

# Ch 6

## Templates, Iterators & the STL

- Template Functions
- Template Classes
- The STL and Iterators
- The Node Template Class
- An Iterator for Linked Lists
- The Bag Class, Again

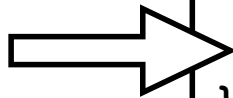
# Template Functions

A C++ Solution to the Genericity Problem

# The Overloading Approach

Overloading is sufficient to meet all functional needs for genericity but may require lots of work.

It's also subject to automatic type conversion problems.



```
int max(int a, int b) {
    return (a > b) ? a : b;
}

double max(double a, double b) {
    return (a > b) ? a : b;
}

int main() {
    cout << max(1, 2) << endl;
    cout << max(1.1, 2.2) << endl;
    //! cout << max(1, 2.2) << endl;
}
```

# The `typedef` Approach

Using `typedef` solves the automatic type conversion problem, but sacrifices intraprogram genericity.

Oops!



```
typedef double item;

item max(item a, item b) {
    return (a > b) ? a : b;
}

int main() {
    bag bag1, bag2;

    cout << max(1, 2) << endl;
    cout << max(1.1, 2.2) << endl;
    cout << max(1, 2.2) << endl;
    //! cout << max(bag1, bag2) << endl;
}
```


# The `template` Approach

Templates reduce the work involved in overloading but, are subject to exact-type matching problems.

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    bag bag1, bag2;

    cout << max(1, 2) << endl;
    cout << max(1.1, 2.2) << endl;
    //! cout << max(1, 2.2) << endl;
    cout << max(bag1, bag2) << endl;
}
```



# The Solution

Provide separate type parameters for each of the function's parameters even if you expect their types to be the same.

```
template <typename T1, typename T2>
T1 max(T1 a, T2 b) {
    return (a > b) ? a : b;
}

int main() {
    bag bag1, bag2;

    cout << max(1, 2) << endl;
    cout << max(1.1, 2.2) << endl;
    cout << max(1, 2.2) << endl;
    cout << max(bag1, bag2) << endl;
}
```

Do the same even for primitive types.

# Template Instantiation

- Each template function (and class) must be instantiated by the compiler for each different parameter list used.
- The full source of templates must be available to the client code before any use.
- Separate compilation is not possible and the implementation, therefore, cannot be hidden.

# Template Classes

Genericity for Objects

# The Old and the New

The bag's contained type was set with a `typedef`.

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

The main difference now is the parameterized type.

```
template <typename T>  
class bag {  
    public:  
        typedef T value_type;  
        // ...  
}
```

# Syntax for a Template class

When implementing functions:

- The template prefix, `template <typename T>`, for example, precedes each external function definition.
- Outside the class definition the class name is written with its parameter list as in `bag<T>`. This does **not** apply to function names.
- Good style dictates that type parameter names be used within the class definition.
- Outside the class definition types must be qualified as in `typename bag<T>::size_type`.

# Organizing Code

- Fully qualify identifiers from other namespaces.  
Do not place any `using` directives in the files.
- Give the implementation file an extension of `.template` or `.tem` and `#include` it at the end of the interface file.
- No separate compilation of the implementation is necessary or even permitted.

# A Friendly Thing

The prototypes for `friend` functions of `template` classes must be formed in a special way:

```
friend std::ostream& operator << <> (std::ostream& out, const bag& b);
```



Note the <> placed between the function name, <<, and the parenthesized parameter list.

# The Bag Class, Again

```
template <class Item>
class bag {
  public:
    typedef Item value_type;
    typedef std::size_t size_type;
    static const size_type DEFAULT_CAPACITY = 30;
    bag(size_type initial_capacity = DEFAULT_CAPACITY);
    bag(const bag& source);
    ~bag( );
    void operator =(const bag& source);
    size_type erase(const Item& target);
    bool erase_one(const Item& target);
    void insert(const Item& entry);
    void operator +=(const bag& addend);
    void reserve(size_type capacity);
    size_type count(const Item& target) const;
    Item grab( ) const;
    size_type size( ) const { return used; }
  private:
    Item *data;
    size_type used;
    size_type capacity;
};

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

#include "bag4.template"
```

# Implementation

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

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) {
    data = new Item[source.capacity];
    capacity = source.capacity;
    used = source.used;
    std::copy(source.data, source.data + used, data);
}

template <class Item>
bag<Item>::~~bag( ) {
    delete [ ] data;
```

