Generic Programming

Michael Hemmer

Tel Aviv University, Israel 🌿

Introduction to Generic Programming
March 14th, 2011
Outline

1. Generic Programming
   - Introduction and Motivation
   - Concepts and Models
   - Traits Classes
   - Dispatching

2. STL - Standard Template Library
   - Containers and Iterators
   - Function Pointer and Function Objects
Generic Programming Paradigm

Definition (Generic Programming)

A discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency.

[MS88]
Generic Programming Paradigm

**Definition (Generic Programming)**

A discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency.

[MS88]

**Translation:**

- You do not want to write the same algorithm again and again!
Generic Programming Paradigm

**Definition (Generic Programming)**

A discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency.

[MS88]

Translation:

- You do not want to write the same algorithm again and again!
- ⇒ You even want to make it independent from the used types.

See also: [http://en.wikipedia.org/wiki/Generic_programming](http://en.wikipedia.org/wiki/Generic_programming)
Motivation - Generic Programming

Reasons to write generic code

- Laziness
  - you don’t want write code again and again
  - code is maintained at one place
Motivation - Generic Programming

Reasons to write generic code

- **Laziness**
  - you don’t want write code again and again
  - code is maintained at one place

- **Flexibility**
  - others can use your code with their types
  - behavior of algorithm can be adjusted later
Outline

1 Generic Programming
   • Introduction and Motivation
   • Concepts and Models
   • Traits Classes
   • Dispatching

2 STL - Standard Template Library
   • Containers and Iterators
   • Function Pointer and Function Objects
Trivial Example: `swap`

```
template <typename T> void swap(T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}
```

- announce that you write a template
- name the template arguments
- write template as if arguments where known types
Concept and Model (brief introduction)

```cpp
template <typename T> void swap(T& a, T& b)
{
    T tmp(a); a = b; b = tmp;
}
```

This code implies certain requirements on \( T \).
template<typename T> void swap(T& a, T& b) {
    T tmp(a); a = b; b = tmp;
}

This code implies certain requirements on T
   • T must have a copy constructor
Concept and Model (brief introduction)

```cpp
template <typename T> void swap(T& a, T& b) {
    T tmp(a); a = b; b = tmp;
}
```

This code implies certain requirements on `T`
- `T` must have a copy constructor
- `T` must have an assignment operator
template <typename T> void swap(T& a, T& b) 
{
    T tmp(a);  a = b;  b = tmp;
}

This code implies certain requirements on T

- T must have a copy constructor that copies
- T must have an assignment operator that assigns
template <typename T> void swap(T& a, T& b) {
    T tmp(a); a = b; b = tmp;
}

This code implies certain requirements on T

- T must have a copy constructor that copies
- T must have an assignment operator that assigns

Or in formal words:

- T must be a Model of the Concept CopyConstructible
- T must be a Model of the Concept Assignable
Concept

A concept is a set of certain requirements.

Four principle categories:
A concept is a set of certain requirements.

Four principle categories:

- Valid Expressions
  - compiling C++ expressions involving the template argument
Concept

A concept is a set of certain requirements.

Four principle categories:

- **Valid Expressions**
  - compiling C++ expressions involving the template argument

- **Associated Types**
  - types that are related to the model as they participate in one or more of the valid expressions
  - e.g. the return type of some required member function
A concept is a set of certain requirements.

Four principle categories:

- **Valid Expressions**
  - compiling C++ expressions involving the template argument

- **Associated Types**
  - types that are related to the model as they participate in one or more of the valid expressions
  - e.g. the return type of some required member function

- **Run-time Characteristics**
  - semantic guarantees, e.g., pre/post-conditions
A concept is a set of certain requirements.

