COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

Templates and Iterators

Slide Set: 6

Learning Objectives

- To understand the importance of **template functions** and **template classes**
- Design and implement template functions and template classes
- Implementing the Bag template class using dynamic array
- Use **iterators** to step through all the elements of an object for any of the STL classes
- **Iterator invalidation rules**
- Manipulate objects of the STL classes using functions from the `<algorithms>` library facility
- Implementing the Node template class
- Implementation of simple forward iterators for data structures
- Implementing the **bag template class** using linked list

Template Functions

Template Functions

Introduction

- Our sequence and node classes can be used in different settings, **however, these classes suffer from the fact that they require the underlying `value_type` to be fixed**

- **This chapter provides a better approach to writing reusable codes**
- The approach, called **templates**, is **applicable to individual functions and to classes**
- Templates are a C++ feature that easily permits reuse

Template Functions

Finding the Maximum of Two Integers

- Example: Suppose we write this function:

```
int maximum(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

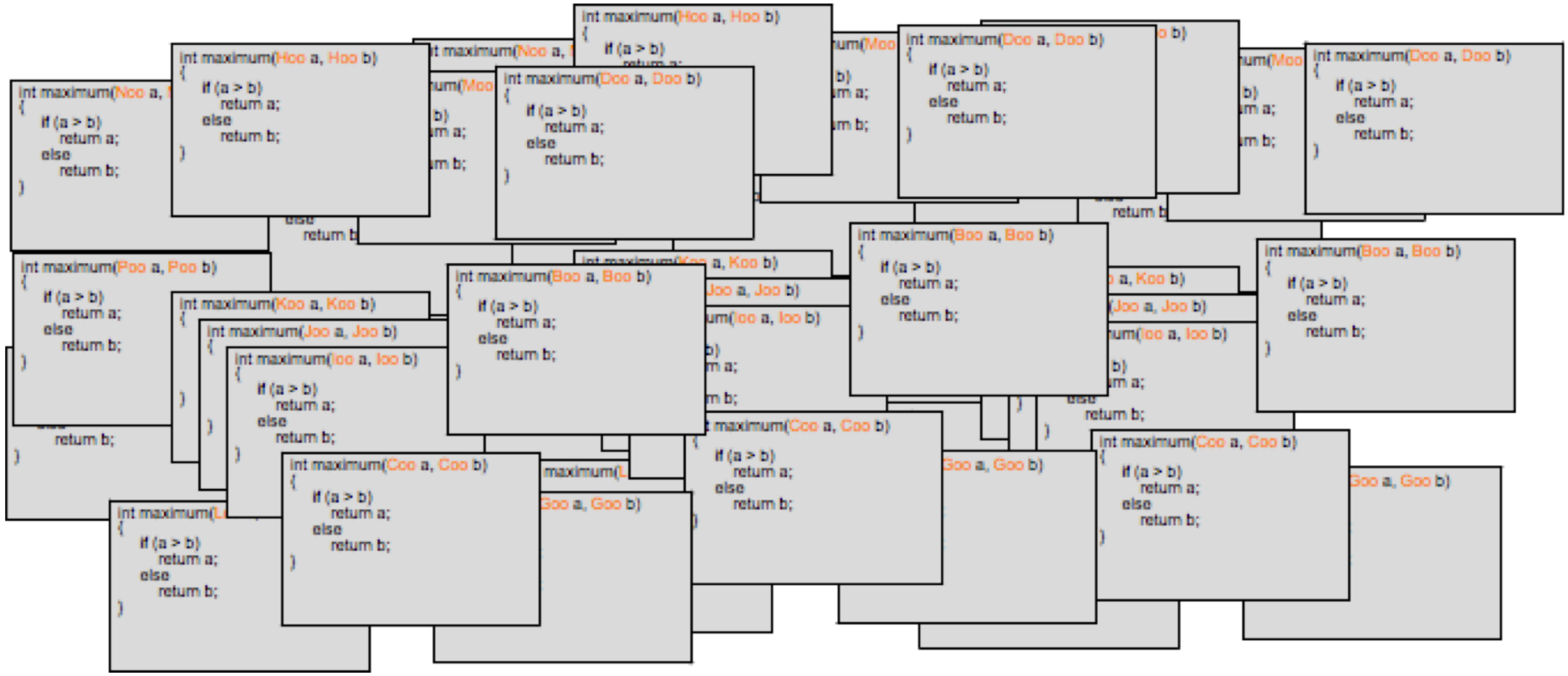
- Suppose that tomorrow you have another program that needs to compute the larger of two **double** numbers

```
double maximum(double a, double b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Template Functions

One Hundred Million Functions...

- Suppose your program uses 100,000,000 different data types, and you need a `maximum` function for each...



Template Functions

Finding the Maximum of Two Items

- When one of the functions is used, the compiler looks at the type of the arguments and selects the appropriate version of the function
- However, **with this approach, you need to write a new function for each type of value that you want to compare**
- **Solution 1:** You could write just one function, along with a `typedef` statement:

```
typedef -------- item;
item maximum(item a, item b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

- A programmer can fill in the `typedef` statement **with any data type that has `operator >` defined and that has a copy constructor**

Template Functions

- **Shortcoming of Solution 1:** Suppose that a single program needs to use several different versions of the function
  - **`typedef` approach does not allow this**

- **Solution 2: Template function**, is similar to an ordinary function with one important difference: **The definition of a template function can depend on an underlying data type**
- The underlying data type is given a name (such as `Item`)
  - `Item` is not pinned down to a specific type anywhere in the function's implementation

- When a template function is used, the **compiler** examines the types of the arguments and at that point the compiler automatically determines the data type of `Item`

Template Functions

Syntax for a Template Function

Using `typedef`

```
typedef int Item;
Item maximum(Item a, Item b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Using template

```
template <class Item>
Item maximum(Item a, Item b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

- Single definition of the template function allows a program to use the function with two integers, or with two doubles, or with two strings … With any data type that has the > operator and a copy constructor
- The expression **`template <class Item>`** is called the **template prefix**, and it warns the compiler that the following definition will use an unspecified data type called `Item`
- The "unspecified type" is called the **template parameter**

Template Functions

Using a Template Function

```
template <class Item>
Item maximum(Item a, Item b)
{
    if (a > b) return a;
    else       return b;
}
```

```
cout << maximum(1000,2000);
```

The function has been **instantiated** with Item equal to `int`

- When the compiler sees this function call, it determines that the type of `Item` must be `int`, and it automatically uses the `maximum` function with `Item` defined as `int`

```
string s1("scu");
string s2("coen");
cout << maximal(s1, s2);
```

The function has been **instantiated** with `Item` equal to `string`
Prints the string that is lexicographically larger

Template Functions

A Program for Using Template Functions

```
#include <cstdlib>     // Provides EXIT_SUCCESS
#include <iostream>    // Provides cout
#include <string>      // Provides string class

using namespace std;

template <class Item>
Item maximal(Item a, Item b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Note:

- You may use "`typename`" instead of "`class`"
- There is no difference

Template Functions

A Program for Template Functions (Cont'd)

```
int main( )
{
 string s1("scu");
 string s2("coen");

 cout << "Larger of scu and coen: " << maximal(s1, s2)<< endl;

 cout << "Larger of 1 and 2 : " << maximal(1, 2) << endl;

 cout << "It's a large world." << endl;

 return EXIT_SUCCESS;
}
```

When the compiler sees this function call:

- It determines that the type of `Item` must be `int`
- Automatically uses the `maximal` function with `Item` defined as `int`

The `maximal` function has been **instantiated** with `Item` equal to `int`

- Violating this rule will likely result in cryptic error messages such as "**Failed unification**"
- **Unification** is the compiler's term for determining how to instantiate a template function

Template Functions

STL `<algorithm>` Facility

- The `<algorithm>` facility in the C++ Standard Library contains the `swap` function, a `max` function that is similar to our `maximal`, and a `min` function that returns the smaller of two items
- All of these functions are **templated functions**

Template Functions

STL <algorithm> Facility (Cont'd)

```
#include <iostream>      // std::cout
#include <algorithm>     // std::swap
#include <vector>        // std::vector
int main () {

  int x=1, y=2;

  std::vector<int> foo (2,x), bar (2,y);   // foo: 2x1   bar: 2x2
  std::swap(foo, bar);                      // foo: 2x2   bar: 2x1

   std::cout << "foo contains:";
   for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

- This example shows the use of STL's `swap` algorithm
- The template function's parameter type is `vector`

```
Output:
foo contains: 2 2
```

Template Functions

Parameter Matching for Template Functions

- Assume that we implement a template function that searches an array for the biggest item and returns the index of that item

```
template <class Item>
size_t  index_of_maximal (const Item data[], size_t n)
```

- When a template function is instantiated, the compiler tries to select the underlying data type so that the type of each argument results in an **exact match** with the type of the corresponding formal parameter

Template Functions

Parameter Matching for Template Functions (Cont'd)

- The compiler **does not convert** arguments for a template function, the arguments must have an **exact match**, with no type conversion
- **The requirement of an exact match applies to all parameters of a template function**

❑ Example: Consider the prototype:

```
template <class Item>
size_t  index_of_maximal (const Item data[], size_t n)
```

- When we call the function, the second argument must be a `size_t` value
- Many compilers cannot accept any deviation:
  - **Invalid:** `int, double, const size_t`
  - **Valid**: `size_t`

Template Functions

Parameter Matching for Template Functions (Cont'd)

- On such a strict compiler, the following will fail:

```
const size_t SIZE = 5;
double data[SIZE];
...

cout<< index_of_maximal(data, SIZE);

cout<< index_of_maximal(data, 5);
```

This fails on many compilers because `SIZE` is declared as `const size_t` rather than a `size_t`

This fails because the compiler takes 5 to be an `int` rather than a `size_t` value

Template Functions

A Template Function to Find the Biggest Item in an Array

- We can deal with the problem by using two template parameters

```
template <class Item, class SizeType>
size_t  index_of_maximal (const Item data[], SizeType n)
```

- With this template function, we have more flexibility:

```
const size_t SIZE = 5;
double data[SIZE];
...
cout<< index_of_maximal(data, SIZE);
cout<< index_of_maximal(data, 5);
```

SizeType will be const size_t

SizeType will be int

Template Functions

A Template Function to Find the Biggest Item in an Array (Cont'd)

Implementation of the `index_of_maximal` template function

```
template <class Item, class SizeType>
std::size_t  index_of_maximal (const Item data[], SizeType n)
{
    std::size_t answer;
    std::size_t i;

    assert(n > 0);
    answer = 0;

    for (i=1; i < n; ++i)
    {
        if (data[answer] < data[i])
            answer = i;
    }

    return answer;
}
```

Template Classes

Template Classes

- A template function is a function that depends on an underlying data type
- In a similar way, when a class depends on an underlying data type, the class can be implemented as a **template class**
- For example, a single program can use a bag of integers, and a bag of characters, and a bag of strings, and …
- You do not have to determine the data type of a data structure when developing a code

Template Classes

Syntax for a Template Class

- Implementing a bag as a template class requires three changes from this original approach:

1. Change the template class definition

```
class bag
{
public:
    typedef int value_type;
    ...
```

```
template <class Item>
class bag
{
public:
        ...
```

- `template <class Item>` is the template prefix, and it warns the compiler that the following definition will use an unspecified data type called `Item`

Syntax for a Template Class (Cont'd)

2. **Implement functions for the template class:** The bag's `value_type` is now dependent on the `Item` type
   - **Outside of the template class definition** some rules are required to tell the compiler about the dependency:
   1. The template prefix `template <class Item>` is placed immediately before each function prototype and definition
   2. Outside of the template class definition, each use of the class name (such as `bag`) is changed to the template class name (such as `bag<Item>`)
   3. Within a class definition or within a member function, we can still use the bag's type names, such as `value_type` or `size_type`
   4. Outside of a member function, to use a type such as `bag<item>::size_type`, we must add a new keyword `typename`, writing the expression `typename bag<item>::size_type`

Syntax for a Template Class (Cont'd)

❑ Example 1

- For our **original class**, the function's implementation began this way:

```
bag operator +(const bag& b1, const bag&b2) ...
```

- For the **template class**, the start of the implementation is shown here:

```
template <class Item>
bag<Item> operator +(const bag<Item>& b1, const bag<Item>&b2) ...
```

Syntax for a Template Class (Cont'd)

❑ Example 2

In the original bag:

```
bag::size_type  bag::count(const value_type& target)  const
```

- Outside of a member function, you must put the keyword `typename` in front of any member of a template class that is the name of a data type

```
template <class Item>
typename bag<Item>::size_type  bag<Item>::count
                                    (const Item& target) const
```

Syntax for a Template Class (Cont'd)

3. **In the header file,** you place the documentation and the prototypes of the functions; **then you must include the actual implementations of all the functions**
   - The reason for the requirement is to make the compiler's job simpler
   - **An alternative:** You can keep the implementations in a separate implementation file, but place an `include` directive at the bottom of the header file to pick up these implementations
   - We include the following line at the end of the header file

```
#include "bag4.template" //Include the implementation
```

Use the Name `Item` and the `typename` Keyword (Cont'd)

- Examples:

In the Original Bag	In the Template Bag Class
`value_type`	`Item`
`size_type` (inside a member function)	`size_type`
`size_type` (outside a member function)	*`typename`* `bag<Item>::size_type`

Do Not Place `using` Directives in a Template Implementation

- Because a template class has its implementation included in the header file, we must not place any `using` directives in the implementation
- Otherwise, every program that uses our template class will inadvertently use the `using` directives

Template Classes

Header File for the Bag Template Class

```
#ifndef SCU_COEN79_BAG4_H
#define SCU_COEN79_BAG4_H
#include <cstdlib>   // Provides size_t

namespace scu_coen79_6A
{
    template <class Item>
    class bag
    {
    public:

        // TYPEDEFS and MEMBER CONSTANTS
        typedef Item value_type;
        typedef std::size_t size_type;
        static const size_type DEFAULT_CAPACITY = 30;

        // CONSTRUCTORS and DESTRUCTOR
        bag(size_type initial_capacity = DEFAULT_CAPACITY);
        bag(const bag& source);
        ~bag( );
```

Template Classes

Header File for the Bag Template Class (Cont'd)

```
    // MODIFICATION MEMBER FUNCTIONS
    size_type erase(const Item& target);
    bool erase_one(const Item& target);
    void insert(const Item& entry);
    void operator =(const bag& source);
    void operator +=(const bag& addend);
    void reserve(size_type capacity);

    // CONSTANT MEMBER FUNCTIONS
    size_type count(const Item& target) const;
    Item grab( ) const;
    size_type size( ) const { return used; }

private:
  Item *data;           // Pointer to partially filled dynamic array
  size_type used;       // How much of array is being used
  size_type capacity;   // Current capacity of the bag
 };
```

Inside the template class definition:
Compiler knows about the dependency on `Item` type

Template Classes

Header File for the Bag Template Class (Cont'd)

Some rules are required outside of the template class definition

```
// NONMEMBER FUNCTIONS
```

`template <class Item>` is placed immediately before each function prototype and definition

```
template <class Item>
bag<Item> operator +(const bag<Item>& b1, const bag<Item>& b2);
}
```

Each use of the template class name is changed to the template class name (`bag<Item>`)

```
#include "bag4.template"
#endif
```

We should then include the implementation of the template in the header file (needed by most compilers)

Instead of that: we keep the implementation in a separate file

Template Classes

Implementation File for the Bag Template Class

```
#include <algorithm>  // Provides copy
#include <cassert>    // Provides assert
#include <cstdlib>    // Provides rand


namespace SCU_COEN79_6A
{

  // MEMBER CONSTANTS
  template <class Item>
  const typename bag<Item>::size_type bag<Item>::DEFAULT_CAPACITY;
```

Some compilers require the default parameters to be both in the prototype and implementation

Remember that:
To refer to `size_type` outside a member function:
`typename bag<Item>::size_type`

Template Classes

Implementation File for the Bag Template Class (Cont'd)

Each definition is preceded by `template <class Item>`

```
// CONSTRUCTORS
template <class Item>
bag<Item>::bag(size_type  initial_capacity)
{
    data = new Item[initial_capacity];
    capacity = initial_capacity;
    used = 0;
}


template <class Item>
bag<Item>::bag(const bag<Item>& source)
// Library facilities used: algorithm
{
    data = new Item[source.capacity];
    capacity = source.capacity;
    used = source.used;
    std::copy(source.data, source.data + used, data);
}
```

Note: We do not include any `using` directive, as the implementation is in the header file

Template Classes

Implementation File for the Bag Template Class (Cont'd)

```
    //DESTRUCTOR
    template <class Item>
    bag<Item>::~bag( )
    {
        delete [ ] data;
    }
    // MODIFICATION MEMBER FUNCTIONS:
    template <class Item>
    typename bag<Item>::size_type bag<Item>::erase(const Item& target)
    {
        size_type index = 0;
        size_type many_removed = 0;
        while (index < used)
        {
            if (data[index] == target)
            {
                --used;
                data[index] = data[used];
                ++many_removed;
            }
            else   --index;
        }
        return many_removed;
    }
         || OTHER FUNCTIONS….
}
```

Template Classes

Parameter Matching for Member Functions of Template Classes

Note:

- Remember that: In the implementation of a **template function**, we were careful to help the compiler by providing a template parameter for each of the function's parameters

- For the **member functions of a template class:** This help is not needed

❑For example, we can use a simple `size_type` parameter for the bag's `reserve` function

- Unlike an ordinary **template function**, the compiler is able to match a `size_type` parameter of a **member function** with any of the usual integer arguments (such as `int` or `const int`)

Template Classes

Using the Template Class

- When the template bag class is ready, then we can declare one bag of characters and one bag of double numbers:

```
bag<char> letters;
```

The template parameter is **instantiated** as a character

```
bag<double> scores;
```

The template parameter is **instantiated** as a double

The Node Template Class

The Node Template Class

Original Node Class vs New Template Node Class

Original Node Class

```
class node
{
 public:
  typedef double value_type;
        . . .
```

New Template Node Class

```
template <class Item>
class node
{
 public:
       . . .
```

The Node Template Class

Original Node Class vs New Template Node Class (Cont'd)

❑ Example: The original prototype for the `list_insert` function (in the toolkit) is:

```
void list_insert (node* previous_ptr,
                                  const node::value_type& entry);
```

- The new prototype for the `list_insert` template function is preceded by the template prefix and uses `Item` within its parameter list

```
template <class Item>
void list_insert (node<Item>* previous_ptr, const Item& entry);
```

The Node Template Class

Original Node Class vs New Template Node Class (Cont'd)

The toolkit's function `list_locate`:

- Our original node had a `const` and a `non-const` version:

```
node* list_locate(node* head_ptr, size_t position);
const node* list_locate(const node* head_ptr, size_t position);
```

- The presence of `size_t` parameter can cause mismatches for an argument type – Remember that this is a non-member function
- The solution is the same approach that we used before: Add a second template parameter

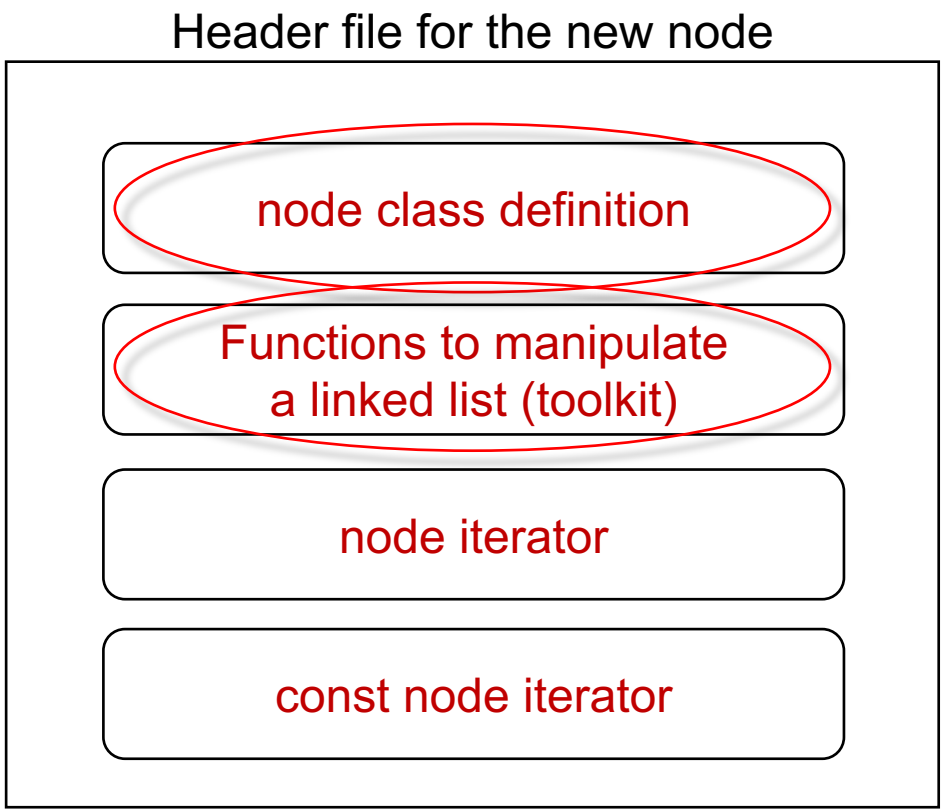
- **We can actually combine the two functions into a single function**

```
template <class NodePtr, class SizeType>
NodePtr list_locate(NodePtr head_ptr, SizeType position);
```

- **The `head_ptr` may be either an ordinary or a const pointer**

The Node Template Class

Header File for the New Node



The Node Template Class

Header File for the New Node (Cont'd)

```
#ifndef SCU_COEN79_NODE2_H
#define SCU_COEN79_NODE2_H
#include <cstdlib>   // Provides NULL and size_t
#include <iterator>  // Provides iterator and forward_iterator_tag

namespace SCU_COEN79_6B
{

  template <class Item>
  class node
  {
    public:


    // TYPEDEF
    typedef Item value_type;


    // CONSTRUCTOR
    node(const Item& init_data = Item( ), node* init_link = NULL)
            { data_field = init_data; link_field = init_link; }
```

The Node Template Class

Header File for the New Node (Cont'd)

```
// MODIFICATION MEMBER FUNCTIONS
Item& data( ) { return data_field; }
node* link( ) { return link_field; }
void set_data(const Item& new_data) {data_field = new_data;}
void set_link(node* new_link) { link_field = new_link; }


// CONST MEMBER FUNCTIONS
const Item& data( ) const { return data_field; }
const node* link( ) const { return link_field; }
```

This enables us to run:
`cursor->data() = "scu";`

The returned value cannot be used to change data field

Examples:

- `const node *c;`
  - `c->link( )` activates the const version of link
  - `list_locate(c,...` calls the const version of `list_search`
- `node *p;`
  - `p->link( )` activates the non-const version of link
  - `list_locate(p,...` calls the non-const version of `list_search`

The Node Template Class

Header File for the New Node (Cont'd)

```
private:
 Item data_field;
 node *link_field;
};

// FUNCTIONS to manipulate a linked list (toolkit):
template <class Item>
void list_clear(node<Item>*& head_ptr);

template <class Item>
void list_copy (const node<Item>* source_ptr,
                    node<Item>*& head_ptr,  node<Item>*& tail_ptr);


template <class Item>
void list_head_insert(node<Item>*& head_ptr, const Item& entry);


template <class Item>
void list_head_remove(node<Item>*& head_ptr);
```

The Node Template Class

Header File for the New Node (Cont'd)

In the original non-template node class we had:

```
void list_insert(node* previous_ptr, const node::value_type& entry);
```

```
template <class Item>
void list_insert(node<Item>* previous_ptr, const Item& entry);


template <class Item>
std::size_t list_length(const node<Item>* head_ptr);
```

The Node Template Class

Header File for the New Node (Cont'd)

Originally (we had two versions):

```
node* list_locate(node* head_ptr, std::size_t position);
const node* list_locate(const node* head_ptr, std::size_t position);
```

- The presence of `size_t` parameter can cause mismatch for an argument type
- Solution: Add a second template parameter
- **We can also combine the two versions**
- Now the return type matches the type of the head pointer

```
template <class NodePtr, class SizeType>
NodePtr list_locate(NodePtr head_ptr, SizeType position);
```

