Iterators

Object-Oriented Programming with C++

Iterators

- Provide a way to visit the elements **in order**, without knowing the details of the container.
  - Generalization of pointers
- Separate container and algorithms with standard iterator interface functions.
  - The glue between algorithms and data structures
  - Without iterators, with N algorithms and M data structures, you need N*M implementations

Iterators

- One of **design patterns** (Gang of Four):

  "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

Usage

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

Usage

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

Usage

```
vector<int> vecTemp;
list<double> listTemp;

vector<int>::iterator  fVecIter, lVecIter;
fVecIter = vecTemp.begin();
lVecIter = vecTemp.end();
fVecIter = find(fVecIter, IVecIter,  3);
if (fVecIter == IVecIter)
    cout<<“3 not found in vecTemp”<<endl;

list<double>::iterator  fListIter, lListIter;
fListIter = listTemp.begin();
lListIter = listTemp.end();
fListIter = find(fListIter, lListIter,  3.0);
```

Requirements

- A unified interface used in algorithms
- Work like a pointer to the elements in a container
- Have ++ operator to visit elements in order
- Have * operator to visit the content of an element

auto_ptr

- An example of overloading * and -> operator

```
template<class T>
class auto_ptr {
private:
    T *pointee;
public:
/*…*/
T& operator *() { return *pointee; }
T* operator ->() { return pointee; }
/*…*/
};
```

Iterators

Example code:

```
template<class T>
class List {
public:
  void insert_front();
  void insert_end();
…
private:
  ListItem<T> *front;
  ListItem<T> *end;
  long _size;
};
```

```
template<class T>
class ListItem {
public:
  T& val() { return _value; }
  ListItem *next() { return
    _next};
…
private:
  T _value;
  ListItem<T> *_next;
};
```

Iterators

```
template<class T>
class ListIter {
  ListItem<T> *ptr;

public:
  ListIter(ListItem<T> *p=0) : ptr(p) {}
  ListIter<T>& operator++()
    { ptr = ptr->next(); return *this; }
  bool operator==(const ListIter& i) const
    { return ptr == i.ptr; }
  …
  T& operator*() { return ptr->val(); }
  T* operator->() { return &(**this);}
};
```

Iterators

How to use `ListIter`:

```
List<int> myList;
… // insert elements

ListIter<int> begin = myList.begin();
ListIter<int> end = myList.end();
ListIter<int> iter;

iter = find(begin, end, 3);
if (iter == end)
    cout << "not found" << endl;
```

Iterators

The associated type of an iterator:

```
// we do NOT know the data type of iter,
// so we need another variable v to infer T

template <class I, class T>
void func_impl(I iter, T& v)
{
    T tmp;
    tmp = *iter;
    // processing code here
}
```

Iterators

The associated type of an iterator:

```
// a wrapper to extract the associated
// data type T

template <class I>
void func(I iter)
{
    func_impl(iter, *iter);
    // processing code here
}
```

However, we might need more type information that associated to iterators

Iterators

Define the type information for an iterator:

```
template <class T>
struct myIter {
  typedef T value_type;

  T* ptr;
  myIter(T *p = 0):ptr(p)
{}

  T& operator*()
  { return *ptr; }
};
```

Iterators

Define the type information for an iterator:

```
template <class T>
struct myIter {
  typedef T value_type;

  T* ptr;
  myIter(T *p = 0):ptr(p)
{}

  T& operator*()
  { return *ptr; }
};
```

```
template <class I>
typename I::value_type func(I iter)
{
  return *iter;
}

// code
myIter<int> iter(new int(8));
cout << func(iter);
```

Iterators

The problem of the `typedef` trick:

- It cannot support pointer-type iterators, e.g., **`int*,double*,Complex*`**, which cripples the STL programming.

Use `iterator_traits` trick:

```
template <class I>
struct iterator_traits {
  typedef typename I::value_type value_type; }

template <class T*>
struct iterator_traits {
  typedef T value_type; }
```

iterator_traits

How to use:

```
template <class I>
typename iterator_traits<I>::value_type
func(I iter) {
  return *iter;
}

// code
myIter<int> iter(new int(8));
cout << func(iter);
int* p = new int[20]();
cout<<func(p); // iterator_traits<int*>??
```

Template specialization

Primary template:

```
template<class T1, class T2, int I>
class A { … };
```

Explicit (full) template specialization:

```
template<>
class A<int, double, 5> { … };
```

Partial template specialization:

```
template<class T2>
class A<int, T2, 3> { … };
```

Iterator traits

The traits technique with template specialization:

```
template<class T>
class C
{
public:
    C() {
    cout<<"template
      T"<<endl;
   }
};
```

```
template<class T>
class C<T*>
{
public:
    C() {
    cout<<"template
     T*"<<endl;
   }
};
```

Iterator traits

The traits technique with template specialization:

```
template<class I>
class iterator_traits
{
public:
    typedef typename I::value_type value_type;
    typedef typename I::pointer_type pointer_type;
    ……
};
```

Iterator traits

The traits technique with template specialization:

```
template<class I>
class iterator_traits
{
public:
 typedef typename
  I::value_type value_type;
 typedef typename
  I::pointer_type pointer_type;
 ……
};
```

```
template<class T>
class iterator_traits
<T*>
{
public:
 typedef T value_type;
 typedef T* pointer_type;
 ……
};
```

