Software Development with Templates, Iterators, and the STL

Learning Objectives

- Understand the importance of template functions and template classes
- Design and implement template functions and template classes
- Use iterators to step through all the elements of an object for any of the STL classes
- Manipulate objects of the STL classes using functions from the <algorithms> library facility
- Implement simple forward iterators for data structures

TEMPLATE FUNCTIONS

Finding the Maximum of Two Integers

- Here's a small function that you might write to find the maximum of two integers

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

Finding the Maximum of Two Doubles

- Here's a small function that you might write to find the maximum of two double numbers

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

Finding the Maximum of Two …

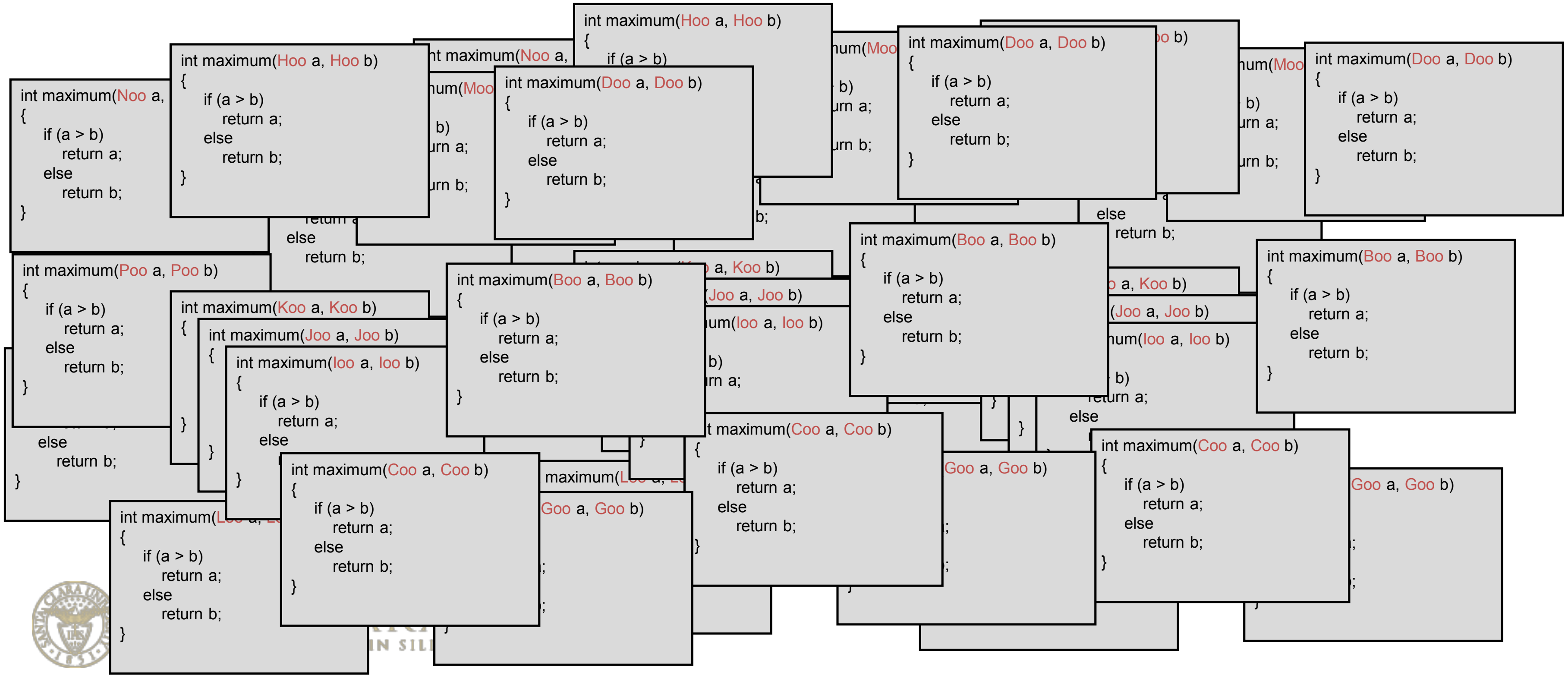
- Here's a small function that you might write to find the maximum of two …using typedef

```
typedef   int   data_type

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

But you need to re-compile your program every time you change the data_type, and you still only have one kind of data type

One Hundred Million Functions...

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



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

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

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

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

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

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

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

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

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

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

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

A Template Function for Maximum

- This template function can be used with many data types.

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

Item:

Underlying data type, template parameter

Using a Template Function

- Once a template function is defined, it may be used with any adequate data type in your program...

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