# Implementation cont'd

```
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;
}

template <class Item>
bool bag<Item>::erase_one(const Item& target) {
    size_type index;
    for (index = 0; (index < used) && (data[index] != target); ++index)
        ;
    if (index == used)
        return false;
    --used;
    data[index] = data[used];
    return true;
}
```

# Implementation cont'd

```
template <class Item>
void bag<Item>::insert(const Item& entry) {
    if (used == capacity)
        reserve(used + 1);
    data[used] = entry;
    ++used;
}

template <class Item>
void bag<Item>::operator =(const bag<Item>& source) {
    Item * new_data;
    if (this == &source)
        return ;
    if (capacity != source.capacity) {
        new_data = new Item[source.capacity];
        delete [ ] data;
        data = new_data;
        capacity = source.capacity; }
    used = source.used;
    std::copy(source.data, source.data + used, data);
}
```

# Implementation cont'd

```
template <class Item>
void bag<Item>::operator +=(const bag<Item>& addend) {
    if (used + addend.used > capacity)
        reserve(used + addend.used);
    std::copy(addend.data, addend.data + addend.used, data + used);
    used += addend.used;
}

template <class Item>
void bag<Item>::reserve(size_type new_capacity) {
    Item *larger_array;
    if (new_capacity == capacity)
        return ;
    if (new_capacity < used)
        new_capacity = used;
    larger_array = new Item[new_capacity];
    std::copy(data, data + used, larger_array);
    delete [ ] data;
    data = larger_array;
    capacity = new_capacity;
}
```

# Implementation cont'd

```
template <class Item>
typename bag<Item>::size_type bag<Item>::count(
    const Item& target) const {
    size_type answer;
    size_type i;
    answer = 0;
    for (i = 0; i < used; ++i)
        if (target == data[i])
            ++answer;
    return answer;
}

template <class Item>
Item bag<Item>::grab( ) const {
    size_type i;
    assert(size( ) > 0);
    i = (std::rand( ) % size( ));
    return data[i];
}

template <class Item>
bag<Item> operator +( const bag<Item>& b1, const bag<Item>& b2) {
    bag<Item> answer(b1.size( ) + b2.size( ));
    answer += b1;
    answer += b2;
    return answer;
}
```

# Demonstration Program

```
const int ITEMS_PER_BAG = 4;
const int MANY_SENTENCES = 3;

template <class Item, class SizeType, class MessageType>
void get_items(bag<Item>& collection,
               SizeType n,
               MessageType description) {
    Item user_input;
    SizeType i;
    cout << "Please type " << n << " " << description;
    cout << ", separated by spaces.\n";
    cout << "Press the <return> key after the final entry:\n";
    for (i = 1; i <= n; ++i) {
        cin >> user_input;
        collection.insert(user_input); }
    cout << endl;
}
```

# Demonstration cont'd

```
int main( ) {
    bag<string> adjectives;
    bag<int>    ages;
    bag<string> names;
    int line_number;

    cout << "Help me write a story.\n";
    get_items(adjectives, ITEMS_PER_BAG, "adjectives that describe a mood");
    get_items(ages,      ITEMS_PER_BAG, "integers in the teens");
    get_items(names,    ITEMS_PER_BAG, "first names");
    cout << "Thank you for your kind assistance.\n\n";

    cout << "LIFE\n";
    cout << "by A. Computer\n";
    for (line_number = 1; line_number <= MANY_SENTENCES; ++line_number)
        cout << names.grab( )      << " was only "
            << ages.grab( )       << " years old, but he/she was "
            << adjectives.grab( ) << ".\n";
    cout << "Life is " << adjectives.grab( ) << ".\n";
    cout << "The ("      << adjectives.grab( ) << ") end\n";
}
```

# The STL Classes and Iterators

Standardized Container Traversal

# The Multiset Template Class

- A multiset is an STL class similar to a bag.
- The base type of a multiset must implement the  $<$  operator in a way that conforms to strict weak ordering.
  - Irreflexivity: if  $(x == y)$  then neither  $(x < y)$  nor  $(y < x)$  may be true.
  - Antisymmetry: if  $(x != y)$  then either  $(x < y)$  or  $(y < x)$  is true, but not both.
  - Transitivity: if  $(x < y)$  and  $(y < z)$  are true then  $(x < z)$  is true.