Iterator traits

The traits technique with template specialization:

```
template<class I>
class iterator_traits
{
public:
 typedef typename
  I::value_type value_type;
 typedef typename
  I::pointer_type pointer_type;
 ……
};
```

```
template<class T>
class iterator_traits
<const T*>
{
public:
 typedef T value_type;
 typedef const T*
  pointer_type;
 ……
};
```

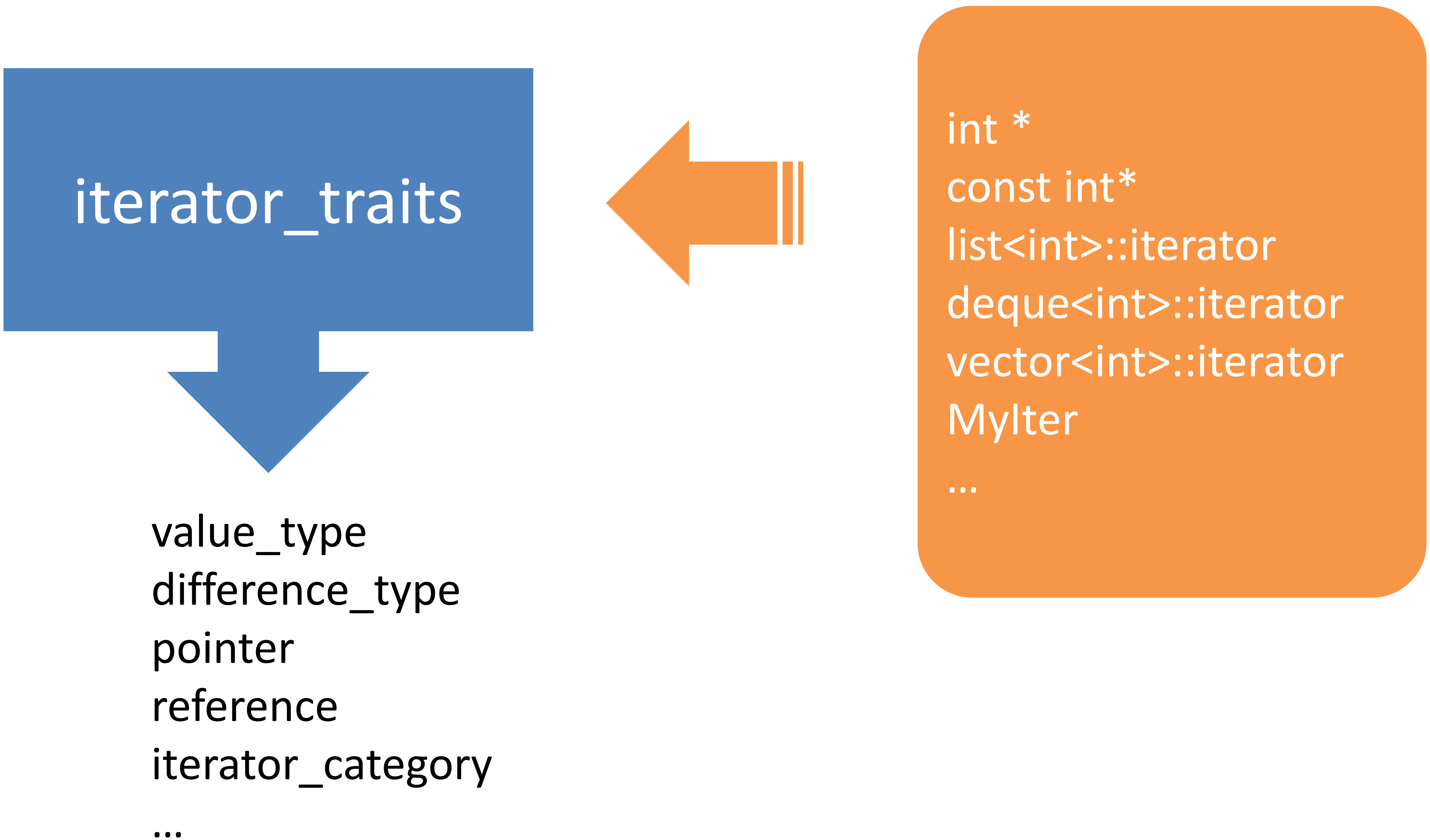
Standard traits in STL

The standard traits technique in STL:

```
template<class I>
class iterator_traits
{
public:
    typedef typename I::iterator_category iterator_category;
    typedef typename I::value_type value_type;
    typedef typename I::difference_type differece_type;
    typedef typename I::pointer pointer;
    typedef typename I::reference reference;
    ……
}
```

Standard traits in STL

The standard traits technique in STL:



Iterators

Iterator category (types):

- `InputIterator`
- `OutputIterator`
- `ForwardIterator`
- `BidirectionalIterator`
- `RandomAccessIterator`

Iterators

- Container knows how to design its own iterator.
- Traits trick extracts type information embedded in different iterators, including raw pointers.
- Algorithms are independent to containers through the design philosophy of iterators.