Lab rapport in C++ OOP

Hergeir Winther Lognberg
Hewi1600

1 Preamble

Assignment was to create a list object with the ability to handle and manipulate 20 items of the type int and float. I am supposed to use the standard library functionality everywhere it's practical to do so.

2 The Code

2.1 placement

I've decided to keep all files associated with the lab in the root of the project folder.

2.2 Template memberfunction definitions

Also relevant is it that I have actually defined all template class functions in a cpp file and instantialized with double there. So that the compiler actually would know which template to compile. (hope i'm being clear enough)

```
1   //bottom of DataFileReader.cpp
2   template class DataFileReader<double>; //instantiating ↩
        template with type double
```

2.3 namespace

I chose to remove

```
1 using namespace std;
```

from all files it was previously used in. I find that this improves readability and clearly seperates std functions from self-made ones.

Still I have not had the need (nor want) for a shared pointer.

2.4 classes

In this lab i've constructed 3 classes of significanse to the lab. These are

- ListManipulator
  - class handles all interaction with list object
- TestApp
  - class handles all interaction between user and ListManipulator after datatype has been chosen
- Interface
  - handles running correct version of TestApp. User chooses datatype here.

2.4.1 ListManipulator

This is a standard template class.

I simply implemented all list aggregations, filling, clearing... etc, here. The results are returned up to Testapp implementations. Get's instantialized with same datatype as TestApp.

2.4.2 TestApp

This is also a template class.

I use this class for giving user results from ListManipulator functions and so on.

2.4.3 Interface

Responsible for instantialising (and running) correct version of testapp. This is done using the following function:

```
template <typename T>
void runMain()
{
   //run temporary instantialization of TestApp with user ↩
       defined type
   TestApp<T>().run();
}
```

and the menu switch looks like this:

```
void Interface::run()
{
   //runing interface wrapper.
   bool again=true;
   while(again)
   {
      switch(menu.getMenuChoice())
      {
         case 1:                  //user chooses int
            runMain<int>();
            break;
         case 2:                  //user chooses double
            runMain<double>();
            break;
         case 3:
            loadFromFile();
            break;
         case 4:
            again=false;
         default:
            break; //won't reach over menuSize anyway
      }
   }
}
```

I give user the ability to load before datatype is chosen. In this case the type get's determined by the first char in the "list.dat" file.

2.5 loading and saving

in case of saving to file I always save the datatype in the first line using the following compiler dependant function.

```
typeid(T).name(); //outputs compiler dependent name for type
```

I used this, because as long as I save to file using the same binary as I read from file with. It will work as long as I use the same function to campare with when I read from file.

Therefore the only drawback would be if I try to read a "list.dat" created by a binary compiled with different compiler than my own.

I write the typename to file like this:

```
ofstream os("list.dat");
os << typename(T).name() << std::endl;
```

and deduce the datatype when loading like this:

```
// loading from file
void Interface::loadFromFile()
{
   std::ifstream is("list.dat");
   char type;
   is >> type;
   if (type==*typeid(int).name()) // check for filtype
   {
      loadRun<int>();      //run apropriate run function.
   }
   else if(type==*typeid(double).name())
   {
      loadRun<double>();
   }
   else
   {
      printPrompt("ERROR TYPE IN FILE","ERROR");
   }

}
```

and loadRun is implemented like this:

```

template <typename T>
void loadRun()
{
   // loading list into Testapp and running it
   TestApp<T>().loadFromFile().run();
}
```

This works as i made the load implementation public in the TestApp class.

2.5.1 loading after initialisation

If user tries to load a "list.dat" loaded with wrong datatype a runtime error is thrown. Therefore I use a try and catch block within TestApp to handle the error and give feedback to user. Nothing is added to list, and app continues to run.

Relevant code:

```
//TestApp.cpp:
template<typename T>
TestApp<T> &TestApp<T>::loadFromFile()
{
   try
   {
      theList->readFromFile();
      menu.enableAll(); //enable all menuoption once list is ↩
          loaded
      printPrompt("elements from file loaded into list");
   }
   catch (std::runtime_error & error)
   {
      printPrompt("Elements in file are of wrong type","ERROR")↩
          ;
   }

   return *this;
}
```

```
//ListManipulator.cpp
template<typename T>
void ListManipulator<T>::readFromFile()
{
   std::ifstream is("list.dat");
   if(is)
   {
      clearList();   // in case something already is in list.
      if (is.get()!= *typeid(T).name())
      {
         throw std::runtime_error("wrong type");
      }
      std::istream_iterator<T> eos;
      std::istream_iterator<T> iit(is);
      std::copy(iit,eos,std::back_inserter(*theList));
   }
   is.close();
```

```
18 }
```

3 conclusion

I believe I found a good solution to the assignment. I could have used polymorphism but chose not to as I believed It wouldn't solve the problem of loading from file as elegantly as the current solution

4 Building/Compiling

Just run *make* in the Lab directory. To run the program run *make run* in same directory.

5 Enviroment

I'm programming on an Arch linux 64-bit system.

I've got the gcc compiler installed and compile using it's g++ alias which links necessary libraries automatically.

To compile I use the recommended flags: "-std=c++11 -Wall -pedantic". The flags let me choose to use c++11 standard and give me useful compiling warnings and errors. For editing of code i currently use VS code with a makefile.

6 Backup

And if anything's missing you can find it on:
github: `https://github.com/Hergeirs/Cpp-Obj/tree/master/Level%202/Lab5`
Cpp-obj/Lab2

October 7, 2017