Templates, Iterators, and STL

- Template Functions
- Template Classes
- The STL's Algorithms and Use of Iterators
- The Node Template Class
- An Iterator for Template Based Linked Lists
- BagTemplate Class
- Bag Class Summary

C++ Templates

- Algorithms and ADTs must be customized to the data type upon which they will operate
  - typedef or alias enables different data types for each compilation
- C++ provides **template** mechanism
  - uses general data type parameters for functions and classes
  - acts as model for constructing many distinct functions
    - compiler automatically builds function for each new type of argument

Function Overloading vs Function Templates

- Template restrictions:
  - At least *one* function parameter must be template type
  - All *operators* used within function must be valid for object
  - Arguments must match *all* parameters exactly
    - No type conversion allowed

'Failed unification' error

int
int&
const int&

- Function templates allow function calls with runtime parameters of different data types
  - function arguments
  - declaration of local objects in function
  - Function return type

Function Overloading versus Function Templates

- Doesn't function overloading already do this?
  - Yes, but *programmer* must define *each* function!
  - Function templates make the *compiler* create *many* overloaded functions automatically
    - *programmer* only writes *one* "generic" function!
  - For multi-file projects, implementation (*definition*) of a template function must be in the same file as its declaration
    - Compiler needs *definition* of template function to create *specific instantiation*

Template Functions

nonempty parameter list of class types (aka template prefix)

- Template function format:

  ```
  template <class Template-parameter> function-definition
  ```

  - multiple template parameters separated by commas
  - all template-parameters must appear at *least once* in function parameter list
  - programming style capitalizes first letter of template parameter
  - template function merely specifies the function
    - does not cause memory to be allocated
  - template functions may be overloaded
    - different parameter list (*type and/or number*)
  - template functions may be specialized
    - customize template function for *specific* set of arguments

Template Functions

- When called, *compiler* associates actual argument data types with formal parameter list
  - compiler creates separate instances of function for each different actual argument list
    - memory used for function instance
- Often easier to convert existing function into template rather than writing one from scratch
  - specific to general

Template Functions: example

function template

```
template <class T>
T lesser(T a, T b) {
   // returns lesser of two arguments
   T result;
   if (a < b)
      result = a;
   else
      result = b;
   return result;
}
```

function calls

lesser(23, -12)
lesser('a', 'A')
lesser(stk1, stk2)

compile

stock must have overloaded operators for < and =

generated function code

```
int lesser(int a, int b) {
   // returns lesser of two arguments
    int result;
   if (a < b)
      result = a;
   else
      result = b;
   return result;
}
```

```
char lesser(char a, char b) {
   // returns lesser of two arguments
    char result;
   if (a < b)
      result = a;
   else
      result = b;
   return result;
}
```

```
stock lesser(stock a, stock b) {
   // returns lesser of two arguments
   stock result;
   if (a < b)
      result = a;
   else
      result = b;
   return result;
}
```

Template Classes

nonempty parameter list of class types (aka template prefix)

- Template class format:

  ```
  template <class Template-parameter> class-declaration
  ```

  - multiple template parameters separated by commas
  - template parameter list *precedes* class declaration
  - scope of template parameters extend throughout entire class definition
  - data types are passed to template class when creating object:

  ```
  class-name <data-type> object-name;
  ```

Template Class Methods

- Template class method can be defined in-line *or* outside of class
- If member method defined external to class,
  - method is treated as template function
  - template parameter list (*same as class*) included in function definition, *even if parameter name not referenced in function*

```
template <class Template-parameter>
return-value class-name <Template-parameter>::function-
  name (function-parameter) function-definition
```

Template Classes: example

class template definition

```
template <class T>
class store {
public:
   store(const T& item = T());
   T getValue() const;
   void setValue(const T& item);
private:
   T value;
};

template <class T>
store<T>::store(const T& item) : value(item))
{}

template <class T>
T store<T>::getValue() const {
   return value;
}
template <class T>
void store<T>::setValue(const T& item) {
   value = item;
}
```

Template Classes: example

variable declaration

```
store <int> iVar;
```

generated function code

```
store::store(const int& item) : value(item))
{}

int store::getValue() const {
   return value;
}

void store::setValue(const int& item) {
   value = item;
}
```

Template Classes: example

variable declarations

```
store <char> cArr[10];
```

generated function code

```
store::store(const char& item) : value(item))
{}

char store::getValue() const {
   return value;
}

void store::setValue(const char& item) {
   value = item;
}
```

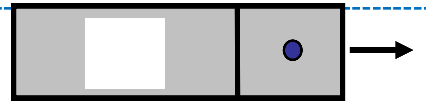
Why Use Templates?

- Templates often used to
  - create type-safe collection classes that can operate on data of *any* type
- Template advantages
  - easier to write
    - create one generic version instead of creating multiple versions
  - easier to understand
    - straightforward way of abstracting type information
  - type-safe
    - types known at compile time--*compiler performs type checking*

Node Template Class for Linked List

- Nodes are the independent items in a linked list
  - data field
    - template parameter data type
  - pointer
    - connects to adjacent item in list
    - also called a link

```
<template class Item>
class nodeTemplate {
public:
  using value_type = Item;
private:
  Item data_field;
  nodeTemplate *link_field;
};
```



