Part IA Computing Course
Lent Term Software Design Exercise

Roberto Cipolla
Department of Engineering
University of Cambridge

January 13, 2005

This handout describes the software design problem which is to be implemented in the C++ programming language in the Lent term laboratory sessions of the Part IA Computing Course. In addition to this handout you will need to review the *Tutorial Guide to C++ Programming*. The programming concepts needed to implement a solution have been covered in the lecture course.

Contents

1 Introduction and Organisation **3**

2 Problem Statement: A Trading Problem **4**

3 The Software Design Process **5**

4 Solution Specification and Problem Modularisation **6**

5 High-Level Design **8**
- 5.1 Modularisation 8
- 5.2 Multiple Source Files 9
- 5.3 Data Structures and the Header File 11
- 5.4 Function Names and Prototypes 14

6 Detailed Design, Implementation and Testing **15**
- 6.1 Getting Started 15
- 6.2 Detailed Design 15
- 6.3 Testing an implemented function 15
- 6.4 Implementation of a function and unit testing 16

7 Notes on Implementation of Functions of Part I **17**
- 7.1 Reading from files 17
- 7.2 Displaying data with the Vogle library functions 18
- 7.3 Method of Least Squares 19
- 7.4 Analysis of Residual Errors 20

8 Notes on Implementation of Functions of Part II **21**
- 8.1 Electronic trading library functions 21
- 8.2 Sample function to initialise trading account and handle exit code. 23
- 8.3 MyTrader Function 25
- 8.4 Closing the account and writing the records to a file 25
- 8.5 Definition of trading library constants using enumeration 26

9 Integration, Final Testing and Evaluation **27**

10 Further Reading **27**

1 Introduction and Organisation

A. Aims

The analysis of a problem and the design of a solution in the C++ programming language. The implementation, testing and evaluation of software. The fostering of software design and cooperative skills through teamwork.

B. Objectives

☐ Problem solving using abstraction and modularisation
☐ Structured programming and program modularisation using functions
☐ Reading and writing to files
☐ Using data structures
☐ Passing parameters to functions by reference
☐ Using library functions and handling exit codes
☐ Compiling programs written in multiple source files

C. Organisation and Marking

- **Preparation (4 hours)**

  You should review your lecture notes on **functions** and **data structures** and the Tutorial Guide to C++ Programming, including section 11, before your first Lent term laboratory session. You should also study the problem statement and background material in this document (sections 1–5).

- **Timetable (8 hours)**

  There are 4 two hour laboratory sessions scheduled for the Lent term computing course.

- **Teamwork**

  The software design exercise is to be completed in laboratory group pairs.

- **Qualification (12 marks)**

  Twelve coursework marks will be awarded for a working program meeting the specification. An additional four marks will be awarded for well-structured programs demonstrating good software engineering practice and for the quality (clarity and simplicity) and performance of the solution.

- **Marking**

  Your solutions will be marked by a laboratory demonstrator who will require a listing of your program and a screen dump of the graphs. The demonstrator will execute the program and evaluate its performance in the final timetabled laboratory session.

2 Problem Statement: A Trading Problem

A company building a high-tech information retrieval system is required to purchase 1000 components within 50 days.

The unit price of the components varies from day to day. At the close of trading, the previous day, the unit price was £20. The price variation over the previous 100 days is recorded in the file `tradingData.dat` and displayed in Figure 1.

The company has £20000 assigned for the purchase of the components. It can buy and sell any volume of components over the 50 day trading period but must ensure it has 1000 components at the end of the period and remains with a positive balance throughout the trading period.

Trading is to be performed automatically by a computer program running on a machine networked to an electronic trading exchange. The trading strategy should aim to maximise the balance whilst securing the purchase of 1000 components by the end of the trading period.

> You are required to **design, implement, test and evaluate** a program written in C++ to display and analyse the past trading price data. Your program should then connect to a trading exchange computer and automatically buy and sell components over the 50 day trading period to secure the purchase of 1000 components at the lowest average price.

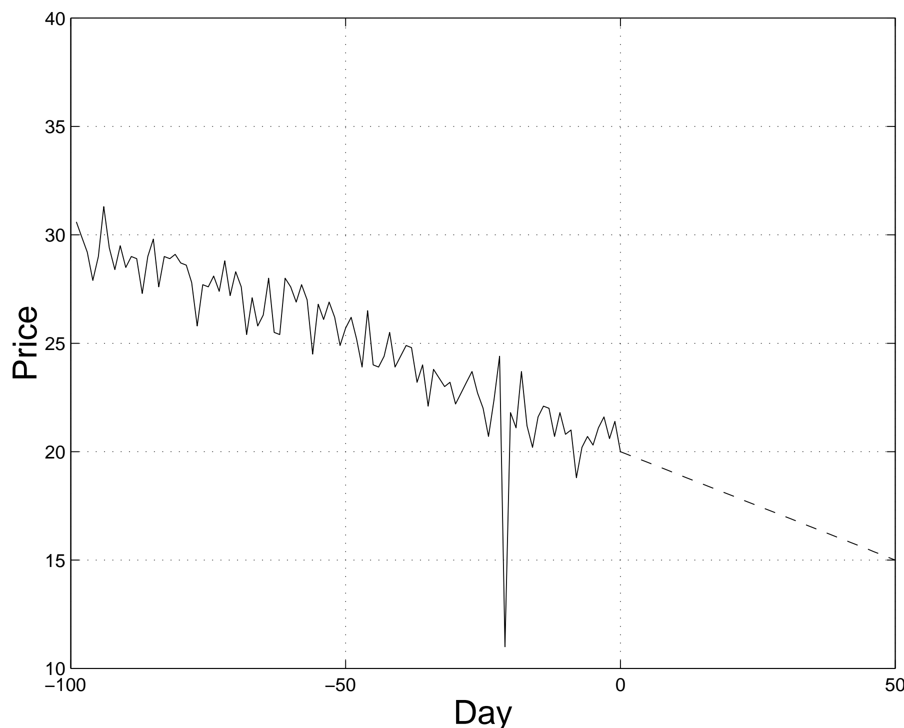

Figure 1: Price variation over the last 100 days with linear prediction.

