System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

4. Templates

Eugene Zouev

C++ Templates:
Template instantiation
Template specialization
Explicit & partial specializations

Explicit Inst-n of Func.Template

Example: the function calculating the number of 32-bit words for an arbitrary type.

```
template < typename T >
int spaceOf ( void )
{
    int bytes = sizeof(T);
    return bytes/4 + bytes%4>0;
}
```

How to call this template?

```
int w = spaceOf();
```

Is it correct?

The problem: how the compiler can **determine the actual type** while instantiating the template?

The solution is to use ***explicit instantiation*** (exactly as for class templates) i.e. explicitly specify the function template arguments.

```
class C { . . . };        // class declaration; C - class type
typedef void (*pf)(int); // pf is pointer-to-function type

int wint = spaceOf<int>();
int wC   = spaceOf<C>();
int wpf  = spaceOf<pf>();
int warr = spaceOf<int[10]>();
```

Explicit Inst-n of Func.Template

Let's consider how explicit instantiation works

```
template < typename T >
int spaceOf ( void )
{
    int bytes = sizeof(T);
    return bytes/4 + bytes%4>0;
}
```

Original call

```
int wint = spaceOf<int>();
```

```
int spaceOf_int ( void )
{
    int bytes = sizeof(int);
    return bytes/4 + bytes%4>0;
}
```

Instantiation

```
int wint = spaceOf_int();
```

Instantiating Function Templates

The function calculating the number of 32-bit words for an **arbitrary type**:

```
template < typename T >
int spaceOf ( void )
{
    int bytes = sizeof(T);
    return bytes/4 + (bytes%4>0);
}
```

The similar function for **objects** of an arbitrary type (**not** for pure types):

The same **name** but different **signature**

```
template < typename T >
int spaceOf ( T x )
{
    int bytes = sizeof(x);
    return bytes/4 + (bytes%4>0);
}
```

The different version of **sizeof** operator

The same algorithm

Instantiating Function Templates

```
int  x1;
bool x2;
int  x3[10];

int s1 = spaceOf(x1); // 1
int s2 = spaceOf(x2); // 1
int s3 = spaceOf(x3); // 1 ????
```

array size is 4x10 = 40

```
template < typename T >
int spaceOf ( T x )
{
   int bytes = sizeof(x);
   return bytes/4 + (bytes%4>0);
}
```

1. Type of x3 is `int[10]`
2. ***Array-to-pointer conversion***: `int[10]->int*`

Argument deducing

3. Instantiation: making **specialization** of `spaceOf` template by substituting the deduced type `int*` for `T`

```
(Compiler generates)
int spaceOfint* ( int* x )
{
   int bytes = sizeof(x);
   return bytes/4 + (bytes%4>0);
}
```

4. Generating the **call** to specialization generated on the previous step

```
(Compiler generates)
int s3 = spaceOfint*(x3);
```

Instantiating Function Templates

So, how **to prevent any standard conversion** while instantiating the template?

Solution: pass *the reference* to the value instead of passing the *value* itself.

```
template < typename T >
int spaceOf ( T& x )
{
    int bytes = sizeof(x);
    return bytes/4 + (bytes%4>0);
}
```

If pass arrays "by value", they **always get converted to pointers**; if pass them "by reference" they **don't**.

Func.Templates: Incomplete Inst.

Example:
Template representing **the raising a value to an integer power**:

$$V^N$$

where

V is of an arbitrary type T, and
N is an integer constant.

```
template < unsigned N, typename T >
T Power ( T v )
{
    T res = v;
    for ( int i=1; i<N; i++ )
        res *= v;
    return res;
}
```

Func.Templates: Incomplete Inst.

```
template < unsigned N, typename T >
T Power ( T v )
{
    T res = v;
    for ( int i=1; i<N; i++ )
        res *= v;
    return res;
}
```

```
int d1 = Power<5,int>(1.2);
```

Complete instantiation; both template actuals are taken from the instantiation.

```
int Power<int> ( int v )
{
    int res = v;
    for ( int i=1; i<5; i++ )
        res *= v;
    return res;
}
```

```
double d2 = Power<5>(1.2);
```

__Incomplete instantiation__; the 1st actual is taken from the instantiation, the 2nd one is deduced from the call's actual.

```
double Power<double>(double v)
{
    double res = v;
    for ( int i=1; i<5; i++ )
        res *= v;
    return res;
}
```

Func.Templates: Incomplete Inst.

