Week 9 Lecture notes

Templates, STL, Algorithm Complexity

Additional reading content can be found in the course textbook written by Goodrich et al chapters 1.5.5, 2.3, 3.2.4, 4.1 – 4.2, 5.1.2, 5.2.2, 5.3.2, 6.1.4, 6.2, 9.1.3, 8.1.6, 8.1.2. For more information on C++ STL, read chapter 18 in the CISP 1010 book, Problem Solving in C++ by Savitch.

Contents

[Templates 1](#_Toc46146158)

[A Linked List Template Class with an Iterator 6](#_Toc46146159)

[Standard Template Library (STL) 16](#_Toc46146160)

[vector 16](#_Toc46146161)

[Iterators 19](#_Toc46146162)

[list, deque 21](#_Toc46146163)

[stack, queue, priority\_queue 21](#_Toc46146164)

[set, map 21](#_Toc46146165)

[Algorithm 23](#_Toc46146166)

[Comparator Classes 23](#_Toc46146167)

[Algorithms and Big-O notation 24](#_Toc46146168)

# Templates

Templates are used to implement **algorithm abstraction**. For example, the algorithm to swap two items (of any type) is:

temp = item1

item1 = item2

item2 = temp

But, if we want to swap integers, we have to write a different function than the one to swap characters than the one to swap doubles, than the one to swap Book objects, etc.

|  |
| --- |
| void swap( int& i1, int& i2 ) {     int temp = i1;     i1 = i2;     i2 = temp;  }  void swap( double& d1, double& d2 ) {     double temp = d1;     d1 = d2;     d2 = temp;  }  // etc … for every type of variable we’d like to swap! |

Seems kind of silly to have to do that. What we can do instead is use templates which parameterizes types. For example, we can make the parameter type of swap’s two parameters a “variable”. (There is already a swap template in the utility header file. See <http://www.cplusplus.com/reference/utility/swap/?kw=swap> ). Note the pre-condition for a templated swap function: If you’re going to use this function to swap two objects (Books, Apples, Avatars, etc.) then those classes have to have a working operator= because the mySwap template function below (and the swap template function that already exists in C++) uses it.

|  |
| --- |
| /\* mySwap: template function that swaps the actual values of it parameters      Parameters: two items to swap     Pre-condition: type T has operator= defined     Returns: nothing  \*/  template<class T>  void mySwap( T& item1, T& item2 ) {     T temp;     temp = item1;     item1 = item2;     item2 = temp;  } |

The steps to create a template function:

1. Precede the function with a template prefix (“template” is a keyword). class doesn’t actually mean we’re creating a class here. Poor keyword choice, really. You could actually use the keyword typename which is a better descriptor but wasn’t originally part of C++. So, you will still see the class keyword used a lot. The T which is the type “variable” is capitalized – best practice/clean code. We could use any valid user-defined name here, but T is pretty standard. If you have more than one type parameter, you can use T1, T2. For example:

template<class T1, class T2>

Other common type parameter names are E (element), K (key), V (value), N (number), S, U, and V.

1. Use the type parameter in your function. Basically, we substitute T for “int” in the very first swap function above.
   1. It is a good strategy when first writing template functions to write it regularly first, using a specific type and not a template. Then, test that function to make sure it works for the specific type before making it a template.
   2. Also, we’re going to need some preconditions regarding operators that need to be overloaded. In this templated swap function, the assignment operator is used. If we were swapping two objects that had dynamically allocated memory, that class would need to overload the assignment operator or the swap wouldn’t work the way we want.
2. Call the function as you would normally:

int i, j; // assume they get values

Book b1, b2; // assume they get values, too

mySwap( i, j );

mySwap( b1, b2 );

Book\* b1Ptr, \*b2Ptr; // assume they point to Book objects

mySwap( \*b1Ptr, \*b2Ptr );

How does the compiler do this? The compiler produces a definition for every version of mySwap that you have used in your program. The compiler uses the templated mySwap to write multiple mySwap functions. The function called mySwap is overloaded with the types that the program uses.

Where do you put the template function? Compilers need the entire template function definition (not just the declaration/prototype) in the file where you call the function, for example, in main. Assume you put the mySwap function declaration/prototype in a file such as a myAlgorithms.h as we have been doing. If you put the mySwap function definition in a file such as myAlgorithms.cpp, then try to compile main.cpp with the following command

g++ -c main.cpp

then how will the compiler be able to write the different versions of mySwap used in main? It won’t is the answer. So, we have to put the entire function declaration/prototype and definition of mySwap in the myAlgorithms.h header file, then #include “myAlgorithms.h” in main.cpp. Therefore, THERE WILL BE NO FILE CALLED myAlgorithms.cpp. The code for the templated function mySwap has to be in the header file so the compiler has access to it when needed.

The function prototype for mySwapis just its heading ended with a semicolon:

template<class T>

void mySwap( T& item1, T& item2 );

See <https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl> for more clarification on separating template function definitions and declarations/prototypes.

If you put your template function prototypes and the function definitions in the same header file, you can still compile the header file separately at the command line:

g++ –c blah.h

and you will have a file ending in gch, for example, blah.gch.

There are many other algorithms whose logic is the same regardless of type. Searching, finding the minimum/maximum, and sorting are three examples. To search for an item in some group of items, we need to be able to compare two items (to have an == operator). A generic search algorithm:

1. for item in group of items
   1. if item == key
      1. return item (or index/pointer to item)
   2. increment index/pointer to next item
2. return not found

A declaration for a templated search function is shown below. The function searches through an array of n items and returns the index where the element is the same as the key or -1 if the key is not found. Note the type T is used for the array base type and the key. They have to be the same type since the search function looks through an array of things to find one of those things. The things have to be the same type. The variable n which represents the number of valid elements in the array of things is an integer in every version of the function. The variable n is not a thing, it is the integer representing the number of things.

template<class T>

int search( T a[], int n, T key );

To find the minimum/maximum or sort a group of items, we need a working < operator (or >), but the algorithms used are not dependent on the type of item. A generic find the minimum algorithm:

1. min = index/pointer of first element
2. for i (an index or pointer) = item 2 to item n
   1. if item i < item min
      1. min = i
   2. increment index/pointer