3 The Software Design Process

Software Engineering is *the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software*. As in other branches of engineering, one of the central ideas is that of decomposing a large difficult problem into a set of simpler sub-problems, these sub-problems may then be further decomposed into yet smaller sub-problems until a level of complexity is reached at which the solution becomes relatively trivial.

Developing computer software involves a design process with the following distinct activities:

Step 1: Requirements analysis and specification (section 4)

Step 2: High-level design (section 5)

- ☐ Modularisation – dividing the problem into parts (modules) and sub-parts (components) which will be implemented by calls to functions
- ☐ Using multiple source files
- ☐ Design and specification of key data structures
- ☐ Specifying the interface between different parts of the program and choosing the names of functions and the parameters to be passed
- ☐ Specifying the user interface for writing prompts, printing error messages and displaying data.
- ☐ Agreeing on a consistent style for writing C++ code, comment statements and the indentation after each set of braces.

Step 3: Detailed design (sections 6–8)

- ☐ Specifying the *function prototype*: function name, parameters and return type.
- ☐ Specifying the *algorithm* (computational steps from input to output) for each function.

Step 4: Implementation of each function and unit testing (sections 6–8)

- ☐ Beginning with the function header, comments and the first and last statement
- ☐ Writing the C++ code for the body of the function
- ☐ Mentally checking that it fits the requirements
- ☐ Compiling and removing syntax errors
- ☐ Checking for errors in logic and algorithm
- ☐ Checking that the specification is met

Step 5: Integration and system testing (section 9)

Step 6: Evaluation and enhancement (section 9)

The following information will guide you through these activities and help you to design and implement a working solution. Step 1 has already been carried out for you and is described in section 4. Suggestions for step 2 are given in section 5.

4 Solution Specification and Problem Modularisation

You should design a solution with the following specification. The program should:

1. **Read the trading price data**

   Read the data (price over the previous 100 days) from the file `tradingData.dat` and store as a data structure (e.g. `ModelData`).

2. **Display the trading price data**

   Display the trading data graphically using Vogle library functions.

3. **Analyse the price variation over the past 100 days**

   Estimate the parameters of a linear model by fitting a straight line to the data using the method of least-squares. Output the equation of the line to the screen. Superimpose the straight line on the display of the data.

4. **Compute the statistics of the residual errors**

   Compute the mean and variance of the errors between the actual price and the model's prediction. Output these values to the screen. Check that the data agrees with the model.

5. **Initialise trading account**

   Use the trading exchange library function `TE_InitialiseTrading()` to establish a connection with the electronic trading exchange server.

6. **Perform trading**

   Use the trading exchange library functions `TE_GetPrice()` to get each day's price (for day 1 to 50).

   Devise your own trading strategy to determine when to buy and sell and the volume of the transaction.You must ensure a positive balance at all times and the purchase of 1000 units by the end of the trading period (i.e. on day 50).

   Implement your strategy for each day by calling a function that decides on whether to buy or sell (i.e. transaction type) and the volume of the transaction. Use the trading exchange library function `TE_Trade()` to perform the transaction.

   At the end of trading on each day determine the new stock and balance and update your accounts (e.g. using the `TradingAccount` data structure).

7. **Display trading accounts and compare to exchange records**

   Output a record of your trading accounts (day, price, transaction type and volume, stock and balance) to a file. The trading exchange library function `TE_CloseTrading()` will compute your balance and the number of components (stock) at the end of trading. Check that these agree with your own accounts.

Timetable

Session number	Objectives and exercises	Time required
Session 1 (Day 1 morning)	Review section 1 to 4 with partner Teamwork and understanding problem and specifications Understanding of environment and data structures List of function prototypes (section 5.4) Testing of implemented functions (section 6.3) Marking of teamwork and design (steps 1 to 3)3	 30 minutes 30 minutes 30 minutes 30 minutes
Session 2 (Day 1 afternoon)	Implementation of functions	60 – 120 minutes
Session 3 (Day 2 morning)	Implementation of remaining functions	60 – 120 minutes
Session 4 (Day 2 afternoon)	Shipping working functions to team directory Testing on test data sets Final evaluation on unseen data Marking	 60 minutes

5 High-Level Design

5.1 Modularisation

Your program must be constructed modularly by dividing it into two parts (*modules*): data analysis and trading. Each part will be implemented independently with calls to **functions**. The following is an example of the modular structure of the final program.

```
// OurTradingMain.cc
// Sample program layout for Lent Term software design exercise

// Standard library header files
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cmath>
#include <vogle.h>
#include <vogleextras.h>
#include <trading.h>
using namespace std;

// User-defined constants, data structures and function declarations
#include "OurTradingHeader.h"

int main()
{
   // Declare data to be a structure of type ModelData
   // Declare account to be a structure of type TradingAccount

   // Call to function to read trading price data from file (100 days)
   // Call to function to display data using Vogle library
   // Call to function to calculate linear model by method of least-squares
   // Call to function to analyse mean and variance of residual errors
   // Call to function to initialise trading account
   // Call to function to trade and update accounts over next 50 days
   // Call to function to close trading account and output accounts

   return 0;
}

// Function definitions of part 1 and 2 are in separate source files
// called OurTradingFunctions1.cc and OurTradingFunctions2.cc
```

Each partner will be responsible for the implementation and testing of different parts (modules) of the program. Divide the required functions between you. For example, the first partner may choose to implement the 4 functions of part/module 1 (up to the analysis of the mean and variance of the residual errors).