Use Power template as follows:

```
template < unsigned N, typename T >
T Power ( T v )
{
    T res = v;
    for ( int i=1; i<N; i++ )
        res *= v;
    return res;
}

void main()
{
    double d1 = Power<5>(1.2);
    double d2 = Power<5,int>(1.2);
    std::cout << d1 << " " << d2;
}
```

Why result values of d1 and d2 are *different*?

Hint: type conversions!

Fun.Template Instantiation Kinds

```
template < typename T1, typename T2 >
void F ( T1 v1, t2 v2 )
{
    . . .
}
```

```
F<int,float>(v1,v2);
```

Complete explicit instantiation; all template actuals ***are taken directly*** from the instantiation.

```
F<int>(v1,v2);
```

Incomplete explicit instantiation; some actuals ***are taken*** from the instantiation, other actuals are deduced from the call's actuals.

```
F(v1,v2);
```

Implicit instantiation; all template actuals ***are deduced*** from the call's actuals.

Explicit Specializations (1)

A simple example: a class template with `less` member

```
template<typename T>
class C {
    public: bool less ( T& v1, T& v2 )
    {
      return v1 < v2;
    }
}
```

How to use the template:

```
C<int> c1;
bool l1 = c1.less(1,2);

C<double> c2;
bool l2 = c2.less(1.2,3.4);

C<const char*> c3;
bool l4 = c3.less("abcd","abcx"); // ???
```

Conclusion: we need

(a) a ***generic*** form of `less` template, and

(b) at least one ***special form*** of this template for the special type:
for comparing *character strings*.

Generic form: Class template

Special form: **Explicit specialization** of the class template

Explicit Specializations (3)

Generic form:

Common type

```
template<typename T>
class C {
    public: bool less ( T& v1, T& v2 )
    {
      return v1 < v2;
    }
}
```

Common algorithm

Special form: **explicit specialization**

Concrete type

```
template<>
class C<const char*> {
   public: bool less ( const char* v1, const char* v2 )
   {
     return strcmp(v1,v2)<0;
   }
}
```

Specific algorithm for the concrete type

*Notice **empty angle brackets**!*

Explicit Specializations :Summary

How to use the template:

Generic form is used for instantiation

Explicit instantiation is used

```
C<int> c1;
bool l1 = c1.less(1,2);

C<double> c2;
bool l2 = c2.less(1.2,3.4);

C<const char*> c3;
bool l4 = c3.less("abcd","abcx");
```

1. It is possible to specify ***explicit specialization(s)*** for a template for special cases of template argument(s).
2. The implementation of explicit specialization ***may differ*** from the implementation of the "primary" template.
3. All explicit specializations of a template together with the "primary" template itself ***form the single family*** of classes.
4. All cases of use of either "primary" template or its explicit specializations ***are processed during compile time***.

Instantiation vs Specialization

Class template

```
template < typename T >
class C
{
    // "Primary"
    // implementation
}
```

The act of instantiating the class template: __template instantiation__

`C<int>`

(Explicit) declaration of __template specialization__

```
template<>
class C<int>
{
    // Instantiated
    // "primary"
    // implementation
}
```

```
template<>
class C<char*>
{
    // An alternative
    // implementation
    // for the concrete type
}
```

Explicitly __instantiated__ template specialization

- Classes-by-template (*non-standard*)
- Template specializations

Explicitly __specialized__ template specialization

Explicit Specializations: Example (1)

One more example: **Factorial**

N! = 1	**for *N=0***
N! = 1	**for *N=1***
N! = N * (N-1) * … 2 * 1	**for *N>=2***

Obvious implementation: recursive function:

```
unsigned long Fact ( unsigned N )
{
   if ( N<2 ) return 1;
   return N*Fact(N-1);
}
```

Let's try to make a **"template" version** of `Fact`.
The first (straightforward) attempt:

```
template < unsigned N >
unsigned long Fact ( void )
{
   if ( N<2 ) return 1;
   return N*Fact<N-1>();
}
```

Explicit Specializations: Example (3)

```
unsigned long f5 = Fact<3>();
```

```
template<>
unsigned long Fact<3> ( void )
{
    if ( 3<2 ) return 1;
    return 3*Fact<3-1>();
}
```

```
template<>
unsigned long Fact<2> ( void )
{
    if ( 2<2 ) return 1;
    return 2*Fact<2-1>();
}
```

```
template<>
unsigned long Fact<1> ( void )
{
    if ( 1<2 ) return 1;
    return 1*Fact<1-1>();
}
```

```
template<>
unsigned long Fact<0> ( void )
{
    if ( 0<2 ) return 1;
    return 0*Fact<0-1>();
}
```

How it works

...An so on!

