Short Course on Programming in C/C++

Organized by Onur Pekcan

Contributor Selim Temizer Instructor Hasan Yılmaz

Week 3 – Lecture2

Today

We will cover;

Standard Template Library (STL)

- **Sequences**
  - vector
  - deque
  - list
- **Container Classes**
  - stack
  - queue
  - priority queue

Sequences

- vector
- deque
- list

Using the `vector`

- Vector: Dynamically growing, shrinking array of elements
- To use it include library header file:

  #include <vector>
- Vectors are declared as

```
vector<int> a(4); //a vector called a,
                  //containing four integers
vector<int> b(4, 3); //a vector of four
          // elements, each initialized to 3.
vector<int> c; // 0 int objects
```

- The elements of an integer vector behave just like ordinary integer variables

```
a[2] = 45;
```

Manipulating vectors

- **The size() member function** returns the number of elements in the vector.

  `a.size()` returns a value of 4.
- **The = operator** can be used to assign one vector to another.
- e.g. v1 = v2, so long as they are vectors of the same type.
- **The push_back() member function** allows you to add elements to the end of a vector.

push_back() and pop_back()

```
vector<int> v;
v.push_back(3);
v.push_back(2);
// v[0] is 3, v[1] is 2, v.size() is
  2
v.pop_back();
int t = v[v.size()-1];
v.pop_back();
```

Double ended queue(deque)

- **deque** (usually pronounced like *"deck"*) is an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are a kind of sequence container. As such, their elements are ordered following a strict linear sequence.

- Deque sequences have the following properties:
  - Individual elements can be accessed by their position index.
  - Iteration over the elements can be performed in any order.
  - Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

list

- Lists are a kind of sequence container. As such, their elements are ordered following a linear sequence
- List containers are implemented as doubly-linked lists
- Doubly linked lists can store each of the elements they contain in different and unrelated storage locations
- The ordering is kept by the association to each element of a link to the element preceding it and a link to the element following it

list

This provides the following advantages to list containers:

- Efficient insertion and removal of elements before any other specific element in the container (constant time).
- Efficient moving elements and block of elements within the container or even between different containers (constant time).
- Iterating over the elements in forward or reverse order (linear time).

list

- Compared to other base standard sequence containers (vectors and deques), lists perform generally better in inserting, extracting and moving elements in any position within the container for which we already have an iterator, and therefore also in algorithms that make intensive use of these, like sorting algorithms

Example 1

```
// constructing lists
#include <iostream>
#include <list>
using namespace std;

int main ()
{
  // constructors used in the same order as described above:
  list<int> first;                          // empty list of ints
  list<int> second (4,100);                 // four ints with value 100
  list<int> third (second.begin(),second.end());  // iterating through second
  list<int> fourth (third);                 // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  cout << "The contents of fifth are: ";
  for (list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
    cout << *it << " ";

  cout << endl;

  return 0;
}
```

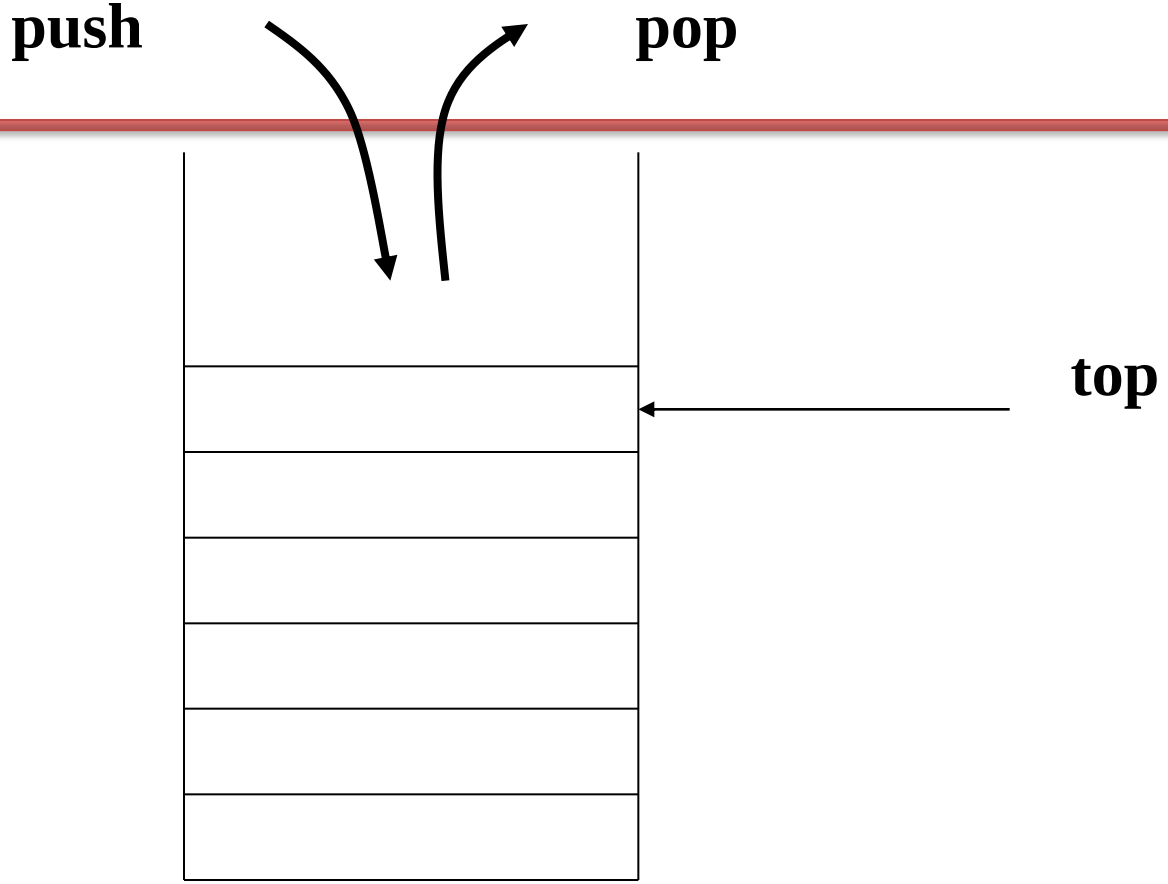
The contents of fifth are: 16 2 77 29

Container Classes

- stack
- queue
- priority queue

The Stack ADT

- The Stack ADT stores arbitrary objects.
- Insertions and deletions follow the *last-in first-out* (LIFO) scheme.
- It is like a stack of trays:
  - Trays can be added to the top of the stack.
  - Trays can be removed from the top of the stack.
- Main stack operations:
  - **push**(object o): inserts element o
  - **pop**(): removes and returns the last inserted element



Stack

The Stack ADT

- Auxiliary stack operations:
  - **top**(): returns a reference to the last inserted element without removing it
  - **size**(): returns the number of elements stored
  - **isEmpty**(): returns a Boolean value indicating whether no elements are stored

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException.

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Stacks and Computer Languages

- A stack can be used to check for unbalanced symbols (e.g. matching parentheses)
- Algorithm

```
1. Make an empty stack.
2. Read symbols until the end of file.
   a. If the token is an opening symbol, push it
      onto the stack.
   b. If it is a closing symbol and the stack is
      empty, report an error.
   c. Otherwise, pop the stack. If the symbol
      popped is not the corresponding opening
      symbol, report an error.
3. At the end of the file, if the stack is not
   empty, report an error.
```