5.2 Multiple Source Files

The design exercise will require you to create a program for which different parts of the C++ source code will be in different files (called project files), all with the same base name, `OurTrading`.

1. The main program file – `OurTradingMain.cc` – will contain the `main()` routine with **function calls** to implement the different parts of the program (see example above).
2. The *header file* – `OurTradingHeader.h` – will contain the definitions of the data structures and the **declaration** of **all** user-defined functions (i.e. a list of function prototypes developed by you and your team partner). See next section for an example.
3. The **function definitions** will be placed in two separate source files. The source code for the function definitions of part 1 (first partner) should be placed in the file `OurTradingFunctions1.cc`.
4. The source code for the function definitions of part 2 (second partner) should be placed in a **separate file** called `OurTradingFunctions2.cc`.

Editing Project Files

C++ source files can be grouped into the same project by giving them the same base name and by *selecting* them (by clicking on their file icons using the middle mouse button) in the **File Manager**. Dragging one of these file icons on the C++ Compiler icon in the **Applications** window will load **all** the project files into the C++ Programming Enviroment.

The project files should be edited using a **single** Emacs window. Select the main project file, `OurTradingMain.cc`, in the C++ Programming Environment (xcc window) [1] and click on the `Edit` button. To edit any of the project files, use the Emacs `Files` and `Buffers` menus to change the edited file from `OurTradingMain.cc` to any of the other project files.

Compiling and Linking using MAKE

Programs which use multiple source files can be compiled and linked by clicking the `Make` button on the C++ Programming Environment (xcc) window.

The complete program can be compiled by loading the file `OurTradingMain.cc` which contains the `main()` function. Clicking the `Make` button will first compile the functions in the files `OurTradingFunctions1.cc` and `OurTradingFunctions2.cc` and then link their object code with the routines from the standard libraries and the compiled version of `OurTradingMain.cc` – see Figure 2. Clicking `Run` will execute the complete and integrated program.

Printing Project Files

You may print any file (including the header) by dragging the file icon onto the `Plotview` icon in the applications window.

[1]The environment has been modified to include an additional `Make` button.