Note, when using template functions with classes, for example, searching through an array of Book objects, the class must overload the appropriate operator(s). For the template search function, that would be the == operator. We must overload the == operator to define what it means for two books to be equivalent. To use a template function on an array of objects for sorting, finding minimum/maximum, the class needs to overload the < (or maybe >) operator. In general, if you overload any relational operator (such as < ), you should overload them all: <, >, !=, ==, >=, <=

When writing templated functions, include in the comment header as a pre-condition, the operators needed for the function to work properly. For example:

/\* search searches for a key in an array

Parameters

a the array

n the number of valid elements in the array

key the key to look for

Pre-condition type T has defined operator==

Returns index where key is found or -1 if not found

\*/

template<class T>

int search( T a[], int n, T key );

Template classes can be created, too, not just functions. The syntax is the same as a function template. For example, let’s create a template for a class that can hold two of anything: a MyPair class with a key and a value associated with the key. With a Book object, the key might be the ISBN (since every book has a unique ISBN) and the value would be a Book object of the book with that ISBN. Other examples of pairs are Taxpayer (key = social security number, data = taxpayer information) and Vehicle (key = vehicle identification number, data = information about the car). As an aside, there actually is a pair template class in the C++ Standard Template Library (STL). See <http://www.cplusplus.com/reference/utility/pair/>. The header file mypair.h is shown below (without comments for brevity).

|  |
| --- |
| template<class T1, class T2>  class MyPair {  private:     T1 key;     T2 value;  public:     MyPair() {};     MyPair(T1 k, T2 v );     void setKey( T1 k );     void setValue( T2 v );     T1 getKey() const { return key; }     T2 getValue() const { return value; }  };    template<class T1, class T2>  MyPair<T1,T2>::MyPair( T1 k, T2 v ) {     key = k;     value = v;  }    template<class T1, class T2>  void MyPair<T1,T2>::setKey( T1 k ) {     key = k;  }    template<class T1, class T2>  void MyPair<T1,T2>::setValue( T2 v ) {     value = v;  }    #endif |

Note the function prototypes/declarations and the function definitions. The function definitions must be preceded with a template prefix that is identical to the class template prefix, even if that function doesn’t use both parameterized types, such as functions setKey and setValue.

template<class T1, class T2>

Also note that the function heading has the parameterized types listed with the class name and scope resolution operator such as

void MyPair<T1,T2>::setKey( T1 k ) {

**Pair Programming 9a:** Create a new workspace/project in the IDE called pp9a. Create two NEW files (don’t add exiting ones) book.cpp and book.h to this project. Copy the code from main9a.cpp, book.h, and book.cpp from the Pair Programming assignment and paste it into these files. Read through main.cpp so you understand the output it should generate since it is the driver to test your template functions. Create a file myPair.h that has the code for the MyPair template class. Create file myAlgorithms.h and write the following template functions in it. Use the Pair Programming tests to test these functions.

* mySwap: has two parameters and swaps their actual values
* myMin: has two parameters and returns the smallest. Pre-condition: operator< has been overloaded to compare the two parameters
* mySort: has two parameters, an array and the number of elements in the array. Sorts the elements. Calls mySwap above. Pre-condition: overloaded operator< in parameter class.
* mySearch: has three parameters, an array, the number of elements in the array and a key to find in the array. Returns the index found or -1 if the key isn’t found in the array. Pre-condition: overloaded operator== in parameter class.

# A Linked List Template Class with an Iterator

What if we wanted to create a generic linked list? The basic algorithm for inserting, traversing, etc. is the same no matter what type of data is in our list. Consider the template class Node definition below in a file called node.h Note the changes in bold from our non-template node class. The type isn’t Node anymore. The type is Node<T> , a node of “T” things, not just integers or books. The type of item being stored isn’t a Book, it’s the parameterized type T. We’ll talk about iterators below.

|  |
| --- |
| // declarations for friend statements below  **template<class T> class LinkedList;**  **template<class T> class Iterator;**    **template<class T>**  class Node {  public:     /\* Default constructor sets next and prev pointers to        NULL and calls item's default ctor      \*/     Node();     /\* 1-arg ctor that creates a node from an item type in the        node. Sets next and prev pointers to NULL      \* Precondition: class T has overloaded operator= that        performs deep copy if T dynamically allocates memory      \*/     Node( const **T**& );     /\* copy constructor. Creates a node from another node.      \*/     Node( const Node**<T>**& );     /\* destructor      \*/     virtual ~Node**<T>**() {}     /\* setItem: sets values for item in node.      \* Precondition: class T has overloaded operator= that        performs deep copy if T dynamically allocates memory      \*/     void setItem( const **T**& );     /\* getItem: returns node's item      \* Precondition: class T has overloaded operator= that        performs deep copy if T dynamically allocates memory      \*/  **T** getItem() const { return item; }     /\* operator= assigns one node's data to another. Does not        copy next and prev pointers      \* Precondition: class T has overloaded operator= that        performs deep copy if T dynamically allocates memory      \*/     Node**<T>**& operator=( const Node**<T>**& );     friend class LinkedList**<T>**;     friend class Iterator**<T>**;  private:  **T** item;     Node**<T>**\* nextPtr;     Node**<T>**\* prevPtr;  }; |

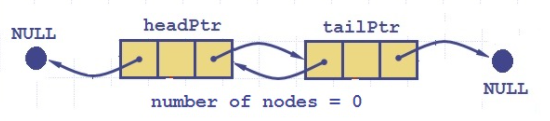
Let’s look at the definitions of the Node template class member functions that should also be placed in file node.h. There will be no node.cpp file. All of the code for the node class will go in the header file so it can be found at compile time so the compiler can write all node functions for the specific types used by the program as explained above. Note the function comment headers. Comments/style is 10% of all lab grades and the code below demonstrates good function comment headers.

