Algorithms in Systems Engineering
IE170

Lecture 1

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - CLRS Chapter 10
- References
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Data Types in C++

- A *data type* is a set of data values and a set of operations that can be performed on those values.
- Data types are a mechanism by which C++ and other object-oriented languages allow programmers to define and implement *data structures*.
- Most of the data types you have probably encountered so far are the C++ built-in data types.

- What are the operations we perform on these data types?

C++ Classes

- In C++, *classes* are used to build new data types.
- A class is composed of
- The *data members* are the values.
- The *member functions* are the operations to be performed on these values.
- There are also *constructors* and *destructors*, by which objects of the specified types are created and destroyed.

C++ Classes

- Ideally, we would like to separate the *definition* of the type from the *implementation*.
- Defining a type consists of specifying the data that needs to be stored and the operations that need to be performed.
- In C++, the definition is contained in a *header file* that must be included in any source file that uses the data type.
- The implementation specifies the method by which these operations should actually be performed.
- What is the main advantage of separating the definition from the implementation?

The Interface

- The *interface* defines the way in which *clients* can actually use the data type.
- In C++, the interface consists of the public members of the class.
- The private members of the class, along with function implementations constitute the *implementation*.
- It is considered good programming style to keep all data members private.
  - The data members specify *how* the data is to be stored and are therefore part of the implementation.
  - Client access to data values should be through provided *access methods*.
  - This means the client does not have to know anything about how the data are actually stored or how the operations are implemented.
  - It also allows changing the implementation without changing the client program.
  - Finally, it prevents the client from manipulating the data directly.

Example: List Data Type

- Suppose we want a new data type for storing a list of integers.
- What are the values to be stored?
- What operations might we want to perform?

- We could implement this data type in two different ways:

A List Class

```
class list {
 private:
   // Here is the implementation-dependent code
   // that defines exactly how the list is stored.
 public:
   // Here is the list of operations to be implemented.
   // Create and destroy a list
   list();
   ~list();
   // Get the number of items in the list
   int getNumItems() const;
   // Get the value of item j
   bool getValue(const int j, int& value) const;
   // Get the value of item j
   bool setValue(const int j, const int value);
   // Add an item before item j
   bool addItem(const int j, const int value);
   // Delete item j
   bool delItem(const int j);
};
```

Implementing with Arrays

This source would be put in a file called `list.h`.

```
class list {
 private:
   // Here is the implementation-dependent code.
   // We'll store the data in this array.
   int* array_;
   // Here is the size of the array.
   int size_;
   // Here is the number of items in the list.
   int numItems_;
 public:
   list();
   ~list();
   int getNumItems() const;
   bool getValue(const int j, int& value) const;
   bool setValue(const j, const int value);
   bool addItem(const int j, const int value);
   bool delItem(const int j);
}
```

Constructing and Destructing

This source would be put in a file called `list.cpp`.

```
#include "list.h"

list::list() :
   array_(new int[MAXSIZE]);
   size_(MAXSIZE);
   numItems_(0);
{}

list::~list() {
   delete array_;
   size_ = 0;
}
```

Implementing List Query Operations

```
int list::getNumItems() const {
   return numItems_;
}

const bool list::getItem(const int j, int& value) {
   if (j > 0 && j < size_){
      value = array_[j];
      return true;
   }else{
      return false;
   }
}
```

Implementing List Modification Operations

```
bool list::addItem(const int j, const int value){
   if (numItems_ == size_ || j < 0 || j > size_){
      return false;
   }else{
      for (int i = size_; i > j; i--)
         array_[i] = array_[i-1];
      array_[j] = value;
      size_++;
   }
}

bool list::delItem(const int j){
   if (j < 0 || j > size_ - 1){
      return false;
   }else{
      for (int i = j; i < size_ - 1; i++)
         array_[i] = array_[i+1];
      size_--;
   }
}
```

Implementing with Linked Lists

- For a linked list implementation, we would replace the array with a linked list.
- To the client, the class would function exactly as before.
- Here is the definition of a node class for a linked list.

- Which implementation should be used?

Stacks and Queues

- A *stack* is a special kind of list in which items can only be removed in "last-in, first-out" (LIFO) order.
- A *queue* is a list in which items can only be removed in "first-in, first-out" (FIFO) order.
- The basic operations on a stack are

- The basic operations on a queue are

A Stack Class

In Lab 3, you will be asked to implement a stack with the following interface.

A Stack Class

In Lab 3, you will be asked to implement a stack with the following interface.

```
class stack {
 private:
   // Implementation-dependent stuff
 public:
   // Contructor and destructor
   stack();
   ~stack();
   // Checks whether the stack is empty
   int empty() const;
   // Puts an item on the stack
   void push(const int value);
   // Takes an item off the stack
   int pop();
}
```

STL

- The *Standard Template Library* or *STL* is a library of commonly used data types and algorithms.
- The classes in the STL are highly optimized.
- Most of the algorithms and data structures we will discuss are implemented in the STL.
- We may use parts of the STL during the semester, but we will also be developing our own implementations of STL classes.
- You can use the STL to test your code.

Some C++ Style and Implementation Requirements

- All objects should be initialized explicitly using a constructor.
  - Incorrect: `int i = 0;`
  - Correct: `int i(0);`
- Constructors themselves should initialize all data members of the class they are constructing.
- Be sure to delete any memory you allocate with `new`.
- Destructors should delete any memory allocated within a class.
- No public data members.
- No executable code in header files.
- No global variables.
- We will try not to have global functions either.