Data Structures and Objects CSIS 3700

Lab 3 — Exceptions

Goal

Create a basic set of exception objects to be used by our container classes to indicate various error conditions.

Preparation

- Create a `Programming` folder in your home directory.
- Inside the `Programming` folder, create the following folders:
  - `include`
  - `lib`
  - `src`

These will hold files we create for containers *et alia*

Background

How does a function indicate whether or not it has successfully completed its work? If it indicates this at all, it is typically via the return value. However, doing so prevents that from being used to return data to the calling function. There are three general solutions to this problem:

1. Use a global variable instead of the return value
2. Use reference parameters to pass back other information
3. Use exceptions instead of the return value

▹*Using global variables*

This is done by many system calls in Unix; the global variable **errno** is set to provide additional information about why a system call failed. The **perror()** function reads this value and displays a (somewhat) informative message.

▹*Using reference parameters*

This is done by many functions; the return value indicates success or failure, allowing the function to be used as a control for an **if-else** statement or a loop. Then, any other information the function provides is necessarily sent via reference parameters.

▹*Using exceptions*

Exceptions provide an out-of-band (*i.e.*, not using the normal parameter-return value communication channel) mechanism for signaling an error condition. Several languages (*e.g.*, Java and Python) use this mechanism. C++ does now, although exceptions were added several years after the language was created.

An exception is a datum sent from a called function / method to a calling function. The datum can be of any type; for our purposes, we will use objects.

Exceptions are sent using a throw-and-catch protocol. The called function throws an exception which the calling function catches. Note that exceptions must be caught; failure to catch an exception will cause a program crash (try it!)

▷Throwing an exception

To throw an exception, use the **throw** statement. Some examples:

```
// throws an int
throw 42;

// throws a string
throw "Danger, Will Robinson!";

// throws a KeyNotFoundException object
throw KeyNotFoundException(badKey);
```

Note the syntax for each of these; your mileage will vary slightly from language to language, but any language that supports exceptions has similar syntax.

▷Catching an exception

To catch an exception, enclose a function call that can throw an exception in a **try-catch** block pair. The first block is a **try** block. An example:

```
try {
  value = dictionary.search(key);
}
```

Note that this is just a code block with the keyword **try** in front of it.

Following the **try** block is one or more **catch** blocks. Each block specifies a type of exception data and a code block. The **catch** blocks are searched in order of appearance; the first **catch** whose data type matches the thrown data has its code block executed and all other **catch** blocks are bypassed. Some examples:

```
catch (int e) { // catches a thrown integer
  cout << "Integer exception " << e;
}
catch (KeyNotFoundException e) { // catches a KeyNotFoundException
  cout << e.toString();
}
catch (...) { // catches anything
  cout << "something was thrown";
}
```

Exceptions and Container Classes

We will use exceptions to indicate various issues that may arise in our use of the containers we will be studying in the course. Following the model used in many instances, we will create an exception *framework* — a collection of related exception classes that provide information about issues that arise.

▹*The root — AbstractException*

We will start by creating an abstract class — a class whose methods aren't all filled in.

1. In the `include` directory, create a file named `exceptions.h`. Open `exceptions.h` in a text editor.
2. Add a guard **#ifndef** and include the **string** header file.
3. Define the AbstractException class:

```
class AbstractException {
 public:
  AbstractException(string src) { this->src = src; }
  ~AbstractException(void) { }
  const string &source(void) { return src; }
  virtual const string toString(void) =0;
 private:
  string
    src;
};
```

This class has a basic constructor and destructor and two methods. Items of interest:

- The constructor has one parameter and it is required. All it does is store the parameter in the **src** field.
- The intent of the **src** field is to allow containers to have a printable name; this name can be passed to the exception to display the name of the offending object.
- The **src** field is private, meaning that nobody outside of **AbstractException** can access it. The **source()** method provides read-only access to the field.
- The **toString()** function is called *pure virtual*. This is an advanced concept; for our purposes, it means that we must define **toString()** for any classes based on **AbstractException**; this is all of our exceptions.
- I'm breaking my rule about code in the class definition. I often do that when the method has at most one instruction.
- **AbstractException** needs no implementation file. Three of the four methods have their bodies defined and the fourth is intentionally left blank.