# Iterators

- Iterators are containers that allow orderly traversal of their contents.
- Operations include
  - initialization from other containers
  - movement
  - element access and modification
  - searching
  - tests for equality (== and !=)
- Types, besides input and output include
  - forward with ++ overloading
  - bidirectional with -- overloading
  - random access with [ ] overloading

# The Multiset Interface

A partial list of the multiset function set:

```
size_type count(const value_type& target) const;  
size_type erase(const value_type& target);  
size_type size() const;  
void erase(iterator i);  
iterator insert(const value_type& entry);  
iterator find(const value_type& target);  
iterator begin();  
iterator end();
```

In addition, iterators define:

```
value_type& operator *();  
value_type& operator []();  
iterator operator ++();  
iterator operator ++(int);  
iterator operator --();  
iterator operator --(int);
```

# Using an Iterator

The `[...)` or left-inclusive pattern may be used to step through the elements of an iterator in order determined by the `<` operator of the base type.

```
multiset<string> actors;
multiset<string>::iterator role;

actors.insert("Moe");
actors.insert("Curly");
actors.insert("Larry");
actors.insert("Curley");

for (role = actors.begin(); role != actors.end(); ++role)
    cout << *role << endl;
```

Pointers (arrays) may be used wherever iterators are expected, as with the `copy()` function.

# Another Example

Here the `find()` and `erase()` functions are illustrated. Other multiset functions are listed in Appendix H of the text.

```
multiset<int> m;  
multiset<int>::iterator cursor;  
  
// ...  
  
cursor = m.find(42);  
if (cursor != m.end())  
    m.erase(cursor);
```

# The Node Template Class

Generalized Nodes

# Node Template Class

```
template <class Item>
class node {
public:
    typedef Item value_type;
    node(
        const Item& init_data = Item( ),
        node* init_link = NULL) {
        data_field = init_data;
        link_field = init_link; }
    Item& data( ) {
        return data_field; }
    const Item& data( ) const {
        return data_field; }
    node* link( ) {
        return link_field; }
    const node* link( ) const {
        return link_field; }
    void set_data(const Item& new_data) {
        data_field = new_data; }
    void set_link(node* new_link) {
        link_field = new_link; }
private:
    Item data_field;
    node *link_field;
};
```

# The Toolkit Prototypes

```
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);
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);
template <class NodePtr, class SizeType>
NodePtr list_locate(NodePtr head_ptr, SizeType position);
template <class Item>
void list_remove(node<Item>* previous_ptr);
template <class NodePtr, class Item>
NodePtr list_search(NodePtr head_ptr, const Item& target);
```

**list\_locate()** and **list\_search()** are now represented by only one prototype each. The **const** version is handled automatically.

# Toolkit Template Implementation...

```
template <class Item>
void list_clear(node<Item>*& head_ptr) {
    while (head_ptr != NULL)
        list_head_remove(head_ptr);
}

template <class Item>
void list_copy(
    const node<Item>* source_ptr,
    node<Item>*& head_ptr,
    node<Item>*& tail_ptr) {
    head_ptr = NULL;
    tail_ptr = NULL;
    if (source_ptr == NULL)
        return ;
    list_head_insert(head_ptr, source_ptr->data( ));
    tail_ptr = head_ptr;
    source_ptr = source_ptr->link( );
    while (source_ptr != NULL) {
        list_insert(tail_ptr, source_ptr->data( ));
        tail_ptr = tail_ptr->link( );
        source_ptr = source_ptr->link( ); }
}
```

# ...Toolkit Template Implementation...

```
template <class Item>
void list_head_insert(node<Item>*& head_ptr, const Item& entry) {
    head_ptr = new node<Item>(entry, head_ptr);
}

template <class Item>
void list_insert(node<Item>* previous_ptr, const Item& entry) {
    node<Item> *insert_ptr;
    insert_ptr = new node<Item>(entry, previous_ptr->link( ));
    previous_ptr->set_link(insert_ptr);
}

template <class Item>
void list_head_remove(node<Item>*& head_ptr) {
    node<Item> *remove_ptr;
    remove_ptr = head_ptr;
    head_ptr = head_ptr->link( );
    delete remove_ptr;
}

template <class Item>
void list_remove(node<Item>* previous_ptr) {
    node<Item> *remove_ptr;
    remove_ptr = previous_ptr->link( );
    previous_ptr->set_link(remove_ptr->link( ));
    delete remove_ptr;
}
```