```
cout << maximum(1,2);
cout << maximum(1.3, 0.9);
...
```

Finding the Maximum Item in an Array

- Here's another function that can be made more general by changing it to a template function:

```
int array_max(int data[  ], size_t n)
{
    size_t i;
    int answer;

    assert(n > 0);
    answer = data[0];
    for (i = 1; i < n; i++)
        if (data[i] > answer) answer = data[i];
    return answer;
}
```

Finding the Maximum Item in an Array

- Here's the template function:

```
template <class Item>
Item array_max(Item data[  ], size_t n)
{
    size_t i;
    Item answer;

    assert(n > 0);
    answer = data[0];
    for (i = 1; i < n; i++)
        if (data[i] > answer) answer = data[i];
    return answer;
}
```

Failed Unification Errors

- Template parameter must appear in the parameter list of the template function
  - Example

    ```
    maximum(Item a, Item b)
    ```

  - Without this rule, the compiler cannot figure out how to instantiate the template function when it is used
  - 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

A Template Function to Swap Two Values

```
template <class Item>
void swap (Item& x, Item& y)
{
    Item temp = x;
    x = y;
    y = temp;
}
```

```
string name1 (”scu");
string name2 (”coen");
swap(name1, name2);
cout << name1;
```

- 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 template functions

Parameter Matching for Template Functions

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

- 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
  - The second argument must be a size_t
  - Many compilers cannot accept any deviation
  - Invalid: int, const size_t

Parameter Matching for Template Functions (Cont.)

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

These won't work with many compilers.

A Template Function to Find the Biggest Item in an Array

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

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

A Template Function to Find the Biggest Item in an Array (cont.)

```
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;
}
```

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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

Syntax for a Template Class

1. Change the template class definition

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

```
template <class Item>
class bag
{
public:
    typedef Item value_type;
            ...
```

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

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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:

- The template prefix `template <class Item>` is placed immediately before each function prototype and definition
- Each use of the class name (such as bag) is changed to `bag<Item>`
- Within a class definition or within a member function, `value_type` or `size_type`, or `Item`
- Outside of a member function, `bag<item>::size_type`

Syntax for a Template Class (Cont'd)

- Non-member function

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

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

- Member function

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

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

Use the Name Item and the `typename` Keyword

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

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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**

- 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
- Include the following line at the end of the header file

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

NOTE: Do not place using directives in a template implementation

Parameter Matching for Member Functions of Template Classes

- Remember: 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, 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)

Using the Template Class

- character bag
  - The template parameter is instantiated as a character

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

- double bag

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

- string bag
  - need <string> header file

```
bag<string> verbs;
```

Demonstration Program for the Bag Template Class

Details of the Story-Writing Program

- A function, *get_items*, in the story-writing program is a template function with this specification:

```
template <class Item, class SizeType, class MessageType>
void get_items(bag<Item>& collection, SizeType n,
                                             MessageType description)
//Postcondition: The description has been written as a prompt to the
//screen. Then n items have been read from cin and added to the
//collection.
```

- The template prefix indicates three classes:
  - `class Item`: This is the type of the item in the bag
  - `class SizeType`: This is the type of n. It may be any integer data type such as int or size_t
  - `class MessageType`: This may be any printable data type such as a string constant or a string variable