|  |
| --- |
| /\* default constructor sets pointers to NULL  \*/  **template<class T>**  Node**<T>**::Node() : nextPtr( NULL ), prevPtr( NULL ) { }    /\* 1-arg copy constructor doesn't copy the next/prev pointers  Pre-condition: T has defined operator=  \*/  **template<class T>**  Node**<T>**::Node( const Node**<T>**& n ) {     \*this = n;     nextPtr = NULL;     prevPtr = NULL;  }  /\* setItem sets the item data values in the node  Parameter  i the item  Pre-condition: T has defined operator=  Returns: nothing  \*/  **template<class T>**  void Node**<T>**::setItem( const **T**& i ) {  this->item = i;  }  /\* operator= sets this' node’s item values to n’s item values  and this' pointers to NULL  Parameter  n the source node to copy from  Pre-condition: T has defined operator=  Returns: a reference to the destination node copied to  \*/  **template<class T>**  Node**<T>**& Node**<T>**::operator=( const Node**<T>**& n ) {       nextPtr = NULL;       prevPtr = NULL;       item = n.item;       return \*this;  } |

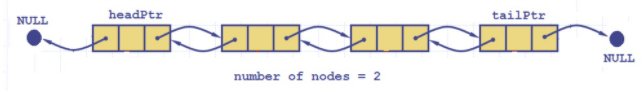
We’ll have a node that can hold any type and a linked list that can hold those nodes. Note the declaration of the LinkedList template class below and the friend statement within the Node class. In order to let the compiler know that there’s a linked list template class with the same type as the Node (we wouldn’t want a list of Car objects with nodes that had Book objects), we have to have that template class declaration. However, as with previous friend statements, the file node.h does not have to #include “linkedList.h”. The linked list template class is covered below.

**Before class to be ready to start this week’s second Pair Programming:** Create a new workspace/project in the IDE called pp9b that we will use to test a template LinkedList class. Add a new file called node.h with the implementation of all of its member functions above. Add two new files, book.h and book.cpp to the project and copy/paste the code for these classes. Compile node.h and correct compilation errors.

Now, let’s create a linked list template class. We will use a linked list that has sentinel nodes, one for the head and one for the tail for a doubly linked list. If we want to use iterators (below), we have to have these “fake” begin and end nodes. So, an empty list would look like the image below with a headPtr that points to a node and a tailPtr that points to a node. So, headPtr->nextPtr = tailPtr and tailPtr->prevPtr = headPtr.



If we have two nodes in the list, our list will look like:



Our template linked list class definition in linkedList.h might have the following private data: a headPtr, a tailPtr, and an integer representing the number of node in the list (such as n).

An algorithm for a default constructor that creates a list with fake head and tail nodes:

1. headPtr = new node
2. tailPtr = new node
3. headPtr->nextPtr = tailPtr
4. tailPtr->prevPtr = headPtr
5. n = 0

Since our linked list template class has a data member that holds the number of elements in the list, a member function that returns the number of elements in the list, getN, would be easy as shown in this one step algorithm.

1. return n

And, a member function that returns whether the list is empty, isEmpty, also has one step:

1. return n == 0

We shouldn’t write a setN function. What would it mean to allow the users, other programmers, of our class to be able to change n? That would mean they could change n from 10 to 3, for example. What 7 nodes should we get rid of? We also won’t write any set or get functions for the head and tail pointers. Having these pointers and fake nodes is an implementation detail that other programmers should not be allowed to access.

What’s an **iterator**? An iterator is like an index in an array or a pointer to an item, but more general. Just like an index or a pointer, it represents the position of a node. An iterator can be incremented to point to the next node or decremented to point to the previous node. Let’s look at an iterator class template definition.

|  |
| --- |
| template<class T> class LinkedList;  template<class T> class Node;    template<class T>  class Iterator {  public:       /\* operator\* dereferences iterator to return a reference  to node in list        \*/       Node<T>& operator\*() const;       /\* operator== returns true if this iterator and  parameter iterator point to same object        \*/       bool operator==( const Iterator<T>& p ) const;       /\* operator!= returns true if this iterator and  parameter iterator don't point to same object        \*/       bool operator!=( const Iterator<T>& p ) const;       /\* operator++ moves iterator to point to next node  in list and returns same iterator        \*/       Iterator& operator++();       /\* operator-- moves iterator to point to previous  node in list and returns same iterator        \*/       Iterator& operator--();       friend class LinkedList<T>;       friend class Node<T>;  private:  /\* the node pointer that this iterator refers to\*/       Node<T>\* curPtr;       /\* Iterator constructor that creates an iterator  that points to Node parameter. No default constructor,  just a private copy constructor        \*/       Iterator( Node<T>\* nPtr );  }; |

Our Iterator class will be friends with our Node and LinkedList template classes so a node and linked list object can use the private data and functions of the iterator class. We make only one iterator constructor (that takes a node) and it’s private for two reasons: 1) it doesn’t make sense to have an iterator if there’s no node to point to and 2) we don’t want any other classes to use our iterator except for Node and LinkedList. It’s a utility class for our linked list. So is the Node class for that matter. The textbook actually has the class definitions for Node and Iterator inside the LinkedList class definition. The iterator class has one data member, curPtr, which will point to the node to which the iterator object refers. Basically, the Iterator class is a fancy pointer with operations defined on it such as ++ and --. For example, with this class we could write code to go through the entire list like this:

Iterator<Book> it = list.begin();

for ( ; it !+ list.end(); ++it ) {

cout << (\*it).getItem() << endl;

}

The C++ STL makes use of iterators, so learning about how they work on the inside is important.

Let’s look at the implementation for the Iterator member functions. The overloaded operator\* is used to dereference an iterator.

|  |
| --- |
| /\* operator\* dereferences list iterator  Returns a reference to the node in list  \*/  template<class T>  Node<T>& Iterator<T>::operator\*() const {       return \*curPtr;  } |

So, if I had an iterator that pointed to a node, I could:

cout << (\*it).getItem() << endl;

assuming a node’s item type had overloaded operator<<. The overloaded operator\* returns a reference to the item to which the iterator refers. If it didn’t return a reference variable, Node<T>&, but just a Node<T> instead, then the function would return a copy of the object to which the iterator refers as opposed to a reference to the actual item which would be less efficient.

