tomorrow's software technologies

Universität Bamberg

Generic Programming with 'Concepts' in C++

Trends in Programming Languages June 20, 2012

Felix Härer

Outline

1. Generic Templates in C++
2. Concepts in C++
3. Summary and Outlook

Generic Programming with Templates

Using templates, *functions* or *classes* can be defined with generic parameters instead of concrete types.

- **Template Definition**
  - user defined or included from a library (e.g. C++ STL)
  - Example for a *class template*:

```
template <class T>
class tuple {
  public:
    T data1, data2;
    tuple(T d1, T d2) { data1 = d1; data2 = d2; }
    void swap() { T tmp = data1; data1 = data2; data2 = tmp; }
};
```

- **Use of template**
  - Instantiation of the template with a concrete type

```
void f()
{
  tuple<int> aTuple(4, 5);
  aTuple.swap();
}
```

Function Template - Example

Template Definition

- calculate a sum of N absolute values
- read values stored in data
- write sum to result

```
template<typename T>
void absoluteSum(T data[], const int N, T &result)
{
  T sum = 0;
  for (int i = 0; i < N; ++i)
  {
    if (data[i] >= 0)
      sum = sum + data[i];
    else
      sum = sum - data[i];
  }
  result = sum;
}
```

Use of Template

- template function call
- use of different types

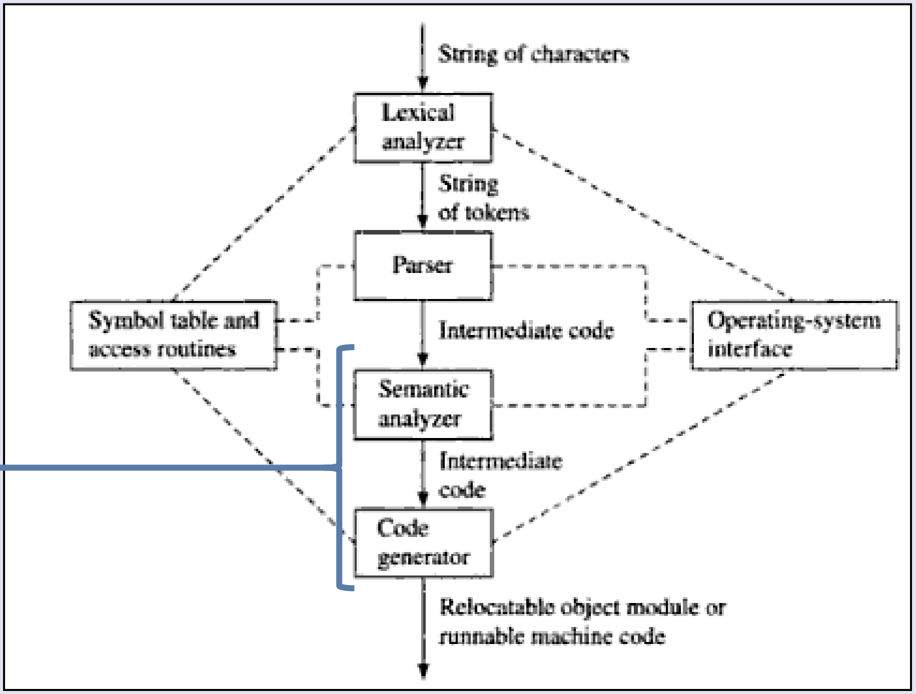
```
void f()
{
  int iArray[] = {-3, 4, 5};
  int iResult = 0;
  absoluteSum(iArray, 3, iResult);


  double dArray[] = {3.3, -4.4};
  double dResult = 0;
  absoluteSum(dArray, 2, dResult);
}
```

Compilation of Templates

Compile Time Instantiation

- functions with concrete types are generated at compile time
- function code is generated for each template instantiation with a unique set of concrete parameters

➢ no run time difference between functions and generic functions (templates)



High-level structure of a simple compiler (Muchnick 1997, p. 2)

Advantages of Generic Templates

- no overhead
- same run time performance
- flexibility through generic types
- type safe, type-checking at compile time

- ➢ less redundant code
  - ➢ more efficient programming
  - ➢ better maintainability
- ➢ generic programming decouples algorithms from data types

Problems of Generic Templates

Type Checking

- late type checking at compile time
- no separate checking of template definition and template usage
- generic parameters can not be constrained
- no overloading

Error Messages

- confusing messages:
  wrong use of a template results in an error inside the template definition
- long messages:
  output for each level of the whole call hierarchy

