CSE 101
Introduction to Data Structures and Algorithms
ADTs in C++

A *class* in C++ is the basic program construct for implementing an ADT. It allows the programmer to group member variables and methods into one entity. A *struct* in C++ is practically the same thing as a class. It can have both methods and fields as members, it can have explicit constructors and destructors, and can use inheritance. The main difference is that struct members are public by default, while class members are private by default.

By contrast, a *struct* in C is more like an array with named elements. There are no methods within the struct itself. For example:

Example: a struct in C

```
struct Blah{    // no typedef
   int foo;
   int bar;
};
.
.
struct Blah x;  // so "struct" is necessary here
x.foo = 6;      // initializing fields
x.bar = 7;
```

In practice, C++ programmers use structs only for very simple classes, with no methods beyond a constructor or destructor.

Example: a struct in C++

```
struct Blah{
   int foo;  // members are public by default
   int bar;

   Blah(int a, int b){  // constructor
      foo = a;
      bar = b;
   }
};
.
.
Blah x(6, 7);  // calling the constructor
```

Example: a class in C++

```
class Blah{
   int foo;  // members are private by default
   int bar;

   Blah(int a, int b){  // constructor
      foo = a;
```

```
      bar = b;
   }

   void set_foo(int a){ // another function
      foo = a;
   }
};
.
.
Blah x(6, 7);  // calling the constructor
x.set_foo(8);  // calling a member function
```

Rather than to rely on default access rules, a good practice is to explicitely separate public and private sections of a class, by using the access modifiers `public` and `private`.

Example: a basic skeleton for a class in C++

```
class Blah{

private:
   // inner classes/structs
   // member variables
   // private functions

public:
   // constructor and copy constructor
   // destructor
   // member methods
   // overloaded operators

};
```

As in C, when we build an ADT in C++, we split the implementation into two files. File `Blah.h` contains the class definition and function prototypes, and file `Blah.cpp` contains the function definitions.

File: `Blah.h`

```
class Blah{

private:
   // inner structs
   // member variables and helper functions

public:
   // prototypes for constructors, destructors, member methods and
   // overloaded operators

};
```

A class defines a *namespace* in C++. That namespace can be accessed from outside the class by using the *namespace resolution operator* `::`

File: `Blah.cpp`

```
// a member method implementing an ADT operation
type Blah::fcn1(..parameter list..){
   // implementation code
}

// another member method
type Blah::fcn2(..parameter list..){
   // implementation code
}

// a private member helper function
type Blah::helper_fcn1(..parameter list..){
   // code
}

// a non-member helper function
type helper_fcn2(..parameter list..){
   // code
}
.
.
```

There are four examples of ADT implementations posted on the webpage in Examples/C++. The Queue and the Stack are both implemented as a linked list, and as an array. Study all of these examples carefully before you begin pa5.

These examples illustrate new conventions for ADT implementations C++. We note these conventions below, as they apply to the Queue implemented as a linked list.

- Queue is not a pointer type, but instead names a new object type.
- We define a QueueElement type using typedef to make it easy to change the type of the container.
- We define at least two constructors:
    - a standard constructor, which may take parameters.
    - a copy constructor taking a `const Queue` reference as input.

  Be aware that constructors and destructors are almost always called implicitely in C++.
- We use the `const` keyword so that the compiler will flag any attempt to change an object.

  A member function declared as

  ```
  type fcn(.....) const;
  ```

cannot change the object pointed to by `this` (the *this pointer*, which refers to the object on the left hand side of the dot (`.`), as in `A.fcn(.....)`.

A member function with `const` in its parameter list cannot change that argument.

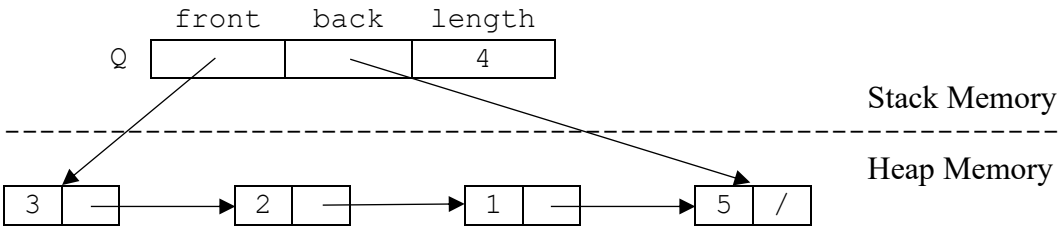
```
type fcn(..., const type& x,...);
```

- The new object created by the constructor is not typically created on the heap, but on the stack. The object may manage some heap memory though. Illustrating again with the Queue example:

  Client View:

```
Q:   3   2   1   5
     |           |
   front       back
```

  Inside View:



- We use *new* and *delete* to allocate and deallocate heap memory. Do not use the C functions malloc, calloc, realloc and free in C++, and **never** mix them with new and delete.

- The idomatically correct way to deal with precondition violations within an ADT operation is to throw an appropriate exception. This exception carries an error message of the form

```
ADT: operation(): error description
```

  The exception's member function `what()` can deliver this message to the client by means of the `try-catch` construct. See the library header `<stdexcept>` for a list of built in exceptions.

- It is very common to overload operators in C++. We almost always overload the three below.

  - `op==()`: compare for equality, as friend
  - `op<<()`: stream insertion, as friend
  - `op=()`: assignment, as member

  The Queue and Stack examples on the webpage show how to overload the above operators. See here for more examples of operator overloading.