The overloaded operator== will tell us whether two iterators point to the same object. This will especially be helpful once we add some iterator code to our linked list.

|  |
| --- |
| /\* operator== returns true if this iterator and parameter  iterator point to same object  Parameter  it iterator that points to a node in the list  Returns true if this iterator’s node == it’s node  \*/  template<class T>  bool Iterator<T>::operator==( const Iterator& it ) const {       return curPtr == it.curPtr;  } |

The overloaded operator++  will allow us to make our iterator point to the next node in the list. This is an overloaded prefix operator, not postfix. For example, if we had an iterator and we incremented it with a prefix ++:

++it;

Returning a reference to the iterator means we can have a statement like this (assuming it2 is another iterator of the same type):

it2 = ++it;

|  |
| --- |
| /\* operator++ moves iterator to point to next node in list  and returns same iterator  Returns reference to current iterator  \*/  template<class T>  Iterator<T>& Iterator<T>::operator++() {       curPtr = curPtr->nextPtr;       return \*this;  } |

Why do we need the fake head and tail nodes? Assume we used the loop above (duplicated here) to iterator through the list.

Iterator<Book> it = list.begin();

for ( ; it !+ list.end(); ++it ) {

cout << (\*it).getItem() << endl;

}

After the loop executes, we should be able to back up from the end of the list like so:

--it;

cout << (\*it).getItem() << endl;

Without the fake head/tail nodes, though, the loop’s Boolean expression would be

curPtr != NULL

as it has been in our linked list code in C and C++ without templates. How do you back up from NULL? You can’t. There is no way to get from NULL to the node before NULL. There is no node before or after, for that matter, NULL. But, you can back up from a fake tail node. Fake head and tail nodes allow us to write code to traverse the list forwards and backwards without falling off the end of the earth -without hitting NULL which cannot be utilized to move further in the list.

**Before class to be ready to start this week’s second Pair Programming:** in the pp9b project, add a file iterator.h with code for all of the above iterator functions. Compile this file and fix compilation errors.

Our next step is to create our template linked list class that uses iterators. Below is the class definition – read through it carefully. The member functions that are inline are in bold. Note the size and empty functions discussed earlier.

|  |
| --- |
| template<typename T>  class LinkedList {  public:     /\* Default ctor creates an empty list with empty head and  tail nodes      \*/     LinkedList();     /\* Copy constructor does deep copy of parameter list      \*/     LinkedList( const LinkedList<T>& );     /\* size returns the size of the list      \*/  **int size() const { return n; }**     /\* empty returns true if the list is empty      \*/  **bool empty() const { return n == 0; }**     /\* begin returns an iterator to the first element  in the list or the tail sentinel node if the list  is empty      \*/  **Iterator<T> begin() const {**  **return Iterator<T>( headPtr->nextPtr ); }**     /\* end returns an iterator to the last element in the list  or the tail sentinel node if the list is empty      \*/  **Iterator<T> end() const { return Iterator<T>( tailPtr ); }**     /\* insert inserts parameter node just before iterator  parameter position      \* Pre-condition: node type T has copy constructor      \*/     void insert( const Iterator<T>&, const Node<T>& );     /\* insertFront inserts node at beginning of list      \* Pre-condition: node type T has copy constructor      \*/     void insertFront( const Node<T>& );     /\* insertBack inserts parameter node at the end of the list      \* Pre-condition: node type T has copy constructor      \*/     void insertBack( const Node<T>& );      /\* erase removes node from list referred to by iterator  parameter      \*/     void erase( const Iterator<T>& );     /\* eraseFront removes the first node in the list  (not headPtr) if list isn’t empty  \*/  **void eraseFront() { erase( begin() ); }**     /\* eraseBack removes node from end of list if list is  not empty (not tailPtr)      \*/  **void eraseBack() { erase( --end()); }**     /\* traverseAndPrint traverses entire list and prints node  items to display.      \* Precondidtion: class T has overloaded <<      \*/     void traverseAndPrint() const;     /\* LinkedList destructor deallocates all nodes including  head and tail sentinels      \*/  **virtual ~LinkedList() { eraseList(); }**     /\* operator= erases current list then deep copies list  parameter to this list      \* Pre-condition: class T has overloaded copy constructor      \* Returns reference to this list      \*/     LinkedList<T>& operator=( const LinkedList<T>& );  private:     /\* eraseList deallocates all nodes including head and tail  sentinels. Used by destructor, operator= and copy ctor      \*/     void eraseList();     Node<T>\* headPtr;     Node<T>\* tailPtr;  // n=num nodes in list not counting sentinel head and tail     int n;  }; |

The beginfunction returns an iterator that points to the first node (of data, not the headPtr) in the list or the tailPtr if the list is empty. The end function returns an iterator that points to the last node in the list, the tailPtr. So, for an iterator, it

if ( it.begin() == it.end() ) // then the list is empty

In other code, we can use it.begin() and it.end()to loop though a list. For example, look at this implementation of the traverseAndPrint member function which traverses and prints the list from head to tail.

|  |
| --- |
| /\* traverseAndPrint traverses entire list and prints node  items to display.   \* Precondition: class T has overloaded <<  \*/  template<typename T>  void LinkedList<T>::traverseAndPrint() const {       Iterator<T> it( begin() );       for( ; it != end(); ++it ) {            cout << it.curPtr->item << endl;       }  } |

Now, let’s look at the code for the insert member function that has two parameters: the iterator that points to the node just after the one we want to insert and a node with the data we want to insert. In order to use this function, we need to have located the position where we want to put the new node. Calling this function with an iterator that refers to the first real data node, the one just after the headPtr, the return value of the list’s begin() function, for example, would insert the new node at the beginning of the list.

|  |
| --- |
| template<class T>  void LinkedList<T>::insert( const Iterator<T>& it, const Node<T>& node ) {       Node<T>\* next = it.curPtr; // node after new node       // node before new node  Node<T>\* prev = it.curPtr->prevPtr;       Node<T>\* newNode = new Node<T>(node);       newNode->nextPtr = next;       newNode->prevPtr = prev;       prev->nextPtr = newNode;       next->prevPtr = newNode;       n++;  } |

The other two insert member functions are easy to write using the more general insert above. The insertFront member function, for example, is just the code below that uses the list’s begin function as the node to insert before.

|  |
| --- |
| template<class T>  void LinkedList<T>::insertFront(const Node<T>& n ) {       insert( begin(), n );  } |

The erasemember functions are similar to the insert functions and follow the general algorithm for removing a node in a list.

Let’s go over one more member function, the overloaded operator= which performs a deep copy of one linked list to another. The single parameter is the linked list to copy, called list below. The algorithm is the same as it is for copying any linked list including the non-templated list class we have already completed, but it is covered again here since students often have trouble with it and it utilizes the new iterator class.