```
  ||Definition of iterator (next section)
}

#include "node2.template"
#endif
```

May be either:
`node<item>` or
`const node<item>*`

The STL's Algorithms and Use of Iterators

The STL's Algorithms and Use of Iterators

Definition

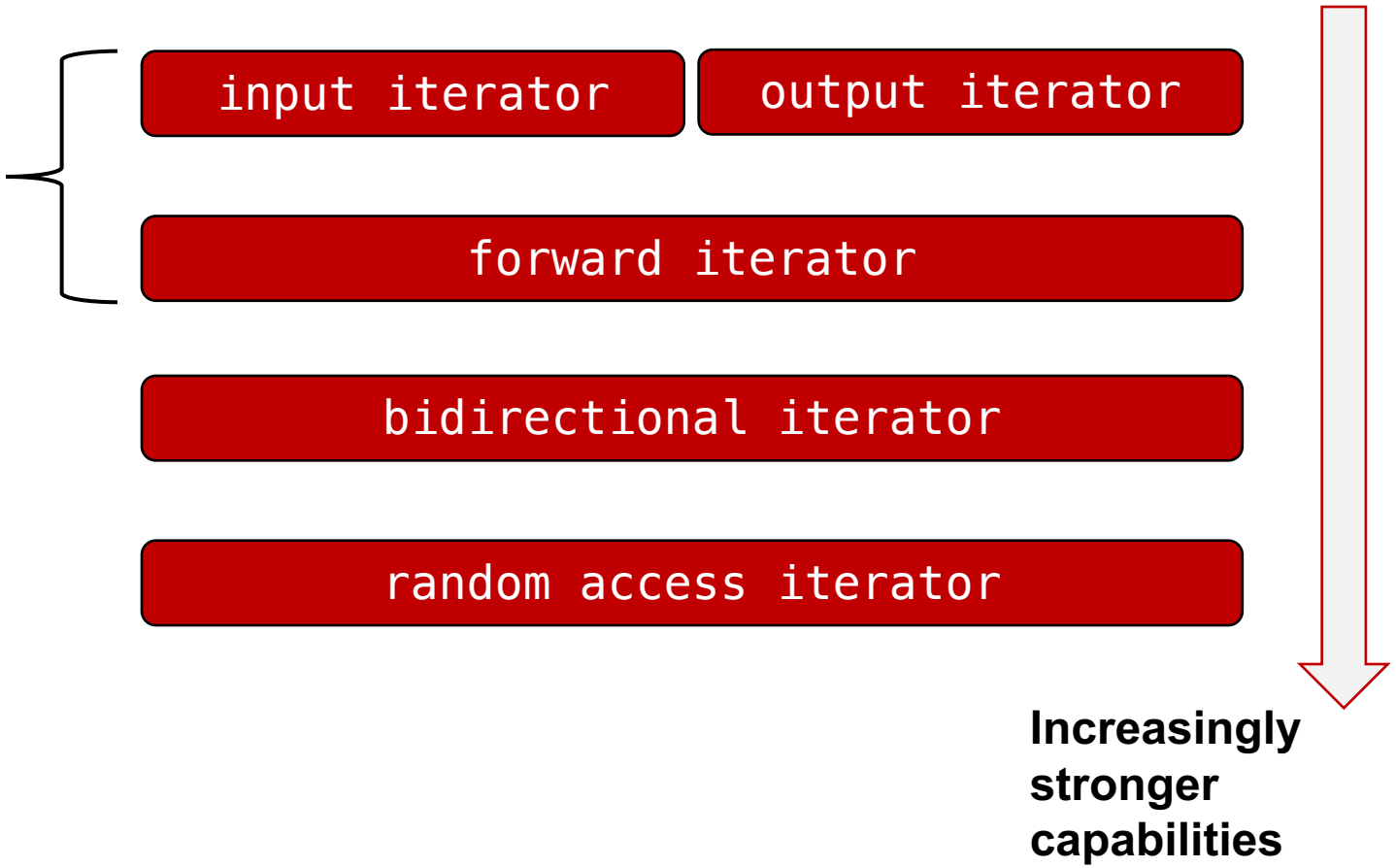
- An **iterator** is any object:
  - **Pointing to some element in a range of elements** (such as an array or a container)
  - Has the ability to iterate through the elements of that range using a set of operators (**with at least the increment (++) and dereference (*) operators**)

- The most obvious form of iterator is a **pointer**
  - A pointer can point to elements in an **array**, and can iterate through them using the increment operator (++)

Standard Categories of Iterators

- The C++ Standard specifies **five significant categories of iterators, based on their capabilities**

Specialized iterators for **retrieving** and **inserting** elements



Standard Categories of Iterators

Iterator form	Description
input iterator	Read only, forward moving
output iterator	Write only, forward moving
forward iterator	Both read and write, forward moving
bidirectional iterator	Read and write, forward and backward moving
random access iterator	Read and write, random access

Standard Categories of Iterators

Input Iterator

- An input iterator is designed to read a sequence of values
- Current element of an input iterator `p` can be retrieved by using the dereferencing `*` operator such as `x = *p`
- The ++ increment operator moves the iterator forward to another item
- The end of an input iterator's elements is usually detected by comparing the input iterator with another iterator that is known to be just beyond the end of the input range

- Produced by: `istream_iterator`

The STL's Algorithms and Use of Iterators

Standard Categories of Iterators (Cont'd)

```
// istream_iterator example
#include <iostream>     // std::cin, std::cout
#include <iterator>     // std::istream_iterator

int main () {
    double value1, value2;
    std::cout << "Insert two values: ";

    std::istream_iterator<double> eos;
    std::istream_iterator<double> iit (std::cin);   // stdin iterator

    if (iit!=eos) value1 = *iit;

    ++iit;
    if (iit!=eos) value2 = *iit;

    std::cout << value1 << "+" << value2 << "=" << (value1+value2) <<'\n';

    return 0;
}
```

The resulting value of a default-constructed object is an **end-of-stream** iterator

Read the first number

Read the second number

```
Output:
Insert two values: 1 5
1+5=6
```

Standard Categories of Iterators (Cont'd)

Output Iterator

- To change the element the iterator refers to, for example: `*p="dance"`
- The ++ increment operator moves the iterator forward to another item
- **The output operator itself cannot be used to retrieve elements**
- The output iterator's usefulness is limited to the situation where some algorithm needs to put a sequence of elements in a container or other object with an output iterator

- Produced by:

```
ostream_iterator; inserter(); front_inserter();
back_inserter()
```

The STL's Algorithms and Use of Iterators

Standard Categories of Iterators (Cont'd)

```
// ostream_iterator example
#include <iostream>     // std::cout
#include <iterator>     // std::ostream_iterator
#include <vector>       // std::vector
#include <algorithm>    // std::copy

int main ()
  {
    std::vector<int> myvector;
    for (int i=1; i<10; ++i)  myvector.push_back(i);

    std::ostream_iterator<int> out_it (std::cout);

    std::copy (myvector.begin(), myvector.end(), out_it);

    return 0;
  }
```

```
Output:
123456789
```

Standard Categories of Iterators (Cont'd)

Forward Iterator

A forward iterator p is an object that provides these items:

- Have all the functionality of **input iterators**
- **If they are not constant iterators**, then they also have the functionality of **output iterators**
- They are limited to one direction in which to iterate through a range (forward)
- **All standard containers support at least forward iterator types**

❑Example: Iterator of a `forward_list` is a forward iterator

The STL's Algorithms and Use of Iterators

Standard Categories of Iterators (Cont'd)

```
#include <iostream>
#include <forward_list>

int main ()
{
    std::forward_list<int> mylist(4);

    for (std::forward_list<int>::iterator it = mylist.begin();
                                          it != mylist.end(); ++it )
        *it = rand();
```

Used as an output iterator

```
    std::cout << "mylist contains:";
    for (std::forward_list<int>::iterator it = mylist.begin();
                                          it != mylist.end(); ++it )
        std::cout << ' ' << *it;
```

Used as an input iterator

```
    return 0;
}
```

Standard Categories of Iterators (Cont'd)

Bidirectional Iterator

- Has all the abilities of a forward iterator, plus it can move backward with the –– operator
- The `--p` operator moves the iterator `p` backward one position and returns the iterator after it has moved backward
- The `p--` operator also moves the iterator backward one position and returns a copy of the iterator before it moved
- Produced by: list; set and multiset; map and multimap

The STL's Algorithms and Use of Iterators

Standard Categories of Iterators (Cont'd)

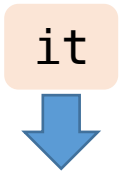
❑ Example 1

```
#include <iostream>     // std::cout
#include <iterator>     // std::insert_iterator
#include <list>         // std::list
#include <algorithm>    // std::copy

int main () {
  std::list<int> foo, bar;
  for (int i=1; i<=5; i++) { foo.push_back(i); bar.push_back(i*10); }
```

foo	1	2	3	4	5
bar	10	20	30	40	50

```
  std::list<int>::iterator it = foo.begin();
```

Returns a bidirectional iterator



foo	1	2	3	4	5

Standard Categories of Iterators (Cont'd)

```
  std::copy (bar.begin(), bar.end(), it);
  std::cout << "foo:";
```

foo	10	20	30	40	50

```
  for ( std::list<int>::iterator it=foo.begin(); it!=foo.end(); ++it )
         std::cout << ' ' << *it;

  std::cout << '\n';
  return 0;
}
```

```
Output:
foo: 10 20 30 40 50
```

Standard Categories of Iterators (Cont'd)

Random Access Iterator

- Has all the abilities of bidirectional iterators
- The term **random access** refers to the ability to quickly access any randomly selected location in a container
- **A random access iterator `p` can use the notation `p[n]` (to provide access to the item that is `n` steps in front of the current item**
- Therefore, distant elements can be accessed directly by applying an offset value to an iterator without iterating through all the elements in between
- Produced by: ordinary pointers; vector; deque

❑ Example: `p[0]` is the current item, `p[1]` is the next item, and so on

Iterators for Arrays

- **C++ allows any pointer to an element in an array to be used as if it were a random access iterator**
- The "current item" of such a pointer is the array element that it points to
- The ++ and –– operators move the pointer forward or backward one spot
- For a pointer `p`, the notation `p[i]` refers to the item that is `i` steps ahead of the current item
- Since a pointer to an array is a random access iterator, we can use these pointers in Standard Library functions that expect an iterator

❑ Example (see next slide)

The STL's Algorithms and Use of Iterators

Iterators for Arrays (Cont'd)

- Example: The `copy` function from `<algorithm>` is a template function with this prototype:

```
template <class sourceIterator, class DestinationIterator>
DestinationIterator copy(
                sourceIterator source_begin,
                sourceIterator source_end,
                DestinationIterator destination_begin );
```

- Both `source_begin` and `source_end` are iterators over the same object
- The first element that is copied comes from `source_begin`, and the copying continues up to (but not including) `source_end`
- The return value is an iterator that is one position beyond the last copied element in the destination

The STL's Algorithms and Use of Iterators

Iterators for Arrays (Cont'd)

- Example: Using array as an iterator in the arguments of the `copy` function

