
Generic Programming in C++

The next level

Gabriel Dos Reis

`dosreis@cmla.ens-cachan.fr`

Centre de Mathématiques et de Leurs Applications

ENS Cachan – CNRS (UMR 8536)

France

1. What is Generic Programming?
2. Current State of Affairs
3. Wishlist for enhanced GP
4. Constrained Templates
5. Signatures
6. Conformance
7. Parametrized signatures

What is Generic Programming?

What is Generic Programming?

[David Musser]:

Generic programming is "**programming with concepts**"

A **concept** is a family of abstractions that are all related by a common set of requirements.

A type T is a model of a concept \mathcal{C} , if it satisfies all the requirements of \mathcal{C} .

Examples:

- **Sequences** (list, vector, deque)
- **Forward, bidirectional, random access iterators**
- **NonCopyables** (standard streams)

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions**

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions** Example: \mathbb{T} is a model of EqualityComparable if both expressions $x == y$ and $x != y$ are syntactically valid and convertible to `bool`

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions**
- **Expression semantics**

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions**
- **Expression semantics** Example: the expression $x \neq y$ should be semantically equivalent to $!(x = y)$ whenever both are valid.

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions**
- **Expression semantics**
- **Complexity requirements**

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions**
- **Expression semantics**
- **Complexity requirements**
- **Invariants**

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions**
- **Expression semantics**
- **Complexity requirements**
- **Invariants**
 - Identity: $\&x == \&y$ implies $x == y$
 - Symmetry: $x == y$ implies $y == x$
 - Transitivity: $x == y$ and $y == z$ implies $x == z$

Programming with Concepts

A concept consists of:

- **(syntactically) Valid Expressions**
- **Expression semantics**
- **Complexity requirements**
- **Invariants**

Note: Except for the first aspect which is **statically** verifiable, the last three are fundamentally **runtime** attributes.

Programming with Concepts (cont'd)

In the same way subtypes are essential for an ordered extension of large software systems, the notion of **concept refinement** is essential for an ordered extension of generic components.

A concept \mathcal{D} refines a concept \mathcal{C} if \mathcal{D} 's set of requirements is a superset of that of \mathcal{C} .

Programming with Concepts (cont'd)

In the same way subtypes are essential for an ordered extension of large software systems, the notion of **concept refinement** is essential for an ordered extension of generic components.

A concept \mathcal{D} refines a concept \mathcal{C} if \mathcal{D} 's set of requirements is a superset of that of \mathcal{C} .

Examples:

- RandomAccessIterator refines BidirectionalIterator
- UpperTriangularMatrix refines SquareMatrix
- DrawingAreaWidget refines CoreWidget

Current State of Affairs

Currently, **concepts are not directly supported** by C++.
Instead, they are described in an eventual
documentation.

Current State of Affairs

Currently, **concepts are not directly supported** by C++. Instead, they are described in an eventual documentation.

- As usual in practice, by Murphy's law, the intent and actual behaviour may disagree.

Current State of Affairs

Currently, **concepts are not directly supported** by C++. Instead, they are described in an eventual documentation.

- As usual in practice, by Murphy's law, the intent and actual behaviour may disagree.
- Diagnostic messages may be hard to decipher.

Current State of Affairs

Currently, **concepts are not directly supported** by C++. Instead, they are described in an eventual documentation.

- As usual in practice, by Murphy's law, the intent and actual behaviour may disagree.
- Diagnostic messages may be hard to decipher.
- Concept refinement is emulated by meta-switches (traits, tags + overloading)

Consider the program construct

```
std::vector<size_t> v(2002, 4); // #1
```

The set of constructor candidates includes:

1. `vector(size_t n, const T& t)`
2. `template<class In>`
`vector(In first, In last)`

Consider the program construct

```
std::vector<size_t> v(2002, 4); // #1
```

The set of constructor candidates includes:

1. `vector(size_t n, const T& t)`
2. `template<class In>`
`vector(In first, In last)`

A specialization of the template constructor (with `In=int`) is selected because it makes the best match.