1. Delete this’ (the current/invoking object) original list (so there’s no memory leak)
2. Allocate this’ headPtr
3. thisPtr = headPtr (used to go through this’ new list)
4. iterator = list.begin() (starting at the beginning of the parameter list)
5. while ( iterator != list.end() )
   1. nodePtr = get new node with contents of iterator’s node
   2. Put the new node in the list by:
      1. nodePtr->prevPtr = thisPtr
      2. thisPtr->nextPtr = nodePtr
   3. thisPtr = nodePtr (incrememt this’ lists pointer)
   4. ++iterator (increment list’s iterator

// get tail node and attach it to list by with steps 6-8

1. tailPtr = new node
2. tailPtr->prevPtr = thisPtr
3. thisPtr->nextPtr = tailPtr

7.     set this’ size to list’s size

8.     Return a reference to the current object

The code for operator= is below.

|  |
| --- |
| template<typename T>  LinkedList<T>& LinkedList<T>::operator=( const LinkedList<T>& l ) {     // thisPtr goes through this' list, curPtr goes thru l's     Node<T>\* thisPtr, \*nodePtr;       // first, delete original list so no memory  leak     eraseList();       // get head of new list then copy l to this     headPtr = new Node<T>;     thisPtr = headPtr;     Iterator<T> it = l.begin();     while( it != l.end() ) {          nodePtr = new Node<T>( \*(it.curPtr) );          nodePtr->prevPtr = thisPtr;          thisPtr->nextPtr = nodePtr;          thisPtr = nodePtr;          ++it;     }     // finally, get tail node of new list     tailPtr = new Node<T>();     tailPtr->prevPtr = thisPtr;     thisPtr->nextPtr = tailPtr;     n = l.n;         // set this lists' size     return \*this;  } |

**Before class to be ready to start this week’s second Pair Programming:** In the pp9b project, create a new file called linkedList.h with the code given above and add it to your project.

**Pair Programming 9b:** Complete files node.h, iterator.h, and linkedList.h with all functions whose declarations are included in the class definitions. Add the input file b1.txt from the Pair Programming Assignment to your project. Copy the code from main9b.cpp and paste it into main.cpp. Read through this main function to understand what it will test. Complete, link, and execute the entire project using the assignment test cases to verify the program’s correctness. Regarding the location of the input file b1.txt:

1. When simply executing, put the input file in the same folder as the workspace file (e.g., pp8.workspace)
2. When using the debugger, put the input file in the same directory as the header and cpp files

# Standard Template Library (STL)

Instead of namespace statements like:

using namespace std;

Let’s start using specific using statements such as those shown below. It’s a best practice.

using std::cout;

using std::endl

using std::vector;

There are a lot of standard abstract data structures. We’ve seen some: stack, queue, and list, for example, with a lot of standard algorithms that apply to them. Pushing/popping values on/off a stack. Traversing a linked list. The C++ Standard Template Library (STL) contains

1. Classes that are lists and queues and other containers that implement these well-known abstract data structures
2. Iterators that let us access items in the containers in different ways (traversals, inserts, updates)
3. Well-known algorithms to use on the container classes such as searching and sorting

It looks like we didn’t really need to write all of that stack/queue/list code from scratch. So, why did I make you do it? And, why will I continue to make you do more of it? Well, I don’t know what job you’ll end up doing. Will you end up using the classes in the STL or will you end up writing those classes? Will you end up working at Google creating apps or will you be on the team that maintains and updates their amazing search engine? Will you use an Oracle database or will you write the code that \*is\* the Oracle database? Will you work for Valve and use their proprietary game engine to create new games, or will you be on the team that creates the game engine itself? I don’t know and neither do you. So, we need to start learning how things work on the inside. Not just how to use them.

All of the classes below are template classes with template functions in the C++ STL. All except the iterators are containers that are meant to hold a collection of items.

## vector

<http://www.cplusplus.com/reference/vector/vector/>

A vector is like an array except it can dynamically (while your program executes) shrink/grow in length. A vector is indexed starting at 0 just like an array. Its size is the number of elements in the vector. A vector also has a capacity which is the amount of space reserved to hold elements, basically the size of the dynamically allocated array inside the vector object. A vector also has a size which is the number of valid elements in that array. The vector class is a template class in the standard namespace.

#include <vector>

using std::vector;

To declare a vector with base type double and size 0:

vector<double> grades; // size = 0, capacity = 0

To declare a vector with a base type int, size 30 with all 30 integers initialized to 0 because 0 is the default value for a numeric type:

vector<int> ages(30);

If we declared a vector of 30 Book objects, the Book class default constructor is called 30 times to initialize all 30 books:

vector<Book> books(30);

AFTER YOU DECLARE A VECTOR, YOU CANNOT USE THE [] OPERATOR AS IN grades[0] = 99.9; THIS IS WRONG. To initially insert a vector element, you must either initialize it when you declare the vector (only in C++ 11) such as:

vector<double> grades = { 99.9, 88.8, 77.7 };

Or use the vector class push\_back member function. For example, to get 10 elements and initialize them to 0:

for( int i = 0; i < 10; i++ ) {

grades.push\_back( 0 );

}

Now, we have a vector of size 10, not 0, and, on our system, a capacity of 16. How do I know the initial capacity? I printed the capacity with:

cout << grades.capacity() << endl;

There’s also a size member function which returns the number of valid elements, and now that I have some elements, I can use [] to access them:

int s = grades.size();

for( i = 0; i < s; i++ ) {

cout << grades[i] << “ “;

}

Remember to use the vector push\_back member function for all new elements. So, if I wanted an 11th element in the grades vector, I’d should use push\_back:

grades.push\_back( 99.9 );

You can create a vector of objects:

// a vector of Book objects with size 0

vector<Book> b;

// a vector of 2 Book objects, both initialized by

// calling the Book class default constructor twice

vector<Book> b2(2);

You can resize the vector:

b.resize( 4 );

for( i = 0; i < b.size(); i++ ) {

cout << b[i] << endl;

}

How are these two new Book objects initialized? With the default constructor of the Book class. When you resize, you don’t need to call the push\_back function to add the elements. You just added them with resize.

