 0;
}
```

unresolved external symbol "public: int __thiscall Array::GetLength(void)" (?GetLength@?$Array@H@@QAEHXZ)

because main.cpp can't see the template definition of Array::GetLength().

Fail to create template instances

Solution 1

- put all of your template class code in the header file
- The upside of this solution is that it is simple.
- The downside here is that if the template class is used in many places, you will end up with many local copies of the template class, which can bloat your code.
- This is our preferred solution unless code bloat becomes a problem.

Solution 2: a three-file approach

- templates.cpp:

```
// Ensure the full Array template definition can be seen
#include "Array.h"
#include "Array.cpp"
template class Array<int>; // Explicitly instantiate template Array<int>
template class Array<double>;
```

The “template class” command causes the compiler to explicitly instantiat e the templated class.

In the above case, the compiler will create both Array inside of templates. cpp.

Because templates.cpp is inside our project, this will then be compiled. T hese functions can then be linked to from elsewhere.

Expression parameters and template specialization

Expression parameters

```
template <typename T, int nSize> // nSize is the expression parameter
class Buffer{
private:
    T m_atBuffer[nSize];
public:
    T& operator[](int nIndex){
        return m_atBuffer[nIndex];
    }
};

// declare an integer buffer with room for 12 chars
Buffer<int, 12> cIntBuffer;
```

A value that has an integral type or enumeration
A pointer or reference to an object
A pointer or reference to a function
A pointer or reference to a class member function

Why do not put nSize as a member data?

Template specialization

```
template <typename T>
class Storage{
private:
    T m_tValue;
public:
    void Print(){
        std::cout << m_tValue << std::endl;
    }
};
Storage<int> nValue(5);    Storage<double> dValue(6.7);
nValue.Print();dValue.Print();
```

This prints:
5
6.7

Template specialization

```
template <>
void Storage<double>::Print()
{
    std::cout << std::scientific << m_tValue << std::endl;
}
```

What's going on here?

```
int main(){
    // Dynamically allocate a temporary string
    char *strString = new char[40];

    …
    Storage<char*> strValue(strString);

    // Delete the temporary string
    delete[] strString;

    // Print out our value
    strValue.Print(); // This will print garbage
```

```
Storage(T tValue){
    m_tValue = tValue;
}
```

Solution: Template specialization

```
template <>
Storage<char*>::Storage(char* tValue){
    int length = strlen(tValue)+1; // +1 to account for null ter
minator

    // Allocate memory to hold the tValue string
    m_tValue = new char[length];

    // Copy the actual tValue string into the m_tValue memory we
just allocated
    strcpy_s(m_tValue, length, tValue);
}
```

```
template <>
Storage<char*>::~Storage(){
    delete[] m_tValue;
}
```

Class template specialization

- specialize member functions of a template class in order to provide different functionality for specific data types.

- it is also possible to specialize an entire class!

```
template <> // the following is a template class with no templated parameters
class Storage<bool>{
private:
     unsigned char m_tType;
public:
    void Set(int nIndex, bool tType) {
        ...
}
    bool Get(int nIndex) {
        ...
    }
};
```

Partial template specialization

Partial template specialization allows us to write functions where some of the t emplate parameters have been fully or partially resolved.

```
template <typename T, int nSize>
void PrintBufferString(Buffer<T, nSize> &rcBuf)
{
    std::cout << rcBuf.GetBuffer() << std::endl;
}

template<int nSize>
void PrintBufferString(Buffer<char, nSize> &rcBuf)
{
std::cout << rcBuf.GetBuffer() << std::endl;
}
```

Partial template specialization for pointers

```
template <typename T>
class Storage{
private:
    T m_tValue;
public:
    Storage(T tValue){
        m_tValue = tValue;
    }
    void Print(){
        std::cout << m_tValue << std::endl;;
    }
};
```

```
Storage<char*>::Storage(char* tValue){
    m_tValue = new char[strlen(tValue)+1];
    strcpy(m_tValue, tValue);
}
```

Partial template specialization for pointers

```
template <typename T> class Storage<T*>{// this is specialization of
Storage that works with pointer types
private:
      T* m_tValue;
public:
    Storage(T* tValue) {
        m_tValue = new T(*tValue);
    }

    void Print() {
        std::cout << *m_tValue << std::endl;
    }
};
```

```
template <typename T>
class Storage{
private:
    T m_tValue;
public:
Storage(T tValue){
    m_tValue = tValue;
}
```