How does #1 have the Right Semantics?

Unusual Semantics (cont'd)

1. `vector(size_t n, const T& t)`
2. `template<class In>`
`vector(In first, In last)`

Magic:

Whenever the actual template-argument bound to `In` is an integral type, the effect should be as if

```
vector(static_cast<size_type>(first),  
        static_cast<T>(last))
```

were called.

Unusual Semantics (cont'd)

1. `vector(size_t n, const T& t)`
2. `template<class In>`
`vector(In first, In last)`

Magic:

Whenever the actual template-argument bound to `In` is an integral type, the effect should be as if

```
vector(static_cast<size_type>(first),  
       static_cast<T>(last))
```

were called.

Does the above resolution repair the unplanned semantics clash?

Unusual Semantics (cont'd)

```
#include <vector>
struct A { }; struct B { };
struct X {
    X() { }
    X(B) { }
    explicit X(int) { }
    explicit X(A) { }
};

int main()
{
    std::vector<X> u(25, B()); // OK
    std::vector<X> v(25, A()); // ERROR
    std::vector<X> w(25, 4);   // OK - ?!
}
```

A classical example

```
#include <list>
#include <vector>
#include <algorithm>

int main()
{
    std::list<std::vector<int> > l;
    // ...
    // put good stuff in 'l'
    // ...
    std::sort(l.begin(), l.end()); // Oops
}
```

A classical example(cont'd)

```
/codesourcery/include/g++-v3/bits/stl_algo.h: In function 'void
  std::sort(_RandomAccessIter, _RandomAccessIter, const Compare, const Allocator&):
  std::_List_iterator<std::vector<int, std::allocator<int> >&, std::vector<int, std::allocator<int> >&>
1.C:11:   instantiated from here
/codesourcery/include/g++-v3/bits/stl_algo.h:213:   instantiated from here
  std::_List_iterator<std::vector<int, std::allocator<int> >&, std::vector<int, std::allocator<int> >&>
  std::allocator<int> >&, std::vector<int, std::allocator<int> >&>
  std::_List_iterator<std::vector<int, std::allocator<int> >&, std::vector<int, std::allocator<int> >&>
  std::allocator<int> >&, std::vector<int, std::allocator<int> >&>
/codesourcery/include/g++-v3/bits/stl_bvector.h: In function 'void
  std::operator-(const std::_Bit_iterator_base&, const std::_Bit_iterator_base&):
  std::_Bit_iterator_base&)
```

A classical example(cont'd)

```
/codesourcery/include/g++-v3/bits/stl_algo.h: In
  std::__final_insertion_sort(_RandomAccessIter
  _RandomAccessIter = std::_List_iterator<std:
  >, std::vector<int, std::allocator<int> >&, s
  std::allocator<int> >*>]':
/codesourcery/include/g++-v3/bits/stl_algo.h:213
1.C:11:   instantiated from here
/codesourcery/include/g++-v3/bits/stl_algo.h:201
  std::_List_iterator<std::vector<int, std::all
  std::allocator<int> >&, std::vector<int, std
  std::_List_iterator<std::vector<int, std::all
  std::allocator<int> >&, std::vector<int, std
```

A classical example(cont'd)

```
/codesourcery/include/g++-v3/bits/stl_bvector.h  
    std::operator-(const std::_Bit_iterator_base&  
    std::_Bit_iterator_base&)  
/codesourcery/include/g++-v3/bits/stl_algo.h:202  
    std::_List_iterator<std::vector<int, std::all  
    std::allocator<int> >&, std::vector<int, std  
    std::<anonymous enum>' operator  
/codesourcery/include/g++-v3/bits/stl_bvector.h  
    std::_Bit_iterator std::operator+(int, const  
/codesourcery/include/g++-v3/bits/stl_bvector.h  
    std::_Bit_const_iterator std::operator+(int,  
    std::_Bit_const_iterator&)  
/codesourcery/include/g++-v3/bits/stl_algo.h:202  
    std::_List_iterator<std::vector<int, std::all  
    std::allocator<int> >&, std::vector<int, std
```