When resizing a vector of pointers to objects, adding elements to the vector just adds pointers. You still have to make those pointers point to something as shown below.

vector<Book\*> bookPtrs;

bookPtrs.resize(2);

bookPtrs[0] = new Book( "123", "a", "b", "c", 1, 2 );

bookPtrs[1] = new Book( "xyz", "z", "y", "x", 9, 8 );

cout << \*bookPtrs[0] << endl << \*bookPtrs[1] <<endl;

In general, it’s not a good idea to have a container of pointers. Imagine you want to sort books by title, but you have a vector of pointers to books, not a vector of books. Then, the STL sort function will be sorting pointers, not books, because the vector is a vector of pointers, not a vector of books.

Remember, if a class that you write dynamically allocates memory then you MUST write a destructor for it (or you’ll get memory leaks when you remove these objects from the vector) and you MUST overload the assignment operator (and copy constructor) or

vector1[i] = vector2[i];

will only do a shallow copy.

A note about efficiency: every time the vector class has to go get more memory, time is added to execution. If a vector has a capacity of 16 and the code adds a 17th element, the vector class dynamically allocated a second array, copies the contents of its original array into the second one, adds the data for the 17th element, and deallocates the original array. That’s a lot of work, sp if you know about how many elements you’ll need, you can specify the capacity using a a vector constructor as shown above or the reserve vector member function:

// maxStudents doesn’t have to be a constant

grades.reserve( maxStudents );

When an STL vector whose base type is a class (like a vector of Book objects) is destroyed for example, with the pop\_back()member function or when the vector goes out of scope, the destructor is called for every element in the vector that is removed. Copying of vector elements for example, with v.push\_back(book), invokes the class’ copy constructor, the Book class’ copy constructor in this case).

I don’t know if you noticed yet, but a lot of the functionality of the vector template class, when it base type is a class, requires this base type to have some member functions. If you have a vector of Book objects, for example, then your Book class had better have a default constructor or

vector<Book> b(2);

which creates a vector of two Book objects initialized by calling the Book default constructer twice, won’t work. If you have code that uses push\_back to put a Book object in the vector, then the Book class better have a copy constructor. And, if you ever assign one vector element to another then the = operator for your class better work.

**Optional before class:** Create another project in the IDE and type code in main.cpp to get a vector with a base type of double and a vector with a base type of Book. Give both vectors some elements with the push\_back member function. Use the [] operator to access the elements (i.e., print them out). Basically, write a little test so you know you understand how to use vectors of different types and how to use the vector member functions. A sample answer will be posted in our online Course Content.

## Iterators

Iterators are like pointers. They are used to reference an item (e.g., element) in a container (e.g., vector) and traverse the container in different ways (e.g., forward in a list, backward in a list). They operate like the iterator class for out linked list. Each container has its own iterator types. For example, maps and sets don’t have numbered elements so someMap[i] doesn’t make any sense. A linked list is not a randomly accessed data structure so someList[i] also doesn’t make sense.

Declaring an iterator is a little long. Here are three examples:

// an iterator for a vector of doubles

vector<double>::iterator p;

// an iterator for a list of Book objects

list<Book>::iterator bookIterator;

// an iterator for a map of Book objects with a string key

map<string, Book>::iterator mapIterator;

We don’t always need to declare an iterator to go through the items in a container. To go through the items in a vector, we can just use the overloaded operator[]function (whose implementation uses an iterator, actually).

All container template classes like vector have the following member functions that return an iterator for that container:

* begin(): returns an iterator for the container that refers to the “first” data item in the container just like our iterator class for our templated linked list.
* end(): returns an iterator for the container that doesn’t really refer to a data item, but a value just past the last valid data item, like the fake tail node in our templated linked list class It’s used a NULL pointer to determine when to stop iterating/recursing through a container. Actually, it returns a fake “end” node just like the iterator of our linked list above.

vector<double>::iterator it = v.begin();

vector<double>::iterator itEnd = v.end();

for( ; it != itEnd; it++ ) {

cout << \*it << " ";

}

Note the use of the dereferencing operator, \*, to access the vector element with the iterator, treating the iterator like a pointer. Again, this is just like the iterator of our templated linked list class. The code above also used the != operator to compare the two iterators it and itEnd, so the iterator classes must have overloaded == and != operators. Below are operators that may be overloaded in some container classes (but not all):

* ++ (prefix and postfix)
* -- (prefix and  postfix)
* == and != which tests if two iterators point to the same item (NOT if two separate items are equal/not equal).
* \* dereferencing operator. Sometimes it allows write access and sometimes read-only. Depends on the iterator. Some iterators are constant and can’t be used to alter what they refer to.

Different kinds of iterators are listed below. They all come in two forms: constant and mutable. An item in a container may not be altered by a constant iterator.

* Forward iterators: ++ will work
* Bidirectional iterators: ++ and -- will work
* Random access iterators: ++, -- and [] and/or (it + *n*) will work

Not all of these iterators make sense for a particular container class. For example, an iterator can move forward and backward in a list, but random access in a list doesn’t make sense. We’ve written code for a list, and it doesn’t make sense to write: myList[i]. How would you implement that? A list doesn’t have any kind of index. A doubly-linked list has a beginning and an end but no way to get something in the middle without traversing the list to that item. The iterator it below

vector<double>::iterator it;

is a mutable, random access iterator because it’s not constant, and it’s used with a vector. If it were a list iterator, it would be a mutable, bidirectional iterator because lists are not random access data structures.

Some iterator types:

* iterator
* const\_iterator
* reverse\_iterator <http://www.cplusplus.com/reference/iterator/reverse_iterator/?kw=reverse_iterator>
* const\_reverse\_iterator

TO USE A REVERSE ITERATOR, YOU DON’T START AT THE END USING THE END FUNCTION. Don’t start at:

vector<double>::reverse\_iterator it = v.end(); X WRONG!

because the end is not a real element/item. So, if you tried: it-- on this iterator, where would that take you? That’s like NULL-- . Instead, use special begin and end functions meant for a reverse iterator, rbegin and rend, respectively. rbegin returns a reference to the last, real item in the container and rend returns a value to indicate one past the first item in the container – like our linked list’s fake head node.

vector<double>::reverse\_iterator it = v.rbegin();

for( ; it != v.rend(); v++ ) {

// do something with \*it

Note: don’t try to declare a C++ iterator object such as

iterator<int> it; // WRONG

This class is for creating your own iterator class. It doesn’t have the functionality of an iterator in it. You have to create a class that inherits from the iterator class and write that functionality yourself which is beyond the scope of this course.

