OOP 3200 – Object Oriented Programming II

Week 7 – The C++ Standard Template Library (STL)

Week 7 Overview

- C++ STL
- In-class Exercise 6
- Quiz 2
- C++ Lab 5

Course Outline

Week	Date	Topic	Evaluation	Weight
1	Sep 09, 2020	- Course Orientation - Object-Oriented Programming overview - Partnering for labs		
2	Sep 16, 2020	REVIEW OF CLASSES & OBJECTS in C++ - Encapsulation - Object Attributes and Behaviours - Classes: The Blueprint for Objects - Relationship Between Class and Objects - Static Class Members - Friend Functions	In-Class Exercises 1: (2%) C++ Assignment 1: (6%)	8
3	Sep 23, 2020	CLASS OPERATORS AND DATA TYPE CONVERSIONS in C++: - Creating Class Operators - How Methods Are Shared - Data Type Conversions	In-Class Exercises 2: (2%) C++ Assignment 2: (6%) C++ Quiz 1: (5%)	13
4	Sep 30, 2020	INHERITANCE AND POLYMORPHISM in C++: - Class Inheritance - Polymorphism - Virtual functions - Interfaces / Abstract Classes	In-Class Exercises 3: (2%) C++ Assignment 3: (6%)	8
5	Oct 07, 2020	COLLECTIONS in C++: - Dynamic Object Creation and Deletion - Pointers As Class Members / Destructors - Copy Constructors / Copy Assignment Operators	In-Class Exercises 4: (2%) C++ Assignment 4: (6%)	8
6	Oct 14, 2020	GENERICS in C++ - Method templates - Class Templates	In-Class Exercises 5: (2%)	2
7	Oct 21, 2020	THE C++ STANDARD TEMPLATE LIBRARY (STL): - Vectors and Linked Lists - Stacks and Queues - Maps and Sets	In-Class Exercises 6: (2%) C++ Assignment 5: (6%) C++ Quiz 2: (5%)	13
READING WEEK				

Objectives

- In this chapter, you will learn about:
  - The Standard Template Library
  - Linked Lists
  - Stacks
  - Queues
  - maps
  - Common Programming Errors
  - Addendum – File Operations

Introduction to the Standard Template Library

CONCEPT :

- The **Standard Template Library (or STL)** contains many templates for useful algorithm s and data structures.

- In addition to its run-time library , which you have used throughout this course , C++ also provides a library of templates . **The Standard Template Library (or STL)** contains numerous templates for implementing data types and algorithms.

- The most important data structures in the STL are the **containers** and **iterators**.
  - A **container** is a class that stores data and organizes it in some fashion.
  - An **iterator** is an object that works like a pointer and allows access to items stored in containers .

Sequential Containers

- There are two types of container classes in the STL:
  - sequential containers
  - associative containers

- **Sequential containers** store items in the form of **sequences**, meaning that there is a natural way to order the items **by their position** within the container.

- An **array** is an example of a sequential container.

DURHAM COLLEGE
SUCCESS MATTERS

Sequential Containers (continued)

- Because a sequential container organizes the items it stores as a sequence, it can be said to have a **`front`** and a **`back`**.
- A container is said to provide **random access** to its contents if it is possible to specify a position of an item within the container and then jump directly to that item without first having to go through all the items that precede it in the container.
- Positions used in **random access** are usually specified by giving an **integer** specifying the position of the desired item within the container.
- The integer may specify a position relative to the beginning of the container, the end of the container, or relative to some other position.
- **Arrays** and **vectors** are examples of sequential containers that provide random access.

Associative Containers

- **Sequential containers** use the position of an item within the sequence to access their data.

- In contrast, **Associative Containers** associate a **key** with each item stored and then use the key to retrieve the stored item.

- A telephone book is an example of an associative container; the values stored are telephone numbers, and each telephone number is ***associated*** with a name. The name can later be used as a **key** to look up, or retrieve, the telephone number.

Associative Containers (continued)

- A **map** is a container that requires each **value** stored to be associated with a **key**.
- Each **key** may be associated with **only one value**; once a key is used, no other value with the same key may be added to the map.
- A **`multimap`** is like a **`map`**, except a **key** may be associated with **multiple values**.
- A **`set`** is like a **`map`** in which only keys are stored, with **no associated values**. No item may be stored twice in a set: That is, duplicates **are not permitted** .
- A **`multiset`** is like a **`set`** in which **duplicates are permitted** .

Iterators

- **Iterators** are objects that **behave like pointers**. They are used to **access items** stored in **containers**.

- A typical **iterator** is an **object of a class** declared inside a container class.

- The **`iterator` overloads pointer operators** such as the increment operator **++** , the decrement operator **`--`** , and the dereferencing operator * in order to provide **pointer-like behaviour**.

- Each STL container object provides member functions **`begin()`** and **`end()`** that return the **beginning** and **ending `iterators`** for the object.