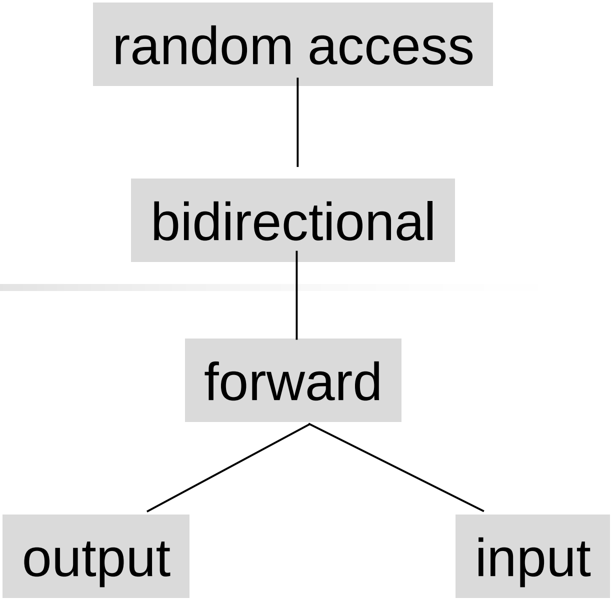
Template Based SLL Toolkit

- Included functions:
  - list_clear → release memory for all nodes in list
  - list_copy → copy all nodes from source list
  - list_head_insert → insert data at beginning
  - list_head_remove → remove node at beginning
  - list_insert → insert data before given node
  - list_length → number of nodes in list
  - list_locate → locate node at given position (1, 2, ...) or nullptr
  - list_remove → remove node linked to given node pointer
  - list_search → search list for given data and return first node pointer to found data or nullptr

STL Iterators

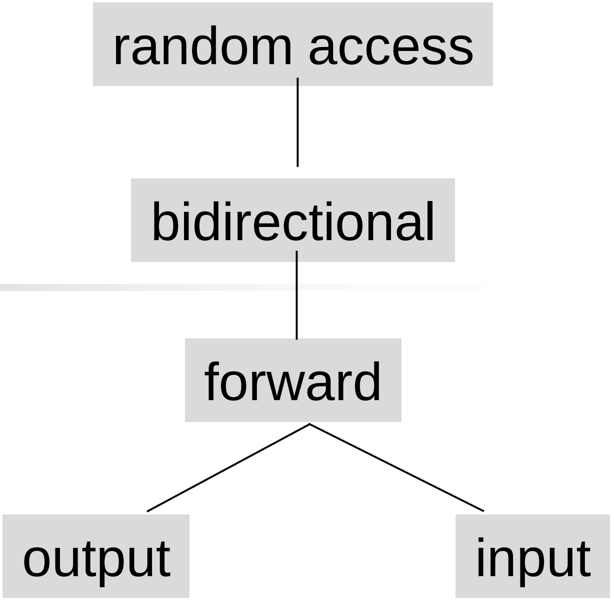
- Iterators are generalizations of ***pointers***
  - used to access information stored in containers
  - can cycle through items in a container in the same way a pointer traverses a linked list
- **Constant** Iterators
  - Used to examine container elements, but not modify them
    - Cannot apply dereference operator (*) on left side of assignment statement
  - ***Must be*** used with constant container objects
- **Non Constant** Iterators
  - Also called mutable iterators
  - Used to modify container elements
  - ***May not*** be used with constant container objects

STL Iterators



- Iterator categories
  - Input
    - De-reference for element retrieval
    - ++ to move from beginning to end
  - Output
    - De-reference to set value for element insertion
    - ++ to move from beginning to end
  - Forward
    - Can de-reference to set value for element insertion or retrieve value
    - ++ to move from beginning to end

STL Iterators

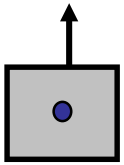


- Iterator categories
  - Bidirectional
    - same as forward, *plus*
    - -- to move from end to beginning
  - Random-Access
    - same as bidirectional, *plus*
    - directly access with index notation
    - determine distance between two iterators with subtraction

Node Iterator Class for Linked List

- Derived from std::iterator
- Constant and non-constant version
- Forward iterator to step through nodeTemplate objects in SLL
  - pointer
    - reference to nodeTemplate object

```
<template class Item>
class nodeTemplate_iterator {
private:
  nodeTemplate<Item>* current;
};
```



Node Iterator Class for Linked List

- Member functions
  - default constructor → assign current to parameter or nullptr
- Overloaded operators *(as member functions)*
  - dereference → return reference to node
  - prefix increment → move forward and return iterator
  - postfix increment → return copy of iterator before moving forward
  - equal
  - not equal

BagTemplate Class

- Rules for implementation
  - Items stored in linked list of nodeTemplate objects
  - First node in list is stored in member variable **`head_ptr`**
  - Total number of items stored in list stored in member variable **`many_nodes`**
- Include iterator and const_iterator members for nodeTemplate navigation through linked list
  - begin() and end()

BagTemplate Class

- Member functions
  - *default constructor → create empty bag*
  - *copy constructor → create bag copied from source*
  - *destructor → clear list of all nodes*
  - *size() → return number of items in bag*
  - *count → count number of item occurrences*
  - *insert → insert an item*

BagTemplate Class

- *erase_one → remove an item, if found*
- *erase → remove all items, return count of items erased*
- *grab → return a random item from bag*
- *+= → copy items from source bag to current*
- *= → reset current bag to source*

- Non-member function
  - *+ → create new* **`bagTemplate`** *object from two added* **`bagTemplate`** *objects*

Converting Container Class to Template

- Include template prefix before each function prototype or implementation
- Outside class definition, include template parameter with class name
- Outside member functions, include typename keyword for any class defined types

BagTemplate Class

- Member functions to provide iterators
  - begin → return nodeTemplate_iterator to parameter constructor (head_ptr)
    - const and non const versions
  - end → return nodeTemplate_iterator to default constructor (nullptr)
    - const and non const versions

Bag Class Summary

- Comparison of Bag Class Containers

Container Approach	Classes
Items stored in fixed sized array	bagFixed
Items stored in dynamic array	bagDynamic
Items stored in dynamic singly linked list	bagList
Items stored in dynamic template based singly linked list	bagTemplate