# ...Toolkit Template Implementation...

```
template <class NodePtr, class SizeType>
NodePtr list_locate(NodePtr head_ptr, SizeType position) {
    NodePtr cursor;
    SizeType i;
    assert(0 < position);
    cursor = head_ptr;
    for (i = 1; (i < position) && (cursor != NULL); ++i)
        cursor = cursor->link( );
    return cursor;
}

template <class NodePtr, class Item>
NodePtr list_search(NodePtr head_ptr, const Item& target) {
    NodePtr cursor;
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link( ))
        if (target == cursor->data( ))
            return cursor;
    return NULL;
}
```

# ...Toolkit Template Implementation

```
template <class Item>
std::size_t list_length(const node<Item>* head_ptr) {
    const node<Item> *cursor;
    std::size_t answer;
    answer = 0;
    for (cursor = head_ptr; cursor != NULL; cursor = cursor->link( ))
        ++answer;
    return answer;
}
```

# An Iterator for Linked Lists

Implementing an External Iterator

# Using the Iterator

An iterator may be used to step through a list of `ints`, changing each to zero if it's odd:

```
node_iterator<int> start(head_ptr);
node_iterator<int> finish;
node_iterator<int> cursor;

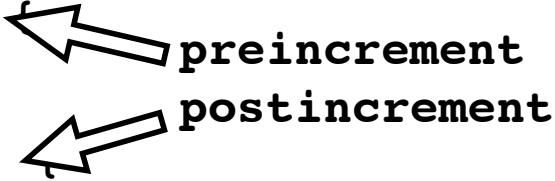
for (cursor = start; cursor != finish; ++cursor)
    if ((*cursor % 2) == 1)
        *cursor = 0;
```

# Iterator Class Definition

```
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( ); }

    current = current->link( );
    return *this; }

    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;
};
```



# const Iterator Class Definition

```
template <class Item>
class const_node_iterator
  : public std::iterator<std::forward_iterator_tag,           > {
  public:
    const_node_iterator(                                     initial = NULL) {
      current = initial; }
      operator *( ) {
        return current->data( ); }
    const_node_iterator& operator ++( ) {
      current = current->link( );
      return *this; }
    const_node_iterator operator ++(int) {
      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:
      current;
};
```

# An `add_values()` Function

```
int add_values(const node<int>* head_ptr) {  
    const_node_iterator<int> start(head_ptr);  
    const_node_iterator<int> finish;  
    const_node_iterator<int> cursor;  
  
    int sum = 0;  
  
    for (cursor = start; cursor != finish; ++cursor)  
        sum += *cursor;  
    return sum;  
}
```

# Linked List Template Bag with an Iterator

Putting it All Together

# Providing an Iterator for a Custom Container Class

- Define an iterator class, inherited from `std::iterator`. Include functions such as `*`, `++`, `==`, `!=` and perhaps `[]` and `--`.
- Also define a `const_iterator` class.
- Provide two `begin()` functions within the container class: an ordinary one and a `const` one.
- Do the same for a pair of `end()` functions.

# Bag with Iterator Class Definition

```
template <class Item>
class bag {
  public:
    typedef std::size_t size_type;
    typedef Item value_type;

    bag( );
    bag(const bag& source);
    void operator =(const bag& source);
    ~bag( );
    size_type erase(const Item& target);
    bool erase_one(const Item& target);
    void insert(const Item& entry);
    void operator +=(const bag& addend);
    size_type count(const Item& target) const;
    Item grab( ) const;
    size_type size( ) const { return many_nodes; }

  private:
    node<Item> *head_ptr;
    size_type many_nodes;
};
```

# Instantiating the Iterator

Since the iterator name is `typedef`d within the container class, its name must be qualified by the object name when it is instantiated:

```
bag<int>::iterator start = myBag.begin();  
bag<int>::iterator stop  = myBag.end();  
bag<int>::iterator cursor;
```

# Bag Implementation...

```
template <class Item>
bag<Item>::bag( ) {
    head_ptr = NULL;
    many_nodes = 0;
}

template <class Item>
bag<Item>::bag(const bag<Item>& source) {
    node<Item> *tail_ptr;
    list_copy(source.head_ptr, head_ptr, tail_ptr);
    many_nodes = source.many_nodes;
}

template <class Item>
bag<Item>::~~bag( ) {
    list_clear(head_ptr);
    many_nodes = 0;
}
```

