An Experimental Data Management System

Alastair Fensome
9038778
School of Physics and Astronomy
The University
Of
Manchester

20th April 2017

1 Abstract

Using an abstract base class as an interface, a class hierarchy was made to manage data for experiments. Run time polymorphism has been used to create new objects for data storage. A vector of base class pointers may then be utilised by functions. The user may input data manually, from a file or create it with a random number generator. All data can then be analysed and saved.

2 Introduction

2.1 Motivation

Data analysis is fundamental to modern science. Physics research in particular requires extensive amounts of data. For example, the Data Centre at CERN shown in figure 1, collects 30 petabytes (3×10^{16} bytes) of data annually [1]. Hence, efficient programming and secure data handling is as crucial to an experiment's success as any other piece of apparatus. Furthermore, safe data management is important to businesses and industry and for this reason there will always be a need for robust data management systems.

Figure 1: Photo of CERN Data Centre (DC) main room. The whole DC contains 1450 m^2 of servers and analysis 30 petabytes of data annually [1].

2.2 Project Description

The project was designed to manage data for an experiment where measurements are divided up into two subcategories, Type-1 and Type-2. These each contained a data point, an error on the data point, a calibration factor and a date stamp. The program may read in these results into classes from any

combination of the following: files, manually input or generated randomly from user input parameters. Once the user has amassed the data they can choose to save it to two separate files for type-1 and type-2 measurements, display the data, calculate the weighted mean and weighted error or clear the data to load a new experiment. When the user wishes to exit, the program destroys all objects, automatically freeing the memory and clearing all data. The program defines a weighting of the i^{th} measurement w_i as in equation 1. where Δx_i is the error on the i^{th} measurement.

$$w_i = \frac{1}{\Delta x_i^2} \tag{1}$$

This allows for the weighted mean $\bar{x}$ to be calculated as in equation 2, where x is the value of data and c is a calibration factor used to account for systematic error.

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i x_i c_i}{\sum_{i=1}^{n} w_i} \tag{2}$$

Hence the weighted error on the mean $\bar{\sigma}$ is calculated with equation 3.

$$\bar{\sigma} = \sqrt{\frac{1}{\sum_{i=1}^{n} w_i}} \tag{3}$$

3 Implementation and Code Design

3.1 Class System

An abstract base class called Measurement was used as an interface for two derived classes, Type-1 and Type-2. These derived classes contained the following private variables:

1. data a double for holding a data point.
2. error a double for the error on the data point.
3. calibration factor a double which accounts for systematic error by functioning as a multiplier to data.
4. date stamp a string that holds the time and date stamp for a measurement.

Measurements contained the following pure virtual functions which are overridden in its derived classes:

1. Get_Data an access function to return the data.
2. Get_Error an access function to return the error.
3. Get_Calib_Factor an access function to return the calibration factor.
4. Get_Date_Stamp an access function to return the date and time at which a measurement was taken.
5. Info a function to print all the data to the screen in a well formatted manner.
6. Set_All a function that allows the variables to be set for an object.

Using these variables and functions allows runtime polymorphism to routinely store and manage data. Because vectors can be easily resized using the push-back function, they were a suitable choice to store base class pointers.

3.2 Main Functions and Template Functions

The base class has two derived classes which means that any function acting on the objects must be generalised to accept multiple data types as inputs. To overcome this, template functions were used throughout the code. This allowed the object's data type to be defined on implementation rather than needing two separate functions to act on the two objects. Hence template functions reduced the number of functions and function overrides. To keep the main code concise and legible most of it was contained in functions or template functions. The following is a list of functions and their features.

1. Menu
   This function is used to output the menu which is shown in figure 2. The menu function is invoked at the beginning of a do loop which is executed while the user has not selected the exit option. This makes the program more user friendly and does not break if the user inputs an incorrect option. When an option is entered, if statements check which function to call. Once the exit option is selected the program destroys all objects and clears all data.

2. Get_Date_Time
   This function uses the localtime_s function in the standard library to return the current date and time as a string in the format year_month_day_hour_minute_second. The date stamp is then used to name saved data files allowing them to be displayed in chronological order.

3. Enter_Data_Manually
   This is a template function which allows the user to enter as much data as they wish by typing it into the console. This function uses the getline function and a string-stream to split the input line into the relevant data types and saved to a new object.

4. Enter_Data_File
   A template function to read in data from files. The function first prompts the user to input a name of a file which it then opens if it exists. If the file does not exist or should it fail to open then the function will print "Error: The file could not open correctly." and return. The code then finds the number of lines in the files and checks they're not blank. The first character of the line is checked to make sure it is not erroneous by confirming that it is a number. A string-stream is used to split up the input line into the relevant data types which are then saved to a new object.

5. Print_Data
   A template function which invokes the info function in a loop for the length of the base class pointer vector. This results in a full list of data being printed to the screen.