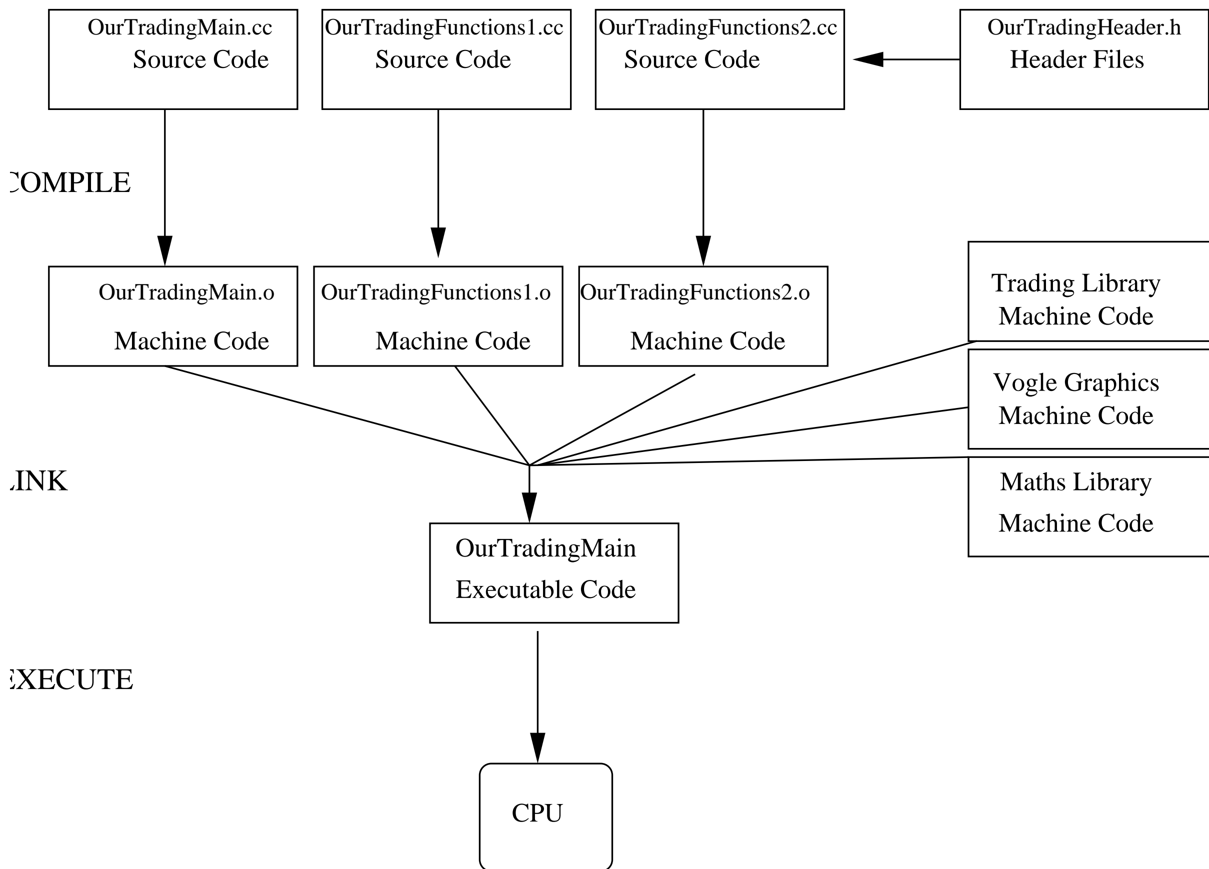

Figure 2: Multiple source files can be grouped into a project and are compiled and linked with the MAKE command.

5.3 Data Structures and the Header File

You will need to select key variables and **data structures** for processing the data. For this design exercise two data structures have already been designed for you (defined below) and can be used without modification.

The ModelData structure

The `ModelData` structure type is made up of 8 *fields* and can be used for the first part of the project to store the trading data and the linear model parameters and least squares analysis.

```
struct ModelData{
   int numPoints;
   float x[MAXSIZE];
   float y[MAXSIZE];
   float a;
   float b;
   float residual[MAXSIZE];
   float mean;
   float variance;
};
```

The day and price data can be stored as arrays in the `x` and `y` *fields.*

The model parameters ($y = a + bx$) can be stored in the `a` and `b` *fields.*

The remaining fields can be used to store the residual errors (also an array) between the model data and actual data and the mean and variance of these errors.

Remember that the data members of a structure are accessed and assigned values by specifying the field and using the *dot operator*. For example:

```
ModelData data;

data.x[0] = 1.0;
data.y[0] = 30.6;
```

declares `data` to be a structure of type `ModelData` and assigns 1.0 and 30.6 to the first elements of the `x` and `y` arrays respectively.

The TradingAccount structure

The `TradingAccount` structure type can be used for the second part of the project to keep a record of the price and transactions performed with the electronic trading exchange.

```
struct TradingAccount{
   int today;
   float price[MAXSIZE];
   int transaction[MAXSIZE];
   int volume[MAXSIZE];
   int stock[MAXSIZE];
   float balance[MAXSIZE];
};
```

The `today` field should record the number of valid entries in the arrays. This should be set to the number of days in which trading has taken place. The remaining fields can be used to record the price, transaction type (buy, sell or pass), volume of trading and the stock and balance at the *end of each day* of trading. The index of these arrays refers to the trading day.

For example:

```
TradingAccount account;

account.price[0] = 20.0;
account.balance[0] = 20000.0;
```

declares `account` to be a structure of type `TradingAccount` and sets the value of the first element of the price and balance arrays (day 0) to be 20.0 and 20000.0 respectively.

Passing by reference

In this design exercise the data members of data structures of these types should be processed by passing them to functions using *passing by reference*. This allows the function to read and change the value of any of the data members. Passing by reference is indicated by including the symbol `&` before the structure name in the function header and prototype. For example the function header:

```
void ReadDataFile(ModelData &data)
```

indicates that a structure of type `ModelData` is passed to the function `ReadDataFile()` and that the function can read and reassign the values of any of the data members.

You should review your lecture notes and also section 7.8 and 11 of the Tutorial Guide to C++ Programming if you are unsure about *passing by reference* or assigning and accessing values of the data members of a structure.

The Project Header File

Structure definitions and declarations of constants should be placed in the project header file `OurTradingHeader.h` (by using the `#include "OurTradingHeader.h"` directive at the top of the program file). [2] A working copy of `OurTradingHeader.h` can be found in the `1ATrading` examples directory (see Getting Started section).

```
// OurTradingHeader.h
// Project header file for trading exercise. Contains definitions of constants,
// data structures and function declarations (prototypes)
// Included at top of all project files

// Constants
const int MAXSIZE = 100;
const int LABGROUP = 150;

// User definition of a data structure for least squares analysis
// x, y and residual are arrays of type float
struct ModelData{
   int numPoints;
   float x[MAXSIZE];
   float y[MAXSIZE];
   float a;
   float b;
   float residual[MAXSIZE];
   float mean;
   float variance;
};

// User definition of a data structure for the trading accounts
struct TradingAccount{
   int today;
   float price[MAXSIZE];
   int transaction[MAXSIZE];
   int volume[MAXSIZE];
   int stock[MAXSIZE];
   float balance[MAXSIZE];
};

// User-defined function declarations for part 1
void ReadDataFile(ModelData &data);

// User-defined function declarations for part 2
void InitialiseAccount(int dataSet, int labGroup, TradingAccount &account);
void HandleExitCode(int errorCode);
```

[2] An *include* directive enclosed in quotes instead of angle brackets (`<` and `>`) indicates that the file is in the same directory as the program file.

5.4 Function Names and Prototypes

After selecting the key variables and data structures you should agree with your partner on the interface between different parts of the program. In particular you must specify the interface between all the functions.

For each function this will require you to fix:

1. **Function parameters:** Specify the type and name of all parameters and whether they will be passed by *value* or *reference*.
2. **Return type:** Agree on the type of the value returned. If no value is returned by the function the return-type will be `void`.
3. **Function name:** Choose a name for your function.

