Generic programming and library development

Topics today:

Concepts and ConceptC++

Sources:

- [Czarnecki and Eisenecker 2000] §6 (Generic programming)
- [Gabriel Dos Reis and Bjarne Stroustrup, Specifying C++ concepts, *ACM SIGPLAN Notices* **41**,1 (2006), 295–308]
- [Douglas Gregor, A `conceptgcc` tutorial, Web document (2006). Available at `http://www.generic-programming.org/software/ConceptGCC/tutorial/`]
- [Douglas Gregor et al., Concepts for C++0x (revision 1), Technical report **N1849=05-0018**, ISO/IEC JTC 1 (2005)]

Course home page:

`http://www.diku.dk/forskning/performance-engineering/Generic-programming/`

Polymorphism

The word **polymorphism** means “the ability to have many forms”.

Parametric polymorphism: C++ templates

Inclusion polymorphism: C++ virtual functions

Overloading: C++ function overloading including partial specialization

Coercion: C++ built-in or user defined conversion operators or constructors to coercion

Bounded polymorphism

Bounded parametric polymorphism (or **constrained genericity**) means that we can specify some constraints on type parameters.

In C++, there is no way to explicitly specify constrains on template parameters, but many clever tricks and workarounds exist to support generic programming (including type mappings, tag dispatching, and SFINAE)

There are two approaches to specifying constraints on type parameters:

1. use an interface defined elsewhere

```
template <LessThanComparable T>
class point {
// ...
}
```

2. list all the required operations in place

```
template <typename T>
  where {bool operator<(T const&, T const&);}
class point {
// ...
}
```

Problems of C++ templates

It is theoretically interesting that the template level of C++ has the power of a Turing machine, but template metaprogramming has its problems, particularly in the areas of

- error reporting,
- debugging,
- code readability,
- code maintainability,
- separate compilation,
- compilation speed,
- internal capacity and robustness of compilers, and
- portability.

Most problems seem to be related to unbounded parametric polymorphism.

Motivating example

```
#include <list>
#include <algorithm>
using namespace std;

void f() {
  list<int> l;
  sort(l.begin(), l.end());
}
```

```
sort.C:7: error: no matching function ↩
  for call to 'sort(std::List_iterator< ↩
  int>, std::List_iterator<int>)'
<path>: note: candidates are: void std:: ↩
  sort(Iter, Iter) [with Iter = std:: ↩
  List_iterator<int>] <where clause>
sort.C:7: note: unsatisfied model ↩
  requirement 'std:: ↩
  MutableRandomAccessIterator<std:: ↩
  List_iterator<int> >'
```

Five definitions

"A **concept** is a set of requirements [on types] bundled together under a single name." [Gregor 2006]

"a type system—called **concepts**—for C++ types and values that can be used for template arguments" [Reis & Stroustrup 2006]

"concepts are compile-time predicates on types and values (e.g. integral constant values). They can be combined with the usual logical operators (*and*, *or*, *not*)." [Reis & Stroustrup 2006]

"The fundamental problem is that a template definition is not (by itself) a good specification of its requirements on its parameters. We need to make those requirements explicit and less ad hoc than the expression of an algorithm. 'Concepts' are such requirements." [Reis & Stroustrup 2006]

"Everybody's first idea for [defining the predicates] is to specify a concept as a set of operations with signatures." [Reis & Stroustrup 2006]

Old style

```
template <typename R>
void stable_sort(R a, R z);
```

Requirements for types

- `R` is a model of random-access iterator.
- `R` is mutable.
- `R`'s `value_type` is strict weakly comparable.

Preconditions

[`a`, `z`) is a valid range.

Postconditions

The elements in [`a`, `z`) are in non-decreasing order.

Complexity guarantees

Let N be $\texttt{z} - \texttt{a}$. The worst-case behaviour is $O(N(\lg N)^2)$ if no auxiliary memory is available, and $O(N \lg N)$ if a large enough auxiliary memory buffer is available.

New style

```
template <MutableRandomAccessIterator R>
  where LessThanComparable<R::value_type>
void stable_sort(R a, R z);
```

Preconditions

```
assert(z − a ≥ 0);
```

Postconditions

```
assert(is_sorted(a, z));
// how to check that no elements are lost
// how to check stability
```

Semantic requirements

operator<() on the set of elements of `R`'s `value_type` is a strict weak ordering.

Complexity guarantees

$\ldots O(N(\lg N)^2) \ldots$

Syntactic concepts

A **syntactic concept** consists of just associated types and function signatures. In particular, no support is provided for handling value members or class members of a class.

Structural conformance relies only on the signatures within a concept.

With **named conformance**, a set of types models a concept only if the user has explicitly declared that the semantics of the concept is met.

For example, one may have an `InputIterator` and `MultiPassInputIterator` concepts that have identical syntax requirements.