```
#include<algorithm>
#include<iostream>

template <class T>
inline T & Max (T & a, T & b)  { return a < b ? b:a; }

int main () {
  int i = 39;
  int j = 20;
  std::cout << "Max(i, j): " << Max(i, j) << endl;

  double f1 = 13.5;
  double f2 = 20.7;
  std::cout << "Max(f1, f2): " << Max(f1, f2) << endl;

  string s1 = "Hello";
  string s2 = "World";
  std::cout << "Max(s1, s2): " << Max(s1, s2) << endl;

  return 0;
}
```

- Class definition

```
template <class A_Type>
class calc
{
public:
   A_Type multiply(A_Type x, A_Type y);
   A_Type add(A_Type x, A_Type y);
};
```

- Class member function implementaion

```
template <class A_Type>
A_Type calc<A_Type>::multiply(A_Type x,A_Type y) { return x*y; }

template <class A_Type>
A_Type calc<A_Type>::add(A_Type x, A_Type y) { return x+y; }
```

- In main,

```
calc <double> a_calc_class;
```

THE STL’S ALGORITHMS AND USE OF ITERATORS

Standard Template Library (STL)

- The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.
  - copy (function template)
  - swap (function template)
  - max, min (function template)
  - …

Standard Template Library (STL)

- Sequence containers (class template)
  - array, vector, deque (Double ended queue), forward_list , list
- Container adaptors (class template)
  - stack, queue, priority_queue
- Associative containers (class template)
  - set, multiset, map, multimap
- Unordered associative containers (class template)
  - unordered_set, unordered_multiset, unordered_map, unordered_multimap

STL Vectors

```
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

- 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

❖ 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

```
// 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
```

```
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**

```
#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;
}
```

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

STL 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

- 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

STL Stack

```
template < class T, class Container = deque<T> >
class stack;
```

Stacks are implemented as *containers adaptors*

Containers adaptors are classes that use an encapsulated object of a specific container class as its *underlying container, providing a specific set* of member functions to access its elements

The container shall support the following operations:

- `empty`
- `size`
- `back`
- `push_back`
- `pop_back`

The standard container classes vector, deque and list fulfill these requirements

STL Stack (cont.)

```
#include <iostream>          // std::cout
#include <stack>             // std::stack

int main ()
{
   std::stack<int> mystack;

   mystack.push(1);
   mystack.push(2);

   mystack.top() += 10;

    std::cout << "mystack.top() is now " << mystack.top() <<
'\n';

    return 0;
}
```

STL Queue

```
template <class T, class Container = deque<T> >
class queue;
```

Queues are implemented as *containers adaptors*

The underlying container must support at least the following operations:

- empty
- size
- front
- back
- push_back
- pop_front

The standard container classes deque and list fulfill these requirements

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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

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

Input Iterator (cont.)

```
#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> iit (std::cin);    // stdin iterator


    value1 = *iit;

     ++iit;
    value2 = *iit;

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

   return 0;
}
```

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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()

Output Iterator (cont.)

```
#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;
  }
```

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

Forward Iterator (cont.)

```
#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();


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

   return 0;
}
```

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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

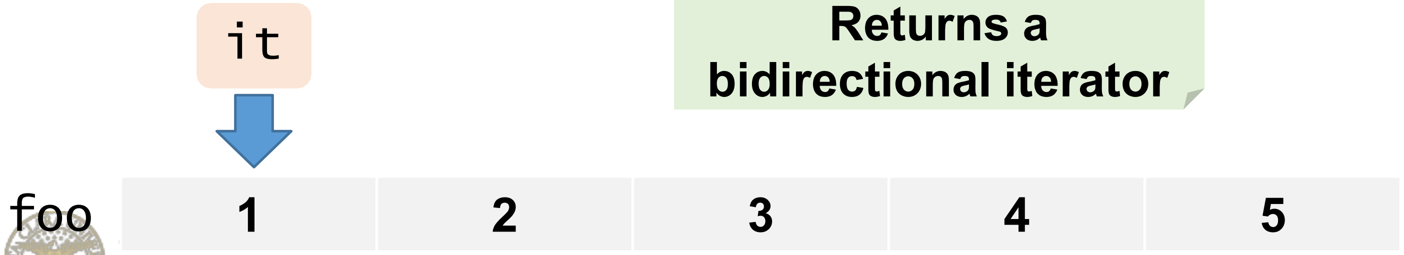
```
#include <iostream>      // std::cout
#include <iterator>      // std::insert_iterator
#include <list>          // std::list
#include <algorithm>     // std::copy