6. Weighted_Mean
   A template function which corrects for systematic error using the calibration factor, calculating values for weighted mean using equation 2 and error on the weighted mean using equation 3. The values are then printed to screen and appended to the end of "Results.txt" under a header name which the user must input.

7. Artificial_Data
   A template function made to produce data using a random number generator. The user must input the parameters for number of data

points, data, error and calibration factor. The data is then created randomly using:

$$random\,data = \frac{data \pm error \times (random\,number\,between\ +1\,and\ -1)}{calibration\,factor}$$

Which means when the Weighted_Mean function corrects for the systematic error, the mean of the data is concordant with the parameters input by the user. Errors are produced randomly as follows:

$$random\,error = error \times \left[1 \pm \frac{random\,number\,between\ +1\,and\ -1}{2}\right]$$

The calibration factor stays constant at the value selected by the user.

8. Save_Data
   A template function designed to save the data, which gives the user three options for naming the file: the type, a date stamp followed by the type or a custom name chosen by the user. Once a suitable name has been entered, the program saves all data to the file and closes it.

9. Clear_Data
   A template function which uses an auto loop to delete the base class pointers. After this the vector is cleared too. This function is invoked if the user picks options 4 or 7, to clear all data or exit respectively.

```
Choose an option from the menu by entering the corresponding number.
1. Enter data manually.
2. Enter data from file.
3. Calculate values, account for calibration factor and display results.
4. Clear all data.
5. Create artificial data.
6. Save all data.
7. Exit.
```

Figure 2: The options menu which allows the user to choose which functions to execute.

3.3 Exception Handling and Safety Measures

One try loop was made to enclose all the main code where errors are likely to be thrown. This helped to keep the main code short. Two exceptions have

been made, as shown in figure 3. The first exception is thrown whenever there is a chance of dividing by zero. The second exception occurs when the user assigns too much memory. It is thrown if the user tries to create more than one million objects. If an exception is caught the program will output an error message, however it does not exit the program as the memory might not have been freed and the user may wish to save the data before exiting.

```
catch (int errorFlag) // error catching!
{
    if (errorFlag == divide_0_flag) // divide by zero error!
    {
        cout << "Error: Can not divide by zero." << endl;
        cout << "It is recomened that you clear memory. Enter option 4 to do this now." << endl;
    }
    if (errorFlag == memory_flag) // entered too high number of measurments
    {
        cout << "Error: Too many artificail measurments input." << endl;
    }
}
```

Figure 3: Catch loop at the bottom of main code, should an error be thrown it will output the corresponding error report.

4 Results

4.1 Inputs

There are three ways of inputting data into the program which may be done any number of times and in any order.

1. Manually
   By selecting the first option data may be entered by typing three numbers separated by a space then pressing enter. Any amount of data can be entered this way. When the user has finished they must enter the character 'x' on a new line to stop entering data.

2. From a file
   The program reads data from files in which the data is formatted as follows: date-stamp data error calibration-factor. The program will also check for errors by making sure the line is not empty and is not made up of letters.

3. Created randomly by the Artificial_Data function using user input parameters.

4.2 Outputs

4.2.1 Calculated Results

The code outputs data and calculated values straight to the console upon entering option 3, "Calculate values, account for calibration factor and display results". However, the program also prompts the user to name the experiment and saves that name, the weighted mean and weighted error to a text file called "Results". The results file is kept as a permanent record of each experiment's weighted mean and weighted error and information may only be appended onto the end of it.

4.2.2 Saving Data

Any data can be saved to a text document by choosing option 6 "Save all data". Furthermore, the data is saved in exactly the same format as the input files which means data can be easily generated, saved, cleared and then reloaded.

5 Discussion

5.1 Smart Pointers

Smart pointers could have been used as an alternative to standard pointers. The benefit of using them is that objects would automatically be destroyed after they go out of scope. This could have improved the program because it eliminates the possibility of creating an object and not freeing the memory once its lifetime is over.

5.2 Template Classes

The program could have been made more generic with the use of template classes. This would allow the class to be used for different data types and would have been useful when making multiple types of measurements. However, template classes were not chosen because there was no need to have a

generic class that holds any data type. Furthermore, runtime polymorphism is better suited to dynamic creation and destruction of objects than static polymorphism.

5.3 Generalisation

It would be possible to generalise the code to accommodate multiple types of measurement. This would allow the user analyse more data before needing to either save or clear the data. This could have been achieved by using runtime polymorphism to create objects which hold data in arrays or vectors and then use a new object per type of measurement.

6 Conclusion

In this project a program was designed and built to manage data. An abstract base class with two derived classes was chosen to use runtime polymorphism effectively. The code applies template functions to generate, read in, save and analyse data for two types of measurements. As discussed in section 5 there are several improvements which could be made to the program such as smart pointers, more generalisation of the code and the use of template classes.

References

[1] Computing — CERN, (2017), (Accessed: 27th April 2017). https://home.cern/about/computing