Four principle categories:

- **Valid Expressions**
  - compiling C++ expressions involving the template argument

- **Associated Types**
  - types that are related to the model as they participate in one or more of the valid expressions
  - e.g. the return type of some required member function

- **Run-time Characteristics**
  - semantic guarantees, e.g., pre/post-conditions

- **Complexity Guarantees**
  - maximum limits on execution time or required resources
Outline

1. Generic Programming
   - Introduction and Motivation
   - Concepts and Models
   - Traits Classes
   - Dispatching

2. STL - Standard Template Library
   - Containers and Iterators
   - Function Pointer and Function Objects
Example: normalize_rational(I)

Code for leda::rational

```cpp
void normalize_rational(leda::rational& x) {
    leda::integer num = x.numerator();
    leda::integer denom = x.denominator();
    leda::integer g = leda::gcd(num, denom);
    x = leda::rational( num/g, denom/g );
}
```
Example: \texttt{normalize\_rational()}(l)

- **Code for \texttt{leda::rational}**

```cpp
void normalize_rational(leda::rational& x) {
    leda::integer num   = x.numerator();
    leda::integer denom = x.denominator();
    leda::integer g     = leda::gcd(num, denom);
    x = leda::rational( num/g , denom/g );
}
```

- **Code for \texttt{CORE::BigRat}**

```cpp
void normalize_rational(CORE::BigRat& x) {
    CORE::BigInt num   = CORE::numerator(x);
    CORE::BigInt denom = CORE::denominator(x);
    CORE::BigInt g     = CORE::gcd(num, denom);
    x = CORE::BigRat( num/g , denom/g );
}
```
Example: \texttt{normalize\_rational (I)}

Algorithm is actually clear:

- Take numerator and denominator.
- Compute the greatest common divisor (gcd).
- Divide both by the gcd and construct the new rational.
Example: `normalize_rational` (II)

Algorithm is actually clear:

- Take numerator and denominator.
- Compute the greatest common divisor (gcd).
- Divide both by the gcd and construct the new rational.

Problems for possible template code:

- **Types** are external and have different interface.
- How to get the type for integer? (associated type)
Example: `normalize_rational(II)`

Algorithm is actually clear:

- Take numerator and denominator.
- Compute the greatest common divisor (gcd).
- Divide both by the gcd and construct the new rational.

Problems for possible template code:

- **Types** are external and have different interface.
- How to get the type for integer? (associated type)

Solution: Unify interface by using a **traits** class that you control!
**Example: Rational_traits (I)**

- a template version using a traits class:

```
template<class Rational>
void normalize_rational(Rational& x) {
    typedef Rational_traits<Rational> Traits;
    typedef typename Traits::Integer Integer;
    Integer num = Traits::numerator(x);
    Integer denom = Traits::denominator(x);
    Integer g = gcd(num, denom);
    x = Traits::make_rational( num/g , denom/g );
}
```

- use traits to get the integer type
- use traits to provide a unified interface
Example: `Rational_traits(II)`

```cpp
template <typename T> struct Rational_traits{}; // default

// full specialization for leda::rational
template<> struct Rational_traits<leda::rational> {

};

// full specialization for CORE::BigRat
template<> struct Rational_traits<CORE::BigRat> {

};
```
Example: Rational_traits (II)

```cpp
template <typename T> struct Rational_traits{}; // default

// full specialization for leda::rational
template<> struct Rational_traits<leda::rational> {  
  typedef leda::integer Integer;
};

// full specialization for CORE::BigRat
template<> struct Rational_traits<CORE::BigRat> {  
  typedef CORE::BigInt Integer;
};
```
Example: Rational_traits(II)

template <typename T> struct Rational_traits{}; // default

// full specialization for leda::rational
template<> struct Rational_traits<leda::rational> {  
  typedef leda::integer Integer;

  static Integer numerator(const Rational& x){
  }
};

// full specialization for CORE::BigRat
template<> struct Rational_traits<CORE::BigRat> {  
  typedef CORE::BigInt Integer;

  static Integer numerator(const Rational& x){
  }
};
Example: `Rational_traits(II)`

```cpp
template <typename T> struct Rational_traits{}; // default

// full specialization for leda::rational
template<> struct Rational_traits<leda::rational> { 
    typedef leda::integer Integer;

    static Integer numerator(const Rational& x){
        return x.numerator();
    }
};

// full specialization for CORE::BigRat
template<> struct Rational_traits<CORE::BigRat> { 
    typedef CORE::BigInt Integer;

    static Integer numerator(const Rational& x){
        return CORE::numerator(x);
    }
};
```
template <typename T> struct Rational_traits{}; // default

// full specialization for leda::rational
template<> struct Rational_traits<leda::rational> {  
    typedef leda::integer Integer;

    static Integer numerator(const Rational& x){  
        return x.numerator();  
    }
    // similar for denominator, make rational
};

// full specialization for CORE::BigRat
template<> struct Rational_traits<CORE::BigRat> {  
    typedef CORE::BigInt Integer;

    static Integer numerator(const Rational& x){  
        return CORE::numerator(x);  
    }
    // similar for denominator, make_rational
};
Traits class

Original definition:

- **A traits class is a class template** that associates information to compile-time entity, e.g., associated types or constants

```cpp
typedef typename Traits::Integer Integer;
```
Traits class

Original definition:

- A traits class is a class template that associates information to compile-time entity, e.g., associated types or constants

```cpp
typedef typename Traits::Integer Integer;
```

- may also provide unified interface

```cpp
Integer num = Traits::numerator(x);
```
Traits class

Original definition:

- A traits class is a class template that associates information to compile-time entity, e.g., associated types or constants

```cpp
typedef typename Traits::Integer Integer;
```

- may also provide unified interface

```cpp
Integer num = Traits::numerator(x);
```

Sometimes the traits class is already used as the template argument:

- Advantage:
  provides all types and functionality the algorithm needs
  ⇒ usually only one template argument

- Disadvantage:
  this overload of the term 'traits' may causes confusion
Outline

1. Generic Programming
   - Introduction and Motivation
   - Concepts and Models
   - Traits Classes
   - Dispatching

2. STL - Standard Template Library
   - Containers and Iterators
   - Function Pointer and Function Objects
Goal:
- Provide a function \texttt{simplify} that is valid for any type.
- If the argument is a rational it should normalize it.
- Otherwise it should do nothing.

Problem:
- \texttt{normalize\_rational} can only be instantiated for rationals.
- This decision must be taken \textit{at compile time}.
First a small modification to `Rational_traits`:

```cpp
struct Tag_false{}; // just two different classes
struct Tag_true{};

template<typename T> struct Rational_traits{ // default
    typedef Tag_false Is_rational;
};

template<> struct Rational_traits<leda::rational> {  
    typedef Tag_true Is_rational;
    ...
}; // the remaining definitions

template<> struct Rational_traits<CORE::BigRat> {  
    typedef Tag_true Is_rational;
    ...
}; // the remaining definitions
```
Dispatching using Tags (III)

Simplify some number:

```cpp
template <class T> inline
void simplify_(T& x, Tag_false){}

template <class T> inline
void simplify_(T& x, Tag_true){normalize_rational(x);}

template <class T> inline
void simplify(T& x){
    typedef Rational_traits<T> Traits;
    typedef typename Traits::Is_rational Is_rational;
    simplify_(x, Is_rational());
}
```
Outline

1. Generic Programming
   - Introduction and Motivation
   - Concepts and Models
   - Traits Classes
   - Dispatching

2. STL - Standard Template Library
   - Containers and Iterators
   - Function Pointer and Function Objects
maximal_element (array version)

Compute the maximal element of an array:

```cpp
template <typename T>
T maximal_element(T* elements, int size)
{
    T result = elements[0];
    for(int i = 1; i < size; i++){
        if (result < elements[i])
            result = elements[i];
    }
    return result;
}

// usage:
int main()
{
    std::cout << maximal_element(A,3) << std::endl;
}
```
The `std::vector` Container (the better array)

- part of the standard template library
  - `#include <vector>`
  - `std::vector<int> my_integers;`
The `std::vector` Container (the better array)

- part of the standard template library
  - `#include <vector>`
  - `std::vector<int> my_integers;`

- Behaves like an array:
  - supports random access via `operator[]`
The `std::vector` Container (the better array)

- part of the standard template library
  - `#include <vector>`
  - `std::vector<int> my_integers;`

- Behaves like an array:
  - supports random access via `operator[]`

- Can grow itself as necessary:
  - Supports `push_back()`, that is, it pushes an element onto the end of the vector (growing it by one).
The `std::vector` Container (the better array)

- part of the standard template library
  - `#include <vector>`
  - `std::vector<int> my_integers;`
- Behaves like an array:
  - supports random access via `operator[]`
- Can grow itself as necessary:
  - Supports `push_back()`, that is, it pushes an element onto the end of the vector (growing it by one).
- Can be asked how many elements it has with `size()`.
The std::vector Container (the better array)

- part of the standard template library
  - #include <vector>
  - std::vector<int> my_integers;

- Behaves like an array:
  - supports random access via operator[]

- Can grow itself as necessary:
  - Supports push_back(), that is, it pushes an element onto the end of the vector (growing it by one).

- Can be asked how many elements it has with size().

- For more details on vector see also: http://www.sgi.com/tech/stl/Vector.html
maximal_element (std::vector version)

Compute the maximal element of an std::vector:

```cpp
template <typename T>
T maximal_element(std::vector<T> elements){ // size not given
    T result = elements[0];
    for(int i = 1; i<elements.size(); i++) { // query for size
        if (result < elements[i])
            result = elements[i];
    }
    return result;
}

// usage:
int main(){
    std::vector<int> V; // no fixed size
    V.push_back(5); // can grow as needed
    V.push_back(7);
    V.push_back(1);
    std::cout << maximal_element(V) << std::endl;
}
```
Containers and Iterators

- So far we have two versions for `maximal_element`:
  - for array
  - for `std::vector`
Containers and Iterators

- So far we have two versions for \texttt{maximal\_element}:
  - \texttt{for array}
  - \texttt{for std::vector}

- The STL actually provides more container
  - \texttt{std::list} // connected via pointers
  - \texttt{std::set} // stores elements only once
  - ...
  - see also: \url{http://www.sgi.com/tech/stl/stl_index_cat.html}
Containers and Iterators

So far we have two versions for maximal_element:
- for array
- for std::vector

The STL actually provides more container
- std::list // connected via pointers
- std::set // stores elements only once
- ...
- see also: http://www.sgi.com/tech/stl/stl_index_cat.html

Can we provide one version of maximal_element for all?
So far we have two versions for \texttt{maximal\_element}:
- \texttt{for} array
- \texttt{for} \texttt{std::vector}

The STL actually provides more container
- \texttt{std::list} // connected via pointers
- \texttt{std::set} // stores elements only once
- ...
- see also: http://www.sgi.com/tech/stl/stl\_index\_cat.html

Can we provide one version of \texttt{maximal\_element} for all? Yes, using Iterators!
Iterators

- Are a generalization of pointers, i.e., pointers are iterators.
- Provide a way of specifying a position in a container.
- Every container \( c \) provides \( c\text{.begin()} \) and \( c\text{.end()} \)
  - \( c\text{.begin()} \) points to the 'first' element in the container
  - \( c\text{.end()} \) points to 'past-the-end' position

\[
\begin{array}{cccccccc}
T & T & T & T & T & T & T & T \\
\end{array}
\]
Iterators

- Are a generalization of pointers, i.e., pointers are iterators.
- Provide a way of specifying a position in a container.
- Every container \( c \) provides \( c\text{.begin()} \) and \( c\text{.end()} \)
  
  - \( c\text{.begin()} \) points to the ‘first’ element in the container
  - \( c\text{.end()} \) points to ‘past-the-end’ position

\[
\begin{align*}
&c\text{.begin()} &c\text{.end()} \\
\text{T} &\text{T} &\text{T} &\text{T} &\text{T} &\text{T} &\text{T} &\text{T}
\end{align*}
\]

- Like pointers iterators can be
  
  - \(*it\) - dereferenced
  - \(it1 == it2\) - equality compared
  - \(it++\) - incremented (not all)
  - \(it--\) - decremented (not all)
  - \(it[i]\) - indexed (not all)
maximal_element (pointer version)

Compute the maximal element of a pointer range

```cpp
template <typename T>
T* maximal_element(T* begin, T* end){
    T* result = begin;
    for(T* ptr = begin; ptr != end; ptr++){
        if (*result < *ptr)
            result = ptr;
    }
    return result;
}
```

// usage:
int main(){
    std::cout << *(maximal_element(A,A+3)) << std::endl;
}
maximal_element (iterator version)

Compute the maximal element of a pointer range

```cpp
template <typename Iterator>
Iterator maximal_element(Iterator begin, Iterator end){
    Iterator result = begin;
    for(Iterator it = begin; it != end; it++)
    {
        if (*result < *it)
            result = it;
    }
    return result;
}

// usage:
int main(){
    std::vector<int> V(A,A+3); // construction from 'iterator' range
    std::cout << *(maximal_element(A,A+3)) << std::endl;
    std::cout << *(maximal_element(V.begin(),V.end())) << std::endl;
}
```

see also: http://www.sgi.com/tech/stl/max_element.html
Iterators (II)

- using iterators abstracts from the used container
  ⇒ results in the most flexible code
Iterators (II)

- using iterators abstracts from the used container ⇒ results in the most flexible code
- However, iterators have different capabilities, for instance, an iterator of a list has no `operator[]` ⇒ STL introduces a hierarchy of iterator concepts
Iterators (II)

- using iterators abstracts from the used container ⇒ results in the most flexible code
- However, iterators have different capabilities, for instance, an iterator of a list has no `operator[]` ⇒ STL introduces a hierarchy of iterator concepts
  - Trivial Iterator // just dereferencable and comparable
Iterators (II)

- using iterators abstracts from the used container ⇒ results in the most flexible code

- However, iterators have different capabilities, for instance, an iterator of a list has no `operator[]` ⇒ STL introduces a hierarchy of iterator concepts
  - Trivial Iterator // just dereferencable and comparable
  - Input Iterator // incrementable, read only
Iterators (II)

- using iterators abstracts from the used container
  ⇒ results in the most flexible code

- However, iterators have different capabilities,
  for instance, an iterator of a list has no operator[]
  ⇒ STL introduces a hierarchy of iterator concepts
  - Trivial Iterator // just dereferencable and comparable
  - Input Iterator // incrementable, read only
  - Output Iterator // incrementable, write only
Iterators (II)

- using iterators abstracts from the used container ⇒ results in the most flexible code
- However, iterators have different capabilities, for instance, an iterator of a list has no `operator[]`
  ⇒ STL introduces a hierarchy of iterator concepts
    - Trivial Iterator // just dereferencable and comparable
    - Input Iterator // incrementable, read only
    - Output Iterator // incrementable, write only
    - Forward Iterator // incrementable with read and write
Iterators (II)

- using iterators abstracts from the used container ⇒ results in the most flexible code
- However, iterators have different capabilities, for instance, an iterator of a list has no `operator[]`
  ⇒ STL introduces a hierarchy of iterator concepts
  - Trivial Iterator  // just dereferencable and comparable
  - Input Iterator  // incrementable, read only
  - Output Iterator  // incrementable, write only
  - Forward Iterator  // incrementable with read and write
  - Bidirectional Iterator  // also decrementable
Iterators (II)