➢ user of a template needs to know the definition

Illustrating Problems of Generic Templates

Two uses of the STL template ‘stable_sort‘:

```
void sortVector()
{
  vector<int> v;
  v.push_back(7);
  v.push_back(2);
  stable_sort(v.begin(), v.end());
}
```

```
void sortList()
{
  list<int> l;
  l.push_back(7);
  l.push_back(2);
  stable_sort(l.begin(), l.end());
}
```

Illustrating Problems of Generic Templates

Two uses of the STL template ‘stable_sort‘:

```
void sortVector()
{
  vector<int> v;
  v.push_back(7);
  v.push_back(2);
  stable_sort(v.begin(), v.end());
}
```

```
void sortList()
{
  list<int> l;
  l.push_back(7);
  l.push_back(2);
  stable_sort(l.begin(), l.end());
}
```

```
$ g++ sortVector.cpp
$ 
```

(compiles without error)

```
$ g++ sortList.cpp
/usr/include/c++/4.2/bits/stl_algo.h: In function 'void std::__inplace_stable_sort(_RandomAccessIterator, _Rando
mAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]':
/usr/include/c++/4.2/bits/stl_algo.h:3892:   instantiated from 'void std::stable_sort(_RandomAccessIterator, _Ra
ndomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]'
sortList.cpp:20:   instantiated from here
/usr/include/c++/4.2/bits/stl_algo.h:3174: error: no match for 'operator-' in '__last - __first'
/usr/include/c++/4.2/bits/stl_bvector.h:182: note: candidates are: ptrdiff_t std::operator-(const std::_Bit_iter
ator_base&, const std::_Bit_iterator_base&)
/usr/include/c++/4.2/bits/stl_algo.h:3892:   instantiated from 'void std::stable_sort(_RandomAccessIterator, _Ra
ndomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]'
sortList.cpp:20:   instantiated from here
/usr/include/c++/4.2/bits/stl_algo.h:3179: error: no match for 'operator-' in '__last - __first'
/usr/include/c++/4.2/bits/stl_bvector.h:182: note: candidates are: ptrdiff_t std::operator-(const std::_Bit_iter
ator_base&, const std::_Bit_iterator_base&)
/usr/include/c++/4.2/bits/stl_algo.h:3182: error: no match for 'operator-' in '__middle - __first'
/usr/include/c++/4.2/bits/stl_bvector.h:182: note: candidates are: ptrdiff_t std::operator-(const std::_Bit_iter
ator_base&, const std::_Bit_iterator_base&)
/usr/include/c++/4.2/bits/stl_algo.h:3182: error: no match for 'operator-' in '__last - __middle'
/usr/include/c++/4.2/bits/stl_bvector.h:182: note: candidates are: ptrdiff_t std::operator-(const std::_Bit_iter
ator_base&, const std::_Bit_iterator_base&)
/usr/include/c++/4.2/bits/stl_algo.h: In function 'void std::__stable_sort_adaptive(_RandomAccessIterator, _Rand
omAccessIterator, _Pointer, _Distance) [with _RandomAccessIterator = std::_List_iterator<int>, _Pointer = int*,
_Distance = int]':
/usr/include/c++/4.2/bits/stl_algo.h:3894:   instantiated from 'void std::stable_sort(_RandomAccessIterator, _Ra
ndomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]'
sortList.cpp:20:   instantiated from here
/usr/include/c++/4.2/bits/stl_algo.h:3809: error: no match for 'operator-' in '__last - __first'
/usr/include/c++/4.2/bits/stl_bvector.h:182: note: candidates are: ptrdiff_t std::operator-(const std::_Bit_iter
ator_base&, const std::_Bit_iterator_base&)
/usr/include/c++/4.2/bits/stl_algo.h:3810: error: no match for 'operator+' in '__first + __len'
/usr/include/c++/4.2/bits/stl_bvector.h:267: note: candidates are: std::_Bit_iterator std::operator+(ptrdiff_t,
const std::_Bit_iterator&)
/usr/include/c++/4.2/bits/stl_bvector.h:353: note:                 std::_Bit_const_iterator std::operator+(ptrdi
ff_t, const std::_Bit_const_iterator&)
/usr/include/c++/4.2/bits/stl_algo.h:3894:   instantiated from 'void std::stable_sort(_RandomAccessIterator, _Ra
ndomAccessIterator) [with _RandomAccessIterator = std::_List_iterator<int>]'
sortList.cpp:20:   instantiated from here
/usr/include/c++/4.2/bits/stl_algo.h:3823: error: no match for 'operator-' in '__middle - __first'
/usr/include/c++/4.2/bits/stl_bvector.h:182: note: candidates are: ptrdiff_t std::operator-(const std::_Bit_iter
ator_base&, const std::_Bit_iterator_base&)
```