## list, deque

<http://www.cplusplus.com/reference/list/list/>

<http://www.cplusplus.com/reference/deque/deque/>

Lists and queues are sequential containers (as opposed to associative containers like map and set that we will look at below) which have a first element, a second element, etc. They are both bidirectional but not random access. The STL list is a doubly-linked list. The function push\_back is used to add an item to a list.

The STL deque is a doubly-ended queue which means you can shift/unshift and push/pop so it’s like a queue and a stack all rolled into one. The functions push\_back and push\_front are used to add items to the back and front of the queue, respectively. with pop\_back and pop\_front removing queue items.

## stack, queue, priority\_queue

<http://www.cplusplus.com/reference/stack/stack/>

<http://www.cplusplus.com/reference/queue/queue/>

<http://www.cplusplus.com/reference/queue/priority_queue/>

Stacks, queues and priority queues are also sequential containers that are implemented with the list template class. They are called adapter containers because they are implemented using other containers. The stack template class is implemented with deque. The queue template class is implemented with deque. The priority\_queue is implemented with vector. A priority queue is a queue where items not only have data (and, maybe, a key), but also have a priority. If all elements have the same priority then they are removed/added as with a regular queue, First-In-First-Out (FIFO). If items have different priorities then all higher priorities (in FIFO order) are removed before lower priority elements.

Stupid compiler idiosyncrasy: Before C++11, if you wanted a template of a template, a vector of pairs, for example, you had to have a space between the two >  >.

vector<MyPair<string, Book>   > vb;

NOT vector<MyPair<string, Book>> vb;

Although, I’d probably use a map for a container of pairs since it’s made for key/value pairs (see below). It’s always important to choose the right data structure based upon your data and what you want to do with it.

## set, map

<http://www.cplusplus.com/reference/set/set/>

<http://www.cplusplus.com/reference/map/map/>

These are associative containers. Data structures that mimic a database in the sense that they store key/value pairs. So, a value isn’t indexed by a numbered index like an array or vector. It is “indexed” by a key. Our MyPair template class is an example. What if we could create an entire container of a bunch of MyPair<string, Book> objects, a map called bookMap. We wouldn’t use an index like 0 or 1 to access an object. Instead we might use the ISBN of a book because that’s a value unique to every book which is a good key. So, the pair would be a pair with an ISBN key (a string) and a book value (the Book object with that ISBN number). Think of this bookMap as an array with ISBN indices such as bookMap[“98-123-456-X”] instead of an index like 0 as in bookMap[0].

The STL set template class stores items without repetition, no duplicate keys. You can add items, delete items, ask whether an item is in the set or not. A set stores items in order and you can specify the ordering although this is optional. If I had a set of Book objects, for example, I might specify the ordering be by book title, smallest to largest. Or, maybe, I would want to order by ISBN. If you don’t specify ordering then the < operator of the value class of the key/value pair is used, so your class, like the Book class, operator< had better be overloaded!

**Optional before class:** Create another project in the IDE. Write code in the main.cpp file that creates a set of lowercase letters and put a few letters in it with the insert function (read the set documentation accessed via the link above for help).

charSet.insert( ‘z’ );

Print the set. Remove a letter. Print the set again. Use a loop to look through the set for every lowercase letter a – z and print found (if the letter is in the set) or not found. Hint: you can use a loop where the loop control variable is a char:

for( i = 'a'; i <= 'z'; i++ )

Then, use the find function to look for the letter in variable i. An answer will be posted in our online Course Content.

The STL map template class uses the pair template (also in the STL). So, to use a map, you can use a pair (found in the header file <utility>)

pair<string, int> p(“c++”, 10);

map<string, int> mymap;

mymap.insert( p );

or, without the pair

mymap[“c++”] = 10;

You can also create a map of objects. Read through the code below (#include and using statements left out).

|  |
| --- |
| int main() {     Book b1("1", "a1", "t1", "p1", 1, 1 );     Book b2("2", "a2", "t2", "p2", 2, 2 );     map<string,Book> bookMap;     bookMap[b1.getIsbn()] = b1;     bookMap[b2.getIsbn()] = b2;       if ( bookMap.find("1") != bookMap.end() ) {        cout << "Found 1\n";     }     if ( bookMap.find("3") == bookMap.end() ) {        cout << "NOT Found 3\n";     }       map<string, Book>::iterator it;     for( it = bookMap.begin(); it != bookMap.end(); it++ ) {        cout << it->second << endl;     }     return 0;  } |

The code example above instantiates some Book objects and a map with a string key (ISBN) and a Book value (the book object with that ISBN). Notice how I call the Book member function getIsbn() as the “index”, the key, when I assign a book to an element in the map. The code then tests the find map member function and uses an iterator to print out all of the books. Since the map really does store information in a pair (even though I didn’t explicitly have a pair), to print each book, I have to print the map data member, second (a pair has data members first, the key, and second, the value) with it->second. Recall the iterator is a pointer.

**Before class to help you get ready for an extra credit lab!** Create another project in workspace pp9 with the code of the main function from above. Include all of the correct header files and using statements. Don’t just use “using namespace std;”. Compile and execute.

## Algorithm

[**http://www.cplusplus.com/reference/algorithm/**](http://www.cplusplus.com/reference/algorithm/)

The algorithm header file in the STL defines a collection of functions designed to be used on many of the STL containers. We’ll just look at one: sort. (You may never write a sort again … well, after this class, anyway). The sort function basically takes two parameters: an iterator to the first thing in the data structure to be sorted and an iterator that points to one past the last thing in the data structure to be sorted. Sound familiar – that begin and end just like our templated linked list? Assume we already have a vector, v, that we’ve added a lot of Book objects to. The Book class MUST have an overloaded operator< because the sort function in the algorithm header file will use it by default to compare one Book object to another. To sort, we just need to:

sort( v.begin(), v.end() );

and our vector of books will be sorted by title since the book class’ operator< we already wrote returns the difference of two titles.