A classical example(cont'd)

```
l.C:11:      instantiated from here
/codesourcery/include/g++-v3/bits/stl_algo.h:194
    std::_List_iterator<std::vector<int, std::al
std::allocator<int> >&, std::vector<int, std
operator
/codesourcery/include/g++-v3/bits/stl_bvector.h
    std::_Bit_iterator std::operator+(int, const
/codesourcery/include/g++-v3/bits/stl_bvector.h
    std::_Bit_const_iterator std::operator+(int,
std::_Bit_const_iterator&)
```

Emulating concept refinement

```
template<typename Bi>
inline void reverse(Bi first, Bi last)
{
    __reverse(first, last,
               __iterator_category(first));
}
```

```
template<typename Bi>
void
__reverse(Bi, Bi, bidirectional_iterator_tag);
```

```
template<typename Random>
void
__reverse(Random, Random,
           random_access_iterator_tag);
```

What about this?

```
enum ShapeType { TRIANGLE, SQUARE, CIRCLE };
struct Shape { Type type; void* data; };

void rotate(Shape* shape, double angle)
{
    switch (shape->type)
    {
        case TRIANGLE:
            // ...
            // ...
        }
    }
}
```

Wishlist for enhanced GP in C++

- If Generic Programming is programming with concepts, then a better support for GP in C++ should encompass a (better) support for concepts within the language:
 - support to enforce statically verifiable requirements
 - support to teach the compiler about dynamic requirements – they may be used as "extra" knowledge in code optimization.

Wishlist for enhanced GP in C++

- If Generic Programming is programming with concepts, then a better support for GP in C++ should encompass a (better) support for concepts within the language:
 - support to enforce statically verifiable requirements
 - support to teach the compiler about dynamic requirements – they may be used as "extra" knowledge in code optimization.
- More uniformity in rules governing control abstraction (functions and function objects).
- ...

Objection

```
#include <cmath>
#include <algorithm>
struct Cos {
    template<typename T>
    T operator()(T x) const
    { return std::cos(x); }
};

int main()
{
    float a[] = { 0.8f, 2.67f /*, ...*/ };
    double b[] = { 0.345, 9.4 /*, ...*/ };
    std::transform(a, a + size(a), a, Cos());
    std::transform(b, b + size(b), b, std::cos);
}
```

Constrained templates

Experience has evidenced needs for some form of expressing constraints on admissible values of template arguments.

Constrained templates

Experience has evidenced needs for some form of expressing constraints on admissible values of template arguments.

Basically there are two forms of constraints:

- "Positive" constraints: a template argument is admissible iff some expressions are well-formed.
- "Negative" constraints: a template argument is admissible iff some expressions are not well-formed.

Problems with classics

```
template<typename T>
struct EqualityComparable {
    static void constraints(const T& x, const T& y)
    {
        x == y;
        x != y;
    }
};
```

```
template<typename T>
void f(const T& a, const T& b)
{
    &EqualityComparable<T>::constraints;
    // ...
}
```

Problems with classics (cont'd)

```
struct X { };
```

```
int main()  
{  
    f(X(), X());  
}
```

```
f.C: In static member function 'static void  
    EqualityComparable<T>::constraints(const T&, co  
f.C:13:   instantiated from 'void f(const T&, co  
f.C:21:   instantiated from here  
f.C:5: no match for 'const X& == const X&' opera  
f.C:6: no match for 'const X& != const X&' opera
```

Problems with classics (cont'd)

- Depends on the compiler's internal details.
- Too coarse grained: either it compiles or it doesn't.
- Things may get more complicated (semantics mismatch).
- No standard support (keep reinventing the wheel).

Signatures: Concept static typing

Constrained templates may be achieved by **a language support for concept static requirements** (valid expressions) **enforcement**.

Signatures: Concept static typing

Constrained templates may be achieved by **a language support for concept static requirements** (valid expressions) **enforcement**.

Signatures conceived **as "types" of types**: a set of constraints imposed on template-arguments used to instantiate a template.

