CS115 Lab: Abstract Data Types (ADT's) Implemented as *CLASSES* in C++

Highlights of this lab:

In this lab, you will:

- Absorb the concepts involved in Data Abstraction.
- Clarify the definitions and format of an ADT.
- Master the implementation details required to code ADT's in a modular fashion.
  - Implementing Member Functions
  - Constructors - a special case

Reference: See the programming example on "Classes" in the C++ Syntax pages.

Lab Exercise: 

- Add operations to an ADT

Click the little computer above for a detailed description.
NOTE: Your lab instructor will tell you what will be marked for this lab.

Data Abstraction.

Your class textbook gives a comprehensive explanation of the purpose of ***data abstraction***. To put it simply, data abstraction is just a way of thinking about how you would solve a problem without worrying about the computer language tools you need to solve this problem. You focus on the problem and what, in ***general terms*** what you would need to implement your solution. Don't be concerned about what data types do (or don't) exist in the computer language. **When you are ready to implement your ADT, consider the following:**

- the data types you need to use, and
- what operations you need to have performed.

These two things constitute an **Abstract Data Type**, or an **ADT**.

The operations for an ADT fall into 4 categories:

Constructor	Creates data for the ADT.
Transformer / Mutator / Setter	Modifies data in the ADT.
Observer / Accessor / Getter	Allows you to "look but not touch" the data in the ADT.
Iterator	Provides the ability to move through components in an ADT, one at a time.

The following is an example of a set of specifications for an ADT. This includes a description of the data and the operations on these values.

```
Type
        Grades
Data or Fields
        name    -    a character array of 20 elements (represents student name)
        id      -    an integer  (represents student #)
        marks   -    an array of integers (for storing marks)
Operations
        setName -    a transformer type operation
        setID   -    a transformer type operation
        setMarks-    a transformer type operation
        reporter-    an observer type operation
```

Theory and abstraction are fine, but sometimes it's helpful to relate this to some practical knowledge in order to put the concepts into place. With that in mind, we'll introduce the term **`classes`** here, which is the most common way in which ADT's are implemented in C++. Here is a comparison between **`classes`**, which you may not have read much about yet, and **`structures`**, which you explored in a previous lab.

- By default, class members are private, whereas structure members are public
- In traditional C, structures did not contain operations.
- Nowadays, structures can have operations, but this approach is not recommended; making all data and operations public does not follow good object oriented design.
- Code that follows good object oriented design typically has data that is private and uses "setters" and "getters" to respectively modify and return the values of the data.

Declaring and using ***classes*** is similar to declaring and using ***structures*** in C++. First you declare the class (or structure), and then you declare the instances of the class (or structure).

For example, here are comparitive examples of how you declare and instantiate a class and a structure.

class Computer	**struct** Student
{	{

// By default: private
// data
// operations
	// By default: public
	// data
	// operations (not recommended)

};	};
// Instantiate a class	*// Instantiate a structure*
Computer pc;	Student stu;

There, now you've got the formal definition of an ADT and had a look at how they are implemented in C++ as **`classes`**. In the next section, you'll examine **`class members`** in more detail.

Definitions for Implementing an ADT with a Class.

Before we proceed, let's just get our terminology in order:

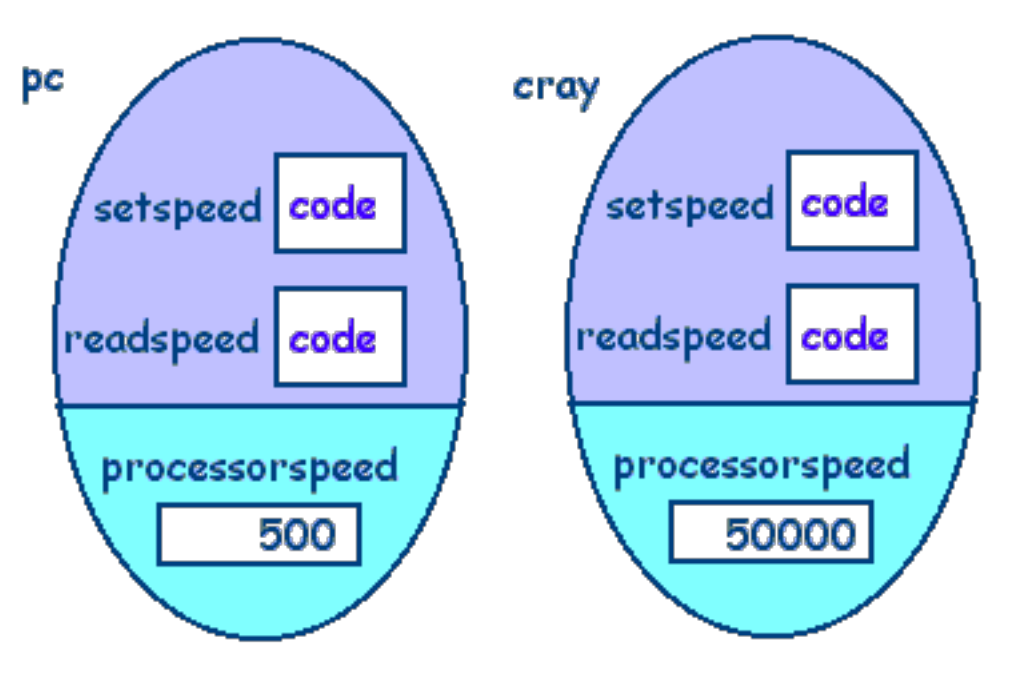
Term	Definition
`class`	An implementation of an ADT.
`class member`	A **`datum`** associated with the class, or a **`function`** used in the class to implement one of the operations associated with the class. These functions are also referred to as **`methods`** of the class.
`class object`	An instance of the class.

Class members are either **`public`**, meaning that they are accessible from outside the class, or **`private`** (the default), meaning that only the class can access them. Generally data is designated as being private (look up "information hiding" in any text) and the methods are designated as being public, so that they can be called from elsewhere.

Here's a simple example:

```
class Computer
{
  private:
        // It's not necessary to include this term since that is the default.
        // However, it's good form to do so and clarifies your meaning.
    int processorSpeed;
  public:
    void setSpeed(int);
    int getSpeed() const;
};
```

Note: It doesn't matter if `private` comes before `public`, or the other way around.



Let's try to visualize what two instances of this `Computer` class would look like. Say you've instantiated a Computer called `pc` and another called `cray`. Each of them would exist as an object and have its own methods and data.

You would use `dot notation` to refer to the members of each instance. e.g. `pc.setSpeed(x);` or `cray.setSpeed(x);` Be careful though - if you tried to access the data member, `pc.processorSpeed` from outside the class you would get an error, because that was declared as being `private`, unlike the methods, which are `public`.

Implementation Details.

Member Functions

A header file - ***`Filename.h`*** - is often used to contain the function prototypes and data members for the class. Another file, generally the same *filename* but with an extension of **`.cpp`**, contains the actual function definitions. You must remember to **`#include "Filename.h"`** in the `Filename.cpp` and in the calling program.

When you enter the code for the function in the ***`Filename.cpp`*** file, you must precede each member function name with the name of the class.

```
e.g.
        void Computer::setSpeed(int p)
```

The double colon **`::`** is known as the **`scope resolution operator`**. It ties the function name to the name of the related class. This is necessary because the function might also exist in another class.

Going back to the example we were just looking at, you could have three files involved in the total program.

`Computer.h`	The header file for the `Computer` class. This contains the private data members and the function prototypes for the operations members.
`Computer.cpp`	The function definition file for the `Computer` class. Must contain: **`#include "Computer.h"`**
`CompServ.cpp`	The main program that calls the class. Must contain: **`#include "Computer.h"`**

You may recall from lab 1 that each **`.cpp`** file must be compiled to an object, **`.o`**, file individually and then linked together to produce the executable file. Click here to review that material. You can use the Makefile introduced last lab to simplify the process. Click here to get instructions.