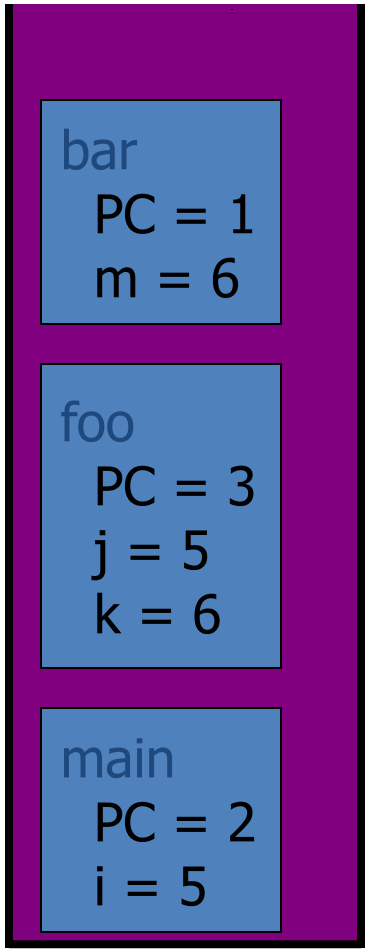
C++ Run-time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```

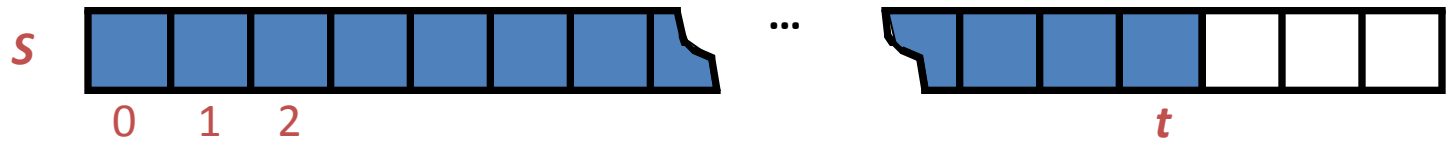


Array-based Stack

- A simple way of implementing the Stack ADT uses an array.
- We add elements from left to right.
- A variable keeps track of the index of the top element

```
Algorithm size()
  return t + 1

Algorithm pop()
  if isEmpty() then
    throw EmptyStackException
  else
    t ← t – 1
    return S[t + 1]
```



Array-based Stack (cont.)

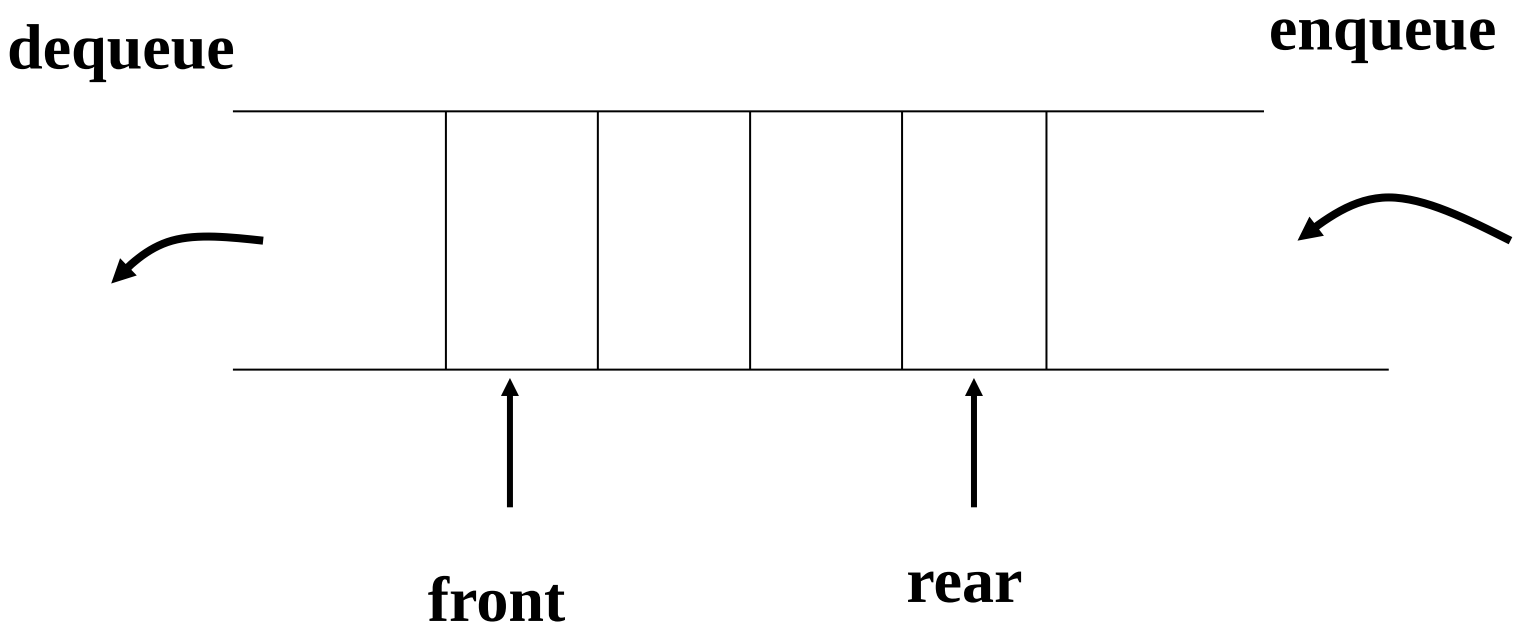
- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    throw *FullStackException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