Signatures aren't concepts. Signatures only tackle the statically verifiable part of concepts: the syntactic validity of expressions.

Each template parameter may be introduced by a declaration that states its "constraints attribute".

Example: Instead of

```
template<typename In> ...
```

we may write

```
template<InputIterator In> ...
```

where `InputIterator` is a known signature that describes constraints for input iterators.

Each template parameter may be introduced by a declaration that states its "constraints attribute".

Example: Instead of

```
template<typename In> ...
```

we may write

```
template<InputIterator In> ...
```

where `InputIterator` is a known signature that describes constraints for input iterators.

At instantiation time (either partial or full), a specialization is considered only if each template-argument conforms to the requirements specified by the corresponding template-parameter signature.

Signatures (cont'd)

- No new major infrastructure is required in order to support the conformance check of template arguments.
- Improved diagnostics.
- The compiler can now check consistency between "comments" and implementations.
- Signatures may be used to implement a form of function template partial specialization (through concept refinement).

Signatures Formalization

New keywords:

- `__signature__` to introduce a signature definition.

Signatures Formalization

New keywords:

- `__signature__` to introduce a signature definition.

Example:

```
__signature__ A { /* ... */ };
```

Signatures Formalization

New keywords:

- `__signature__` to introduce a signature definition.
- `__this_type__` (similar to the `this` keyword for classes) is an expression designating the actual type for which conformance check is being conducted.

Signatures Formalization

New keywords:

- `__signature__` to introduce a signature definition.
- `__this_type__` (similar to the `this` keyword for classes) is an expression designating the actual type for which conformance check is being conducted.

Example:

```
__signature__ DefaultConstructible {  
    __this_type__();  
};
```

Signatures Formalization

New keywords:

- `__signature__` to introduce a signature definition.
- `__this_type__` (similar to the `this` keyword for classes) is an expression designating the actual type for which conformance check is being conducted.

New semantics:

- `typename` is a (built-in) signature: is it the signature of all types, pretty much in the same way `void*` is the generic type of pointers to data.

Signatures Formalization (cont'd)

- Once defined, a signature may appear wherever a typename can be used to declare a type template-parameter

```
template<InputIterator In,  
        EqualityComparable T>  
In find(In first, In last, const T& t);
```

Signatures Formalization (cont'd)

- Once defined, a signature may appear wherever a typename can be used to declare a type template-parameter

```
template<InputIterator In,  
        EqualityComparable T>  
In find(In first, In last, const T& t);
```

- Concept refinement is implemented by signature inheritance

```
__signature__ UnaryPredicate  
                : UnaryFunction {  
    typename result_type = bool;  
};
```

Signatures Formalization (cont'd)

- It may not be necessary to prefix a dependent name with the `typename` keyword

Signatures Formalization (cont'd)

- It may not be necessary to prefix a dependent name with the `typename` keyword

```
__signature__ BinaryFunction {  
    typename first_argument_type;  
    typename second_argument_type;  
    typenme result_type;  
};
```

Signatures Formalization (cont'd)

- It may not be necessary to prefix a dependent name with the `typename` keyword

```
template<BinaryFunction Op>
class binder1st : UnaryFunction {
public:
    typedef Op::second_argument_type
        argument_type;
    typedef Op::result_type result_type;
    binder1st(const Op&,
              Op::first_argument_type&);
    result_type
    operator()(const argument_type&) const;
};
```

A guiding principle:

It should be possible to "retro-type" existing templates without changes.

A guiding principle:

It should be possible to "retro-type" existing templates without changes.

Possible conformance strategies:

- **named conformance:** require explicit conformance declaration before use in a context where constraints are enforced. Enables retargetting of types. Does not seem to support retro-typing.

A guiding principle:

It should be possible to "retro-type" existing templates without changes.

Possible conformance strategies:

- **named conformance:** require explicit conformance declaration before use in a context where constraints are enforced. Enables retargetting of types. Does not seem to support retro-typing.
- **structural conformance:** Deduction from the type **structure**. No conformance declaration required. Two different signatures may specify identical (or isomorphic) constraints.

