Software↵
Languages.Lab

# Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

Vrije Universiteit Brussel - Faculty of Science and Bio-Engineering Sciences - Computer Science Department - Pleinlaan 2 - 1050 Brussels - Belgium

Software↵
Languages.Lab

# Chapter 6 - Generic Programming using the STL

Structuur van Computerprogramma's 2

## Generic Programming?

*“**Generic Programming** is a technique where one implements the essence of an algorithm, abstracting from the data types on which it operates by a set of requirements for such data types.”*

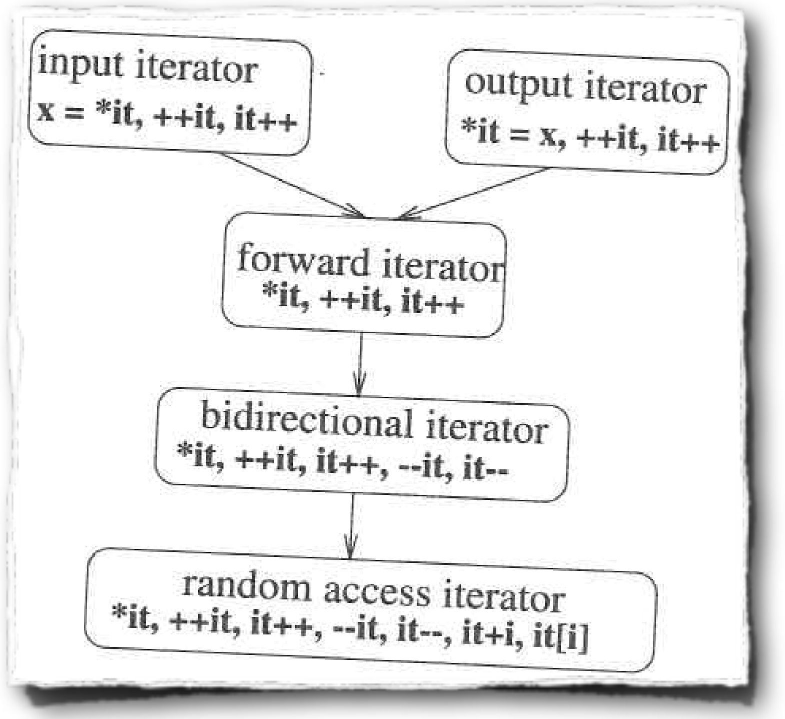
*“Applying generic programming to container types and associated algorithms leads to the introduction of so-called **iterators**.”*

*“An **iterator** abstracts from a particular container type.”*

Software Languages Lab

# Kinds of Iterators

- STL considers 5 kinds of iterators with increasing functionality:
  - **input iterator**: read-only access
    `x = *it    it++      ++it`
  - **output iterator**: write-only access
    `*it = x    it++      ++it`
  - **forward iterator**: read-write access
    `it++      ++it`
  - **bidirectional iterator**: read-write access
    `it++      ++it      it--      --it`
  - **random access iterator**: read-write access
    `it++      ++it      it--      --it      it[4]      it + 14`



(`it` is an example iterator object)

Software Languages Lab

# Algorithms and Iterator Kinds

- Some algorithms require a certain kind of iterator
  - What kind is required by **`find()`** ?
- Pointers are random access iterators !

```
extern std::string a[];
extern unsigned int a_size;

find(a, a + a_size, "abc");
```

## Example Algorithm: sum

```
template<typename InputIterator, typename T>
T sum(InputIterator first, InputIterator last) {
  T result(*first++); // assume first!=last !!
  while (first != last)
     result += *first++;
  return result;
}
```

Problem:

```
extern Node<int>* l;
typedef Node<int>::Cursor list_iterator;

// error; implicit template instantiation, but
// compiler cannot deduce T=int
sum(l->begin(), l->end());

// ok
sum<list_iterator, int>(l->begin(), l->end());
```

The template argument deduction procedure does not take return types into account !

Software Languages Lab

```
template<typename InputIterator, typename T>
T sum(InputIterator first, InputIterator last, T& result) {
   if (first == last)
      return result;
   do
      result += *first++;
   while (first != last);
   return result;
}

int r;
extern Node<int>* l;

sum(l->begin(), l->end(), r); // ok
```

This works, but it is not a clean solution:

- the algorithm designer is forced to include a **T** argument
- the type parameter **T** is actually since it is determined by the **InputIterator**