Explicit Specializations: Example (4)

The second attempt: explicit specializations:

```
template < unsigned N >
unsigned long Fact ( void )
{
    return N*Fact<N-1>();
}
```

Primary template

N! = N*(N-1)!

```
template<>
unsigned long Fact<0> ( void )
{
    return 1;
}
```

Explicit specialization for N=0

0! = 1

```
template<>
unsigned long Fact<1> ( void )
{
    return 1;
}
```

Explicit specialization for N=1

1! = 1

Explicit Specializations: Example (5)

```
template<>
unsigned long Fact<3> ( void )
{
    if ( 3<2 ) return 1;
    return 3*Fact<3-1>();
}
```

```
unsigned long f5 = Fact<3>();
```

Automatically instantiated template specialization

```
template<>
unsigned long Fact<2> ( void )
{
    if ( 2<2 ) return 1;
    return 2*Fact<2-1>();
}
```

How it works

Automatically instantiated template specialization

See Task 1

```
template<>
unsigned long Fact<1> ( void )
{
    return 1;
}
```

Explicitly given template specialization

...Process terminated!

Partial Specializations (1)

Common form:

```
template<typename T>
class C {
    public: bool less ( T& v1, T& v2 ) { return v1 < v2; }
}
```

```
C<double> c2;
bool l2 = c2.less(1.2,3.4);
```

Explicit specialization

```
template<>
class C<const char*> {
   public: bool less ( const char* v1, const char* v2 )
   { return strcmp(v1,v2)<0; }
}
```

```
C<const char*> c3;
bool l4 = c3.less("abcd","abcx");
```

Comparing <u>two pointers</u>: which template to apply?

```
int* x = ...
int* y = ...
```

Partial Specializations (2)

Generic form: all types *(except those mentioned below)*

```
template<typename T>
class C {
   public: bool less ( const T& v1, const T& v2 )
              { return v1<v2; }
}
```

Explicit specialization: **const char*** type

```
template<>
class C<const char*> {
   public: bool less ( const char* v1, const char* v2 )
             { return strcmp(v1,v2)<0;  }
}
```

Partial specialization for pointer types (*except* **const char***)

See Task 2

```
template<typename T>
class C<T*> {
   public: bool less ( T* v1, T* v2 ) { return *v1<*v2; }
}
```

Partial Specializations (3)

Notation Rules:

```
template < typename T >
class C
{
   // common implementation
}
```

Primary template: for *all* types T

"Normal" template header

```
template < typename T >
class C<T*>
{
   // implementation for
   // the specified subset
}
```

Partial specialization:
for a *subset* of types

Different implementation

*Type **subset** specification: **any pointer***

Partial Specializations (4)

How to specify type subsets?
Several possible cases (most typical):

> `(T)` represents lists where at least one type contains `T`;
> `()` represents lists where no type contains `T`.

`const T`	constant types
`T*`	pointer types
`T&`	reference types
`T[`*`integer-const`*`]`	arrays
`type` `(*)(T)`	pointers to functions with parameter(s) of type `T`
`T(*)()`	pointers to functions returning type `T`
`T(*)(T)`	pointers to functions with parameter(s) of type `T` and returning type `T`

Template Concept: Summary

Template Concept

```
template < typename T >
class C {
    . . .
}
```

A template for a set of types and/or values (general case): “usual” template

```
template < >
class C<int> {
    . . .
}
```

A version of the original template for a particular type and/or value (special case): Explicit Specialization

```
template < typename T >
class C<T*> {
    . . .
}
```

A version of the original template for a **subset** of types (special case): **Partial Specialization**

Template Parameters: a Summary

Template Parameters

```
template < typename T >
class C1 {
    . . .
}
```

```
C1<int> c1;
```

Type parameter:
actual argument is a “real” type

```
template < int N, int* P>
class C2 {
    . . .
}
```

```
C2<10,&p> c2;
```

Non-type parameter:
actual argument is a constant, non-local variable, or address

```
template < template X <typename T> >
class C3 {
    . . .
}
```

Template parameter:
actual argument is a template

Tasks

1

a) Make a function template which calculates ***Fibonacci numbers*** using the following equations:

Fib(1) = 1;
Fib(2) = 1;
Fib(N) = Fib(N-1) + Fib(N-2)

Use explicit specializations for cases 1,2.

b) Is it possible to make a functionally equivalent **class template**? Try to develop this.

2

Write the complete family of class templates with `less` member:

- Generic template
- Explicit specialization for `const char*`
- Partial specializations for pointers… **and functions (!)**

Write the small program demonstrating how all kinds of templates are used.