Lab 2: Streams and Templates
CMSC 162
Spring 2025
Due: Feb. 21, 2025 by 11:59:59pm

Warmup Exercise

Abe's Land and Water Craft Limited sells two types of vehicles: boats and cars. Abe has a wide selection of craft with many different features. The various features on each vehicle affect the eventual sale price of the vehicle. For instance, a car with a large gas tank usually sells at a higher price than a car with a small gas tank.

Abe would like to keep track of which vehicles have the best features and can bring in the most money. If he knows which ones are his big money makers, he can use his high-pressure sales pitch to market those products more heavily.

He keeps a list of cars and boats in a file named "inventory.dat". The file consists of a series of lines that look like this:

```
car             43      5150    10      32.5    120             true
```

Or like this:

```
boat            46      28178   5       true    30
```

Notice that the lines have different values. For instance, the fifth value for a car is a floating point value, while for a boat it's a boolean value. This makes it much more difficult to read the file, because **until** we know if a line represents a boat or car, we don't know how many other fields to read or what types to store them in.

We can get around this problem by reading the whole line into a single string and then breaking that string up into pieces using the "istringstream" and "ostringstream" classes that the C++ standard library provides. To use them, you need to include <sstream>. They are in the standard (std) namespace.

istringstream stands for "input string stream". Like other input streams, it allows us to read from a data source. The cin stream object that you're familiar with allows us to read from the console window. An input string stream, instead lets us read pieces of a string we pass to its constructor. Just like cin, we can use the >> operator and getline function to out pieces from a string stream.

For example:

```
string myName = "Robert Marmorstein";
istringstream myStream(myName);

string firstName;
string secondName;

myStream >> firstName >> secondName;
```

This reads "Robert" into firstName and "Marmorstein" into secondName.

ostringstream stands for "output string stream" and it creates an output stream that can print whatever we send it into an internal buffer. We can use the **<<** operator we use with “cout” on an output string stream. It provides a special function named “str()” that allows us to access every thing we’ve printed to the buffer as a string. For instance, I can do:

```
ostringstream outStream;

int x = 50;
outStream << "This is a number: " << x << endl;

cout << outStream.str();
```

This prints "This is a number: 50" followed by a new line.

We are going to help Abe out by writing a program that will produce a list of the estimated sales prices for each vehicle. We will calculate this price using a formula which takes into account all the features of each vehicle.

Step 1. Creating a "Car" class

Make a new directory named "Lab2". In that folder create a file named "car.h" In that file, **add #ifndef guards** and then create a class which has the following private variables:

id : a unique integer identification number for each vehicle
price : the base asking price (MSRP) of each vehicle
age : how old the vehicle is in years
tank_size: the number of gallons the fuel tank can hold
max_MPH: the top speed the car can reach
is_red: a true or false value indicating whether the car is red or not (red cars cost more)

Add a default constructor, standard accessor functions, and standard mutator functions for these variables. In addition, write a function named print that has no parameters, but returns a string describing all the features of the car. For instance, it might print:

Car 000453 has a MSRP of $4500.63, is 30 years old, has an 8 gallon fuel tank, can reach a top speed of 70 MPH, and is red.

Then create a "car.cpp" file which implements the functions from your car.h file.

Notice that I do not tell you what types to use. You should be able to figure that out from the descriptions I provided.

Step 2. Creating a "Boat" class

Now make a file named "boat.h" which contains a "Boat" class. The boat class has the following private variables:

id : a unique integer identification number for each vehicle
price : the base asking price (MSRP) of each vehicle
age : how old the vehicle is in years
is_yacht : a true or false value indicating whether the boat is a yacht
length: the number of feet from the prow to the stern

As with the "Car" class, write a constructor, mutators, and accessors for these variables and implement them in a file named "boat.cpp". Also provide a print function.

Step 3. Using STL vectors

In 160, you mostly stored collections of data items using arrays. An alternative to using arrays is to use the STL vector class. The STL is the "Standard Template Libray" -- basically it's lot of code that comes with most C++ compilers. It provides some very nice data structures (that you don't have to write – someone else already did!) which can be used generically. What does that mean? It means that they can be used to store ANY type of data, not just ints, floats, doubles, etc.

In particular, the STL provides a template class called a "vector" which is a special kind of dynamic array. To create a vector of integers, you would type something like this:

```
vector<int> myVector;
```

You can then do things like:

```
myVector.push_back(20);
```

which adds the number "20" to the vector.

Notice that we didn't have to specify an initial size. When we do push_back, C++ grows the vector to be large enough for our data. Just as we can create vectors of integers, we can create vectors of "Cars" and "Boats".

We could declare a vector of Car objects like this:

```
vector<Car> carList(10);
```

The number ten is passed to the constructor and specifies an initial size. Initially, the vector will have 10 car objects.

You could access the 6th car like this:

```
carList[5].setId(30);
```

Vectors avoid many of the disadvantages of an array, but they are often slower and use more memory.

To use STL vectors in your code, you have to do two things:

1. Add ***#include <vector>*** to the top of your file.

2. Add ***using namespace std;*** to your file (if it is not already there) or put ***std::*** in front of each vector declaration.

Create a file named "warmup.cpp". In that file, add a main function which:

A. Creates a vector of Car objects named "car_list" and a vector of Boat objects named "boat_list".

B. Opens a file named "inventory.dat" (using an ifstream object).

C. Enters a loop in which it:
    1. Reads one line of the file

    2. Creates either a Car or Boat object depending on the first word of the file

3. Uses istringstream to break the line into pieces and stores them in the Car or Boat object.

4. Adds the object to the appropriate vector.

You will need to use a priming read or an input loop for this to work appropriately.