Constructors - a special case

When we talked about ADT's one of the classes of operation we identified was ***Constructor***. Constructors initialize data that wasn't there before.

An ADT can have two different general types of constructor. The first constructs the ADT itself. The second constructs pieces of the ADT if the ADT contains other ADTs, as in a **list** of **addresses**.

The first type of constructor generally has special language support. In C++ there are a few variations that allow for all, some or no data to be specified at object creation time. Any further changes would be made using ***transformers*** or the other type of constructor. The details of how to implement the first type of constructor follow in the next subsection.

The second type of constructor works just like any other member function, but at some point it may call a constructor for the contained data type.

C++ Constructors

- Constructors are used to initialize data members for a class object.
- A constructor is automatically invoked when a class object is created.
- If there is **no** constructor, then the data members for the new class object are not initialized and can contain garbage values. It is best to use a constructor to initialize the values.
- The ***name*** of the constructor function is the ***same as the name of the class***. For example, if your class name was `Grades` then that would also be the name of your constructor function.
- A class can have more than one constructor, but each constructor would take a different number of parameters.
- A constructor without ***any*** parameters is called a ***default constructor***.
- You do ***not*** indicate the data type of a constructor function because the function cannot return any value.
  WRONG: **int** *class_name::constructor_name(parameter-list)*
  CORRECT: *class_name::constructor_name(parameter-list)*
  For example this is what a default constructor for that `Grades` class would look like:

```
        Grades::Grades()
        {
            //  include C++ statements here to initialize the data members
        }
```

- You can have several constructors with the same name, as long as they have a different number of parameters or different types of parameters. Thus, you could have a constructor with no parameters, with one parameter, with two parameters, etc. For example, two constructors that take one integer parameter and three integer parameters would look like:

```
        Grades::Grades(int a)
        {
            //  include C++ statements here to initialize the data members
        }
        Grades::Grades(int a, int b, int c)
        {
            //  include C++ statements here to initialize the data members
        }
```

- To invoke a constructor, just create an object, giving the parameters for the constructor after the new object name. The constructor with the appropriate number of parameters will be executed. For example if there was a class called `Grades`, here are some statements to create new Grades objects, using constructors which each take different parameter lists:

```
        Grades student1;        // create an object using the default constructor
        Grades student2(123);   // create an object passing a single parameter
        Grades student3(85, 75, 99);    // create an object passing three parameters
```

  Note that you would have had to create each of the constructor functions in the class file.

Copy Constructors

A copy constructor creates a clone of an existing object by initializing a new object, with an existing object of the same class. Suppose you had a class called `Tree` and had defined a Tree object called `pine1` To create a clone, you could enter: `Tree pine2(pine1);` or: `Tree pine2 = pine1;` If you had no copy constructor defined, then the compiler would supply a default "copy constructor" to create the clone.
Note: The first notation "`Tree pine2(pine1);`", is the current preferred syntax. The second notation "`Tree pine2 = pine1;`", is now considered obsolete. However both forms are given here because a programmer is likely to encounter both.

Copying an object using the default copy constructor may work for simple objects. However, if there were pointers in the original object, only the pointers would be duplicated, not the data that was being pointed to. Following the previous example, suppose Tree objects had an integer data member, and a pointer data member to a character string. pine1's pointer would have the address of the ***same data*** as pine2's pointer! There would then be *two ways* of accessing (and modifying!) the same data. This type of a copy operation is called a **`shallow copy`** because the "pointed-to" data is **not** copied when a 'clone' is made. This is not a true clone.