- The **`begin()` `iterator`** points to the item at the beginning of the container if the container is non-empty, while the **`end()` `iterator`** points to **just past the end** of the container.

The Use of Iterators

- Each STL container class defines an inner class called **`iterator`** that can be used to create iterator objects. For example,

```
vector<int>::iterator
list<string>::iterator
```

- are the inner classes that represent iterators to containers of type **`vector<int>`** and **`list<string>`**, respectively.

- Here is an example of how to define iterators for a vector of **`int`** and a **`list`** of **`string`** and initialize both iterators to the beginning of the container:

```
vector<int> vect;
list<string> mylist ;
vector<int>::iterator vIter = vect.begin();
list<string>::iterator listIter = mylist.begin();
```

The Use of Iterators (continued)

- In C++ 11, you can declare the type of an iterator as auto and the compiler will infer its correct type from the type of the container:

```
auto vIter = vect.begin();
auto listIter = mylist.begin();
```

- Moreover, C++ provides the **`begin(c)`** and **`end(c)`** functions, which when applied to an STL container **`c`**, return the **`c.begin()`** and **`c.end()`** iterator, respectively.

- So we can use the alternative forms:

```
auto vIter = begin(vect);
auto listIter = mylist.begin(mylist);
```

The Use of Iterators (continued)

- An iterator works like a pointer that indicate s a position within a container. If **`iter`** is an **iterator**, then **`*iter`** represents the **element stored** in the container at the position of the **iterator**.

- Moreover, writing **`iter++`** causes the iterator to advance to the next position in the container. The **`end()`** iterator always indicates a position past the last element in the container, so it should **never be dereferenced**.

- The **`end()`** iterator is only used as a sentinel to prevent "falling off the end" of a container.

The Use of Iterators (continued)

- For example, the following code will print all values in a vector:

```
vector<int> vect { 10 , 20 , 30 , 40 , 50} ;

auto iter = vect.begin();
while (iter != vect.end())
{
   // Print element at iter and advance
   cout << *iter << " ";
   iter ++ ;
}
```

The Standard Template Library (cont'd.)

Table 13.1 STL Lists

List Type	Classification	Use
Vector	Sequence	Dynamic arrays
List	Sequence	Linked lists
Deque	Sequence	Stacks and queues
Set	Associative	Binary trees without duplicate objects
Multiset	Associative	Binary trees that might have duplicate objects
Map	Associative	Binary trees with a unique key that doesn't permit duplicate objects
Multimap	Associative	Binary trees with a unique key that permits duplicate objects

The Standard Template Library (cont'd.)

- STL **`list`** types:
  - **Sequence `list`**: list object determined by its position in list
  - **Associative `list`**: automatically maintained in sorted order

- Example: alphabetical list of names is an associative list
  - Depends on name and sorting criteria rather than exact order that names were entered onto list

The Standard Template Library (cont'd.)

- **Arrays** and STL lists are both referred to as containers (collections)

- **Arrays** (built-in list types): most often used to store built-in numerical data types
  - Retain general characteristics common to more advanced lists provided by STL

- STL **`lists`** (class types): must provide means for accessing individual objects
  - Data structure: list that provides for individual data location

The Standard Template Library (cont'd.)

- STL **`lists`** are commonly used to store and maintain objects (records)
  - Lists can also store built-in data types
- Once an object's structure has been defined, a means for collecting all of the objects into single list is required
- A mechanism is also needed to store all of the records in some order
  - Allows individual records to be located, displayed, printed, and updated