For example, a suitable *function header* for the function to read the trading price data from a file is:

```
void ReadDataFile(ModelData &data)
```

A suitable header for the function to initialise an electronic trading account is:

```
void InitialiseAccount(int dataSet, int labGroup, TradingAccount &account)
```

Specify the **function header** of all of the functions for part 1 and part 2, listed below. You and your team partner are required to implement the following functions.

Part I

☐ Function to read trading price data from file (already implemented)
☐ Function to display data using Vogle library routines
☐ Function to calculate linear model by method of least-squares
☐ Function to analyse mean and variance of residual errors

Part II

☐ Function to initialise trading account (already implemented)
☐ Functions to trade and update accounts over next 50 days
☐ Function to close trading account, check final stock and balance and display account.

Ask a demonstrator to check your list before proceeding. A copy of each header (*the function prototype*) should be placed in the project *header file*, `OurTradingHeader.h`. Each separate source file (e.g. `OurTradingFunctions1.cc`) should also *include* a copy of this header file.

6 Detailed Design, Implementation and Testing

6.1 Getting Started

After working through section 4 and 5 with your partner you are in a position to begin the implementation. If you have not already done so, log onto the teaching system by typing in your user identifier and password. Start the File Manager environment by typing:

```
start 1AComputingLent
```

The above command will create a directory called `1ATrading`. This contains incomplete, but working copies of the following files:

1. A data file called `tradingData.dat` which contains the price data over the last 100 days.
2. A sample header file, `OurTradingHeader.h`, which contains the definitions of sample data structures which should be used in the implementation of the solution.
3. A program file called `OurTradingMain.cc` which contains a sample `main()` routine. This sample outlines the modules required and shows how they will be implemented by calls to functions.
4. Two source files, `OurTradingFunctions1.cc` and `OurTradingFunctions2.cc`, containing the definitions of some of the functions that will be called from the `main()` routine. Each partner will add the C++ source code of the functions they implement to these files.

This command will also set up a directory called `1ASoftwareDesign` which can be read and written to by members of your team. This directory will be used for testing the final integrated product.

6.2 Detailed Design

Sections 7 and 8 give suggestions and details on the algorithms that need to be implemented for each function and how to interface with the trading library routines. Read section 7.1 or 8.1 and 8.2 now so that you can test the functions that have been implemented for you already.

6.3 Testing an implemented function

Two functions have already been designed and implemented in C++ for you. The functions `ReadDataFile()` and `InitialiseAccount()` are defined in the source files `OurTradingFunctions1.cc` and `OurTradingFunctions2.cc` respectively.

You can test them by:

1. **Source code in OurTradingFunctions**

   Check that the source code for the functions is in the project file `OurTradingFunctions1.cc` or `OurTradingFunctions2.cc`.

   Make sure you understand the C++ source code of the function definitions and how they use the data members of the data structures, `data` and `account`. Note that

the data structures are passed to the functions by reference. Ask a demonstrator for help if you are unsure of any of the details.

2. **Declaring the function in the project header file**

   Check that a copy of the function header (*function prototype*) is in the project header file `OurTradingHeader.h`.

3. **Adding a Function Call**

   Call the functions from the `main()` routine in the main project file, `OurTradingMain.cc`. For example, by adding the statements:

   ```
   ReadDataFile(data);
   InitialiseAccount(TEST1, LABGROUP, account);
   ```

   You will also have to declare the data structures that you will use (e.g. `data` and `account`).

4. **Compiling and linking the project files using MAKE**

   Compile and link the project files (`OurTradingMain.cc`, `OurTradingFunctions1.cc` and `OurTradingFunctions2.cc`) using the `Make` button.

5. **Executing the binary**

   Run the program `OurTradingMain` and check that the two functions execute correctly.

6.4 Implementation of a function and unit testing

You are now ready to implement your first function (e.g. function to display the data graphically). Follow these steps for each function that you implement:

1. Begin the implementation of each function by typing in the function header (a copy should already have been included in header file) and opening and closing braces in one of the source files, e.g. `OurTradingFunctions1.cc`.
2. Type in the body of the function and check the syntax of each statement.
3. Mentally check that the function fits the requirements.
4. Compile the file containing the function definition and check for syntax errors.
5. Check that the function has been correctly declared (*function prototype*) in the project header file, `OurTradingHeader.h`.
6. Call the function from the `main()` routine in `OurTradingMain.cc`.
7. Compile and link the project files using the `MAKE` button.
8. Run the executable `OurTradingMain` to test the algorithm implemented.
9. Proceed with the development cycle of editing, compiling and testing for syntactical and logical errors.

7 Notes on Implementation of Functions of Part I

7.1 Reading from files

The following function uses an `ifstream` (input file stream) *object* called `fin` to read the trading data from a file and store it in a user-defined data structure of type `ModelData`. The data structure is *passed by reference*. The function definition has been placed in the source file, `OurTradingFunctions1.cc`.

```
// Function that loads trading data from file
// Price data is stored in the y field of a structure of type ModelData
// Structure is passed by reference (&)

void ReadDataFile(ModelData &data)
{
   char fileName[80] = "tradingData.dat";
   int number = 100;
   int i, day;
   float price;

   // Associate file name with fin object of class ifstream
   ifstream fin;
   fin.open(fileName);

   // Prompt for new file name if not able to open
   while(!fin.good())
   {
      cout << "Unable to open trading data file. Enter a new name: ";
      cin >> fileName;
      fin.open(fileName);
   }

   data.numPoints = number;
   for(i=0; i< number; i++)
   {
     fin >> day >> price;
     data.x[i] = day;
     data.y[i] = price;
   }

   fin.close();
}
```

7.2 Displaying data with the Vogle library functions

The Vogle (Very Ordinary Graphics Learning Environment) graphics package provides C++ functions for doing simple graphics and plotting data.