Illustrating Problems of Generic Templates

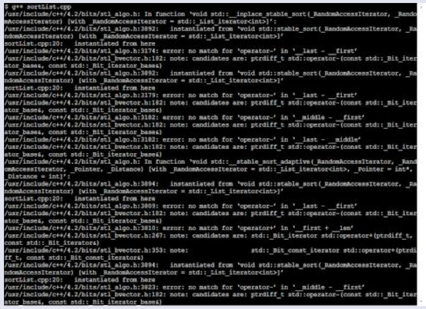
Two uses of the STL template ‘stable_sort‘:

```
void sortVector()
{
  vector<int> v;
  v.push_back(7);
  v.push_back(2);
  stable_sort(v.begin(), v.end());
}
```

```
void sortList()
{
  list<int> l;
  l.push_back(7);
  l.push_back(2);
  stable_sort(l.begin(), l.end());
}
```

```
$ g++ sortVector.cpp
$
```

(compiles without error)



➢ stable_sort requires random access (RandomAccessIterator), but a linked list only provides sequential access

Improving Templates Using Documentation

Requirements in C++98

24.1.2 Output iterators **[lib.output.iterators]**

A class or a built-in type `X` satisfies the requirements of an output iterator if `X` is an Assignable type (23.1) and also the following expressions are valid, as shown in Table 73:

Table 73—Output iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X(a)`			`a = t` is equivalent to `X(a) = t`. note: a destructor is assumed.
`X u(a);` `X u = a;`			
`*a = t`	result is not used		
`++r`	`X&`		`&r == &++r`.
`r++`	convertible to `const X&`	`{ X tmp = r; ++r; return tmp; }`	
`*r++ = t`	result is not used		

(C++98, p. 511)

Improving Templates Using Concepts

Idea

- add requirements as seen in the C++98 standard to the language itself
- allow constraints on generic template parameters
- ➢ provide a **type system for template parameters**

Addresses the Problems

- long and complicated error messages
- no overloading
- template user needs to know the template definition
- late type checking

Outline

1. Generic Templates in C++

2. Concepts in C++

3. Summary and Outlook

Initial Aims of C++ Concepts

- A system as flexible as current templates
- **Enable better checking of template definitions**
- **Enable better checking of template uses**
- **Better error messages**
- Selection of template specialization based on attributes of template arguments
- **Typical code performs equivalent to existing template code**
- Simple to implement in current compilers
- **Compatibility with current syntax and semantics**
- **Separate compilation of template and template use**
- **Simple/terse expression of constrains**
- Express constraints in terms of other constraints
- Constraints of combinations of template arguments
- **Express semantics/invariants of concept models**
- The extensions shouldn’t hinder other language improvements

(Stroustrup and Dos Reis 2003)

Six Aims of C++ Concepts

1. **modular type checking for template definitions and uses**
2. **better error messages**
3. **same performance for typical code**
   - at run time
   - compile time not more than +20%
4. **separation of author and user of a template**
   - type check of a template implementation
   - type check of all template uses against template signature
5. **simple and terse expression**
   - concepts are only used if they are usable
   - not just an academic concept
6. **express semantics and invariants of concept models**

Concepts – Overview

1. **Concept Definition**
   - specifies the required behavior of types
   - syntactic definition of required operations
   - semantic definition of axioms (optional)
   - implicitly or explicitly mapped to types
2. **Concept Map**
   - for explicitly mapping types to a concept
   - specifies how types meet the requirements of a concept
3. **Constrained Template**
   - template definition with ‘requires’ clause

Concepts – Abstract Example

The STL template ‘stable_sort‘ can only handle data structures which allow random access.

1. **Concept Definition**
   - define a concept for types that allows random access
   - requires a type to have random access operators []

2. **Concept Map**
   - the STL container class ‘vector’ allows random access
   - map ‘vector‘ to the random access concept

3. **Constrained Templates**
   - add the concept as a requirement to the template definition

1. Concept Definition

A concept definition specifies the required behavior of types.

```
concept Addable<typename T>
{
  // syntactic:
  T operator+(T x, T y);

  // semantic:
  axiom symmetry(T x, T y)
  {
    // x+y is equivalent to y+x
    x+y <=> y+x;
  }
}
```

Syntactic Definition

- defines all operations a type needs to implement
- used for type checking
- type check based on available operations

