C++ Project: Experimental Data Management System

Robin Kennedy-Reid

August 2019

Abstract

This report outlines the design and implementation of a C++ class hierarchy for storing experimental data. A console application has been written to demonstrate its features. An abstract base class for measurements has been written, as well as a derived class for multi-variable numerical measurements. A class for experiments, that serve as containers for multiple measurements, has been designed. The experiment class has member functions to build experiments from file and user input, as well as produce reports.

1 Introduction

This project had three main specifications. The first was to design a class hierarchy for measurement. The class had to have an abstract base class, used as interface, and derived classes for specific types of measurement. All measurement types are expected to contain a time stamp, a measurement value and two errors for that value. The second specification was to design a class for experiments, used as containers for multiple measurements. The user should be able to calculate errors for experiments and input experiment data from console or file input. Finally, a program had to be written to demonstrate the features of these classes and how to generate experiments.

The purpose of this project is to demonstrate the typical features and methods important to object-orientated programming. It should use a number of advanced features of C++11, as lectured on in the module 'Object Orientated Programming in C++'.

2 Code Design and Implementation

2.1 Measurement Classes

Two template classes have been written for measurements; the abstract base class, `Measurement`, and a derived class for multi-variable numeric measurements, `NumericMeasurement`. While the base class has no template-type member data, it is templated to allow for virtual member functions to return values and errors of template types, while retaining polymorphism. See Listings 1 and 2 for the declarations of each class.

```cpp
// In base_measurement.h
template <class T> class Measurement {
public:
   virtual T GetValue(const int) const = 0;
   virtual T GetTotalError(const int) const = 0;
   /* More pure virtual access functions...*/
protected:
   time_t timestamp_;
   Measurement() : timestamp_(time(nullptr)) {} // assign timestamp
};
```

Listing 1: Template class declaration for measurement interface. The class is templated to allow for virtual functions to return various types and retain polymorphism. The protected constructor instantiates the time stamp and can only be called from derived classes.

Measurement has one data member, a time stamp of time_t type. This variable is initiated in the default constructor of Measurement. This constructor is protected to ensure it can't be called outside of the class hierarchy. Derived classes can thus instantiate the time stamp by calling Measurement() in their constructors. Functions defined in the <ctime> header file are used in member functions to print formatted time stamps.

Vectors and arrays are a natural choice for the member data in NumericMeasurement; a measurement should be able to hold an arbitrary number of variables, unknown at compilation, but these variables should be constrained to have only 3 elements of the same type. Rather than standard C++ arrays, those defined in std::array have been used. This adds a layer of member functions common to other standard containers, thus using iterators to pass over variables and components is simpler. This is also an argument against using an alternative container such as tuple, which provides access to components using std::get. Tuple also has no advantage over an array in the case of all elements being the same type.

```
// In numeric_measurment.h
template <class T> class NumericMeasurement : public
    base_measurement::Measurement<T> {
public:
   /* Member function declarations... */
private:
   std::vector< std::array<T, 3> > data_;
   std::vector< std::string > units_;
   };
```

Listing 2: Template class declaration for a numeric measurement. A measurement is a vector of 3-element arrays which contain a value and two sources of error. A measurement with one variable would be declared as vector with one element. A vector of strings that contain the units of each variable is also contained in the class. Note the base class is nested in a namespace, base_measurement.

The constructors of NumericMeasurement check that the vectors of units and variables are the same size and display an error if not. The program does not exit in the case of this fault, as this could result in the destruction of other experiments during runtime.

2.2 Experiment Class

The class for experiments has been implemented as a container for multiple experiments. The class is templated, as it holds a vector of Measurement<T> pointers. This allows for member functions to return experiment errors of type T. The class also holds a string for the experiment title and a date stamp of time_t type, instantiated at construction. See Listing 3 for the declaration of Experiment.

Experiment has as member functions two print commands, Print and PrintToFile. Print takes as its argument a reference to an output stream and prints into it the measurements in table form. Access functions of Measurement are used to print values and display the units for each column. PrintToFile takes no arguments and generates an output file into which it prints the data using Print. The name of this file is automatically generated from the user-input title and the date stamp of the experiment. Printing reports to the console can be done simply by calling Print(std::cout).

```
// In experiment_class.h
template <class T> class Experiment {
public:
   Experiment(); // Default constructor

   void AddMeasurement(std::shared_ptr<base_measurement::
      Measurement<T>>);
   void SetTitle(const std::string&);

   T GetWeightedMean() const;
   T GetMeanError() const;

   /* Constructors, getter functions, print functions, destructor
      ... */
private:
   time_t datestamp_;
   std::string title_;
   std::vector<std::shared_ptr<base_measurement::Measurement<T>>>
      data_;
};
```

Listing 3: Template class declaration for experiments. Default construction is expected, then `AddMeasurement` used to add measurements during runtime, which pushes measurement pointers into `data_`. Note that the mean and error functions are able to return the type that `Measurement<T>` is declared to be.