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();
```

$\underbrace{\texttt{foo.begin()}}_{\text{Returns a bidirectional iterator}}$

Returns a bidirectional iterator



```
  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;
}
```

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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

Iterators for Arrays (cont.)

- 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

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[5]

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

copy(p, mid, small_front);

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

THE NODE TEMPLATE CLASS

The Node Template Class

Original Node Class:

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

New Template Node Class:

```
template <class Item>
class node
{
public:
   typedef Item value_type;
      . . .
```

The Node Template Class (cont.)

- The original prototype for the `list_insert` function 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 (cont.)

- `list_locate`

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

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

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

Functions That Return a Reference Type

❖ A change that will simplify later usages of the node

- For the member function that retrieves a copy of the node's data field:

```
Item data() const {return data_field;}
```

- It returns only a **copy** of `data_field`

- We can add the & symbol to the return type, and alter the function so that it is no longer a const function:

```
Item& data() {return data_field;}
```

Functions That Return a Reference Type (Cont.)

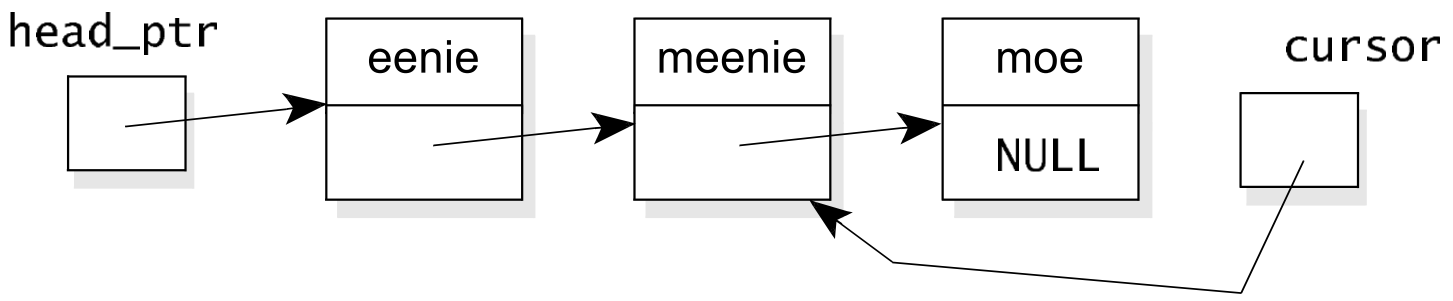
- The use of a reference type has these effects:

  1. The return value must be a variable or object that will still exist after the function returns. The return value must not be a local variable

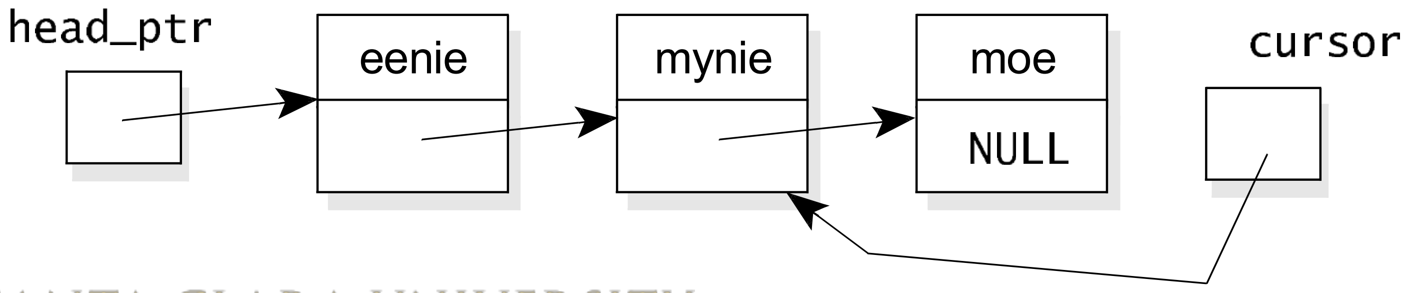
  2. The function returns this actual variable or object (not a copy of the object)

Functions That Return a Reference Type (Cont.)

- Suppose that `cursor` is pointing to the second node of this **linked list of strings**:



- By Executing `cursor->data() = "mynie";`



SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

Data Member Function Requires Two Versions

- Changing the `data()`'s return value to a reference type has a drawback: **This version of the `data` function can no longer be used as a constant member function**
- Solution: Provide a second constant version of the `data` function