Semantic Definition

- optional, not used for type checking
- only used for compiler optimization
- axioms with logical connectives

1. Concept Definition

Example for a ‘Stack’ Concept

```
concept Stack<typename S> {
  typename value_type;
  bool empty(S&);
  void push(S&, value_type);
  void pop(S&);
  value_type& top(S&);
}
```

(Gregor and Stroustrup 2006, p. 25)

- typename value_type:
  defines value_type as a name for an arbitrary type
- concrete and generic types can be mixed

1. Concept Definition

Implicit and Explicit Concept Definitions

Implicit Concept or auto Concept

- duck typing:
  types are automatically mapped to a concept if they implement all operations defined in the concept definition
- use 'auto' keyword:

```
auto concept GreaterEqualsComparable<typename T>
{
  bool operator>=(T x, T y);
};
```

Explicit Concept

- each type needs to be mapped explicitly to the concept
- ➢ a concept map needs to be specified

2. Concept Map

- explicit mapping of types to a concept
- specifies how types meet the requirements of a concept, e.g. ‘make a vector behave like a stack’:

```
concept Stack<typename S> {
  typename value_type;
  bool empty(S&);
  void push(S&, value_type);
  void pop(S&);
  value_type& top(S&);
}
// Make a vector behave like a stack
template<Regular T>
concept_map Stack<std::vector<T> > {
  typedef T value_type;
  bool empty(std::vector<T>& vec) { return vec.empty(); }
  void push(std::vector<T>& vec, value_type value) { vec.push_back(value); }
  void pop(std::vector<T>& vec) { vec.pop_back(); }
  value_type& top(std::vector<T>& vec) { return vec.back(); }
}
```

(Gregor and Stroustrup 2006, p. 25)

2. Concept Map

- explicit mapping of types to a concept
- specifies how types meet the requirements of a concept, e.g. ‘make a vector behave like a stack’:

```
concept Stack<typename S> {
  typename value_type;
  bool empty(S&);
  void push(S&, value_type);
  void pop(S&);
  value_type& top(S&);
}
// Make a vector behave like a stack
template<Regular T>
concept_map Stack<std::vector<T> > {
  typedef T value_type;
  bool empty(std::vector<T>& vec) { return vec.empty(); }
  void push(std::vector<T>& vec, value_type value) { vec.push_back(value); }
  void pop(std::vector<T>& vec) { vec.pop_back(); }
  value_type& top(std::vector<T>& vec) { return vec.back(); }
}
```

```
// concept for well-behaved types
auto concept Regular<typename T> {
  T::T();                // default constructor
  T::T(const T&);        // copy constructor
  // ...
};
```

(Gregor and Stroustrup 2006, p. 25)

3. Constrained Templates

A templates definition *requires* a concept for a type.

- templates with ‘requires’ keyword are type checked

```
auto concept GreaterEqualsComparable<typename T>
{
  bool operator>=(T x, T y);
};
```

```
template<typename T>
requires GreaterEqualsComparable<T>
bool isGreaterEquals(T x, T y)
{
  return (x >= y);
}
```

Concepts for the ‘absoluteSum‘ Template

A template may use any number of concepts in the require clause.

```
auto concept Assignable<typename T>                 { T& operator=(T& x, T y); };
auto concept IntConstructible<typename T>           { T::T(int); };
auto concept GreaterEqualsComparable<typename T> { bool operator>=(T x, T y); };
auto concept Addable<typename T>                    { T operator+(T x, T y); };
auto concept Subtractable<typename T>               { T operator-(T x, T y); };
```

```
template<std::CopyConstructible T>
requires Assignable<T>, IntConstructible<T>, GreaterEqualsComparable<T>,
  Addable<T>, Subtractable<T>
void absoluteSum(T data[], const int N, T &result)
{
  T sum = 0;
  for (int i = 0; i < N; ++i)
  {
    if (data[i] >= 0)
      sum = sum + data[i];
    else
      sum = sum - data[i];
  }
  result = sum;
}
```

Concept Inheritance

Concepts can be reused by other concepts

- concepts may inherit any number of other concepts
- allows specialized concepts
- allows composed concepts
- e.g. create an AbsoluteSummable concept:

```
auto concept AddableSubtractable<typename T>
  : Assignable<T>, Addable<T>, Subtractable<T> {};

auto concept AbsoluteSummable<typename T> : AddableSubtractable<T>
{
  // IntConstructible
  T::T(int);
  // GreaterEqualsComparable
  bool operator>=(T x, T y);
};
```

```
template<std::CopyConstructible T>
requires AbsoluteSummable<T>
void absoluteSum(T data[], const int N, T &result)
```