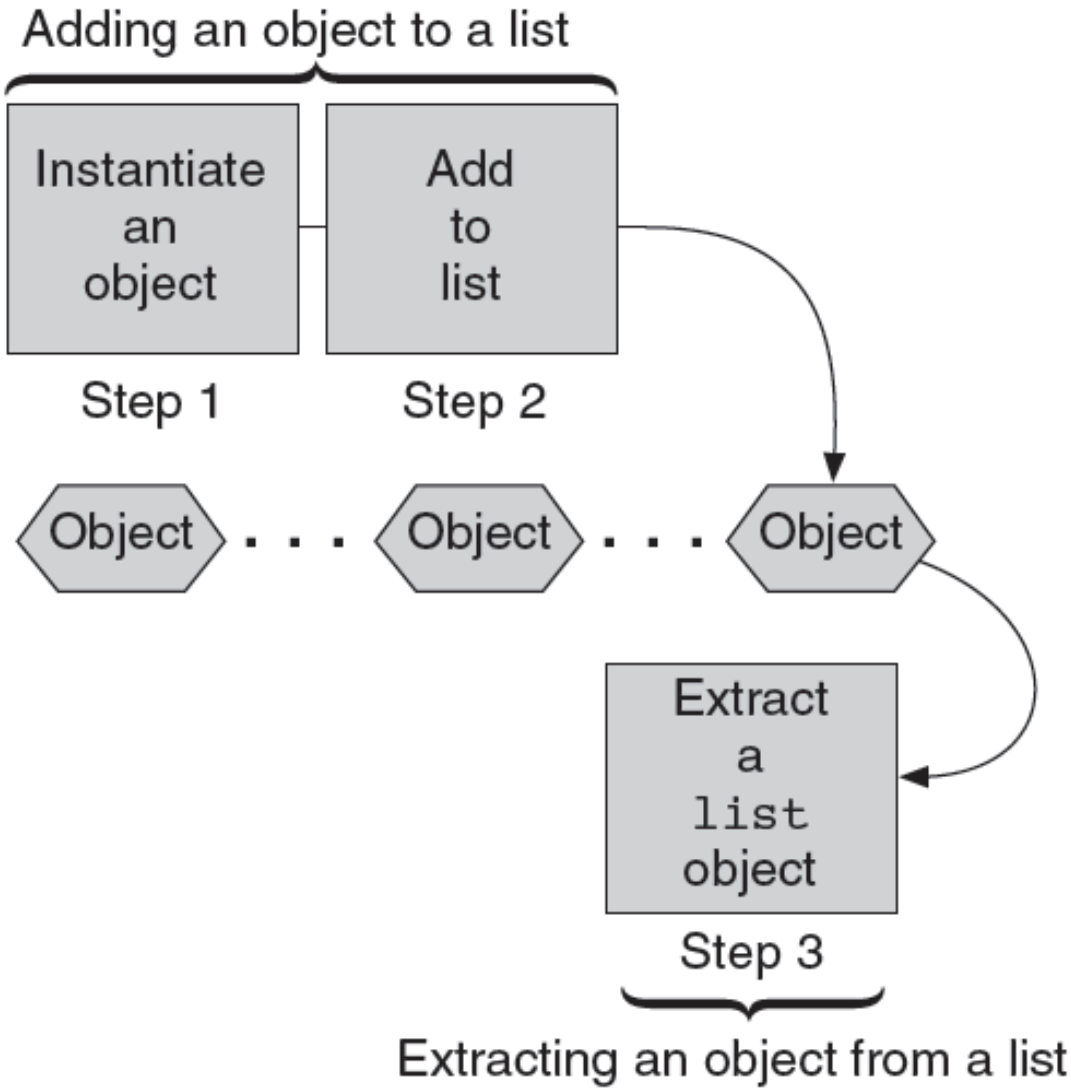


Figure 13.1 The list creation process

The Standard Template Library (cont'd.)

- STL class methods: provided for each STL class
- STL general methods (algorithms): can be applied to objects stored in any STL-created list
  - Listed in Table 13.2
- **STL iterators**: used to specify the means of accessing list objects
  - Iterators operate in a similar manner as indices do for arrays

The Standard Template Library (cont'd.)

- Procedure for creating and using STL lists:
  - Use STL class to construct desired container type
  - Store objects within list
  - Apply either STL class's methods or more general STL algorithms to stored objects

- These steps will be used in the following sections to create three commonly used lists:
  - Linked lists, **stacks**, and **queues**

Linked Lists

- Classic data-handling problem: making additions or deletions to list of objects maintained in specific order
- Example: alphabetical telephone list
  - Want to add new objects to list such that alphabetic ordering of objects is maintained
- Arrays or vectors are not efficient representations for adding or deleting objects internal to list
  - Creates an empty slot that requires shifting up all objects below deleted object to close empty slot

Linked Lists (cont'd.)

- Linked list provides method for maintaining a constantly changing list without the need for continually reordering and restructuring
- Each linked list object contains one variable that specifies the location of the next object in the list
  - This variable is a pointer
- It is not necessary to physically store each object in proper order
  - Instead, each new object is physically stored in whatever memory space is currently free

Linked Lists (cont'd.)

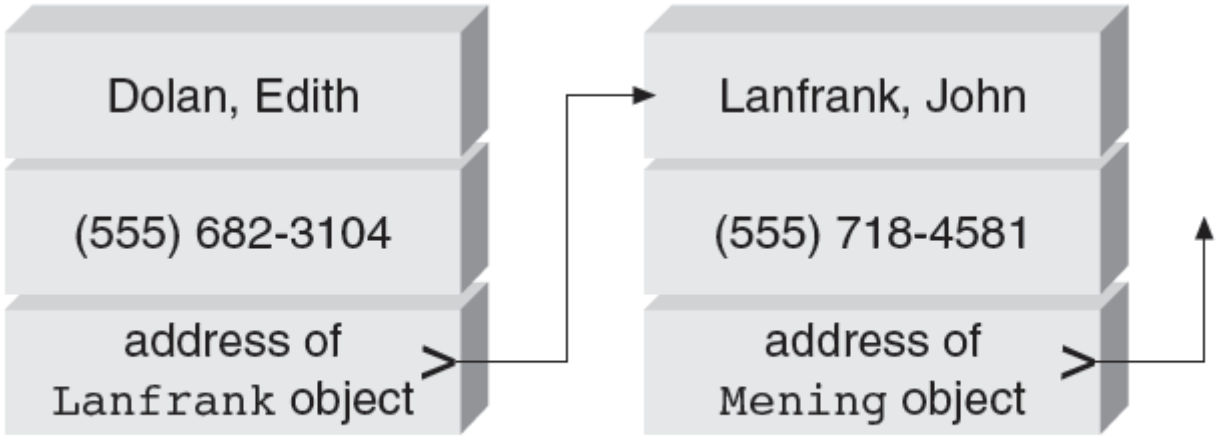


Figure 13.2 Using pointer variables to link objects

Linked Lists (cont'd.)

- Use of pointer variable
  - Add June Hagar to alphabetical list shown in Figure 13.3
  - The data for June Hagar is stored in data object using the same type as that used for existing objects
  - Value in pointer variable in Dolan object must be altered to locate Hagar object
  - Pointer variable in Hagar object must be set to the location of Lanfrank object

Linked Lists (cont'd.)

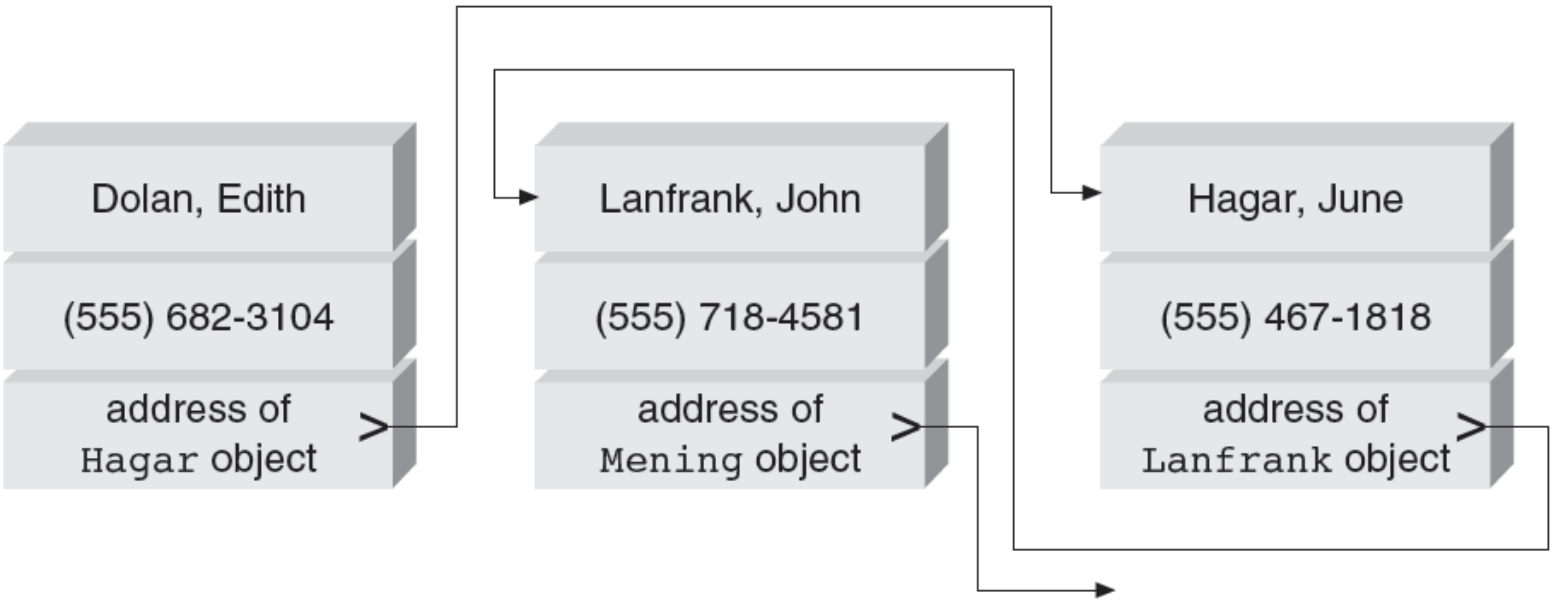


Figure 13.3 Adjusting pointer variables to point to the correct objects

Linked Lists (cont'd.)

- Pointer variable in each object locates next object in list, even if that object is not physically located in the correct order

- Removal of object from list: done by changing pointer variable's value in object preceding it to location of object immediately following deleted object
  - Removal is the reverse process of adding an object

Linked Lists (cont'd.)