The Abstract Data Type Queue

- A ***queue*** is a list from which items are deleted from one end (**front**) and into which items are inserted at the other end (**rear,** or **back**)
  - It is like line of people waiting to purchase tickets:
- ***Queue*** is referred to as a **first-in-first-out (FIFO)** data structure.
  - The first item inserted into a queue is the first item to leave
- Queues have many applications in computer systems:
  - Any application where a group of items is waiting to use a shared resource will use a queue. e.g.
    - jobs in a single processor computer
    - print spooling
    - information packets in computer networks.
  - A *simulation*: a study to see how to reduce the waiting time involved in an application

A Queue



ADT Queue Operations

- ***createQueue()***
  - Create an empty queue
- ***destroyQueue()***
  - Destroy a queue
- ***isEmpty():boolean***
  - Determine whether a queue is empty
- ***enqueue(in newItem:QueueItemType) throw QueueException***
  - Inserts a new item at the end of the queue (at the **rear** of the queue)
- ***dequeue() throw QueueException***

  dequeue(out queueFront:QueueItemType) throw QueueException
  - Removes (and returns) the element at the **front** of the queue
  - Remove the item that was added earliest
- ***getFront(out queueFront:QueueItemType) throw QueueException***
  - Retrieve the item that was added earliest (without removing)

Some Queue Operations

Operation	***Queue after operation***
x.createQueue()	an empty queue *front* ↓
x.enqueue(5)	5
x.enqueue(3)	5 3
x.enqueue(2)	5 3 2
x.dequeue()	3 2
x.enqueue(7)	3 2 7
x.dequeue(a)	2 7 (a is 3)
x.getFront(b)	2 7 (b is 2)

priority queue

- Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering condition.

- Priority queues are implemented as *container adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *popped* from the *"back"* of the specific container, which is known as the *top* of the priority queue.

priority queue

- The underlying container may be any of the standard container class templates or some other specifically designed container class. The only requirement is that it must be accessible through random access iterators and it must support the following operations:

front()

push_back()

pop_back()

Example 2

```
#include<iostream>
#include<queue>

using namespace std;

int main()
{
   priority_queue<int> q1;
   int number;

   while(cin >> number)                //get numbers until you see a character which is not an integer
   {
      q1.push(number);
   }

   while(!q1.empty())                  //while q1 is not empty
   {
      cout << q1.top() << endl;        //get top of priority queue
      q1.pop();//pop
   }
return 0;
}
```