Pseudo-signatures

Pseudo-signatures permit conversions of the argument and result types.

```
template <typename T>
concept LessThanComparable {
  bool operator<(T const&, T const&);
  bool operator>(T const&, T const&);
  bool operator<=(T const&, T const&);
  bool operator>=(T const&, T const&);
};
```

The declaration of **operator**<() requires the existence of a < operator, either built in, as a free function, or as a member function, that can be passed two values convertible to type `T` and returns a value convertible to `bool`.

Associated types

Associated types are represented as nested types within the concept; they replace traits and permit checking of template definitions.

```
template <typename X>
concept IteratorAssociatedTypes {
  typename value_type = X::value_type;
  typename difference_type = X::difference_type;
  typename reference = X::reference;
  typename pointer = X::pointer;
};
```

If a model does not specify a type definition for an associated type, then the model uses the default.

Default implementations

Default implementations serve the same purpose as the comparison operators in the `std::rel_ops` namespace, but without its problems.

```
template <typename T>
struct concept LessThanComparable {
  bool operator<(T const&, T const&);
  bool operator<=(T const& x, T const& y) {
    return !(y < x);
  }
  bool operator>(T const& x, T const& y) {
    return y < x;
  }
  bool operator>=(T const& x, T const& y) {
    return !(x < y);
  }
};
```

This is a structural concept, so any type with a $<$ operator matching the given signature will model this concept, and will have the other operators defined automatically.

Some standard concepts

```
template <typename T, typename U = T>
struct concept Assignable {
  T& operator=(T&, U const&);
};

template <typename T, typename U = T>
struct concept EqualityComparable {
  bool operator≡(T const&, U const&);
  bool operator≠(T const& x, T const& y) {
    return !(x ≡ y);
  }
};

template <typename T, typename U>
struct concept Convertible {
  operator U(T const&);
}; // built-in, constructor, or member operation

template <typename T>
struct concept DefaultConstructible {
  T::T();
  T::~T();
};

template <typename T>
struct concept CopyConstructible {
  T::T(T const&);
  T::~T();
  T* operator&(T&);
  T const* operator&(T const&);
};
```

Refinement

```
template <typename X>
concept InputIterator
  : IteratorAssociatedTypes<X>,
    CopyConstructible<X>,
    Assignable<X>,
    EqualityComparable<X> {
  where SignedIntegral<difference_type>;
  where Convertible<reference, value_type>;
  where Arrowable<pointer, value_type>;

  typename postincrement_result = X;
  where Dereferenceable<postincrement_result,
    value_type>;

  pointer operator->(X);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
  reference operator*(X const&);
};
```

Constrained templates

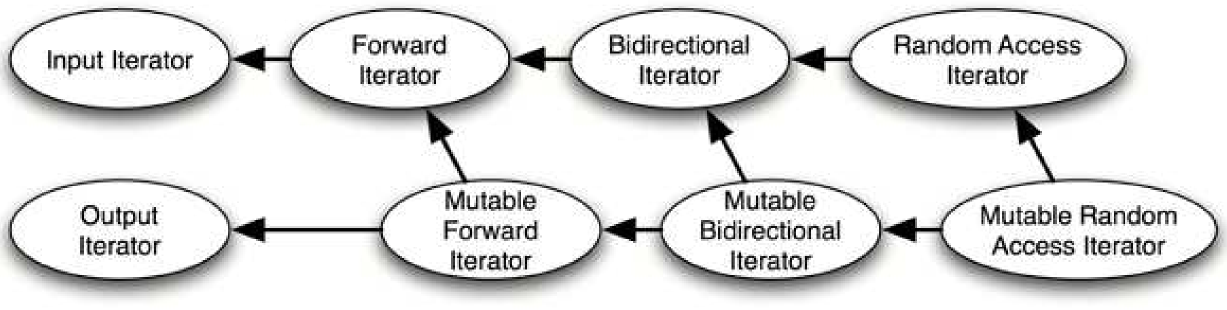
```
template <typename T>
  where CopyConstructible<T> && Assignable<T>
void swap(T& x, T& y) {
  T temp(x);
  x = y;
  y = temp;
}

template <typename T>
  where CopyConstructible<T>
class list {
public:
  ...
  where LessThanComparable<T>
  void sort();
  ...
};
```

Iterator concepts in pairs

```
template <typename X>
concept ForwardIterator
  : InputIterator<X>, DefaultConstructible<X> {
  where Convertible<reference, value_type const&>;
  where Arrowable<pointer, value_type const&>;
  where Convertible<postincrement_result,X const&>;
};

template<typename X>
concept MutableForwardIterator
  : ForwardIterator<X>, BasicOutputIterator<X> {
  where reference ≡ value_type&;
  where Arrowable<pointer, value_type&>;
};
```



Source: [Gregor 2006]

Concept-based overloading

```
template <InputIterator Iter>
void advance(Iter& i, difference_type n) {
  while (n != 0) {
    ++i;
    --n;
  }
}

template <BidirectionalIterator Iter>
void advance(Iter& i, difference_type n) {
  while (n > 0) {
    ++i;
    --n;
  }
  while (n < 0) {
    --i;
    ++n;
  }
}

template <RandomAccessIterator Iter>
void advance(Iter& i, difference_type n) {
  i += n;
}
```

Concept maps

A **model declaration** illustrates how a set of types will model a particular concept.

```
template <typename T>
concept_map ForwardIterator<T*> {
  typedef T value_type;
}
```

Each model must meet all of the requirements in the concept.