# ...Bag Implementation...

```
template <class Item>
typename bag<Item>::size_type bag<Item>::count(const Item& target) const {
    size_type answer;
    const node<Item> *cursor;
    answer = 0;
    cursor = list_search(head_ptr, target);
    while (cursor != NULL) {
        ++answer;
        cursor = cursor->link( );
        cursor = list_search(cursor, target); }
    return answer;
}

template <class Item>
Item bag<Item>::grab( ) const {
    size_type i;
    const node<Item> *cursor;
    assert(size( ) > 0);
    i = (std::rand( ) % size( )) + 1;
    cursor = list_locate(head_ptr, i);
    return cursor->data( );
}
```

# ...Bag Implementation...

```
template <class Item>
typename bag<Item>::size_type bag<Item>::erase(const Item& target) {
    size_type answer = 0;
    node<Item> *target_ptr;
    target_ptr = list_search(head_ptr, target);
    while (target_ptr != NULL) {
        ++answer;
        target_ptr->set_data( head_ptr->data( ) );
        target_ptr = target_ptr->link( );
        target_ptr = list_search(target_ptr, target);
        list_head_remove(head_ptr); }
    return answer;
}

template <class Item>
bool bag<Item>::erase_one(const Item& target) {
    node<Item> *target_ptr;
    target_ptr = list_search(head_ptr, target);
    if (target_ptr == NULL)
        return false;
    target_ptr->set_data( head_ptr->data( ) );
    list_head_remove(head_ptr);
    --many_nodes;
    return true;
}
```

# ...Bag Implementation...

```
template <class Item>
void bag<Item>::insert(const Item& entry) {
    list_head_insert(head_ptr, entry);
    ++many_nodes;
}

template <class Item>
void bag<Item>::operator +=(const bag<Item>& addend) {
    node<Item> *copy_head_ptr;
    node<Item> *copy_tail_ptr;
    if (addend.many_nodes > 0) {
        list_copy(addend.head_ptr, copy_head_ptr, copy_tail_ptr);
        copy_tail_ptr->set_link( head_ptr );
        head_ptr = copy_head_ptr;
        many_nodes += addend.many_nodes; }
}
```

# ...Bag Implementation

```
template <class Item>
void bag<Item>::operator =(const bag<Item>& source) {
    node<Item> *tail_ptr;
    if (this == &source)
        return ;
    list_clear(head_ptr);
    many_nodes = 0;
    list_copy(source.head_ptr, head_ptr, tail_ptr);
    many_nodes = source.many_nodes;
}

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

## Project E

Complete Project 4 from Chapter 6 of the text. Directions are on page 347.

- Name your **class** `keyed_bag4`. You will submit its definition in `keyed_bag4.h` and the implementation in `keyed_bag4.cpp`.
- Be sure to define a **const\_iterator class** as well as `keyed_bag4`. The interface file, `keyed_bag4.h`, should look like:

```
// referencing (forward) declaration:
template <class K, class V>
class const_iterator;

template <class K, class V>
class keyed_bag4 {
    //...
};

template <class K, class V>
class const_iterator
    : public std::iterator<std::forward_iterator_tag, K, V> {
    // ...
};

#include "keyed_bag4.cpp"
```

- The `keyed_bag4` public member functions are:

```
keyed_bag4();
keyed_bag4(const keyed_bag4& source);
~keyed_bag4();
keyed_bag4& operator =(const keyed_bag4& source);
void insert(const V& entry, K key);
bool erase(K key);
V get(K key) const throw(std::invalid_argument);
size_type size() const;
size_type count(const V& target) const;
bool has_key(K key) const;
const_iterator<K, V> begin();
const_iterator<K, V> end(); // hint: should position cursor just
                           // past the end of the pair array
```

- The `const_iterator` public member functions are:

```
const_iterator( const std::pair<K, V>* initial = NULL);
const_iterator& operator ++();
const_iterator operator ++(int);
bool operator !=(const const_iterator item) const;
bool operator ==(const const_iterator item) const;
const std::pair<K, V> operator*();
```

- If an attempt is made to `insert()` an entry with a duplicate key, the existing entry should be replaced by one with the new value.
- If an attempt is made to `get()` a value with a nonexistant key a `std::invalid_argument` exception should be thrown. It should be constructed with an appropriate message string.
- Do not create a `namespace` for your class .
- Test your `classes` thouroughly with your own interactive test program similar to the one presented on pp. 133-135 of the text. You will not submit the test program.
- My test program is [here](#).

When you're satisfied with the definition and implementation, submit your work via the [CATE form for Project E](#).

*Legend: `method/function` `keyword` `literal`*