▹*The generic Exception*

Next, we will create a generic **Exception** class. The **AbstractException** forces all of our exceptions to define the **toString()** method, but it can't be thrown or caught because it lacks a body for **toString()**. To create a throwable exception, we need to create a subclass — a new class, based on **AbstractException**.

1. In the `exceptions.h` file, add the following class:

```
class Exception : public AbstractException {
 public:
  Exception(string src="") : AbstractException(src) { }
  ~Exception(void) { }
  const string toString(void);
};
```

2. In the `src` directory, create an `exceptions.cc` file and open it in a text processor.
3. In `exceptions.cc`, include exceptions.h. Use <···>.
4. Add the following to exceptions.cc:

```
const string Exception::toString(void) {
  string
    msg;

  if (source().length() > 0)
    msg = source() + ":␣";
  else
    msg = "";
  msg += "generic␣exception";

  return msg;
}
```

Items of interest:

- `class Exception :  public AbstractException` says that **Exception** is a class based on **AbstractException**. Everything that **AbstractException** can do, **Exception** can also do — except access the **src** field, since it's private to **AbstractException**. This means that **Exception** can call **source()**, even though it's not listed in the class.
- `Exception(string src="") :  AbstractException(src) { }` says that the **src** parameter to the **Exception** constructor is passed off to **AbstractException**'s constructor. In Java, this is the **super()** call that you often see. The constructor has no other work to do, so its body is empty.
- We must implement **toString()**. In fact, that's the only method we need to implement for each of our classes.

▹*Next step — An exception for an empty container*

Sometimes our containers will be empty. In this case, we don't want to do things like look at elements in the container (*e.g.*, the **peek()** method for stacks) or remove something (*e.g.*, the **dequeue()** method for queues). We will create an exception specifically for these situations.

1. Add the following to the `exceptions.h` file:

```
class ContainerEmptyException : public Exception {
 public:
  ContainerEmptyException(string src="") : Exception(src) { }
  ~ContainerEmptyException(void) { }
  const string toString(void);
};
```

2. Add the following to the `exceptions.cc` file:

```
const string ContainerEmptyException::toString(void) {
  string
    msg;

  if (source().length() > 0)
    msg = source() + ":␣";
  else
    msg = "";
  msg += "container␣empty";

  return msg;
}
```

Items of interest:

- This class is based on **Exception**, which is then based on **AbstractException**.
- Because of the parental relationship in the hierarchy of exception classes, you can throw a **ContainerEmptyException** object and catch it either as a **ContainerEmptyException** or an **Exception**. This provides you the flexibility of looking for a specific exception or a generic one.

▹*Finishing the lab*

There are three other types of exceptions that we'll encounter in our containers:

- **ContainerFullException**
  Additional items cannot be added to our container; *e.g.*, the backing array is full.
- **KeyNotFoundException**
  A search did not find the desired key. Note that if you search an empty container, the preferred exception to throw is **KeyNotFoundException** rather than **ContainerEmptyException**.
- **InvalidIndexException**
  Accessing a list or tree position used an index that is out of bounds. Like **KeyNotFoundException**, this is the preferred exception to throw when you provide an index into an empty container.

Add class definitions and implementations for these three exceptions; they will look very similar to **ContainerEmptyException**. Some necessary tweaks you'll need to make:

- The **KeyNotFoundException** constructor should take two parameters: **key** and **src**. Both are strings, both have a default value of an empty string. Send **src** to the **Exception** constructor just like you do with **ContainerEmptyException**. Store **key** in a private field. Incorporate **key** in the string returned by the **toString()** method.
- The **InvalidIndexException** also has two parameters: **index** and **src**; **index** is an integer. Aside from the difference in data type, treat it just like you do **key** in **KeyNotFoundException**.

What to turn in

Turn in the `exceptions.h` and `exceptions.cc` files.