Error Messages

Call absoluteSum with a user defined type ‘my_int‘

```
12 template<typename T>
13 requires AbsoluteSummable<T>
14 void absoluteSum(T data[], const int N, T &result)
15 {
16   T sum = 0;
17   for (int i = 0; i < N; ++i)
18   {
19     if (data[i] >= 0)
20       sum = sum + data[i];
21     else
22       sum = sum - data[i];
23   }
24   result = sum;
25 }
```

```
27 struct my_int
28 {
29   int i;
30 };
31
32 int main()
33 {
34   const int LEN = 3;
35   my_int myArray[LEN] = { 1, 2, 3 };
36   my_int myResult;
37   absoluteSum(myArray, LEN, myResult);
38   return 0;
39 }
```

Error messages from ‘conceptg++’:

```
summable.cpp: In function ‘int main()’:
summable.cpp:37: error: no matching function for call to ‘absoluteSum(my_int [3], int, my_int&)’
summable.cpp:14: note: candidates are: void ‘absoluteSum(T*, int, T&) [with T = my_int] <requirements>
summable.cpp:37: note:   no concept map for requirement AbsoluteSummable<my_int>’
```

Outline

1. Generic Templates in C++
2. Concepts in C++
3. Summary and Outlook

Concepts Improve Generic Templates

- ✓ **modular type checking for template definitions and uses**
- ✓ **better error messages**
- ○ **same performance for typical code**
  - ✓ same at run time
  - ○ compile time about 10x (aim: not more than +20%)
- ✓ **separation of author and user of a template**
  - ✓ separate type checking of a template and its use against a concept
- ○ **simple and terse expression**
  - ✓ concept definitions are usually not hard to write
  - ○ concept maps can be very complicated
- ✓ **express semantics and invariants of concept models**

Outlook

In 2008, concepts were voted into the C++0x standard – in 2009, the C++0x committee voted for a removal of concepts from the standard.

Reasons

- implementation not 'production quality', but proof of concept
- compile time increases (conceptgcc: 10x)
- ease of use for the mainstream programmer
- technical issues about implicit and explicit concepts
  - should implicit auto concepts be used (ambiguity problems)?
  - give the programmer a choice of using implicit or explicit concepts?

Outlook

Uncertain Future of Concepts

- no official statements about the future of concepts
- the new standard is not on the horizon, yet
- Bjarne Stroustrup (2009):
  "I hope we will see 'concepts' in a revision of C++ in maybe five years."

Universität Bamberg

Generic Programming with ‘Concepts’ in C++

Trends in Programming Languages

June 20, 2012

Felix Härer

References

Douglas Gregor, Bjarne Stroustrup (2005): Concepts (revision 1).
Technical Report N2081=06-0151.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2081.pdf

Douglas Gregor, Bjarne Stroustrup, Jaakko Jäarvi, Gabriel Dos Reis, Jeremy Siek, Andrew Lumsdaine (2006): Concepts: Linguistic Support for Generic Programming in C++.
In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. ACM, New York, Pages 291-310.
http://www2.research.att.com/~bs/oopsla06.pdf

Gabriel Dos Reis, Bjarne Stroustrup (2006): Specifying C++ Concepts.
In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, New York, Pages 295 - 308.
http://www.research.att.com/~bs/popl06.pdf

Gabriel Dos Reis, Bjarne Stroustrup, Alisdair Meredith (2009): Axioms: Semantics Aspects of C++ Concepts.
Technical Report N2887=09-0077.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2887.pdf

References

ISO
(1998): Programming languages -- C++.
International Standard, ISO/IEC 14882:1998(E).
http://www-d0.fnal.gov/~dladams/cxx_standard.pdf

Bjarne Stroustrup
(2009): Simplifying the use of concepts.
Technical Report N2906=09-0096.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2906.pdf

Bjarne Stroustrup, Gabriel Dos Reis
(2003): Concepts – design choices for template argument checking.
Technical Report N1522=03-0105.
http://www.open-std.org/jtc1/sc22/wg21

Steven Muchnick
(1997): Advanced Compiler Design and Implementation.
Morgan Kaufmann Publishers.

Andrew Sutton, Bjarne Stroustrup
(2012): Design of Concept Libraries for C++.
In: Lecture Notes in Computer Science, 2012, Volume 6940/2012, pages 97-118, DOI: 10.1007/978-3-642-28830-2_6.
http://www2.research.att.com/~bs/sle2011-concepts.pdf