The function prototypes and an explanation of the parameters are listed.

- `void vogleinit(float xlo, float xhi, float ylo, float yhi);`
  Creates a window with default background (WHITE), default foreground (BLACK) and a default margin around the graph. The parameters specify the minimum and maximum x and y values to be plotted.
- `void xaxis(float xlo, float xhi, int nxticks, float y, float ticksize, char title[], float labello, float labelhi);`
  Draw an axis between (`xlo, y`) and (`xhi, y`). The axis should have `nxticks` of length `ticksize`, and a `title` (`char` array) which is placed under the axis. The ticks themselves are labelled with values ranging from `labello` to `labelhi`.
- `void yaxis(float ylo, float yhi, int nyticks, float x, float ticksize, char title[], float labello, float labelhi);`
  Draw an axis between (`x, ylo`) and (`x, yhi`). The axis should have `nyticks` of length `ticksize`, and a `title` (`char` array) which is placed at the side of the axis. The ticks themselves are labelled with values ranging from `labello` to `labelhi`.
- `void color(int color);`
  Options for `color` include `BLACK`, `RED`, `GREEN`, `YELLOW`, `BLUE`, `MAGENTA`, `CYAN`, `WHITE`.
- `void point2(float x, float y);`
  Draw a point at x, y.
- `void move2(float x, float y);`
  Move graphics position to point (x, y).
- `void draw2(float x, float y);`
  Draw from current graphics position to point (x, y).
- `int getkey();`
  Return the character (ASCII ordinal) of the next key typed at the keyboard.
- `void vexit();`
  Reset the window/terminal (must be the last Vogle function called).

An example of a program which uses these functions can be found in `PlotNormal.cc` in the 1AC++Examples directory. The `PlotResultsVogle()` function can be easily modified to display the trading data and linear model.

For example, to draw an x-axis at `y=0.0` from `xlo=0` to `xhi=100` with `nxticks=6` ticks at $0, 20, \ldots 100$ and labelled with the text `Day`, the Vogle library function `xaxis()` can be called with the following arguments:

```
xaxis(0, 100, 6, 0.0, 0.005, "Day", 0, 100);
```

7.3 Method of Least Squares

The method of least squares can be used to fit a straight line $y = a + bx$ to a discrete set of data points $(x_1, y_1) \ldots (x_n, y_n)$ where $n > 2$. The values of the model, a and b, which minimise the sum of the squares of the errors (measured in the y-direction) of the data points to the straight line satisfy:

$$an + b\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \quad (1)$$

$$a\sum_{i=1}^{n} x_i + b\sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i \quad (2)$$

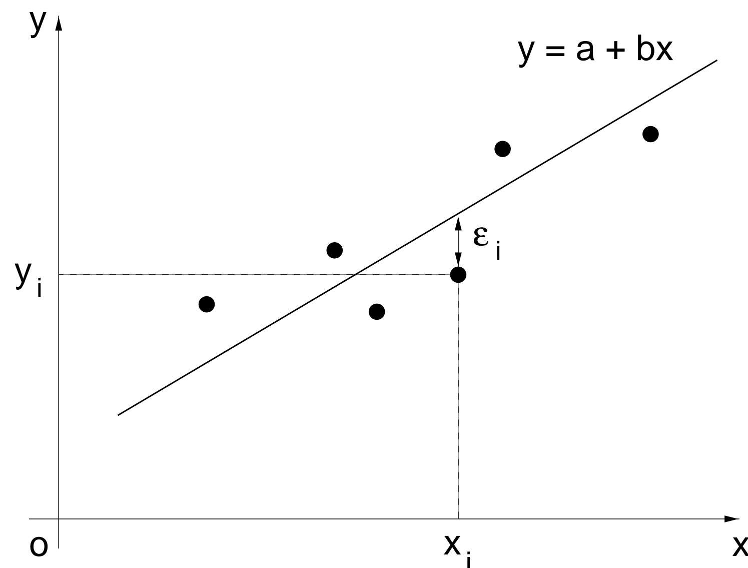

Figure 3: Method of Least Squares

Do not worry about the derivation of these equations. To determine the straight-line (model) parameters, a and b, you will need to solve the two linear simultaneous equations above after evaluating the sum of the x-values, y-values, x^2 values and xy values (e.g. S_x, S_y, S_{xx} and S_{xy} respectively).

$$a = \frac{(S_{xx}S_y - S_x S_{xy})}{(nS_{xx} - S_x S_x)} \quad (3)$$

$$b = \frac{(nS_{xy} - S_x S_y)}{(nS_{xx} - S_x S_x)} \quad (4)$$

7.4 Analysis of Residual Errors

After fitting the straight line to the data, we can analyse the residual errors ϵ_i. The error for each measurement, ϵ_i, is given by the difference between the measured value, y_i, and the prediction based on the model:

$$\epsilon_i = y_i - (a + bx_i) \tag{5}$$

The sample mean (μ) and the sample variance (σ^2) of the residual errors are given by:

$$\mu = \frac{1}{n}\sum_{i=1}^{n}\epsilon_i \tag{6}$$

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\epsilon_i - \mu)^2 \tag{7}$$

The mean should be approximately zero if the linear model fits the data correctly. Use the fields (`residual`, `mean` and `variance`) of the `ModelData` structure to record the residual errors (ϵ_i), mean (μ) and variance (σ^2) respectively.