- There are two fundamentally different approaches to constructing a linked list
  - Use STL list class (recommended approach)
  - Programmer develops own list class
    - Includes an object's declaration and the code for creating and maintaining the list

- Major benefit of STL list class: can be constructed without programmer having either to understand or program internal details of pointer variables

Using the STL `list` Class

- **Link variables**: contain location information, allowing for access of individual objects of a linked list (Figure 13.4)
  - A new object is inserted into list simply by storing new object in any available memory location and adjusting location information in at most two link variables
  - Not necessary to store list objects in contiguous memory locations (as with arrays)
  - An object is removed by adjusting link information in two link variables

Using the STL `list` Class (cont'd.)

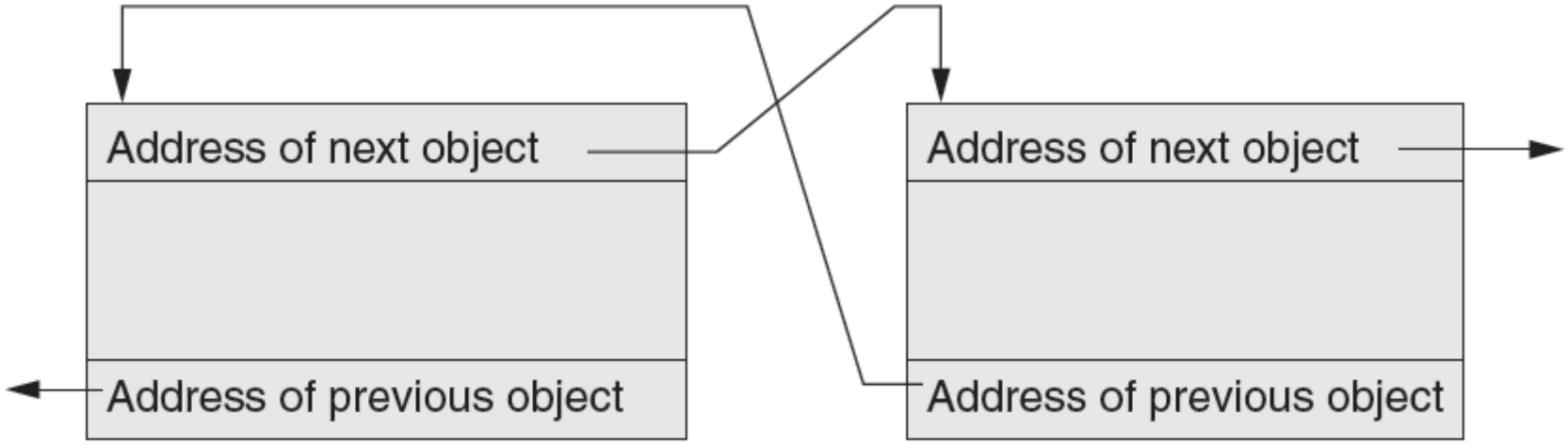


Figure 13.4 A class with four link variables

Using the STL `list` Class (cont'd.)

- Methods included in the **`list`** class (Table 13.3): provide for adding, removing, and locating objects from front and rear of list
  - No random access
  - To access any internal object, list must be sequentially traversed from front or back of list
- "Popping" (removing) an object from the **`list`**: requires traversing list and removing all objects before desired object
- Program 13.1 demonstrates use of lists

Using the STL `list` Class (cont'd.)

Program 13.1

```
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

int main()
{
  list<string> names, addnames;
  string n;
  // add names to the original list
  names.push_front("Dolan, Edith");
  names.push_back("Lanfrank, John");
```

Using the STL `list` Class (cont'd.)

Program 13.1

```
    // create a new list
    addnames.push_front("Acme, Sam");
    addnames.push_front("Mening, Stephen");
    addnames.push_front("Zemann, Frank");
    names.sort();
    addnames.sort();
    // merge the second list into the first
    names.merge(addnames);
    cout << "The first list size is: " <<
        names.size() << endl;
    cout << "This list contains the names:\n";
    while (!names.empty())
    {
      cout << names.front() << endl;
      names.pop_front(); // remove the object
    }
}
```

Using the STL `list` Class (cont'd.)

- The output produced by Program 13.1 is:

```
The first list size is: 5
This list contains the names:
Acme,Sam
Dolan,Edith
Lanfrank,John
Mening, Stephen
Zemann,Frank
```

Using the STL `list` Class (cont'd.)

```
2  // Standard library list class template.
3  #include <iostream>
4  #include <vector>
5  #include <list> // list class-template definition
6  #include <algorithm> // copy algorithm
7  #include <iterator> // ostream_iterator
8  using namespace std;
9
10 // prototype for function template printList
11 template <typename T> void printList(const list<T>& listRef);
12
13 int main() {
14    list<int> values; // create list of ints
15    list<int> otherValues; // create list of ints
16
17    // insert items in values
18    values.push_front(1);
19    values.push_front(2);
20    values.push_back(4);
21    values.push_back(3);
22
23    cout << "values contains: ";
24    printList(values);
25
26    values.sort(); // sort values
27    cout << "\nvalues after sorting contains: ";
28    printList(values);
29
30    // insert elements of ints into otherValues
31    vector<int> ints{2, 6, 4, 8};
32    otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
33    cout << "\nAfter insert, otherValues contains: ";
34    printList(otherValues);
35
36    // remove otherValues elements and insert at end of values
37    values.splice(values.cend(), otherValues);
38    cout << "\nAfter splice, values contains: ";
39    printList(values);
```

Stacks

- **Stack**: special type of list in which objects can only be added to and removed from the top of list
  - Stack is **last-in, first-out (LIFO)** list

- Example: stack of dishes in cafeteria, where last dish placed on top of stack is first dish removed

- In computer programming, stacks are used in all function calls to store and retrieve data to and from function

Stacks (cont'd.)

- Stack example
  - Has a list of three names
  - If list access is restricted so that names can only be added and removed from the top of list, then list becomes a stack
  - Top and bottom of list must be identified
    - Since Barney is above the other names, he is considered the top of list (designated by arrow)
- Another stack example (Figure 13.7): illustrates how stack expands and contracts as names are added and deleted

Stacks (cont'd.)

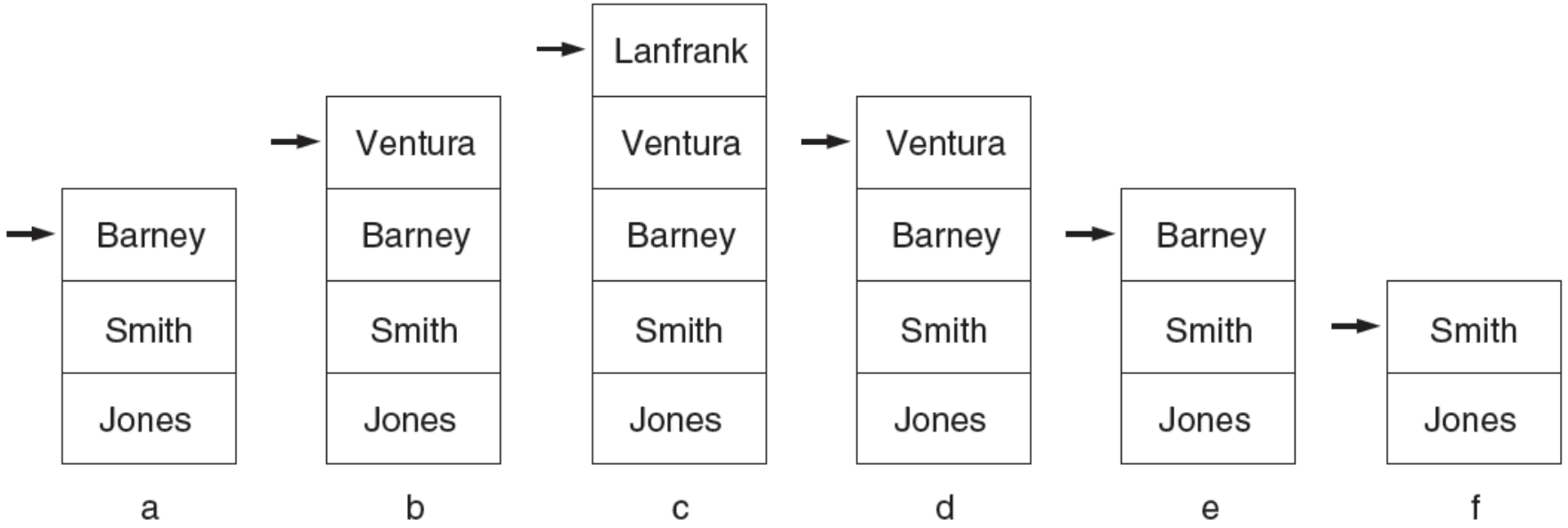
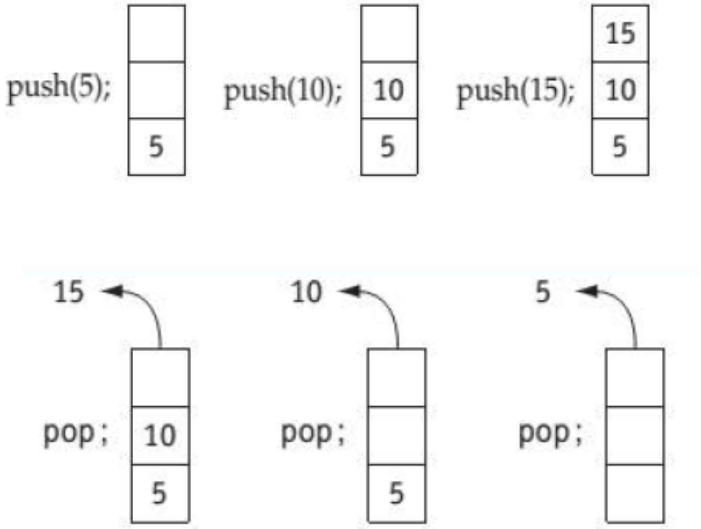


Figure 13.7 An expanding and contracting list of names

Stack Implementation with the `deque` Class

- Creating stack requires the following four components:
  - Container for holding items in list
  - Method of designating current top stack item
  - Operation for placing a new item on stack
  - Operation for removing an item from stack
- **Push**: operation of putting new item on top of stack
- **Pop**: operation of removing item from stack



Stack Implementation with the **deque** Class (cont'd.)

- In C++, a stack can be easily created using STL's `deque` class
- **deque** class creates double-ended list
  - Objects can be pushed and popped from either end of list
- To create stack, only front end of deque is used
- A summary of `deque` class's methods and operations is listed in Table 13.4

Queues

- **Queue**: list in which items are added to one end of list, called top, and removed from other end of list, called bottom
  - Items are removed from list in exact order in which they were entered
  - Queue is **first-in, first-out (FIFO)** list

- Example of a queue: list of people waiting to purchase season tickets to professional football team
  - First person on list should be called when first set of tickets becomes available, second person should be called for second available set, and so on

Queue Implementation with the `deque` Class

- A queue is easily derived using STL `deque` container
- Enque (push): place new item on queue
- Serve (pop): remove item from queue

- **Enqueuing** is similar to pushing on one end of stack, and serving from queue is similar to popping from other end of stack
- How each of these operations is implemented depends on list used to represent queue

deque Sequence Container

- Class **deque** provides many of the benefits of a vector and a list in one container.
- The term deque is short for “double-ended queue.” Class **deque** is implemented to provide efficient indexed access (using subscripting) for reading and modifying its elements, much like a **`vector`** .
- Class **deque** is also implemented for efficient **insertion** and **deletion** operations at its front and back, much like a list (although a **`list`** is also capable of efficient **insertions** and **deletions** in the middle of the list).
- Class **deque** provides support for **random-access iterators**, so **deque**s can be used with all Standard Library algorithms.
- One of the most common uses of a deque is to maintain a first-in, first-out queue of elements.
- In fact, a **deque** is the default underlying implementation for the **queue** adaptor.

deque Sequence Container (continued)

```
2  // Standard Library deque class template.
3  #include <iostream>
4  #include <deque> // deque class-template definition
5  #include <algorithm> // copy algorithm
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  int main() {
10    deque<double> values; // create deque of doubles
11    ostream_iterator<double> output{cout, " "};
12
13    // insert elements in values
14    values.push_front(2.2);
15    values.push_front(3.5);
16    values.push_back(1.1);
17
18    cout << "values contains: ";
19
20    // use subscript operator to obtain elements of values
21    for (size_t i{0}; i < values.size(); ++i) {
22       cout << values[i] << ' ';
23    }
24
25    values.pop_front(); // remove first element
26    cout << "\nAfter pop_front, values contains: ";
27    copy(values.cbegin(), values.cend(), output);
28
29    // use subscript operator to modify element at location 1
30    values[1] = 5.4;
31    cout << "\nAfter values[1] = 5.4, values contains: ";
32    copy(values.cbegin(), values.cend(), output);
33    cout << endl;
34 }
```

DURHAM COLLEGE
SUCCESS MATTERS

maps

- The **`map`** is an associative container that holds **key-value** pairs. **Keys** are **sorted** and **unique**.
- A map is also implemented as a balanced **binary tree/graph**. So now, instead of one value per element, we have two.
- To use a map, we need to include the **map** header.
- To define a map, we use the **`std::map<type1, type2> map_name`** syntax.
- Here the **`type1`** represents the type of the **key**, and **`type2`** represents the type of a **value**.

maps (continued)

- To initialize a map of **`int char`** pairs, for example, we can write:

```
#include <map>
int main()
{
   std::map<int, char> mymap = { {1, 'a'}, {2, 'b'}, {3,'z'}};
}
```

- In this example, integers are **keys**, and the characters are the **values**.

