System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

7. Variadic Templates

Eugene Zouev

Generic Programming: Introduction

Lectures 1-6: To remind

- Explicit & Partial Specializations
- Functional Objects & Templates
- C++ Standard Template Library
- The notion of iterators

The plan for today:

- `decltype` specifier
- **Variadic templates**
- Fold expressions

`decltype` specifier

decltype type-specifier

Since C++11

```
int multiply(int v1, int v2)
{
    return v1 * v2;
}
```

Suppose we have the function…

…and would like to generalize it:

```
template<typename T1, typename T2>
??? multiply(T1 v1, T2 v2)
{
    return v1 * v2;
}
```

The problem:
How to specify the type of `multiply`??

decltype type-specifier

The problem:
How to specify the type of `multiply`??

The solution:
Just a formal (and a bit funny ☺) approach

```
template<typename T1, typename T2>
decltype(v1*v2) multiply(T1 v1, T2 v2)
{
   return v1 * v2;
}
```

Incorrect…

Here we claim that the return type of the function **is the same** as the type of the product of values of types `T1` and `T2`…

While instantiating the template, the compiler will deduce the type of the product from the actual types.

decltype type-specifier

The new problem here:

v1 and v2 **are not known** at this point!!

```
template<typename T1, typename T2>
decltype(v1*v2) multiply(T1 v1, T2 v2)
{
    return v1 * v2;
}
```

The solution:

```
template<typename T1, typename T2>
auto multiply(T1 v1, T2 v2)->decltype(v1*v2)
{
    return v1 * v2;
}
```

Trailing return type

decltype type-specifier

Since C++11

```
template<typename T1, typename T2>
auto multiply(T1 v1, T2 v2)->decltype(v1*v2)
{
    return v1 * v2;
}
```

Instantiation

```
auto multiply_int,double(int v1, double v2)->double
{
    return v1 * v2;
}
```

```
int main()
{
    auto i = 1;
    auto j = 1.3;
    auto k = sum(a,b);
    cout << c << endl;
}
```

Actual types:
int, double

decltype type-specifier

Common syntax

```
decltype(expression)
```

Important: expression is not evaluated! - As in `sizeof(expression)`

Informal rules on examples

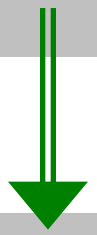
```
int i;
decltype(i) x1;       // int x1;
decltype(i+1) x2;     // int x2;
decltype((i)) x3;     // int& x3;
decltype(i=4) x4;     // int& x4;

int foo();
decltype(foo()) x5;   // int x5;
```

decltype type-specifier

One more benefit from trailing return types:
Simplification of syntax!

```
double (*(foo()))()
{
    return …
}
```



```
auto foo()->double(*)()
{
    return …
}
```

```
double (*(foo()))()
```

1
2
3
4

1. foo is a function…
2. returning a pointer…
3. to a function…
4. returning double

auto/decltype/trailing return: References

- https://stackoverflow.com/questions/23986932/purpose-of-decltype-specifier
- https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/introduction_to_the_c_11_feature_trailing_return_types?lang=en
- https://habrahabr.ru/post/206458/

Variadic templates

Preface: variable number of function parameters

```
#include <iostream>
#include <cstdarg> // needed to use ellipsis

double findAverage(int count, ...)
{
    double sum = 0;
    va_list list;
    va_start(list, count);

    // Loop through all arguments
    for (int arg=0; arg < count; ++arg)
        sum += va_arg(list, int);

    va_end(list);
    return sum/count;
}
```

The ellipsis must be the last parameter.
count is how many additional arguments we're passing

We access the ellipsis through a va_list

We initialize the va_list using va_start. The first parameter is the list to initialize. The second parameter is the last non-ellipsis parameter.

We use va_arg to get parameters out of our ellipsis. The first parameter is the va_list we're using. The second parameter is the type of the parameter

Cleanup the va_list when we're done.

Preface: variable number of function parameters

```
#include <iostream>
#include <cstdarg> // needed to use ellipsis

double findAverage(int count, ...)
{
    double sum = 0;
    va_list list;
    va_start(list, count);

    // Loop through all arguments
    for (int arg=0; arg < count; ++arg)
        sum += va_arg(list, int);

    va_end(list);
    return sum/count;
}
```

Low-level feature (came from C)

- Throws away argument types
- No information about number of arguments

Try to change 1 for 1.0…
…or for "Hello world" ☺

```
int main()
{
    cout << findAverage(5,1,2,3,4,5) << '\n';
    cout << findAverage(6,1,2,3,4,5,6) << '\n';
}
```

Template parameters

```
template<typename T1, typename T2>
class Pair {
    T2 element1;
    T2 element2;
    ...
}
```

Fixed number of parameters
=> fixed number of template arguments

```
Pair<int, int> p1;
```

```
Pair<long> p0;
Pair<int,int,int> p00;
```

```
Pair<float,double> p2;
```

```
Pair<anotherClass,yetAnotherClass> p3;
```

Russian:
Вариативный шаблон

Variadic Templates

Since C++11

The template can be instantiated with arbitrary number of arguments

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

Template parameter pack

```
C<int> c1;
C<double,float> c2;
C<OtherClass> c3;
C<> c4;
...
```

Variadic Templates

The same is about function templates:

```
template<typename ...Args>
void foo(Args... args)
{
    // do something
}
```

```
foo("Hello",42, 3.14);
foo(42);
foo();  // !!
```

The same is about **non-type** template parameters

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

```
C<1> c1;
C<2,7,10> c2;
C<> c3;
...
```