D. Prints out all of the cars and THEN all of the boats to cout.

Main Project

In this part of the lab, I want you to design and implement a "Dynamic Array" template class. A dynamic array is like a usual C++ array, except that:

1. The dynamic array class contains an array allocated **on the heap**. The user never accesses the internal array directly. Instead, he (or she) can only access it using functions of the dynamic array class.

2. If the user tries to access a position beyond the end of the array, instead of crashing, the program replaces the internal array with a larger one after first copying all the values from the old array into the new one. This gives the illusion that the dynamic array has "grown".

3. The dynamic array provides a "shrink" function that takes a single parameter (an integer named size) and replaces the internal array with a smaller one of appropriate size. Any values still in range are copied into the new array, but any values out of range are lost.

4. The dynamic array provides a "size" function that returns the current size of the array.

The C++ vector class is actually one implementation of a dynamic array – but for this lab you are going to write your own.

Step 1. Defining the Interface

Create a new file named "dynarray.h". In that file, add the following code:

```
#ifndef DYNARRAY_H
#define DYNARRAY_H

template <typename T>
```

```
class Dynarray {

        private:
        T* m_list;
        int m_size;

        void resize(const int new_size)
        { }

        public:
        Dynarray()
        { }

        ~Dynarray()
        { }

        //Mutators

        T& operator[](const int pos)
        { }

        void shrink(int size)
        { }

        //Accessor
        int size() const
        { }
};

#endif
```

Notice that all of these functions are "empty" functions or "function stubs". This is an example of **top-down design**. First we specify what the functions are (so we can test that the outline works correctly). Then we will fill them in one at a time.

The #ifndef and #define and #endif directives are called "guards". They prevent us from accidentally including the dynarray template multiple times. The template class has two private variables. An array named "m_list" and an integer named "m_size". In addition to the constructor and destructor, it provides three two functions and one overloaded operator.

Step 2. Testing

To test your code, we need to create a main function that can create and manipulate a dynamic array.

Create a file named “main.cpp” and add the following:

```
#include <iostream>
#include "dynarray.h"
using namespace std;

int main()
{
        Dynarray<int> d;
        for (int i=0; i < 1000; ++i) {
                d[i] = i;
        }

        d.shrink(10);

        for (int i=0; i < d.size(); ++i) {
                cout << d[i] << endl;
        }

        d[30] = -6;

        for (int i=0; i < d.size(); ++i) {
                cout << d[i] << endl;
        }
}
```

If you've done everything right, this should compile and run, but not print anything yet. You should recompile and run this test file every time you make a change to the template.

Step 3. Constructor and Destructor

In dynarray.h:

1. Implement a constructor that allocates an array of ten “T” objects on the heap and stores the pointer in the “m_list” private variable. It should then set m_size to 10.

2. Implement a destructor that properly frees (using the delete[] operator) the memory used by the list. Save and test your code.

Step 4. Size accessor

In dynarray.h:

Implement the size accessor (which should simply return m_size). If this seems trivial – it

should be!

Step 5. Helper Function

In dynarray.h, add code for the resize function which:

1. Creates a new array on the heap of size “new_size” and stores the pointer to it in a “T pointer” named “new_list”.

2. Copies as many values as possible from m_list into new_list. If new_size is bigger than m_size, you should copy only m_size values. If new_size is smaller, you should copy only new_size values.

3. Copies 0's into any remaining elements of new_list. That is, if new_size is larger than m_size, copy values into positions m_size through (new_size - 1).

4. Deletes the m_list array (using delete[]).

5. Copies the new_list pointer into the m_list pointer.

6. Sets m_size to new_size.

Step 6. Implementing Shrink

In dynarray.h:

Implement a “shrink” function that calls the resize helper if (and only if) the new size is smaller than m_size.

Step 7. Overloading the array operator

To implement the array operator, you must:

1. Check that the item accessed (position “pos”) is in range. If not, call the resize helper to make the array large enough for the item. That means making the size one MORE than the position of the item.

2. Return the value at position “pos”. C++ will automatically make a reference to it.

Glitter

Extend the project in a clever and distinctive way. Glitter is NOT extra credit. It is a fundamental part of your grade. I grade glitter using two criteria:

1. How creative and unusual it is.
2. How well it demonstrates technical ability

One thing you could do for glitter would be to use your dynamic array to implement some sort of game or other application. Or you could extend the dynamic array by adding additional functions or operators. You should expect to spend almost as much time on glitter as on the rest of the lab.

Submitting

To submit, make a tarball by typing:

```
cd ..
```

(To get to the parent directory)

```
tar czvf Lab2.tar.gz Lab2/
```

If you've done this properly, it should list all the files you just created. Something like:

```
Lab2/
Lab2/boat.cpp
Lab2/boat.h
Lab2/car.cpp
Lab2/car.h
Lab2/warmup.cpp
Lab2/dynarray.h
Lab2/main.cpp
```

If not, make sure you're in the right folder.

Once you've created the tarball, you can submit by going to the course web site: http://marmorstein.org/~robert/submit/

Make sure you select “cs162” in the combo box to the right before typing in your “student” username and your password.

BE SURE TO PROPERLY CITE ANY SOURCES YOU USE!