- Every map element is a **pair**. The pair's first element (the **key**) is accessed through a **`first()`** member variable, the second element (the **value**) is accessed through the **`second()`** member function variable.

maps (continued)

- To print out our map, we can use:

```
#include <iostream>
#include <map>
int main()
{
   std::map<int, char> mymap = { {1, 'a'}, {2, 'b'}, {3,'z'} };
   for (auto el : mymap)
   {
      std::cout << el.first << ' ' << el.second << '\n’;
   }
}
```

maps (continued)

- We can also construct a map through its default constructor and some help from its key subscript operator **[ ]**. If the key accessed through a subscript operator does not exist, the entire key-value pair gets inserted into a map. Example:

```
#include <iostream>
#include <map>
int main()
{
   std::map<int, char> mymap;
   mymap[1] = 'a';
   mymap[2] = 'b';
   mymap[3] = 'z';

   for (auto el : mymap)
   {
      std::cout << el.first << ' ' << el.second << '\n’;
   }
}
```

maps (continued)

- To insert into a map, we can use the **`insert()`** member function:

```
#include <iostream>
#include <map>
int main()
{
   std::map<int, char> mymap = { {1, 'a'}, {2, 'b'}, {3,'z'} };

   mymap.insert({ 20, 'c' });

   for (auto el : mymap)
   {
      std::cout << el.first << ' ' << el.second << '\n';
   }
}
```

maps (continued)

- To search for a specific key inside a map, we can use the map's **`find(key_value)`** member function, which **returns an iterator**.

- If the key was not found, this function returns an iterator with the value of **`end()`**. If the key was found, the function returns the iterator pointing at the pair containing the searched-for key:

```
#include <iostream>
#include <map>

int main()
{
   std::map<int, char> mymap = { {1, 'a'}, {2, 'b'}, {3,'z'} };
   auto it = mymap.find(2);

   if (it != mymap.end())
   {
      std::cout << "Found: " << it->first << " " << it->second << '\n';
   }
   else
   {
      std::cout << "Not found.";
   }
}
```

Common Programming Errors

- Two common programming errors related to using STL's list and deque classes are:
  - Inserting objects instantiated from different classes into same list
  - Attempting to use indices rather than iterators when using STL class methods and algorithms

Common Programming Errors (cont'd.)

- The five most common programming errors related to linked lists, stacks, and queues, which occur when programmers attempt to construct their own lists are:
  - Not checking pointer provided by new operator when constructing non-STL list
  - Not correctly updating all relevant pointer addresses when adding or removing records from dynamically created stacks and queues
  - Forgetting to free previously allocated memory space when space is no longer needed
  - Not preserving integrity of addresses contained in top-of-stack pointer when dealing with stack and queue-in and queue-out pointers when dealing with queue
  - Not correctly updating internal record pointers when inserting and removing records from stack or queue

Summary

- An object permits individual data items to be stored under common variable name
  - Objects can then be stored together in list
- Linked list is list of objects in which each object contains pointer variable that locates next object in list
- Linked lists can be automatically constructed using STL's **`list`** class
- Stack is list consisting of objects that can only be added and removed from top of list
  - LIFO (last-in, first-out) list
  - Can be implemented using STL's **deque** class
- Queue is list consisting of objects that are added to top of list and removed from bottom of list
  - FIFO (first-in, first-out) list
  - Can be implemented using STL's **deque** class

Addendum – File Operations - Writing

```
2  // Creating a sequential file.
3  #include <iostream>
4  #include <string>
5  #include <fstream> // contains file stream processing types
6  #include <cstdlib> // exit function prototype
7  using namespace std;
8
9  int main() {
10    // ofstream constructor opens file
11    ofstream outClientFile{"clients.txt", ios::out};
12
13    // exit program if unable to create file
14    if (!outClientFile) { // overloaded ! operator
15       cerr << "File could not be opened" << endl;
16       exit(EXIT_FAILURE);
17    }
18
19    cout << "Enter the account, name, and balance.\n"
20       << "Enter end-of-file to end input.\n? ";
21
22    int account; // the account number
23    string name; // the account owner's name
24    double balance; // the account balance
25
26    // read account, name and balance from cin, then place in file
27    while (cin >> account >> name >> balance) {
28       outClientFile << account << ' ' << name << ' ' << balance << endl;
29       cout << "? ";
30    }
31 }
```

Addendum – File Operations - Reading

```
2  // Reading and printing a sequential file.
3  #include <iostream>
4  #include <fstream> // file stream
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 void outputLine(int, const string&, double); // prototype
11
12 int main() {
13    // ifstream constructor opens the file
14    ifstream inClientFile("clients.txt", ios::in);
15
16    // exit program if ifstream could not open file
17    if (!inClientFile) {
18       cerr << "File could not be opened" << endl;
19       exit(EXIT_FAILURE);
20    }
21
22    cout << left << setw(10) << "Account" << setw(13)
23       << "Name" << "Balance\n" << fixed << showpoint;
24
25    int account; // the account number
26    string name; // the account owner's name
27    double balance; // the account balance
28
29    // display each record in file
30    while (inClientFile >> account >> name >> balance) {
31       outputLine(account, name, balance);
32    }
33 }
34
35 // display single record from file
36 void outputLine(int account, const string& name, double balance) {
37    cout << left << setw(10) << account << setw(13) << name
38       << setw(7) << setprecision(2) << right << balance << endl;
39 }
```