Remember: If you want to use STL data structures and any algorithms on objects of classes you write, them those classes need to have

* Copy constructor
* Destructor
* Overloaded relational operators like <
* Overloaded = operator

## Comparator Classes

In order to use the sort template function in the algorithm header file, we need to have a way to compare the objects we are sorting. Above, we overloaded operator< in the class whose objects we wanted to sort. We could, instead, create a comparator class that has code in it that defines what it means to compare two objects. In C++, a comparator class is a class with one member function in it, overload operator(). This function takes two parameters, the objects we want to compare. Let’s call the first parameter a and the second b. The function returns true if a < b otherwise it returns false.

Consider the two comparator classes below. One compares two Book titles and one compares two Book ISBNs.

|  |
| --- |
| class bookTitleMore {  public:  bool operator()(Book b1, Book b2) {  return b1.title > b2.title;  }  };  class bookIsbnLess {  public:  bool operator()(Book b1, Book b2) {  return b1.isbn < b2.isbn;  }  }; |

In order for the above code to work, the Book class must make the two comparator classes above friends.

I can instantiate a bookIsbnLess object and use it to compare two Book objects, b1 and b2:

bookIsbnLess isbnLess;

if ( isbnLess( b1, b2 ) ) // then b1’s ISBN < b2’s

The sort template function is overloaded. One of its versions takes three parameters: beginning and ending vector locations and a comparator object like one of the two above. So, if I had a vector of Book objects, the code below would sort that vector in two different ways:

bookIsbnLess isbnLess;

sort( v.begin(), v.end(), isbnLess );

// then do what you want with the vector sorted by ISBNs

bookTitleMore titleMore;

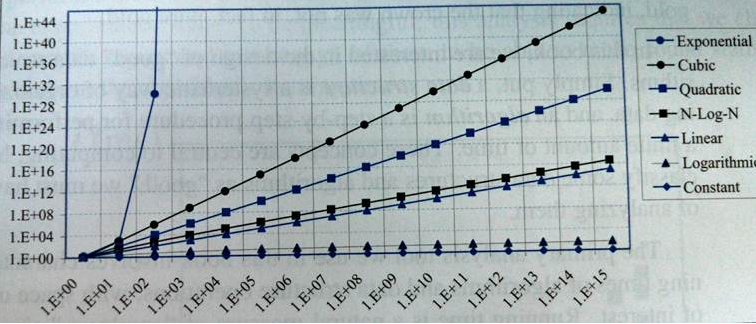
sort( v.begin(), v.end(),titleMore );

// then do what you want with the vector sorted by title

# Algorithms and Big-O notation

Some functions where *c* is a constant and *n* is the number of items operated on and a graph showing their growth as *n* increases are shown below. Note the scale of the axes.

* Constant: *f(n) = c*
* Logarithmic: *f(n) = log n*
* Linear: *f( n ) = n*
* N-Log-N: *f( n ) = n log n*
* Quadratic: *f( n ) = n2*
* Cubic: *f( n ) = n3*
* Exponential: *f( n ) = cn*



(Image from Data Structures and Algorithms by Goodrich, et al)

We use these functions to describe the efficiency of operations/algorithms. For example, consider popping a value off a stack as the operation. How much time as a function of the number of items on the stack, *n*, does it take? Well, it doesn’t matter how many items are on the stack, the time to pop doesn’t change. So, we say that popping an item off the stack is a constant function, *f( n ) = c*. Theoretically/mathematically, we don’t really care what the constant *c* is. In practice, though, we might care what the constant is.

Examples of constant time operations are shown below. It doesn’t matter what the constant is, whether you have 10 arithmetic operations or just one.

* Variable assignments such as x = y
* Arithmetic operations such as x + y
* Logical or relational comparisons such as: a || b, x < y
* Indexing an array or dereferencing a points such as a[i] or \*ptr

Examples of logarithmic operations:

* A binary search of a sorted array
* A search in a binary search tree (we’ll cover how to do this later in the semester)

Examples of linear operations:

* Searching in a non-sorted array (key to find could be at the end)
* Inserting in a list, in order (might have to insert at the end of the list)

Examples of N-log-N operations:

* Some sorts such merge sort (we’ll study these later)

Examples of quadratic operations:

* Selection sort and bubble sort
* Any algorithm with nested loops, both loop control variables going from 0/1 to n (number of items to be operated on)

Big-O notation gives us a way to talk about the efficiency of algorithms in terms of the worst-case scenario. It is an upper-bound estimate and is not a count of the number of operations for a single execution of an algorithm on a particular data set. When calculating the number of operations needed to complete an algorithm, we could get results such as (where *n* is the number of data items to process):

6*n* + 3

*n*2 + *n* + 10

In the first equation, the constant 3 isn’t really important. Take n = 100. Is it the 6\*100 = 600 operations that matter or the 3? That’s still a 3 no matter how many items we have to process. Even the ‘6’ isn’t that important. Basically, we have an algorithm whose number of operations increases linearly with the number of elements: a straight line. Theoretically, the slope, the 6, doesn’t matter. In general, any algorithm that executes with *a n+b* operations where *a* and *b* are constants and *n* is the number of data items, is considered a linear algorithm of O*(n)*.

In the second equation, the dominant term is *n2*. Consider *n* = 1024. The *n2* = 1,048,576 while *n* is just 1,024. The constant 10 is definitely irrelevant. An algorithm whose calculated number of operations is of the form *ax2 + bx + c* is a quadratic algorithm of O(*n2*.). This is a pretty slow algorithm. Bubble sort and selection sort are both O(*n2*) algorithms. Take a look at the general form to see:

for ( i = 0; i < n; i++ ) { // executes n times

for ( j = 0; j < n; j++ ) { // executes n times

// do stuff – these statements execute n2 times

If the first loop executes *n* times and each time that loop executes, the inner loop executes *n* times, then the code to do stuff executes *n2* times. So, O(*n2*). I know this may not be \*exactly\* what the for loops look like in each sort. In the selection sort, for example, the second loop’s counter looks more like: j = i+1; j < n; j++. So the loop executes *n-1* times the first execution, *n-2* the second, *n-3* the third, etc. But that’s just *(n-1)!* which might as well be *n!* which, if you look at a graph, grows “quadratically” and definitely NOT linearly.