```
int numbers[7] = {0, 10, 20, 30, 40, 50, 60};
int small [4] = {0, 0, 0, 0};

int *p = numbers + 2;       // an iterator that starts at numbers[2]
int *mid = numbers + 6;     // an iterator that starts at numbers[6]

int *small_front = small;   // an iterator that starts at small[0]


copy(p, mid, small_front);
```

20	30	40	50	small

```
copy(numbers+4, numbers+7, small);
```

40	50	60	50	small

Iterators Invalidation Rules

Insertion

- **vector**: all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated)
- **list**: all iterators and references unaffected

Erasure

- **vector**: every iterator and reference after the point of erase is invalidated
- **list**: only the iterators and references to the erased element is invalidated

An Iterator for Linked Lists

An Iterator for Linked Lists

The Node Iterator

- We will use the node template class to build various data structures
- **We start by defining iterators that can step through the nodes of a linked list**
- We put this node iterator into `node2.h`, so that any container class that uses a node can also use the node iterator

- **The node iterator has two constructors:**
  - A constructor that **attaches the iterator to a specified node in a linked list**
  - A default constructor that **creates a special iterator that marks the position that is beyond the end of a linked list**
- We will be able to use our iterator to step through a linked list following the usual […) left-inclusive pattern

An Iterator for Linked Lists

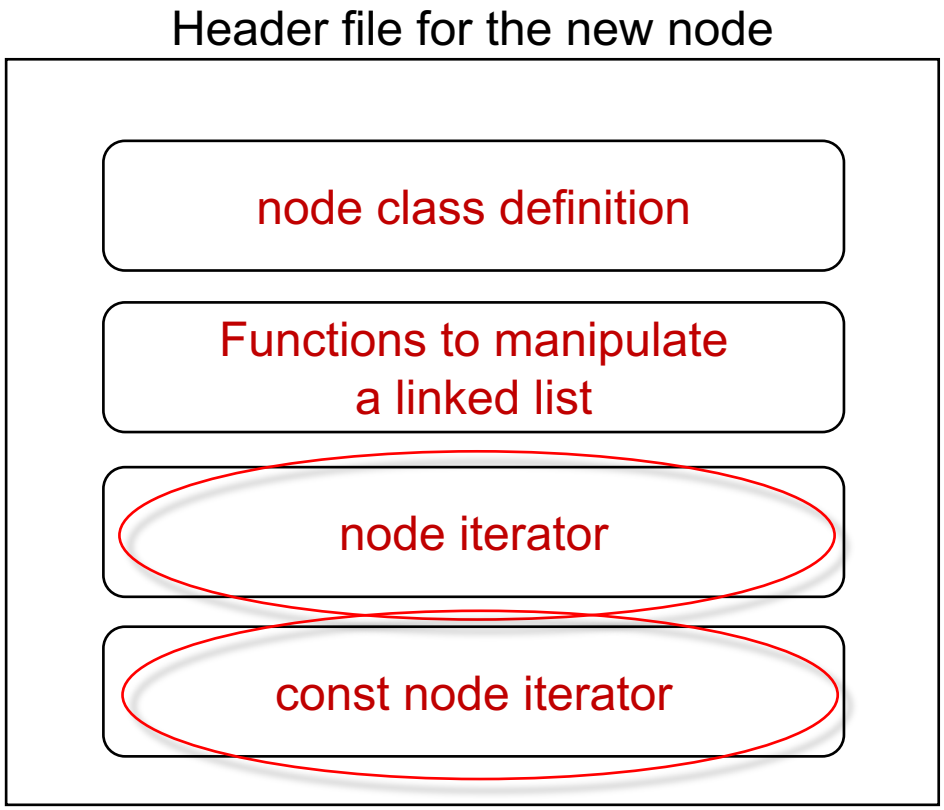
The Node Iterator (Cont'd)

- Example: Suppose that `head_ptr` is the head pointer for a list of integers
- Following loop will step through the list, changing to zero any number that is odd

```
node_iterator<int> start(head_ptr);  // start is the first node
node_iterator<int> finish;           // finish is beyond the end
node_iterator<int> position;         // position moves through list

for (position = start; position != finish; ++position)
{
    if ((*position%2) == 1)   // the number is odd
        *position = 0;        // Change the odd number to zero
}
```

An Iterator for Linked Lists

Header File for the New Node



An Iterator for Linked Lists

Definition of the **Iterator** for the Node Template Class

```
template <class Item>
class node_iterator :
   public std::iterator<std::forward_iterator_tag, Item>
{
```

We plan to create a forward iterator

The data type of the item the iterator refers to

```
// CONSTRUCTOR
public:
  node_iterator(node<Item>* initial = NULL)
     { current = initial; }
```

- The constructor can be called with no argument
- Result: An iterator that is off the end of the list

An Iterator for Linked Lists

Definition of the **Iterator** for the Node Template Class (Cont'd)

```
Item& operator *( ) const
{ return current->data( ); }
```

Pre-increment

```
node_iterator& operator ++( )
{
      current = current->link( );
      return *this;
}
```

Note that we can return a reference because the iterator is still valid when the function returns

❑ **Example:**

An Iterator for Linked Lists

Definition of the **Iterator** for the Node Template Class (Cont'd)

```
node_iterator operator ++(int)
{
     node_iterator original(current);
     current = current->link( );
     return original;
 }
```

Post-increment

Can we change the return type to be a reference?

```
bool operator ==(const node_iterator other) const
{ return current == other.current;}

bool operator !=(const node_iterator other) const
{ return current != other.current; }
```

Two iterators are equal if the `current` variables of the two iterators are equal

```
 private:
       node<Item>* current;
};
```

A pointer to the node that contains the current item

An Iterator for Linked Lists

The Node Iterator is Derived from `std::iterator`

- At the front of our iterator definition, we have the line:

```
: public std::iterator<std::forward_iterator_tag, Item>
```

- It allows our iterator to pick up some features of the Standard Library iterators
- We plan to create a forward iterator so we use the tag `std::forward_iterator_tag`, this will allow Standard Library algorithms (such as `copy`) to determine which operations our iterator has
  - Other tags provided by STL: `input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag, random_access_iterator_tag`
- Inside the angle brackets, we indicate the data type of the items that our iterator will refer to (in our case, `Item`)

An Iterator for Linked Lists

The Node Iterator's Private Member Variable

- Our iterator has one private member variable, `current`
- This is a pointer to the node that contains its current item
- If the iterator has moved beyond the end of a list, then `current` will be null

An Iterator for Linked Lists

Node Iterator — the * Operator

- The node iterator implements the * operation:

```
Item& operator *( ) { return current->data( ); }
```

- For a node iterator p, this operator allows us to use the notation *p to access p's current item
- The function returns a **reference** to the actual item in the node
- The return type of the * operator is also a reference to the item (indicated by the symbol & in the return type of Item&)
- **The return value from *p allows us to both access and change p's current item**

❑Example:

```
cout << *p <<endl;  // prints the value of p's item
*p = 2;             // changes the value of p's item to 2
```

An Iterator for Linked Lists

Node Iterator — Two Versions of the ++ Operator

- The node iterator has two versions of the `++operator` and allows us to write either ++p or p++
- **The prefix version** begins this way:

```
node_iteraror& operator ++()      //Prefix ++
```

- When ++p is used as part of a larger expression, it is important to know that the return value of ++p is the iterator p itself *after* it has already been moved forward

An Iterator for Linked Lists

Node Iterator — Two Versions of the ++ Operator (Cont'd)

```
node_iteraror& operator ++()  //Prefix ++
{
    current = current -> link();
    return *this;
}
```

- When ++p is activated, the first statement moves p's current pointer forward one node
- The second statement is `return *this`
  - **The statement uses the keyword `this`, which is always a pointer to the object that activated the function**

An Iterator for Linked Lists

Node Iterator — Two Versions of the ++ Operator (Cont'd)

- **The postfix version.** begins this way:

```
node_iteraror operator ++(int)    //Postfix ++
```

- Using the keyword `int` (where the parameters usually go) indicated that this is the postfix version of the ++ operator
- This function allows us to write p++, with the ++ following p
- The return value of p++ is a copy of p before it was changed

An Iterator for Linked Lists

Iterators for Constant Collections

Justifying the need for a constant iterator:

- **We must take care when a pointer is declared with the `const` keyword**, because we must forbid any change to the associated linked list

❑ Example: Consider this function, which is supposed to traverse a linked list of integers, adding up the value of all the integers

```
int add_values(const node<int>* head_ptr)
{
    node_iterator<int> start(head_ptr); // start is at the first node
    node_iterator<int> finish;          // finish is beyond the end
    node_iterator<int> position;        // position moves through list
    int sum = 0;

    for (position = start; position != finish; ++position)
    {
        sum +=*position;
    }
    return sum;
}
```

An Iterator for Linked Lists

Iterators for Constant Collections (Cont'd)

- The problem is that `head_ptr` is declared as `const node<int>*`, and therefore it cannot be used as the argument to the constructor of the node iterator
- That constructor has the following prototype, which requires an ordinary pointer to a node:

```
node_iterator(node<Item>* initial);
```

- **The solution is to provide another iterator that can be used with a `const` node:** The **`const_node_iterator`**

- Our `const_node_iterator` differs from the ordinary in a way that each use of the data type `Item` or `node<Item>*` is now written as `const Item` or `const node<Item>*`

An Iterator for Linked Lists

Definition of the **Const Iterator** for the Node Template Class

```
template <class Item>
class const_node_iterator :
        public std::iterator<std::forward_iterator_tag, const Item>
{

 public:
     const_node_iterator(const node<Item>* initial = NULL)
        { current = initial; }


     const Item& operator *( ) const
         { return current->data( ); }


     const_node_iterator& operator ++( )   // Prefix ++
        {
            current = current->link( );
            return *this;
        }
```

The return type is a const reference, so it cannot be used to change the item

An Iterator for Linked Lists

Definition of the **Const Iterator** for the Node Template Class (Cont'd)

```
   const_node_iterator operator ++(int) // Postfix ++
      {
             const_node_iterator original(current);
             current = current->link( );
             return original;
      }

    bool operator ==(const const_node_iterator other) const
     { return current == other.current; }


    bool operator !=(const const_node_iterator other) const
      { return current != other.current; }

 private:
        const node<Item>* current;
    };
}
#include "node2.template"
#endif
```

Linked-List Version of the Bag Template Class with an Iterator

Linked-list Version of The Bag Template Class with Iterator

- We can use the template version of the node class to implement another bag template class using linked list
- Our new version will be a **template** class, making use of **the template version of the linked-list toolkit**

```
template <class Item>
class bag
{

  public:
    ...

  private:
    node<Item> *head_ptr;     // Head pointer for the list
    size_type many_nodes;     // Number of nodes on the list
};
```

Linked-list Version of The Bag Template Class with Iterator

❑Example: Consider this declaration:

```
bag<string> names;
```

- This declares a bag with the `item` instantiated as a `string`
- This bag has a private member variable, `head_ptr`, which is a pointer to a `node<string>`

Linked-list Version of The Bag Template Class with Iterator

How to Provide an Iterator for a Container Class that You Write

Provide these items in the public section of the class definition:

- Add a `typedef` for the **`iterator`** class
- Add a `typedef` for the **`const_iterator`** class
- The container needs a **`begin`** member function, which creates and returns an iterator that refers to the container's first item
  - We will need two versions of the `begin` function: an ordinary version that returns a bag `iterator`, and a constant member function that returns a bag `const_iterator`
- The container needs two member functions that return an `iterator` (or a `const_iterator`), indicating a position that is beyond the end of the container

Linked-list Version of The Bag Template Class with Iterator

Why the Iterator is Defined Inside the Bag

- **By putting the `iterator` class definition inside the definition of the bag template class, the `iterator` becomes a member of the bag class**
- To use this iterator, a program specifies the bag, followed by `::iterator`

❑ Example:

```
bag<int>::iterator position;
```

Linked-list Version of The Bag Template Class with Iterator

Header File for the Fifth Bag Template Class

```
#ifndef SCU_COEN79_BAG5_H
#define SCU_COEN79_BAG5_H
#include <cstdlib>   // Provides NULL and size_t and NULL
#include "node2.h"   // Provides node class

namespace SCU_COEN79_6B
{
   template <class Item>
   class bag
   {
   public:
       // TYPEDEFS
         typedef std::size_t size_type;
         typedef Item value_type;
         typedef node_iterator<Item> iterator;
         typedef const_node_iterator<Item> const_iterator;
```

This enables us to specify the iterator as, for example:
`bag<int>::iterator position;`

Linked-list Version of The Bag Template Class with Iterator

Header File for the Fifth Bag Template Class (Cont'd)

```
// CONSTRUCTORS and DESTRUCTOR
bag( );
bag(const bag& source);
~bag( );


// MODIFICATION MEMBER FUNCTIONS
size_type erase(const Item& target);
bool erase_one(const Item& target);
void insert(const Item& entry);
void operator +=(const bag& addend);
void operator =(const bag& source);


// CONST MEMBER FUNCTIONS
size_type count(const Item& target) const;
Item grab( ) const;
size_type size( ) const { return many_nodes; }
```

Linked-list Version of The Bag Template Class with Iterator

Header File for the Fifth Bag Template Class (Cont'd)

```
// FUNCTIONS TO PROVIDE ITERATORS
iterator begin( )
  { return iterator(head_ptr); }

const_iterator begin( ) const
 { return const_iterator(head_ptr); }
```

Returns an iterator that refers to container's first item

- The return statement will create a temporary `iterator` object, using the `head_ptr` as the argument to the iterator constructor
- A copy of this temporary object is returned by the `begin` member function

```
iterator end( )
   { return iterator( ); }

const_iterator end( ) const
   { return const_iterator( ); }
```

Returns an iterator that refers to a position beyond the end of the container

Linked-list Version of The Bag Template Class with Iterator

Header File for the Fifth Bag Template Class (Cont'd)

```
private:
      node<Item> *head_ptr;     // Head pointer for the list of items
      size_type many_nodes;     // Number of nodes on the list
    };


    // NONMEMBER functions for the bag
    template <class Item>
    bag<Item> operator +(const bag<Item>& b1, const bag<Item>& b2);
}

// The implementation of a template class must be included in its
// header file:
#include "bag5.template"
#endif
```

Linked-list Version of The Bag Template Class with Iterator

node.h

- `node` class definition and implementation
- `node` toolkit
- **`node_iterator` class definition and implementation**
- **`const_node_iterator` class definition and implementation**

bag.h

- `bag` class definition, **including functions to provide iterators**

STL Vectors vs. STL Lists

STL Vectors vs. STL Lists

- STL includes three similar container classes:
  - **Vectors**: uses a dynamic array
  - **Lists**: uses a doubly linked list
  - **Deques:** uses a third mechanism that we will see in the future lectures

STL Vectors vs. STL Lists

vector

```
template < class T, class Alloc = allocator<T> >
class vector;
```

- `T:`
  - Type of the elements
  - Aliased as member type `vector::value_type`
- `Alloc:`
  - Type of the allocator object used to define the storage allocation model
  - **Allocators:**
    - Are an important component of the C++ Standard Library
    - A common trait among STL containers is their ability to change size during the execution of the program
    - To achieve this, some form of dynamic memory allocation is usually required
    - Allocators handle all the requests for allocation and deallocation of memory for a given container

STL Vectors vs. STL Lists

vector (Cont'd)

- Similar to arrays:
  - **vectors use contiguous storage locations for their elements**
  - Elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays
- Unlike arrays:
  - vector size can change dynamically

- vectors use a **dynamically allocated array** to store their elements
  - vectors do not reallocate each time an element is added to the container
  - vector containers may allocate some extra storage to accommodate for possible growth
- vectors provide efficient element access (just like arrays) and relatively efficient adding or removing elements from its end

STL Vectors vs. STL Lists

vector (Cont'd)

- `std::vector::begin`

```
iterator begin();
const_iterator begin() const;
```

- Returns an iterator pointing to the first element in the vector
- If the container is empty, the returned iterator value should not be dereferenced

- `std::vector::end`

```
iterator end();
const_iterator end() const;
```

- Returns an iterator referring to the **past-the-end** element in the vector container

STL Vectors vs. STL Lists

vector (Cont'd)

```
// vector::begin/end
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int>  myvector;
    for (int i=1; i<=5; i++)  myvector.push_back(i);

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin() ;
                                     it != myvector.end(); ++it)
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

```
Output:
myvector contains: 1 2 3 4 5
```

STL Vectors vs. STL Lists

vector (Cont'd)

```
void push_back (const value_type& val);
```

- Adds a new element **at the end of the vector**
- This effectively increases the container size by one, which causes an **automatic reallocation** of the allocated storage space **if -and only if- the new vector size surpasses the current vector capacity**

STL Vectors vs. STL Lists

vector (Cont'd)

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;
    int myint;

    std::cout << "Please enter some integers (enter 0 to end):\n";

    do {
         std::cin >> myint;
       myvector.push_back (myint);
     } while (myint);

    std::cout << "myvector stores " << myvector.size() <<
                                          " numbers.\n";

    return 0;
}
```

STL Vectors vs. STL Lists

vector (Cont'd)

```
void pop_back();
```

- **Removes the last element in the vector**, effectively reducing the container size by one

`back()`: Returns a reference to the last element in the vector

```
#include <iostream>
#include <vector>
int main ()
{
   std::vector<int> myvector;
   int sum (0);
   myvector.push_back (100);
   myvector.push_back (200);
   myvector.push_back (300);

   while (!myvector.empty())
   {
      sum += myvector.back();
      myvector.pop_back();
   }

   std::cout << "The elements of
   myvector add up to " << sum << '\n';
   return 0;
}
```

```
Output:
The elements of
myvector add up to 600
```

STL Vectors vs. STL Lists

vector (Cont'd)