8 Notes on Implementation of Functions of Part II

8.1 Electronic trading library functions

The second part of the design exercise requires that you devise a trading strategy for the next 50 days. Trading is to be carried out automatically under computer control. You will need to call the following library functions to interface with a networked trading server that handles buying and selling of the components:

1. `TE_InitialiseTrading()` – Establishes communication with the trading exchange server and initialises an account for the 50 day trading period.
2. `TE_GetPrice()` – Gets the quoted price for the component (to be called at the beginning of each day).
3. `TE_Trade()` – Makes a transaction (buying or selling an integer volume of components).
4. `TE_CloseTrading()` – Closes the trading account and disconnects from the electronic exchange. The function will return the final balance and stock at the end of the trading period recorded by the exchange.

You do not need to know how these library functions have been implemented. You are only required to understand how your program should interface with them. This information is readily available from the function prototypes (found in the header file `trading.h` and listed below) for the electronic trading library functions.

Below each prototype is a list of parameters. All these library functions return the integer value of an exit (error) code which is set to `TE_OKAY` (i.e. 0) if the function execution was successful. Any other return value indicates that an error has occurred.

1. **Initialise trading**

```
int TE_InitialiseTrading(int dataSet, int groupNo);
//
// Parameters list:
// dataSet      Which set of prices to use? (TEST1, ... TEST10, FINAL)
// groupNo      Lab Group Number
//
// Return values:
// TE_OKAY               trading session initialised (no error)
// TE_ALREADY_CONNECTED already connected to server
// TE_AUTH_FAILED        authorisation failed
```

2. **Get today's price**

```
int TE_GetPrice(int day, float &price);
//
// Parameter list:
```

```
// day          trading day, starting at 1
// price        the price for today (passed back by reference)
//
// Return values:
// TE_OKAY          price will be valid (no error)
// TE_NOT_CONNECTED you weren't trading anyway
// TE_TOO_EARLY     cannot look into the future! - you must
//                  make a transaction before incrementing the day.
```

3. **Make a transaction**

```
int TE_Trade(int transaction, int volume);
//
// Parameter list:
// transaction   buy, sell or pass (TT_BUY, TT_SELL or TT_PASS)
// volume        how many components to buy or sell (ignored for TT_PASS)
//
// Return values:
// TE_OKAY          transaction succeeded (no error)
// TE_NOT_CONNECTED you weren't trading anyway
// TE_NO_FUNDS      insufficient funds to buy that many
// TE_NO_STOCK      insufficient stock for sale
// TE_TOO_LATE      you have already made a transaction today
```

4. **Close trading account**

```
int TE_CloseTrading(float &finalbalance, int &finalstock);
//
// Parameter list:
// finalbalance       balance at close of trading (passed by reference)
// finalstock         number of components held at close of trading (by reference)
//
// Return values:
// TE_OKAY            balance and stock are valid
// TE_NOT_CONNECTED   you weren't trading anyway
```

Note that the parameters which are passed by reference (indicated by the symbol `&` before the variable name in the function header) will have their values assigned or changed by the library function. For example, to get the price on `day=1` we can call the `TE_GetPrice()` function:

```
ec = TE_GetPrice(day, price);
```

and check that the return value of the function is `ec == TE_OKAY`.

8.2 Sample function to initialise trading account and handle exit code.

An example of the implementation of a function to initialise the trading account and to establish a connection with the exchange is given below and listed in the file `OurTradingFunctions2.cc`.

The function is passed a data structure of type `TradingAccount`. The `today` data member of the data structure is used to record the trading day (e.g. on the first trading day `today = 1`). The other fields are used to record the price, transaction type (buy, sell or pass), number of components bought/sold, stock at the end of each transaction and the balance after each transaction. Trading can take place from `day=1` to `day=50`.

The exit code (return value) of the library function is checked and an appropriate error message is produced if a connection was not established. The system library function, `exit()`, is used to terminate the program instead of the conventional `return` from the main function. This forces the program to terminate after detecting an error in one of the function calls.

The symbolic constants (integers) (e.g. `TE_OKAY`, `TT_PASS`) are defined in the following sub-section using enumerations.

```
// Function to initialise trading account and connect to exchange
// Structure is passed by reference
void InitialiseAccount(int dataSet, int labGroup, TradingAccount &account)
{
   int ec, day= 0;

   // Set data members of TradingAccount structure for day = 0
   account.today = day;
   account.price[day] = 20.0;
   account.transaction[day] = TT_PASS;
   account.volume[day] = 0;
   account.stock[day] = 0;
   account.balance[day] = 20000.0;

   // Call trading library function and handle exit (return) code
   ec = TE_InitialiseTrading(dataSet, labGroup);
   HandleExitCode(ec);
}
```

```
// Function definition to display error messages from trading exchange
// The program is terminated with exit(-1) if there is an error
void HandleExitCode(int errorCode)
{
   switch(errorCode)
   {
      case TE_OKAY:
         break;

      case TE_FAIL:
         cout << "Trading error: bad parameter." << endl;
         exit(-1);

      case TE_ALREADY_CONNECTED:
         cout << "Trading error: already connected to server." << endl;
         exit(-1);

      case TE_NOT_CONNECTED:
         cout << "Trading error: not connected to server." << endl;
         exit(-1);

      case TE_AUTH_FAILED:
         cout << "Trading error: authorisation failed." << endl;
         exit(-1);

      case TE_TOO_EARLY:
         cout << "Trading error: must agree price every day." << endl;
         exit(-1);

      case TE_TOO_LATE:
         cout << "Trading error: transaction already made for today." << endl;
         exit(-1);

      case TE_NO_FUNDS:
         cout << "Trading error: insufficient funds for purchase." << endl;
         exit(-1);

      case TE_NO_STOCK:
         cout << "Trading error: insufficient stock for sale." << endl;
         exit(-1);

      default:
         cout << "Trading error: trading system failure." << endl;
         exit(-1);
   }
}
```

8.3 MyTrader Function

For each day of the trading period (from `day=1` to `day=50`) you are required to:

1. Get today's price by calling the trading library function `TE_GetPrice()`. The quoted price is passed back by reference.
2. Determine whether to buy, sell or not trade at this price (i.e. transaction-type) and the volume of trading. This should be implemented with a call to a function which will return the transaction type and the volume of trading. Begin with a simple strategy to ensure the purchase of 1000 components.
3. Perform the transaction by calling the trading library function `TE_Trade()`. The transaction must be one of `TT_SELL`, `TT_PASS` or `TT_BUY`. The volume must be an integer.
4. Record the transaction by setting the price, transaction type, volume, stock and balance entries in the `account` data structure.

This can be done by defining a function, for example `MyTrader()` below, which has additional function calls.

```
void MyTrader(TradingAccount &account)
{
     // Call library function to get today's price
     // Call user-defined function to decide on transaction and volume
     // Call library function to trade
     // Call user-defined function to update accounts
}
```

8.4 Closing the account and writing the records to a file

At the end of the trading period (on day 50) you should close the trading account by calling the `TE_CloseTrading()` library function. The final balance and final stock recorded by the electronic trading exchange will be returned by reference. You must compare these values with the balance and stock entries for day 50 in your own records.

Call a function to record the accounts in a file (i.e. write a file, see lecture notes or section 11.6 of the Tutorial Guide). Include price, transaction, volume, stock and balance details for each day.

8.5 Definition of trading library constants using enumeration

The following symbolic constants are used in the electronic trading library functions. They have already been defined for you using **enumeration** statements. These definitions are found in the header file `trading.h`.

1. Trading error codes

   ```
   enum TrError {
      TE_OKAY = 0,
      TE_FAIL,
      TE_ALREADY_CONNECTED,
      TE_NOT_CONNECTED,
      TE_AUTH_FAILED,
      TE_TOO_EARLY,
      TE_TOO_LATE,
      TE_NO_FUNDS,
      TE_NO_STOCK,
      TE_NUM_ERROR_CODES
   };
   ```

   The enumeration is simply assigning integer values (0 to 9) to the symbolic constants (`TE_OKAY, ....TE_NUM_ERROR_CODES`). You should use the symbolic constant names in your programs to make them more readable.

2. Training data sets

   ```
   enum TrDataSet {
      TEST1 = 1,
      TEST2,
      TEST3,
      TEST4,
      TEST5,
      TEST6,
      TEST7,
      TEST8,
      TEST9,
      TEST10,
      FINAL = -1
   };
   ```

3. Transaction types

   ```
   enum TrTransType {
      TT_SELL = -1,
      TT_PASS =  0,
      TT_BUY  =  1
   };
   ```

9 Integration, Final Testing and Evaluation

Integration of Part 1 and 2

The final program should integrate all the tested and working functions. When the functions have been successfully tested the source file should be copied (*shipped*) to the team directory, `1ASoftwareDesign`. The two files containing the working functions (`OurTradingFunctions1.cc` and `OurTradingFunctions2.cc`) can be copied into the team directory (`1ASoftwareDesign`) by dragging the program file icon in the file manager window (e.g. `OurTradingFunctions1.cc`) onto the `ship` icon in the applications window.

The `ship` command delivers a copy of the file to the group directory where it can be read and rewritten by any member of the team. You will also have to `ship` the `OurTradingHeader.h` header file (after adding a complete list of function prototypes) and the trading data file. You will have to create a new, integrated version of `OurTradingMain.cc` to call all of the functions of the program.

Final Testing and Evaluation

10 sets of sample/training data are available for testing by calling `TE_InitialiseTrading()` with the `dataSet` parameter set to one of `TEST1, ...TEST10`. Final testing will be on unseen data and will only be available once for the final evaluation.

Record the final balance and average price obtained with each data set. In the final laboratory session it will be executed on unseen data (with the same statistical characteristics as the test data and historical data) with the `dataSet` parameter set to `FINAL`.

The performance of different groups on the same data will be compared. Four additional marks will be awarded for the quality of the solution (including simplicity and clarity), evidence of teamwork and the performance during evaluation on the unseen data and on the test data.

10 Further Reading

The following references provide a comprehensive treatment of the C++ Programming Language and useful tips on good programming practice.

1. *C++ How to Program*, Deitel, H.M and Deitel, P.J.
   Prentice Hall, Englewood (NJ), 1994.

2. *Code Complete: a Practical Handbook of Software Construction*, McConnell, S.
   Microsoft Press, 1993.