```
const Item& data( ) const{ return data_field; }
```

  - The return value refers directly to the node's data field, but because of the `const` keyword, it cannot be used to change that data field
  - Since the return value cannot be used to change the node, the function can be declared as a constant member function

- If pointer `p` is a pointer to a `const` node, then `p->data()` will activate the `const` version of `data`; otherwise the `non-const` version is used

AN ITERATOR FOR LINKED LISTS

The Node Iterator

- Use the node template class to build various data structures
- Start by defining iterators that can step through the nodes of a linked list
- 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**

The Node Iterator (cont.)

- Example: Suppose that `head_ptr` is the head pointer for a list of integers
- Following loop steps 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
}
```

Definition of Node Iterator Class

```
template <class Item>
class node_iterator
: public std::iterator<std::forward_iterator_tag, Item>
{
public:
        node_iterator(node<Item>* initial = NULL)       { current = initial; }
        Item& operator *( ) const       { return current->data( ); }
        node_iterator& operator ++( ) {         // Prefix ++
                current = current->link( );
                return *this;   }
        node_iterator operator ++(int)  {       // Postfix ++
                node_iterator original(current);
                current = current->link( );
                return original;        }
        bool operator ==(const node_iterator other) const   { return current == other.current; }
        bool operator !=(const node_iterator other) const   { return current != other.current; }
private:
        node<Item>* current;
};
```

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

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

```
: 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`
  - 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`)

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

Inheritance

```
#include <iostream>
using namespace std;

class Polygon {
 protected:
   int width, height;
 public:
   void set_values (int a, int b)
     { width=a; height=b;}
};

class Rectangle: public Polygon {
 public:
   int area ()
     { return width * height; }
};

class Triangle: public Polygon {
 public:
   int area ()
     { return width * height / 2; }
};

int main () {
 Rectangle rect;
 Triangle trgl;
 rect.set_values (4,5);
 trgl.set_values (4,5);
 cout << rect.area() << '\n';
 cout << trgl.area() << '\n';
 return 0;
}
```

Node Iterator — the * Operator

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

- 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**

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

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

Node Iterator — Two Versions of the ++ Operator

- The Prefix version

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

- 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

Node Iterator — Two Versions of the ++ Operator

- The Postfix version

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

- Using the keyword `int` (where the parameters usually go) indicated that this is the postfix version of the ++ operator
- The return value of p++ is a copy of p before it was changed

Iterators for Constant Collections

- We must take care when a pointer is declared with the **`const`** keyword
- Example

```
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;
}
```

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
  - The constructor has 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: **`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>*`

Definition of a Const Iterator

```
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;    }
    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;
};
```

LINKED-LIST VERSION OF THE BAG TEMPLATE CLASS WITH AN ITERATOR

Linked-list Bag Template with an 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
};
```

How to Provide an Iterator for a Container Class

- 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
    - ✓ 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

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

Five Bag Classes

Approach	Define item with	Files
Store the items in an array with a fixed size	typedef	bag1.h and bag1.cpp
Store the items in a dynamic array	typedef	bag2.h and bag2.cpp
Store the items in a linked list, using the node class	typedef	bag3.h and bag3.cpp
Store items in a dynamic array	template parameter	bag4.h and bag4.template
Store items in a linked list, using the template version of the node class. This implementation also has an iterator.	template parameter	bag5.h and bag5.template

Copyright Notice

Part of this lecture was adapted from the slides of Dr. Behnam Dezfouli

SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY