System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++

8. Concepts

Eugene Zouev

Generic Programming: Introduction

Lectures 1-7: To remind

- Explicit & Partial Specializations
- Functional Objects & Templates
- C++ STL; the notion of iterators
- Variadic templates & fold expressions

The plan for today:

- Variadic templates: examples
- **Concepts!**

Example 0: Prime Numbers

The code finds out **at compile time** whether a given number is a prime number

```
// p: number to check, d: current divisor
template<unsigned p, unsigned d>
struct DoIsPrime {
  static constexpr bool value =
                 (p%d != 0) && DoIsPrime<p,d-1>::value;
};
template<unsigned p> // end recursion if divisor is 2
struct DoIsPrime<p,2> {
  static constexpr bool value = (p%2 != 0);
};
template<unsigned p> // primary template
struct IsPrime {
  // start recursion with divisor from p/2:
  static constexpr bool value = DoIsPrime<p,p/2>::value;
};
```

Variadic templates & fold expressions: some examples

Example 1: Variadic Expressions

```
template<typename... T>
void printDoubled (const& T... args)
{
  print (args + args...);
}
```

For the following call:

```
using namespace std;
printDoubled(7.5, string("hello"), complex<float>(4,2));
```

The effect is:

```
using namespace std;
print(7.5 + 7.5,
      string("hello") + string("hello"),
      complex<float>(4,2) + complex<float>(4,2);
```

Example 2: Variadic Expressions

```
template<typename... T>
void addOne (const& T... args)
{
  print (args + 1 ...);
}
```

Adds the value of 1 to each actual argument

Notice the whitespace between 1 and ...

```
args + 1 ...
```

Variadic expression

Example 3: Variadic Templates & Fold Expressions

```
using namespace std;
template<typename T1, typename... TN>
constexpr bool isHomogeneous (T1, TN...)
{
  return (is_same<T1,TN>::value && ...);
}
```

Since C++17

```
isHomogeneous(43, -1, "hello")
```

Yields `false`

```
is_same<int,int>::value && is_same<int,const char*>::value
```

```
isHomogeneous("hello", " ", "world", "!")
```

Yields `true`

Example 4: Variadic Indices

```
template<typename C, typename... Idx>
void printElems (const C& coll, Idx... idx)
{
  print(coll[idx]...);
}
```

```
using namespace std;
vector<string> coll = {"good", "times", "say",  "bye"};
printElems(coll,2,0,3);
```

```
print (coll[2], coll[0], coll[3]);
```

```
template<size_t... Idx, typename C>
void printIdx (const C& coll)
{
  print(coll[Idx]...);
}
printIdx<2,0,3>(coll);
```

The Future of C++: Constraints & Concepts

The problem with C++ templates

Trivial (but typical) example

```
double m2 = Max(1.0,x);
```

```
class C {
 int m;
public:
 C() : m(0) { }
};
```

```
C c1, c2;
 … Max(c1,c2) …
```

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

```
double Max_double (double a, double b)
{
   return b<a ? a : b;
}
```

Error!!

```
C Max_C (C a, C b)
{
   return b < a ? a : b;
}
```

The problem with C++ templates

Disadvantage:

No way to clearly specify
<u>the requirements on actual parameter types</u>

```
// This function template implements the common case
// of calculating maximum.
// Actual type for the template instantiation
// should have '>' operator defined.
template < typename T>
T Max ( T, T );
```

Compiler cannot check this informal requirement

Therefore, a user of a template doesn't know whether the template is applicable to his/her needs

Templates in Other Languages

Much simpler:
- **Only type parameters**
- **Single inheritance**

Compiler can check this requirement

```
class SORTED_LIST[G -> COMPARABLE]
feature
  extend (x: G) do ... end
  item (i: INTEGER): G do ... end
  sort do ... end
end
```

Eiffel

This generic class can only be instantiated with a G that is also COMPARABLE, because this is necessary to implement the sort routine.

Templates in Other Languages

C#

```
public class Dictionary<KeyType,ValueType>
                    where KeyType : IComparable
{
    // Compare keys to find the place for
    // the new dictionary element
    public void add (KeyType key,ValueType val)
    {
        switch ( key.CompareTo(x) )  // OK
        ...
    }
    ...
}
```

It is assumed that `CompareTo()` method signature is specified in the `IComparable` interface.

Requirement on actual types:

Only types implementing `IComparable` interface can be actual types for instantiations of `Dictionary` template.

Templates in Other Languages

C#

```
interface IComparable
{
    int CompareTo ( int x ); // signature
}

class Comparator : IComparable
{
    int CompareTo(int) { ... } // implementation
}

class AnotherClass : AnotherInterface { ... }
```

Correct

```
Dictionary<Comparator,Employee> dict1 = ...;

Dictionary<AnotherClass,Employee> dict2 = ...;
```

Compile-time error

Templates in Other Languages

```
public class MyTemplate<Type1,Type2>
                  where Type1 : IComparable,
                        Type2 : MyInterface,
                        Type2 : MyBaseClass
{
    ...
}
```

Requirements on actual types:

- Actual type for `Type1` formal type must implement `IComparable` interface;
- Actual type for `Type2` formal type
  (a) must implement `MyInterface` interface, and
  (b) must be derived from `MyBaseClass` type.