```
template <class InputIterator>
void assign (InputIterator first, InputIterator last);


void assign (size_type n, const value_type& val);
```

- Assigns new contents to the vector, replacing its current contents, and modifying its size accordingly
- **`first, last`**
  - Input iterators to the initial and final positions in a sequence
  - The range used is [first, last)
- **`n`**: New size for the container
- **`val`**
  - Value to fill the container with
  - Each of the *n* elements in the container will be initialized to a copy of this value

STL Vectors vs. STL Lists

vector (Cont'd)

```
#include <iostream>
#include <vector>

int main ()
{
   std::vector<int> first, second, third;

   first.assign (7,100);
```

Assigns 7 ints with a value of 100 to `first`


```
   std::vector<int>::iterator it;
   it = first.begin() + 1;
```


```
   second.assign (it, first.end()-1);
```

Assigns the five central values of `first` to `second`
Note: `end-1` is not assigned

STL Vectors vs. STL Lists

vector (Cont'd)

```
    int myints[] = {1776, 7, 4};
    third.assign (myints, myints+3);


    std::cout << "Size of first: " << first.size() << '\n';
    std::cout << "Size of second: " << second.size() << '\n';
    std::cout << "Size of third: " << third.size() << '\n';

    return 0;

}
```

Assigns all the elements of `myints` to `third`

```
Output:
Size of first: 7
Size of second: 5
Size of third: 3
```

STL Vectors vs. STL Lists

vector (Cont'd)

```
iterator insert (iterator position, const value_type& val);

void insert (iterator position, size_type n, const value_type& val);

template <class InputIterator>
void insert (iterator position, InputIterator first,
                                                InputIterator last);
```

- **`position`**: Position in the vector where the new elements are inserted
- **`val`**: Value to be copied (or moved) to the inserted elements
- **`n`**: Number of elements to insert; Each element is initialized to a copy of `val`
- **`first, last:`** Iterators specifying a range of elements; Copies of the elements in the range `[first,last)` are inserted at `position` (in the same order)

STL Vectors vs. STL Lists

vector (Cont'd)

```
// inserting items into a vector
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector (3,100);

    std::vector<int>::iterator it;


    it = myvector.begin();
    it = myvector.insert( it, 200 );


    myvector.insert (it, 2, 300);
```

STL Vectors vs. STL Lists

vector (Cont'd)

```
// we need to get a new iterator, because "it" is no longer valid
it = myvector.begin();
```

- An Iterator is invalidated when the container it points to changes its shape internally
- e.g., Move elements from one location to another and the initial iterator still points to old invalid location

- **Rule: For "vector", all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated)**

STL Vectors vs. STL Lists

vector (Cont'd)

```
std::vector<int> anothervector (2,400);
```

400	400

```
myvector.insert (it+2,anothervector.begin(),anothervector.end());
```

300	300	400	400	200	100	100	100

```
int myarray [] = { 501, 502, 503 };
myvector.insert (myvector.begin(), myarray, myarray+3);
```

501	502	503	300	300	400	400	200	100	100	100

```
std::cout << "myvector includes these elements:";
for (it=myvector.begin(); it<myvector.end(); it++)
   std::cout << ' ' << *it;

std::cout << '\n';

return 0;
}
```

Output:

```
myvector contains these elements: 501
502 503 300 300 400 400 200 100 100
100
```

STL Vectors vs. STL Lists

vector (Cont'd)

```
iterator erase (iterator position);
iterator erase (iterator first, iterator last);
```

- Removes from the vector either a single element (`position`) or a range of elements (`[first,last)`)
- Invalidates iterators and references at or after the point of the erase, including the end() iterator

STL Vectors vs. STL Lists

vector (Cont'd)

```
#include <iostream>
#include <vector>
int main ()
{
    std::vector<int> myvector;
    for (int i=1; i<=10; i++) myvector.push_back(i);

    //erase the 6th item of the vector
    myvector.erase (myvector.begin()+5);

    //erase the first three items of the vector
    myvector.erase (myvector.begin(), myvector.begin()+3);

    std::cout << "myvector includes these items:";
    for (unsigned i=0; i<myvector.size(); ++i)
        std::cout << ' ' << myvector[i];

    std::cout << '\n';
    return 0;
}
```

Output:

myvector includes these items:

4 5 7 8 9 10

STL Vectors vs. STL Lists

list

```
template < class T, class Alloc = allocator<T> >
class list;
```

- `lists`
  - Allow constant time insert and erase operations anywhere within the sequence
  - **It is a doubly linked list**
  - **Allows iteration in both directions**

- `list` is very similar to forward_list
  - However, `forward_list` objects are single-linked lists, and thus they can only be iterated forwards

STL Vectors vs. STL Lists

list (Cont'd)

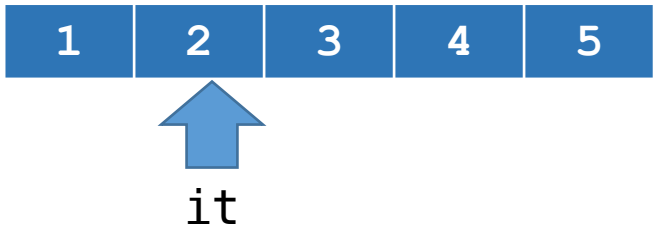
```
iterator insert (iterator position, const value_type& val);


void insert (iterator position, size_type n, const value_type& val);


template <class InputIterator>
void insert (iterator position, InputIterator first,
                                                InputIterator last);
```

STL Vectors vs. STL Lists

list (Cont'd)

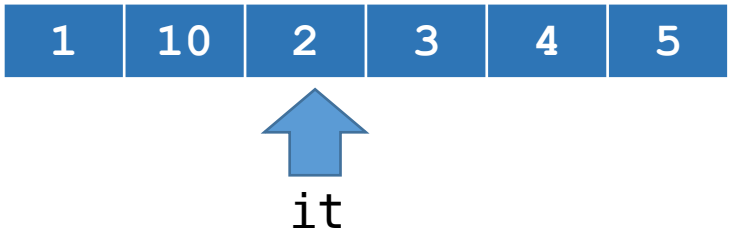
```
// inserting items into a linked list
#include <iostream>
#include <list>
#include <vector>

int main ()
{
    std::list<int> mylist;
    std::list<int>::iterator it;

   for (int i=1; i<=5; ++i) mylist.push_back(i);

    it = mylist.begin();
    ++it;

   mylist.insert (it,10);
```

Note that the items of a liked list are not necessarily stored contiguously



STL Vectors vs. STL Lists

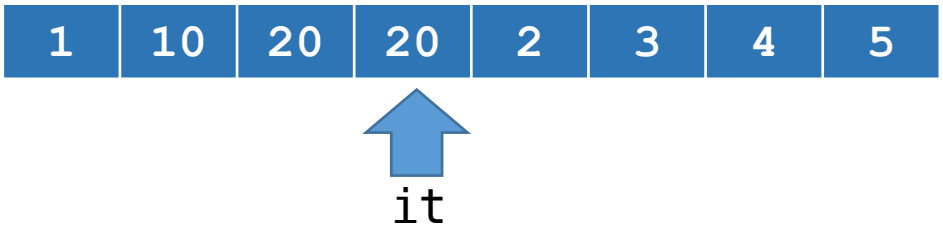
list (Cont'd)



Rule: For `list`, all iterators and references are unaffected after an insertion

```
mylist.insert (it, 2, 20);
```


```
--it;
```




```
std::vector<int> myvector (2,30);
mylist.insert (it, myvector.begin(), myvector.end());
```

STL Vectors vs. STL Lists

list (Cont'd)

```
    std::cout << "mylist included these items:";

    for (it=mylist.begin(); it!=mylist.end(); ++it)
        std::cout << ' ' << *it;


    std::cout << '\n';


    return 0;
}
```

```
Output:
mylist contains: 1 10 20 30 30 20 2 3 4 5
```

STL Vectors vs. STL Lists

list (Cont'd)

