Object-Oriented Programming (OOP)
Lecture No. 40

Recap

- Generic algorithm requires three operations (++, *, !=)
- Implementation of these operations in `vector` class
- Problems
  - No support for multiple traversals
  - Supports only a single traversal strategy
  - Inconsistent behavior
  - Operator !=

Cursors

- A better way is to use *cursors*
- A cursor is a pointer that is declared outside the container / aggregate object
- Aggregate object provides methods that help a cursor to traverse the elements
  - T* first()
  - T* beyond()
  - T* next(T*)

Vector

```
template< class T >
class Vector {
private:
  T* ptr;
  int size;
public:
  Vector( int = 10 );
  Vector( const Vector< T >& );
  ~Vector();
  int getSize() const;
```

...Vector

```
  const Vector< T >& operator =( const
                  Vector< T >& );
  T& operator []( int );
  T* first();
  T* beyond();
  T* next( T* );
};
```

...Vector

```
template< class T >
T* Vector< T >::first() {
  return ptr;
}

template< class T >
T* Vector< T >::beyond() {
  return ( ptr + size );
}
```

...Vector

```
template< class T >
T* Vector< T >::next( T* current )
{
  if ( current < (ptr + size) )
     return ( current + 1 );
  // else
  return current;
}
```

Example – Cursor

```
int main() {
  Vector< int > iv( 3 );
  iv[0] = 10;
  iv[1] = 20;
  iv[2] = 30;
  int* first = iv.first();
  int* beyond = iv.beyond();
  int* found = find(first,beyond,20);
  return 0;
}
```

Generic Algorithm

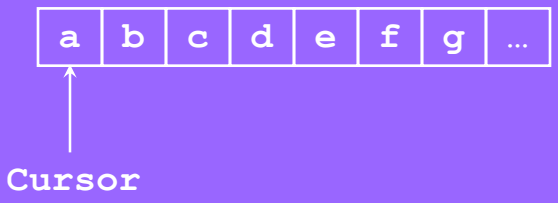
```
template< typename P, typename T >
P find( P start, P beyond,
                        const T& x ) {
  while ( start != beyond &&
                         *start != x )
     ++start;

  return start;
}
```

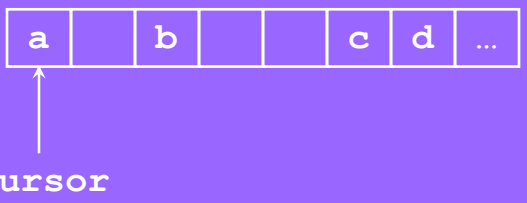
...Cursors

- This technique works fine for a contiguous sequence such as Vector
- However it does now work with containers that use complicated data structures
- There we have to rely on the container traversal operations

Example – Works Fine



Example – Problem



Example – Problem

```
int main() {
  Set< int > is( 3 );
  is.add( 10 );
  is.add( 20 );
  is.add( 30 );
  ET* first = iv.first();
  ET* beyond = iv.beyond();
  ET* found = find(first, beyond, 20);
  return 0;
}
```

...Example – Problem

```
template< typename P, typename T >
P find( P start, P beyond,
                        const T& x ) {
  while ( start != beyond &&
                         *start != x )
     ++start; // Error

  return start;
}
```

Works Fine

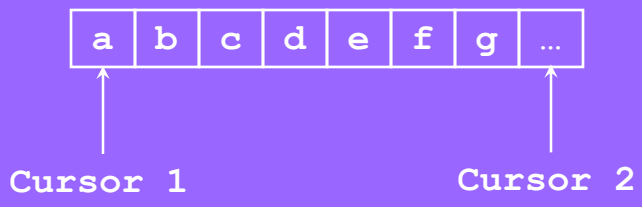
```
template< typename CT, typename ET >
P find( CT& cont, const ET& x ) {
  ET* start = cont.first();
  ET* beyond = cont.beyond();
  while ( start != beyond &&
                        *start != x )
     start = cont.next( start );
  return start;
}
```

...Works Fine

```
int main() {
  Set< int > is( 3 );
  is.add( 10 );
  is.add( 20 );
  is.add( 30 );
  int* found = find( is, 20 );
  return 0;
}
```

Cursors – Conclusion

- Now we can have more than one traversal pending on the aggregate object



…Cursors – Conclusion

- However we are unable to use cursors in place of pointers for all containers

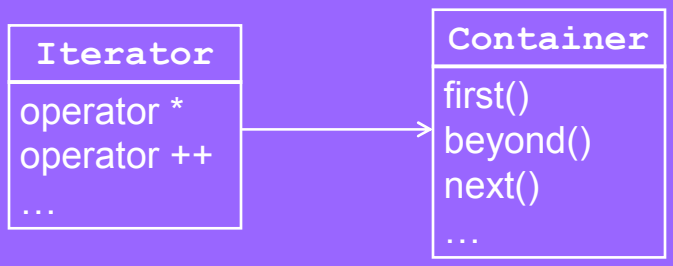
Iterators

- Iterator is an object that traverses a container without exposing its internal representation
- Iterators are for containers exactly like pointers are for ordinary data structures

Generic Iterators

- A generic iterator works with any kind of container
- To do so a generic iterator requires its container to provide three operations
  - T* first()
  - T* beyond()
  - T* next(T*)

Example – Generic Iterator



Generic Iterator

```
template< class CT, class ET >
class Iterator {
  CT* container;
  ET* index;
public:
  Iterator( CT* c,
            bool pointAtFirst = true );
  Iterator( Iterator< CT, ET >& it );
  Iterator& operator ++();
  ET& operator *();
```

...Generic Iterator

```
  bool operator !=(
          Iterator< CT, ET >& it );
};
```

...Generic Iterator

```
template< class CT, class ET >
Iterator< CT, ET >::Iterator( CT* c,
              bool pointAtFirst ) {
  container = c;
  if ( pointAtFirst )
     index = container->first();
  else
     index = container->beyond();
}
```

...Generic Iterator

```
template< class CT, class ET >
Iterator< CT, ET >::Iterator(
     Iterator< CT, ET >& it ) {
  container = it.container;
  index = it.index;
}
```

...Generic Iterator

```
template< class CT, class ET >
Iterator<CT,ET>& Iterator<CT,ET>::
                        operator ++() {
  index = container->next( index );
  return *this;
}
```

...Generic Iterator

```
template< class CT, class ET >
ET& Iterator< CT, ET >::operator *()
{
  return *index;
}
```

...Generic Iterator

```
template< class CT, class ET >
bool Iterator< CT, ET >::operator !=(
     Iterator< CT, ET >& it ) {
  if ( container != it.container ||
                    index != it.index )
     return true;
  // else
  return false;
}
```

...Generic Iterator

```
int main() {
  Vector< int > iv( 2 );
  Iterator< Vector<int>, int >
     it( &iv ), beyond( &iv, false );
  iv[0] = 10;
  iv[1] = 20;
  Iterator< Vector<int>, int > found
         = find( it, beyond, 20 );
  return 0;
}
```

...Generic Iterator

```
template< typename P, typename T >
P find( P start, P beyond,
                        const T& x ) {
  while ( start != beyond &&
                         *start != x )
     ++start;

  return start;
}
```

Iterators – Conclusion

- With iterators more than one traversal can be pending on a single container
- Iterators allow to change the traversal strategy without changing the aggregate object
- They contribute towards data abstraction by emulating pointers