Structural conformance

For each signature-member-specification s_{ms} in \mathcal{S} , the compiler looks for a corresponding valid expression for T :

- If s_{ms} names a type (or template) N , then the expression $T :: N$ should be valid; must match in kind.

Structural conformance

For each signature-member-specification s_{ms} in \mathcal{S} , the compiler looks for a corresponding valid expression for T :

- If s_{ms} names a type (or template) N , then the expression $T :: N$ should be valid; must match in kind.
- If s_{ms} declares a function-member fn , the compiler checks for the validity of the postfix expression $expr . fn (args)$; the compiler **synthetizes** $expr$ of type T and $args$ of appropriate types.

Structural conformance

For each signature-member-specification s_{ms} in \mathcal{S} , the compiler looks for a corresponding valid expression for T :

- If s_{ms} names a type (or template) N , then the expression $T :: N$ should be valid; must match in kind.
- If s_{ms} declares a function-member fn , the compiler checks for the validity of the postfix expression $expr.fn(args)$; the compiler **synthetizes** $expr$ of type T and $args$ of appropriate types.
- If s_{ms} declares a friend function, then the compiler looks for valid **free** function call with appropriate argument types.

Structural conformance

For each signature-member-specification sms in \mathcal{S} , the compiler looks for a corresponding valid expression for T :

- If sms names a type (or template) N , then the expression $T :: N$ should be valid; must match in kind.
- If sms declares a function-member fn , the compiler checks for the validity of the postfix expression $expr.fn(args)$; the compiler **synthetizes** $expr$ of type T and $args$ of appropriate types.
- If sms declares a friend function, then the compiler looks for valid **free** function call with appropriate argument types.

Access is checked in the context of the point of use.

Allocator Requirements

```
__signature__ Allocator {
    typename value_type;
    typename pointer = value_type*;
    typename const_pointer = const value_type*;
    typename reference = value_type&;
    typename const_reference = const value_type&;
    UnsignedIntegral size_type;
    SignedIntegral difference_type;
    template<typename> class rebind;
    Allocator rebind<typename>::other;

    __this_type__();
    __this_type__(rebind<typename>::other);
};
```

Allocator Requirements (cont'd)

```
void construct(pointer, value_type);
void destroy(pointer);

pointer address(reference);
const_pointer address(const_reference);
pointer allocate(size_type);
pointer allocate(size_type, const_pointer);
void deallocate(pointer, size_type);
size_type max_size();

friend bool
operator==( __this_type__, __this_type__ );
friend bool
operator!=( __this_type__, __this_type__ );
};
```

Parametrized signatures

How to account for different but similar signatures, conceptually parameterized? Jeremy Siek example: use of `std::complex<double>` as

- a real vector space (Euclidian inner product)
- a complex vector space (Hermitian inner product)

Parametrized signatures

How to account for different but similar signatures, conceptually parameterized? Jeremy Siek example: use of `std::complex<double>` as

- a real vector space (Euclidian inner product)
- a complex vector space (Hermitian inner product)

For parametrized signatures, then there ought to be a way of making them effective in typing templates:

- either, make them non-specializable;
- or require every specialization has the same structure as the primary template.

declaration:

block-declaration

function-definition

template-declaration

explicit-instantiation

explicit-specialization

linkage-specification

namespace-definition

signature-specifier

signature-specifier:

signature-head { signature-member-specification

Formal syntax (cont'd)

signature-head:

__signature__ identifier base-clause_{opt}

signature-member-specification:

signature-member-declaration signature-member

signature-member-declaration:

signature-name identifier;

decl-specifier-seq_{opt} signature-member-declar

signature-member-declarator:

declarator value-specifier_{opt}

Formal syntax (cont'd)

value-specifier
= *type-specifier*

signature-name:
typename
identifier

Acknowledgements

- Jeremy Siek and Fergus Henderson
- Mark Mitchell (<http://www.codesourcery.com>)
- Benjamin Kosnik (bkoz@redhat.com)