Lecture 9
Templates, Iterators and STL

Instructor: George Wolberg
Department of Computer Science
City College of New York
Topics

- **Template Functions and Template Classes**
  - for code that is meant to be reused in a variety of settings in a single program

- **Iterators**
  - step through all items of a container in a standard manner

- **Standard Template Library (STL)**
  - the ANSI/ISO C++ Standard provides a variety of container classes in the STL
Chapter 6 introduces templates, which are a C++ feature that easily permits the reuse of existing code for new purposes.

This presentation shows how to implement and use the simplest kinds of templates: template functions.
Here’s a small function that you might write to find the maximum of two integers.

```c
int maximum(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```
Here’s a small function that you might write to find the maximum of two `double` numbers.

```c
double maximum(double a, double b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```
Here’s a small function that you might write to find the maximum of two Gongfus.

```c
Gongfu maximum(Gongfu a, Gongfu b) {
    if (a > b) return a;
    else return b;
}
```
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;
}
Suppose your program uses 100,000,000 different data types, and you need a maximum function for each...
A Template Function for Maximum

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

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

Item:
Underlying data type, template parameter
With two features...
When you write a template function, you choose a data type for the function to depend upon...

template <class Item>
Item maximum(Item a, Item b)
{
    if (a > b)
        return a;
    else
        return b;
}
A template prefix is also needed immediately before the function’s implementation:

```cpp
template <class Item>
Item maximum(Item a, Item b)
{
    if (a > b)
        return a;
    else
        return b;
}
```
Once a template function is defined, it may be used with any adequate data type in your program...

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

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

What’s behind the scene?
Here’s another function that can be made more general by changing it to a template function:

```c
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;
}
```
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;
}
A template function depends on an underlying data type – the template parameter.

More complex template functions and template classes are discussed in Chapter 6.
Template Classes

- How to turn our node class into node template class
  - template <class Item> precedes the node class definition
  - value_type -> Item
  - Outside the template class definition
    - template prefix precedes each function prototype and implementation
    - node -> node <Item>

- Exercise: Turn node into node template class
  - handout node1 ....then node2
Template Classes

- How to turn our node class into node template class (continued)
  - The implementation file name with `.template` extension (instead of `.cpp`) – cannot be compiled!
  - it should be included in the header by
    - `#include “node2.template”`
  - eliminate any using directives in the implementation file, so you must write
    - `std::size_t, std::copy, etc.`
  - More changes ... please read Chapter 6
Template Classes

How to use it?

```
node<int>* ages = NULL;
list_head_insert(ages,18);
node<string> name;
name.set_data("Jorge");
node<point> *seat;
seat = new node<point>;
(*seat).set_data(point(2,4));
```
All you need to know about Templates

- Template Function
  - a template prefix before the function implementation
  - template <class Item1, class Item2, ...>

- Function Prototype
  - a template prefix before the function prototypes

- Template Class
  - a template prefix right before the class definition

- Instantiation
  - template functions/classes are instantiated when used

Better Understanding of classes and functions
Write a small program `n2demo.cpp` with the lines in the previous slide, make sure you have the correct include and using directives. Then print out the data in `node *ages`, `name` and `*seat`.

Try to run the program with `point.h`, `point.cpp` (online with lecture 3)

`node2.h`, `node2.template` (online today)

Note: you only need to compile `point.cpp` with your `n2demo.cpp`

Turn in `n2demo.cpp` and the output in paper version on Wednesday
Iterators

- We are going to see how to build an iterator for the linked list
- so that each of the containers can build its own iterator(s) easily
- A node_iterator is an object of the node_iterator class, and can step through the nodes of the linked list
Reviews: Linked Lists Traverse

- How to access the next node by using link pointer of the current node
- the special for loop still works with template

```cpp
template <class Item>
std::size_t list_length(const node<Item>* head_ptr)
{
    const node<Item>* cursor;
    std::size_t count = 0;
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link())
        count++;
    return count;
}
```
It would be nicer if we could use an iterator to step through a linked list following the [...] left-inclusive pattern

```cpp
template <class Item>
std::size_t list_length(const node<Item>* head_ptr)
{
    const_node_iterator<Item> start(head_ptr), finish, position;
    std::size_t count = 0;
    for (position = start; position != finish; ++position)
        count++;
    return count;
}
```
node_iterator key points:

- Derived from std::iterator (may NOT exist!)  
  - node_iterator<Item> position;
- A private variable - a pointer to current node  
  - node<Item>* current;
- * operator – get the current data  
  - using the notation *position
- Two versions of the ++ operator  
  - Prefix version: ++position; Postfix ver: position++
- Comparison operators == and !=
- Two versions of the node_iterator  
  - node_iterator and const_node_iterator
Linked List Version the bag Template Class with an Iterator

- Most of the implementation of this new bag is a straightforward translation of the bag in Chapter 5 that used an ordinary linked list

- Two new features
  - Template class with a underlying type Item
  - iterator and const_iterator – defined from node_iterator and const_node_iterator, but use the C++ standard [...] left inclusive pattern
The C++ standard [\ldots] pattern

- You can use an iterator to do many things!

```cpp
bag<int> b;
bag<int>::iterator position; // this iterator class is defined in the bag class
std::size_t count = 0;

b.insert(18);
...
for (position = b.begin(); position != b.end(); ++position) // step through nodes
{
    count++;
    cout « *position « endl; // print the data in the node
}
```
Standard Template Library (STL)

- The ANSI/ISO C++ Standard provides a variety of container classes in the STL
  - set, multiset, stack, queue, string, vector
- Featured templates and iterators
- For example, the multiset template class is similar to our bag template class
- More classes summarized in Appendix H
Summary

- Five bag implementations
- A template function depends on a underlying data type (e.g. Item) which is instantiated when used.
- A single program may have several different instantiations of a template function
- A template class depends on a underlying data type
- A iterator allows a programmer to easily step through the items of a container class
- The C++ STL container classes are all provided with iterators