# Associating Types with Iterators

```
template<typename T>
class Node {
public:
   // ...
   class Cursor {
   public:
      // Node::Cursor::value_type is type of *Cursor
      typedef T value_type;
      Cursor(Node* node = 0) : node_(node) { }
      const T& operator*();
      Cursor& operator++();
      bool operator!=(const Cursor& c) const;
   private:
      Node* node_;
   };
   Cursor begin() const { return Cursor(this); }
   Cursor end() const { return Cursor(); }
private:
   // ...
};
```

## Sum with only 1 (deducible) Template Parameter

```
template<typename InputIterator>
typename InputIterator::value_type // what is ‘‘typename’’?
sum(InputIterator first, InputIterator last) {
   assert(first != last);
   typename InputIterator::value_type result(*first++);
   while (first != last)
      result += *first++;
   return result;
}

extern Node<int>* list;
// ok
// InputIterator = Node<int>::Cursor &&
// InputIterator::value_type
// = Node<int>::Cursor::value_type
// = int
sum(list->begin(), list->end());

// BUT:
extern int a[10];
sum(a, a+10); // error! why?
```

## Iterator Types Problem

**Q:** Iterators may not be classes: e.g. what is `value_type` for a pointer type?

**A:** Use a compile-time function to compute `T::value_type` from `T`. This can be done using metaprogramming

## Iterator Traits

```
template<typename Iter>
// default; ok if Iter is a class type
struct iterator_traits {
   typedef typename Iter::iterator_category iterator_categ
   typedef typename Iter::value_type value_type;
   typedef typename Iter::difference_type difference_type;
   typedef typename Iter::pointer pointer;
   typedef typename Iter::reference reference;
};
```

defines a compile-time "function"

*Iterator* → **iterator_traits**<*Iterator*>

returning e.g., `iterator_traits<Iterator>::value_type`, the type of `*i` where `i` is of type `Iterator`

## Iterator Categories

Found by **`iterator_traits<Iterator>::iterator_category`**, which indicates the kind of iterator

```
struct input_iterator_tag { };
struct output_iterator_tag { };

// inheritance: see later
struct forward_iterator_tag:
   public input_iterator_tag { };

struct bidirectional_iterator_tag:
   public forward_iterator_tag { };

struct random_access_iterator_tag:
   public bidirectional_iterator_tag { };
```

## Iterator Category of Cursor

```
template<typename T>
class Node {
public:
  // ...
  class Cursor {
  public:
    // type of *Cursor
    typedef T value_type;
    // Cursor is a forward iterator
    typedef forward_iterator_tag iterator_category;
    Cursor(Node* node = 0);
    const T& operator*() const;
    Cursor& operator++();
    bool operator!=(const Cursor& c) const;
  private:
    Node* node_;
  };
private:
  // ...
};
```

Software Languages Lab

## Other Iterator Traits Components

`iterator_traits<Iterator>::value_type`

- ➠ type of `*i` with `Iterator i`

`iterator_traits<Iterator>::category`

- ➠ iterator kind of `Iterator`

`iterator_traits<Iterator>::reference`

- ➠ usually `Iterator::value_type&`

`iterator_traits<Iterator>::pointer`

- ➠ `&Iterator::reference`

`iterator_traits<Iterator>::difference_type`

- ➠ distance (`ptrdiff_t`)

## Specializing Iterator Traits for Pointer Types

```
template<typename T>
// iterator_traits specialization for pointer types
struct iterator_traits<T*> {
   typedef random_access_iterator_tag iterator_category;
   typedef T value_type;
   typedef ptrdiff_t difference_type;
   typedef T* pointer;
   typedef T& reference;
};
```

## Sum with Iterator Traits

```
template<typename InputIt>
typename iterator_traits<InputIt>::value_type
sum(InputIt first, InputIt last) {
   assert(first != last);
   typename
      iterator_traits<InputIt>::value_type result(*first++);
   while (first != last)
      result += *first++;
   return result;
}

int a[10];
sum(a, a+10); // ok; why?
```

## Dispatching on Iterator Category I

```
template<typename InputIt>
inline void // version for input and forward iterators
advance(
     InputIt& i,
     typename iterator_traits<InputIt>::difference_type n,
     input_iterator_tag) {
   for (; n > 0; --n) ++i;
}

template<typename BidirectIt>
inline void // version for bidirectional iterator
advance(
     BidirectIt& i,
     typename iterator_traits<BidirectIt>::difference_type n,
     bidirectional_iterator_tag) {
   if (n >= 0)
      for (; n > 0; --n) ++i;
   else
      for (; n < 0; ++n) --i;
}
```

## Dispatching on Iterator Category 2

```
template<typename RandomIt>
inline void // version for random access iterators
advance(
    RandomIt& i,
    typename iterator_traits<RandomIt>::difference_type n,
    random_access_iterator_tag) {
   i += n;
}

// the general version of advance dispatches to a more
// specialized one, depending on the iterator kind
template<typename Iterator>
inline void
advance(
    Iterator& i,
    typename iterator_traits<Iterator>::difference_type n) {
        advance(
            i,
            n,
            typename iterator_traits<Iterator>::iterator_category());
}
```

# Advance Call Resolution

```
// I is the iterator type
advance(I& i, typename iterator_traits<I>::difference_type n)
```

calls an overloaded function with an extra `I::Iterator_category` argument

```
advance(i, n,
      typename iterator_traits<I>::iterator_category())
```

which will resolve, depending on the type of

```
iterator_traits<I>::iterator_category()
```

to the most efficient implementation.
For example:

```
int* a;
advance(a,10); // a+=10
```

will eventually resolve (explain!) to `a += 10`

## Example Containers

All in namespace `std`

- pair (of values)
- map (key `->` value)
- set (key `->` bool)
- (vector, list, dequeue, multimap, multiset)
- (adaptors: stack, queue, priority queue)
- (hash_map, hash_set)

## Pair

```
template <typename T1, typename T2>
struct pair {
   typedef T1 first_type;
   typedef T2 second_type;
   T1 first;
   T2 second;
   pair() : first(T1()), second(T2()) { }
   pair(const T1& a, const T2& b) : first(a), second(b) { }
};
```

## Pair

```
template<typename T1, typename T2>
inline bool
operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
   return x.first == y.first && x.second == y.second;
}

template<typename T1, typename T2>
inline bool
operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
   return x.first < y.first
         || (!(y.first < x.first) && x.second < y.second);
}

// easier to type, e.g. make_pair(20, "abc")
template<typename T1, typename T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
   return pair<T1, T2> (x, y);
}
```

Software
Languages.Lab

# The "less" Function Template Class

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

useful, e.g. as template parameter for sort:

```
template<typename RandomAccessIt,
        typename StrictWeakOrdering>
void
sort(
   RandomAccessIt first,
   RandomAccessIt last,
   StrictWeakOrdering comp =
      less<iterator_traits<RandomAccessIt>::value_type> ()
);
```

## Example Container: Map

```
// a map implements a [Key->T] finite function
template <typename Key,
          typename T,
          typename Compare=less<Key>,
          typename Alloc = alloc>
class map { // usually reptype is red-black tree
public:
   typedef rep_type::iterator iterator;
   typedef ... const_iterator;
   // type of *iterator is <const Key,T> pair
   typedef pair<const Key, T> value_type;
   map(); // constructor
   // put [*first,..,*last[ in a map
   template<class InputIt> map(InputIt first, InputIt last);
   iterator begin();
   const_iterator begin() const;
   iterator end();
   const_iterator end() const;
//...
```

# Example Container: Map

```
//...
   size_type size() const;
   iterator find(const Key& x); // end() if not found
   const_iterator find(const Key& x) const;
   size_type count(const Key& x);
   // result.first = where inserted,
   // result.second = true iff ok
   pair<iterator,bool> insert(const value_type& x);
   T& operator[](const Key& k); // can be assigned to
   template <class InputIt>
        void insert(InputIt first, InputIt last);
   void erase(iterator position);
   // find and erase, return 1 iff ok
   size_type erase(const Key& x);
   // erase [*first .. *last[
   void erase(iterator first, iterator last);
   void clear(); // erase [begin(),..,end()[
};
```

# Map Usage Example

```
typedef std::map<std::string, int> Examen;

int main(int, char**) {
   Examen scores;
   // insert
   scores["john"] = 18; // std::string::string(const char*)
   scores.insert(std::make_pair("fred", 5));
   // retrieve
   for (Examen::const_iterator i=scores.begin();
        i!=scores.end(); ++i)
      std::cout << (*i).first << ": " << (*i).second << std::endl;
   // update
   scores["fred"] = 11; // 2de zittijd
   Examen::iterator i = scores.find("john");
   if (i != scores.end()) {
      (*i).second = 13; // another way to update
      std::cout << (*i).first << ": " << (*i).second << std::endl;
   }
   return 0;
}
```

# Set (Specialization of Map)

```
// elements are kept in sorted order
template <typename Key,
          typename Compare = less<Key>,
          typename Alloc = alloc>
class set {
public:
   typedef Key value_type;
   // there are only "constant" iterators. why?
   typedef rep_type::const_iterator iterator;
   set(); // constructor
   set(const set<Key, Compare, Alloc>& x);
   // create set from [*first,..,*last[
   template<class InputIt> set(InputIt first, InputIt last);
   iterator begin();
   iterator end();

   //...
```

## Set

```
  //...

  size_type size(); // cardinality
  iterator find(const Key& x); // end() if not found
  size_type count(const Key& x); // 1 or 0
  // result.first = where inserted,
  // result.second = true iff ok
  pair<iterator, bool> insert(const value_type& x);
  void erase(iterator position); // erase at position
  size_type erase(const Key& x); // 1 if ok, 0 if not
  // erase [*first,..,*last[
  void erase(iterator first, iterator last);
  void clear(); // erase(begin(),end());
};
```

## for_each, find, find_if Algorithms

```
template<typename InputIt, typename Function>
Function for_each(InputIt first, InputIt last, Function f) {
   for (; first != last; ++first)
      f(*first);
   return f;
}

template<typename InputIt, typename T>
InputIt find(InputIt first, InputIt last, const T& value) {
   while (first != last && *first != value)
      ++first;
   return first;
}

template<typename InputIt, typename Predicate>
InputIt find_if(InputIt first, InputIt last, Predicate pred) {
   while (first != last && !pred(*first))
      ++first;
   return first;
}
```

# The copy Algorithm

```
// code is a simplification of the real thing
template<typename InputIt,
         typename OutputIt>
inline OutputIt
copy(InputIt first, InputIt last, OutputIt result) {
  for (; first != last; ++result, ++first)
    *result = *first;
  return result;
}
```

- How to copy to e.g., a set container?
- What if the receiver is a vector that is “too small”?

## insert_iterator I

```
// an iterator that translates *it = v to a container
// insert operation e.g.
// copy(c1.begin(), c1.end(), inserter(c2,c2.begin()))
// will work fine
template<typename Cont>
class insert_iterator {
protected:
   Cont* container;
   typename Cont::iterator iter;
public:
   insert_iterator(Cont& x, typename Cont::iterator i) :
      container(&x), iter(i) { }
   insert_iterator<Cont>& operator=(
         const typename Cont::value_type& value) {
      iter = container->insert(iter, value);
      ++iter;
      return *this;
   }
   //...
```

## insert_iterator I

```
   //...
   // the next member functions do nothing
   insert_iterator<Cont>& operator*() { return *this; }
   insert_iterator<Cont>& operator++() { return *this; }
   insert_iterator<Cont>& operator++(int) { return *this; }
};


// this function makes it easy to use an insert iterator
template<typename Cont, typename Iterator>
inline insert_iterator<Cont>
inserter(Cont& x, Iterator i) {
   typedef typename Cont::iterator iter;
   return insert_iterator<Cont> (x, iter(i));
}
```

## Other Iterator Adaptors

Other `iterator adaptors` are available: e.g.,

- `back_insert_iterator` (push_back)
- `front_insert_iterator` (push_front)
- stream iterators

# istream_iterator

```
// input iterator that reads values from a stream
template<typename T, typename Distance = ptrdiff_t>
class istream_iterator {
   friend bool operator==(
        const istream_iterator<T, Distance>& x,
        const istream_iterator<T, Distance>& y);
protected:
   // e.g. value = *iter++; will read value from stream
   istream* stream; // from which data is read
   T value; // last read from stream
   bool can_read; // true iff not yet at end
   void read() {
      can_read = (stream && *stream) ? true : false;
      if (can_read)
         *stream >> value;
      can_read = (*stream) ? true : false;
   }
   //...
```

# istream_iterator

```
   //...
public:
   typedef T value_type; // for iterator_traits
   typedef const T* pointer; // for iterator_traits
   typedef const T& reference; // for iterator_traits
   istream_iterator() :
      stream(0), can_read(false) { }
   istream_iterator(istream& s) : stream(&s) { read(); }
   reference operator*() const { return value; }
   pointer operator->() const { return &(operator*()); }
   istream_iterator<T, Distance>& operator++() {
      read();
      return *this;
   }
   istream_iterator<T, Distance> operator++(int) {
      istream_iterator<T, Distance> tmp = *this;
      read();
      return tmp;
   }
};
```

Software Languages Lab

# ostream_iterator

```
// an output iterator that writes to a stream
// e.g. *it = value will write value on the stream of it
template<typename T>
class ostream_iterator {
protected:
   ostream* stream;
   const char* string; // what is this used for?
public:
   typedef void value_type; // ...
   ostream_iterator(ostream& s) : stream(&s), string(0) { }
   ostream_iterator(ostream& s, const char* c) : stream(&s),{ }
   ostream_iterator<T>& operator=(const T& value) {
      *stream << value;
      if (string)
         *stream << string;
      return *this;
   }
   ostream_iterator<T>& operator*() { return *this; }
   ostream_iterator<T>& operator++() { return *this; }
   ostream_iterator<T>& operator++(int) { return *this; }
};
```