- using iterators abstracts from the used container ⇒ results in the most flexible code
- However, iterators have different capabilities, for instance, an iterator of a list has no `operator[]` ⇒ STL introduces a hierarchy of iterator concepts
  - Trivial Iterator  // just dereferencable and comparable
  - Input Iterator  // incrementable, read only
  - Output Iterator  // incrementable, write only
  - Forward Iterator  // incrementable with read and write
  - Bidirectional Iterator  // also decrementable
  - Random Access Iterator  // operator[]
Iterators (II)

- using iterators abstracts from the used container ⇒ results in the most flexible code
- However, iterators have different capabilities, for instance, an iterator of a list has no `operator[]` ⇒ STL introduces a hierarchy of iterator concepts
  - Trivial Iterator // just dereferencable and comparable
  - Input Iterator // incrementable, read only
  - Output Iterator // incrementable, write only
  - Forward Iterator // incrementable with read and write
  - Bidirectional Iterator // also decrementable // operator[]
  - Random Access Iterator

- There is `std::iterator_traits<Iterator>` providing `iterator_category` tag for dispatching.

see also: [http://www.sgi.com/tech/stl/Iterators.html](http://www.sgi.com/tech/stl/Iterators.html)
Outline

1. Generic Programming
   - Introduction and Motivation
   - Concepts and Models
   - Traits Classes
   - Dispatching

2. STL - Standard Template Library
   - Containers and Iterators
   - Function Pointer and Function Objects
Function Pointers

- Even more flexibility using function pointers:

```cpp
template <typename Iterator, typename Comp>
Iterator maximal_element(Iterator begin, Iterator end, Comp comp){
    Iterator result = begin;
    for(Iterator it = begin; it != end; it++){
        if (comp(*result,*it))
            result = it;
    }
    return result;
}

bool less(const int& a, const int& b){return a<b;}
bool larger(const int& a, const int& b){return a>b;}

int main(){
    *(maximal_element(A,A+3, *less)) // returns 7
    *(maximal_element(A,A+3, *larger))  // returns 1
}
```
**Function Objects**

- Even more flexibility using function objects:

```cpp
.. // definition of maximal_element as before

template <typename T> struct Compare { // function object class

    bool m_less; // carries a state
    Compare(bool less = true):m_less(less){}

    // make it look like a function
    bool operator() (const T& a, const T& b) {
        if (m_less) return a < b;
        else return b < a;
    }
};

int main(){
    *(maximal_element(A, A+3, Compare<int>(true)))  // returns 7
    *(maximal_element(A, A+3, Compare<int>(false))) // returns 1
}
```

- More flexibility due to carried state.
Generic Programming Paradigm

Definition (Generic Programming)
A discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency.

[MS88]
Generic Programming & the STL Bibliography

Andrei Alexandrescu.
*Modern C++ Design: Generic Programming And Design Patterns Applied.*
Addison-Wesley, Boston, MA, USA, 2001.

Matthew H. Austern.
*Generic Programming and the STL.*
Addison-Wesley, Boston, MA, USA, 1999.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides.
*Design Patterns — Elements of Reusable Object-Oriented Software.*
Addison-Wesley, Boston, MA, USA, 1995.

David R. Musser, Gillmer J. Derge, and Atul Saini.
*STL tutorial and reference guide: C++ programming with the standard template library.*

David Vandevoorde and Nicolai M. Josuttis.
*C++ Templates: The Complete Guide,*
Addison-Wesley, Boston, MA, USA, 2002.

Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine.
*The BOOST Graph Library,*
Addison-Wesley, Boston, MA, USA, 2002.

David A. Musser and Alexander A. Stepanov.
Generic programming.
The planned new standard for the C++ programming language.
http://en.wikipedia.org/wiki/C++0x#References.

The SGI STL.