```
iterator erase (iterator position);
iterator erase (iterator first, iterator last);
```

- Removes from the list container either a single element (`position`) or a range of elements (`[first, last)`)

Return value:

- An iterator pointing to **the element that followed the last element erased by the function call**
- The container "end" is returned if the operation erased the last element in the sequence

STL Vectors vs. STL Lists

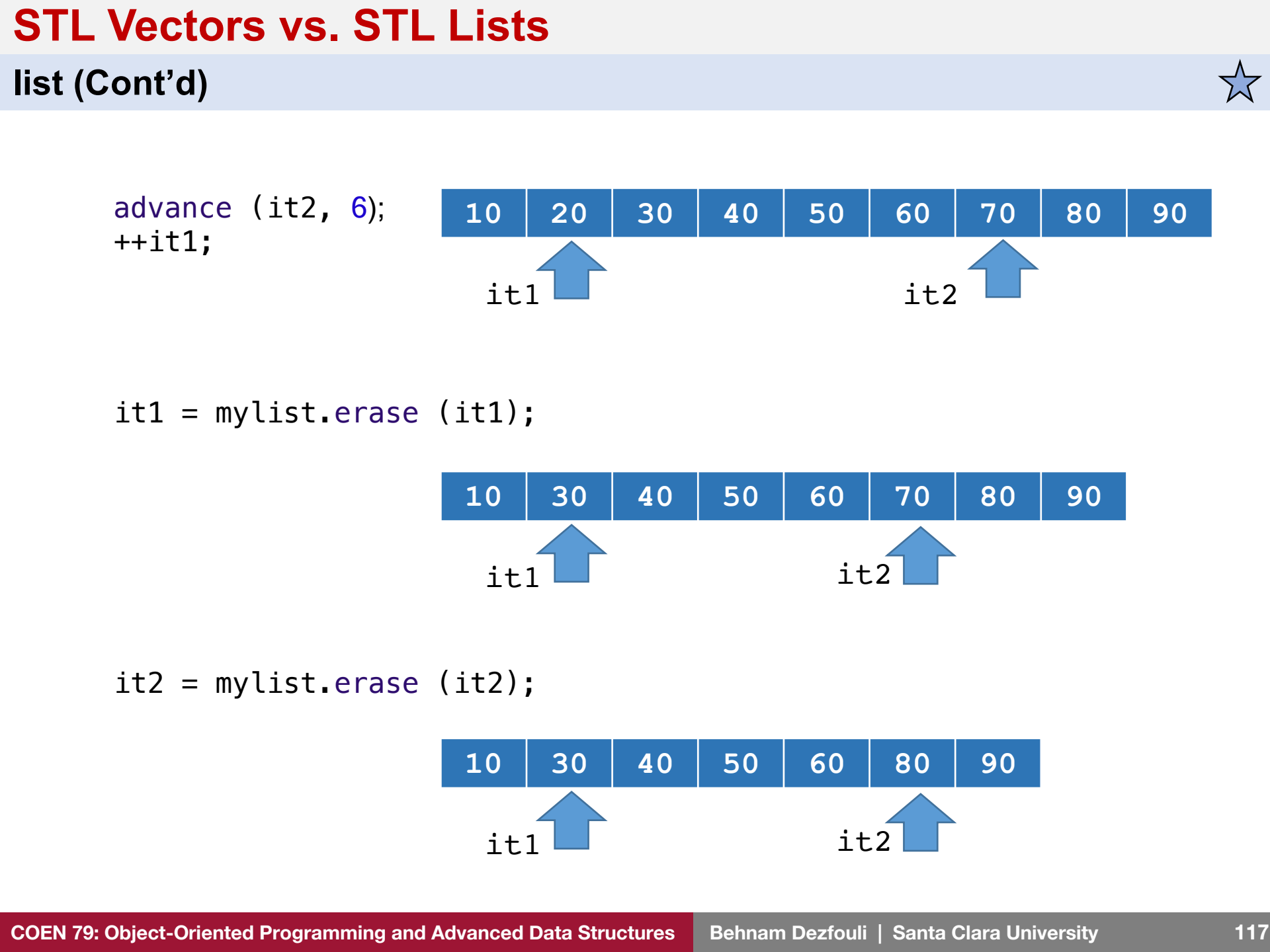
list (Cont'd)

```
// erasing from list
#include <iostream>
#include <list>

int main ()
{
    std::list<int> mylist;
    std::list<int>::iterator it1,it2;

    // set some values:
   for (int i=1; i<10; ++i) mylist.push_back(i*10);
```


```
   it1 = it2 = mylist.begin();
```

STL Vectors vs. STL Lists

list (Cont'd)



```
advance (it2, 6);
++it1;
```



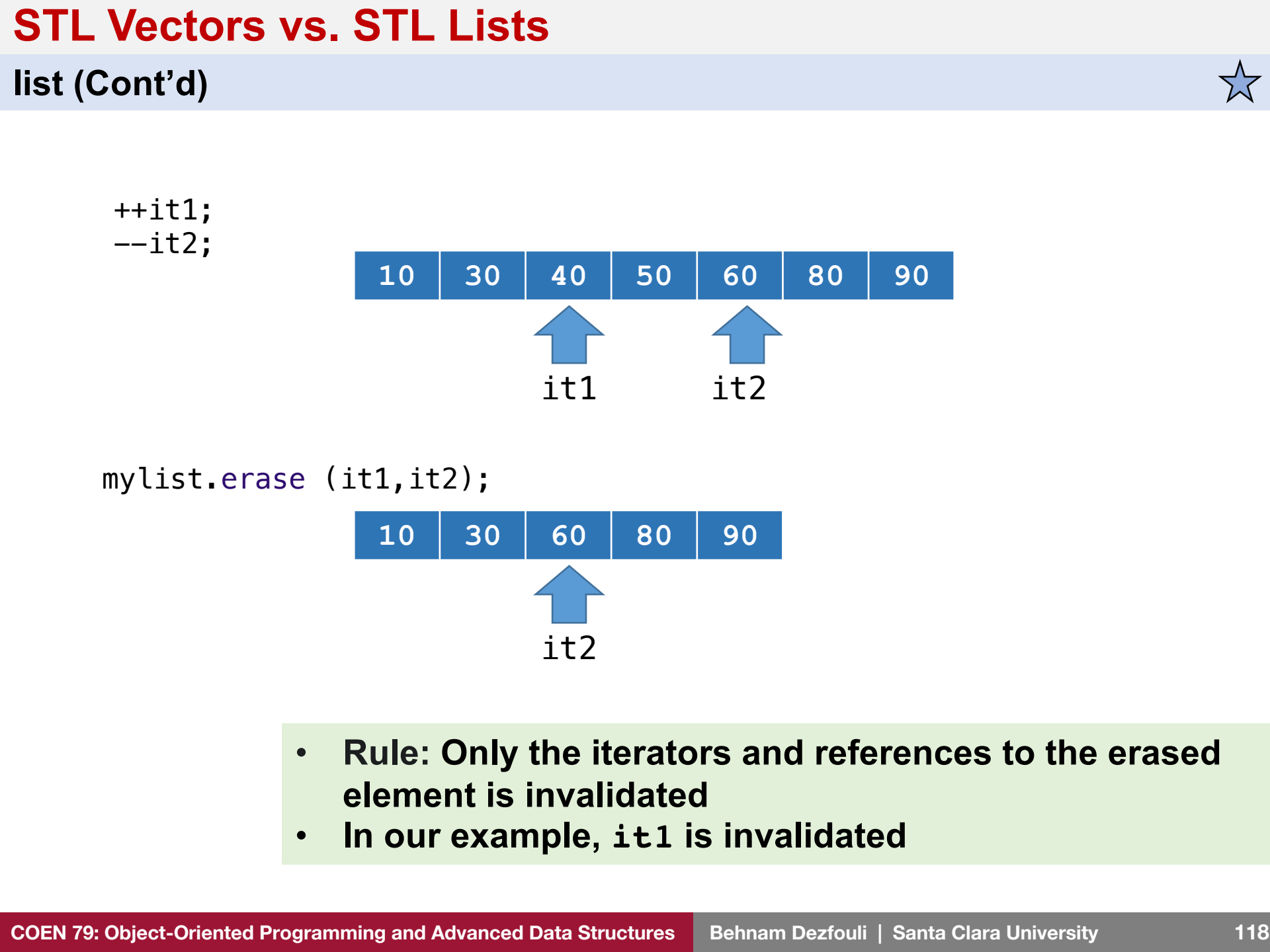
```
it1 = mylist.erase (it1);
```



```
it2 = mylist.erase (it2);
```

STL Vectors vs. STL Lists

list (Cont'd)



```
++it1;
--it2;
```



```
mylist.erase (it1,it2);
```


- **Rule: Only the iterators and references to the erased element is invalidated**
- **In our example, `it1` is invalidated**

STL Vectors vs. STL Lists

list (Cont'd)

```
    std::cout << "mylist includes these items:";

    for (it1 = mylist.begin(); it1 != mylist.end(); ++it1)
       std::cout << ' ' << *it1;


   std::cout << '\n';

    return 0;
}
```

```
Output:
mylist contains: 10 30 60 80 90
```

STL Vectors vs. STL Lists

- With your knowledge of dynamic arrays and linked lists, you can figure out why there are certain differences between the vector and the list classes

❑ Example:

  - `list` has versions of `push` and `pop` that work at the front of the linked list
  - It is easy to add or remove an item from the front of a linked list
    - This is not easy for a vector, so `pop_front` and `push_front` are not even part of the vector class

- The index access functions are provided only for the `vector`
- Some functions, such as `insert`, are provided for both containers, but one version will be more efficient than the other

Summary

Summary

- A **template function** is similar to an ordinary function with one important difference: The definition of a template function depends on an underlying data type
- When a template function is used, the compiler examines the types of the arguments during **instantiation** of the template
- In a single program, several different usages of a template function can result in several different instantiations
- When a class depends on an underlying data type, the class can be implemented as a **template class**
- An **iterator** allows a programmer to easily step through the items of a container class