If pointers are part of an object, what you may need is a **`deep copy`** which you would have to define yourself. Here is the general syntax for a deep copy constructor.

```
type :: type (const type  & object_name)
```

In that example of the general syntax, *type* refers to the class type name. Notice how the parameter is passed to the copy constructor. You do ***not*** want to do any harm to the existing object, so you declare the object parameter as type `const` and use the ampersand `&` to pass it as a ***reference*** rather than a ***value***. For example:

```
Tree::Tree(const Tree & otherTree)
    {
    age = otherTree.age;
    descrip = new char[strlen(otherTree.descrip) + 1];
    strcpy(descrip, otherTree.descrip);
    }
```

Actually, we haven't seen the `new` operator yet. It defines a dynamic variable in "free store" - a topic of a later lab. However the code is given here for the sake of giving a complete example of a `deep copy` constructor.

Destructors

The ADT concept is supported by many languages. Some of them can **garbage collect** things that you are done with. C++ does not. There are times when you will need special things done to an object when it is **destroyed**.

- A destructor is invoked automatically when a class object is destroyed.
  Objects are destroyed under many circumstances:
  - you **delete** the instance
  - you exit the scope for which the object was instanced
  - the program ends
- The ***name*** of the destructor function is the ***same as the name of the class*** but it is preceded with the ***tilde*** symbol **~**. For example, if your class name was `Grades` then the destructor name would be `~Grades`
- You cannot pass parameters to a destructor function.
- You do ***not*** indicate the data type of a destructor function because the function cannot return any value.
- Reasons for using destructors will be presented later in the course when dynamic allocation is discussed.
- you never call a destructor instead you use the **delete** keyword.

Lab Exercise -- ADTs Implemented with C++ Classes

Create a "Date" class that contains:

- three private data members:
  - `month`
  - `day`
  - `year`
  (I leave it to you to decide the type)
- "setters" and "getters" for each of the data (6 functions in total)
  - One advantage of a "setter" is that it can provide error checking. Add assert statements to the setter to enforce reasonable conditions. For example, day might be restricted to between 1 and 31 inclusive.
- one default constructor (no arguments)
- one constructor with three arguments: month, day, and year
  - Add assert statements to enforce reasonable conditions.
- a `printDate` function. This function will have no arguments and return void
- a `sameDay` function. This function will have one `Date` argument and a boolean return type

In main (in the following order):

1. instantiate one date object (`date1`) using the default constructor
2. use the `get`ters to display the `month`, `day`, and `year` of `date1` (should print the default values)
3. read keyboard input from the user for a month, day and year
4. use the `set`ters to set the values of `date1` to the values that came from the user
5. read keyboard input from the user for a second date
6. use the constructor with three arguments to instantiate `date2` to the second date input from the user
7. print both objects using `printDate`
8. print a message to say if the two days are the same (testing the `sameDay` function)

Your code should be in three files:

- `Date.h`
  - contains the class definition
- `Date.cpp`
  - includes "Date.h"
  - contains the functions for the class
- `main.cpp`
  - includes "Date.h"
  - tests the class

Sample Output (Two Runs)

```
>./main
Testing the default constructor and the getters
The initialized date is (M-D-Y):1-1-1

Please enter a date:(Month Day Year): 11 03 1976
Please enter a second date:(Month Day Year): 03 03 1999

Printing the two days:
The date is (M-D-Y): 11-3-1976
The date is (M-D-Y): 3-3-1999
The days are the same
>./main
Testing the default constructor and the getters
The initialized date is (M-D-Y): 1-1-1

Please enter a date:(Month Day Year): 12 15 2009
Please enter a second date:(Month Day Year): 12 25 2020

Printing the two days:
The date is (M-D-Y): 12-15-2009
The date is (M-D-Y): 12-25-2020
The days are different
```

If you are having trouble compiling your class, check for these common errors:

1. You have forgotten the semi-colon(`;`) after the closing curly bracket (`}`) for the class definition (in the `Date.h` file)
2. You have forgotten the scope resolution as in: `void Date::setYear(int y)` (in the `Date.cpp` file)
3. You forgot the `()` after the function name. For example, you should write: `cout << date1.getYear()` (in `main.cpp`)

For the StudentClass that we go over this week, see StudentClass.cpp