Implemented into `Experiment` are two functions that compute the weighted mean and error of an experiment of single-variable measurements. These functions return the templated type `T` and use `static_cast<T>` throughout to ensure arguments to `std::pow` are of matching type. The functions also check that `data_` is not empty and that the measurements within it are single-variable before attempting calculation. If these conditions are not met, the functions return 0 and print errors. The program does not exit, to preserve other experiment objects that may exist at runtime.

2.3 Main Program and User Interface

`main()` has been written as a short demonstration of the features of the measurement class hierarchy and the experiment class. Vectors of `Measurement` pointers are created in float and double types. These vectors are then printed to the console. This demonstrates the correct implementation of templating and overridden print functions.

The program can then build experiments based on file or console input. The user can choose between these two options and build an arbitrary number of experiments, all stored in a vector of `Experiment` pointers. After completing input, the program outputs a file report and the user can choose to print a report to console. Throughout the program user inputs such as integers, doubles and multiple-choice string responses are validated.

Functions are defined to reduce repeated code and validate user inputs. For example `MakeExperiment` takes a `std::function` type as a parameter, into which functions that build experiments from user or file input can be passed without repeating the surrounding code, such as report printing.

3 Advanced Features

3.1 Templates, Header Files and Namespaces

All three classes described above are declared in separate header files, with appropriate include guards. As all the classes are templated, their definitions have not been separated into source files. The measurement class header files are wrapped in namespaces to prevent naming conflicts, and help ensure base or derived class objects are declared in the main program appropriately.
As well as template classes, template functions are defined in the main program in order to take in input to the template class `Experiment<T>`.

3.2 Smart Pointers

As can be seen in Listing 3, smart pointers have been used in this project. The experiment class contains a vector of shared pointers to `Measurement`. In `main()`, vectors of unique pointers to measurements are used in the initial demonstration. For a data-centric project such as this, the automatic garbage collection implemented by smart pointers makes the code much safer, especially as all member data in classes use standard containers. During runtime, measurements are added to experiment classes using `std::make_shared`.

3.3 Lambda Functions

Lambda functions are used a number of times in this program in order to reduce repeated code, or condense small pieces of code that may otherwise be written in verbose for-loops. Listing 4 is an example of a lambda used in the program.

```
bool is_integer(const std::string& input){
   return !input.empty() && std::find_if(input.begin(),input.end(),
      [&](char c) { return !std::isdigit(c); }) == input.end();
}
```

Listing 4: Function to check a given string input is an integer. The function uses `find_if` with a lambda expression to search over the string for non-digit characters. If only digits are found `find_if` returns `input.end()`. Thus if the string is not empty and contains no non-digit characters, the function will return `true`.

Lambda functions have been used in the main program for taking user input. In the case of multi-variable experiments, the program must repeatedly request inputs that go into the same vectors. It would confuse the program to write different output requests for each variable, and repeat the same input code after each. It would be confusing to write such specific functions alongside the more important function definitions in the code. As such, lambda expressions have been defined within these functions using `auto`, which can then be called multiple times without repeated code.

4 Discussion and Conclusion

The code as written has some limitations. The use of shared pointers to measurements in the experiment class is not the optimal choice, as the safest option would be for an experiment to be the sole owner of its data. No experiment or measurement objects are copied or intended to be copied, so this does not affect the safety of the program, as no member data is shared between objects. To change these pointers to unique pointers would make the constructors of the experiment class more complicated and require some restructuring of the measurement class hierarchy in order to implement deep-copying of data in experiments.

The project has a number of possible extensions. The experiment class could be extended to include functions that compute curve fits for multi-variable data. File input could also be made more sophisticated; currently the user must input the number of variables per measurement, which the program then reads. The program could be adapted to automatically determine this from the file. It would also be useful for the program to be able to read in data in the format that it is printed in, with time stamps and column headings.

This project has met the specification and demonstrated the key features of object-orientated programming in C++, despite some small limitations. A number of advanced features of C++11 have also been used to enhance the efficiency of the code. The project has a number of useful extensions that could be readily added to the code.

Word count, excluding headings and listings: 1650