Note that for the `Assignable` concept, we never have to write a model declaration, because it was declared as a structural concept. The `ForwardIterator` concept, however, was not declared with the `struct` modifier, so the user must write model declarations.

More facilities

```
template <typename T>
concept Container {
  bool T::empty() const;
  ...
};

template <typename T>
concept Sequence : Container<T> {
  template <InputIterator In>
  T::T(In, In);
  ...
};
```

“not” constraints

```
template <typename In, typename Out>
Out unique_copy(In a, In z, Out r);
```

The expression `*r = *a` must be valid. If neither `In` nor `Out` meets the requirements of forward iterator, then the value type of `In` must be `CopyConstructible`. Otherwise `CopyConstructible` is not required.

```
template <InputIterator In, typename Out>
  where OutputIterator<Out, In::value_type> &&
        EqualityComparable<In::value_type> &&
        Assignable<In::value_type> &&
        CopyConstructible<In::value_type> &&
        !ForwardIterator<In> &&
        !MutableForwardIterator<Out>
Out unique_copy(In, In, Out);

template<ForwardIterator In, typename Out>
  where OutputIterator<Out, In::value_type> &&
        EqualityComparable<In::reference>
Out unique_copy(In, In, Out);

template<InputIterator In, MutableF...Iterator Out>
  where Assignable<Out::reference, In::reference> &&
        EqualityComparable<Out::reference,
        In::value_type> && !ForwardIterator<In>
Out unique_copy(In, In, Out);
```

Online exercise

```
template <typename T, typename U = T>
struct concept Assignable {
  T& operator=(T&, U const&);
};

template <typename T, typename U = T>
struct concept Movable {
  T& operator=(T&, U&);
};

template <InputIterator In, OutputIterator Out>
  where {}  // What should we write here?
Out copy(In a, In z, Out r) {
  for (In p = a; p != z; ++p, ++r) {
    *r = *p;
  }
}
```

Pros and cons of concept-constrained genericity

+ improved error messages
+ debugging easier for library authors
+ explicit descriptions of the import interfaces
+ new opportunities for overloading
+ separate compilation possible
+ improved static type checking
+ lower barrier to novices
− more to learn
− more to type
− duplication of the interface information
− flexibility of lazy type checking lost
− possibility for over-specification

Semantic concepts

Algebraic concepts (such as monoid, group, ring, field), ordering concepts (such as strict weak ordering), sequential computation concepts (such as container, iterator, range)

- detection of range violations (e.g. dereferencing a past-the-end iterator)
- detection of multi-pass property of forward iterators
- detection of iterator invalidation
- checking for proper use of algorithms that require a concept (sortedness property required by `binary_search()`)
- concept-based rewriting ($x \oplus 0 \rightarrow x$ when $(x, +)$ models the monoid concept)
- checking for satisfaction of the axioms (comparator for `sort()` should obey the axioms of the strict weak order concept)

Related literature: static analysis, program transformation, proof checking, etc.

Monoid

Let $(A, \star)$ be an algebraic system, where $\star$ is a binary operation on A. $(A, \star)$ is called a **monoid** if the following conditions are satisfied:

1. $\star$ is a closed operation.
2. $\star$ is an associative operation.
3. There is an identity.

Example: Let A be a set of people of different heights, and let $\triangle$ be a binary operation such that $a \triangle b$ is equal to the taller one of a and b. We note that $(A, \triangle)$ is a monoid where the identity is the shortest person in A.

Strict weak ordering

A binary relation $\bigcirc$ on set S is **irreflexive** if $x \bigcirc x$ is false for all $x \in S$, and it is **transitive** if $x \bigcirc y$ and $y \bigcirc z$ implies $x \bigcirc y$ for all $x, y, z \in S$.

A binary relation $\olessthan$ is a **strict weak ordering** if

1. it is irreflexive,
2. transitive, and
3. if the relation $\circledequal$, defined by

   $x \circledequal y \iff$ both $x \olessthan y$ and $y \olessthan x$ are false,

   is transitive.

Example: $<$ is a strict weak ordering on the set of integers.

Conclusions

In the CPH STL project, all development will be moved from C++ to ConceptC++ after the prototype compiler `conceptgcc` has been installed to our computers.