Variadic Templates: How to Use

Example 1: **The printing function**

```
void print() { cout << endl; }

template<typename T>
void print(const T& t)
{
  cout << t << endl;
}

template<typename First, typename ...Rest>
void print(const First& first, const Rest& ...rest)
{
  cout << first << ", ";
  print(rest...);
}
```

Parameter pack expansion

Variadic Templates: How to Use

```
void print() { cout << endl; }

template<typename T>
void print(const T& t) { cout << t << endl; }

template<typename First, typename ...Rest>
void print(const First& first, const Rest& ...rest)
{
  cout << first << ", ";
  print(rest...);
}
```

Compile-time recursion!!

```
print();  // calls first overload
print(1); // calls the second overload

// The third overload
print(10,20);
print(100,200,300);
print("first",2,"third",3.14159);
```

Variadic Templates

Under the Hood

```
template<typename First, typename ...Rest>
void print(const First& first, const Rest& ...rest)
{
  cout << first << ", ";
  print(rest);
}
```

Initial template

```
print(10,"twenty",30);
```

First is **int**
Rest is **char***, **int**

```
template<typename First, typename ...Rest>
void print0(const First& first, const Rest& ...rest)
{
  cout << 10 << ", ";
  print("twenty",30);
}
```

First is **char***
Rest is **int**

```
template<typename First, typename ...Rest>
void print1(const First& first, const Rest& ...rest)
{
  cout << "twenty" << ", ";
  print(30);
}
```

First is **int**
Rest is **empty**

```
template<typename T>
void print(const T& t)
{
  cout << t << endl;
}
```

Variadic Templates: How to Use

Example 2: **Tuples**

The idea: double generalization ☺

```
class SimplePair {
   int element1;
   double element2;
   ...
}
```

```
template<typename T1, typename T2>
class CommonPair {
   T2 element1;
   T2 element2;
   ...
}
```

```
template<????>
class Tuple {
   ????
   ...
}
```

Variable number of template parameters!

True tuple ⟹

Side-step: partial specializations

From the "Programming paradigms" course

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: pointer types *(except* `const char*` *)*

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

Variadic Templates: Tuples

The first step: the primary template
Tuple is a structure consisting of an arbitrary number of elements of arbitrary types

```
template<typename ...Args>
struct Tuple;
```

Never gets instantiated

The second step: template specialization
Recursive definition: **Tuple** is a structure consisting of an element of some type ("head"), **and a** (possibly empty) **tuple** ("tail")

```
template<typename Head, typename ...Tail>
struct Tuple<Head, Tail...> : Tuple<Tail...>
{
  Tuple(Head h, Tail ...tail) : Tuple<Tail...>(tail...), head(h) { }
  typedef Tuple<Tail...> base_type;
  typedef Head           value_type;

  base_type& base = static_cast<base_type&>(*this);
  Head head;
}
```

The third step:
template specialization for empty tuple

```
template<>
struct Tuple<> { }
```

Variadic Templates: Tuples

How to declare tuples:

```
Tuple<int,double,const char*> t(12,2.34,"Hello");
```

The task for your homework:
Try to "bootstrap" this declaration like we did for the `print` function before.

How to work with tuples:

```
int x = t.head;
double y = t.base.head;
const char* z = t.base.base.head;
```

Inconvenient? – Yes.
The special template function `std::get<int>()` for tuples exists.

The task for your homework (optional):
Write your own version of `get` function for this `Tuple` implementation.

Fold expressions

Fold expressions

Since C++17

The idea:
To support operations **on all variadic template arguments; i.e., unpack variadic arguments** and perform an operator on them all

Initial example:

```
template<typename ...Args>
bool all(Args... args)
{
    return (... && args);
}
```

The unary left fold expression…

```
bool b = all(true,true,true,false);
```

…expands as
`((true && true) && true) && false`

Fold expressions: syntax & semantics (1)

Unary right fold

```
( args op ... )
```

`args`: a parameter pack

$(args_1\ op\ (\ldots\ op\ args_{N-1}\ op\ args_N))$

Unary left fold

```
( ... op args )
```

$((args_1\ op\ args_2)\ op\ldots\ )\ op\ args_N$

Fold expressions: syntax & semantics (2)

Binary right fold

```
( args op ... op init )
```

`args`: a parameter pack
`init`: an expression

$$(args_1\ op\ (\ldots\ op\ args_{N-1}\ op\ (args_N\ op\ init)))$$

Binary left fold

```
( init op ... op args )
```

$$(((init\ op\ args_1)\ op\ args_2)\ op \ldots)\ op\ args_N$$

Fold expressions: syntax & semantics (3)

Possible operator signs in fold expressions:

```
+   -   *   /   %   ^   &   |
=
<   >  <=   >=  ==  !=
<<   >>
+=   -=   *=   /=   %=   ^=   &=   |= <<=   >>=
&&   ||
,
.*   ->*
```

The task for your homework:
Write a template function with variadic parameters. The body of the function should contain various kinds of fold expressions with the following operators: +, +=, <<, , (comma).

Variadic templates: references

- C++ ISO Standard, Sect. 17.1 (template parameters), 8.1.6 (fold expressions), 23.5.3 (tuple).
- http://en.cppreference.com/w/cpp/language/variadic_arguments
- http://en.cppreference.com/w/cpp/language/parameter_pack
- http://en.cppreference.com/w/cpp/language/fold
- https://msdn.microsoft.com/ru-ru/library/dn439779.aspx (Russian)
- https://en.wikipedia.org/wiki/Variadic_template
- https://habrahabr.ru/post/228031/ (Russian)
- http://artlang.net/article/view/13/ (Russian)
- https://habrahabr.ru/post/101430/ (Russian)