Several interfaces can be specified as constraints for a certain type but **only one base class**.

Explicit & partial specs: a partial solution

From the previous lecture(s)

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; }
}
```

New C++: Constraints etc

Since C++20

Constraint:
The new notion for **specifying requirements** on template arguments.

It's applicable to all kinds of templates:

- Class templates
- Function templates
- Non-template functions that are members of class templates

1. Any compile-time expression.
2. A **concept**: **a named set of constraints**

```
template<typename T> requires expression
T Max ( T a, T b )
{
   return a>b ? a : b;
}
```

Can be treated as compile-time boolean predicate

Constraints: Examples

```
template<int N> requires N<=1000
class C
{ ... }
```

Expressions after the requires keyword are calculated while compile-time

```
template<typename T>
        requires std::is_integral_v<T>
class C
{ ... }
```

```
template<typename T, T t>
        requires std::is_integral_v<T> && t<=1000
class C
{ ... }
```

Constraints: Syntax

template-head:
    template < *template-parameter-list* > *requires-clause*$_{opt}$

template-parameter-list:
    template-parameter { , *template-parameter* }

requires-clause:
    requires *constraint-logical-or-expression*

constraint-logical-or-expression:
    constraint-logical-and-expression
        [|| *constraint-logical-and-expression*]

constraint-logical-and-expression:
        primary-expression [&& *primary-expression*]

(The syntax rules were taken from the C++ Draft Standard and transformed to EBNF notation for better understanding)

Concepts

Concept:

A **concept** is by definition just a **named set of constraints** on one or more template parameters.

A side-step:

While C++11 was being developed, **a very rich concept system** was designed for it, but integrating the feature into the language specification ended up requiring too many committee resources, and that version of concepts **was eventually dropped from C++11**…

☹☹☹

Defining Concepts

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

Our favorite Max function template ☺

So, we need to express the requirement on the T type parameter. The requirement looks like as follows: <u>T should support (contain, define) < operator</u>.

```
template<typename T>
concept LessThanComparable = requires(T x, T y)
{
   { x < y } -> bool;
}
```

```
template<typename T> requires LessThanComparable<T>
T Max ( T a, T b )
{
   return b<a ? a : b;
}
```

Using Concepts

Concept is compile-time expression

Full form of using concepts

```
template<typename T> requires LessThanComparable<T>
T Max ( T a, T b )
{
   return b<a ? a : b;
}
```

Shorthand

```
template<LessThanComparable T>
T Max ( T a, T b )
{
   return b<a ? a : b;
}
```

Using Concepts

Another example

```
template<typename T>
    requires Integral<T> || FloatingPoint<T>
T Power ( T b, T p )
{
   ...
}
```

Two concepts compose
constraint-logical-or-expression

Concepts: Syntax

```
template-declaration:
     template-head declaration
   | template-head concept-definition

concept-definition:
   concept identifier = logical-or-expression ;
```

(Simplified EBNF version)

An addition to expression syntax

```
require-expression:
   requires [ parameters ] req-body

req-body:
   { requirement { requirement } }
```

Non-empty sequence or requirements
enclosed in curly braces

(Simplified EBNF version)

Concepts: Syntax with Examples

```
requirement:
      simple-requirement
    | type-requirement
    | compound-requirement
    | nested-requirement
```

Just an expression

```
template<typename T>
concept C = requires(T a, T b) {
  a + b;
};
```

C<T> is true if a+b is a **valid expression**.
Note: **expression is not evaluated**!

Type requirement: a type feature

```
template<typename T>
concept C = requires
  typename T::inner;
  typename S<T>;
};
```

- T should contain an inner type inner.
- There should be a specialization of template S for type T.

Concepts: Syntax with Examples

requirement:
    simple-requirement
  | *type-requirement*
  | *compound-requirement*
  | *nested-requirement*

Special form

```
template<typename T>
concept C = requires(T x) {
   { *x } -> typename T::inner;
};
```

Requirements:

- *x is a **valid expression**;
- T::inner is a valid type;
- *x can be implicitly converted to T::inner

Additional constraints for local parameters:

```
template<typename T>
concept C = requires(T a) {
   requires sizeof(a)==4;
};
```

Requirement is satisfied if size of a is 4.

Concepts as Type Placeholders

```
auto x = f(y);
```

The type of x is deduced from the return type of the f function.

Here, auto is called **unconstrained placeholder**.

```
template<typename T>
concept Sortable = expression;
```

```
Sortable x = f(y);
```

The type of x is deduced from the return type of the f function. **Also**, it compiles only if the type satisfies Sortable concept.

Here, Sortable is a **constrained placeholder**.

Constraints & concepts: references

C++ Draft Standard (document N4687), Sect. 17.1 (template parameters), 8.1.7 (requires expression).

C++ Templates: The Complete Guide
David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor

http://en.cppreference.com/w/cpp/language/constraints

The task for your homework (optional):

Try the newest versions of **gcc** and **clang** compilers to check if they support requirements and concepts.