

G64OOS Lab Manual

Spring 2014 (revision 1.06 # 31/03/2014)

Author: Peer-Olaf Siebers

(Based on previous lab companions which were co-authored with Michel Valstar)

1. Introduction

This lab manual fulfils two purposes. It is a companion guide to the taught lab sessions of G64OOS Object Oriented Systems as well as the detailed description of the assessed coursework. The lab companion guide provides a short description of the topics addressed in each lab session, references to relevant sections in the self-study books listed on the G64OOS website, example code, extra exercises, and links to additional resources on the net. The assessed coursework section explains in great detail what you need to do for your coursework, and how the coursework will be graded. The coursework requires you to design and code a medium sized program in C++ to demonstrate the understanding of object oriented programming principles.

2. Lab Companion Guide

The lab companion guide provides a description of what we will cover each week in the lab. It does not cover everything we will discuss during the actual lab sessions and therefore working through the companion guide at home is not a suitable substitute for attending the lab sessions.

The questions and exercises often require you to go beyond the material discussed during the lectures. For these cases please remember: The internet is your friend. Please try to find the answers to questions online first before asking a lab assistant or lecturer. A particular good resource for this purpose is: www.cplusplus.com and www.codeproject.com.

Lab 1: Introduction to C++ - Hello World; Operators; Functions; Class

This week's lab work is intended to get you started writing some code. Throughout the labs we will use either Visual Studio or Code::Blocks for writing the C++ code. You are free to choose your preferred IDE but we might not be able to provide you with any technical support on other IDE's. There is a free version of Visual Studio available online that you can download for your self-studies at home. Code::Blocks is an open source cross-platform lightweight IDE and freely available online. Both programs are installed on the A32 lab machines.

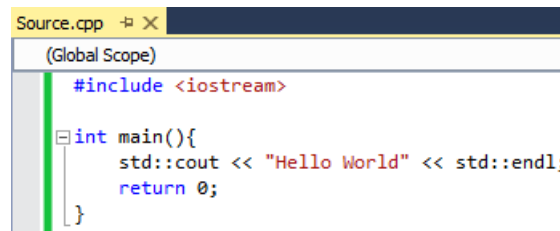
Let's have a look at visual Studio. The first step in writing a Visual C++ program is to create a project for it. Choose [FILE > New > Project] from the main menu bar. Here you have the choice of different project types. All examples in the lab manual are Win32 console applications. So you could choose Win32 Console Application, click [OK] and then [Finish] in the next window. This creates the project and a default class file. But there is a catch! The project contains a certain amount of excess baggage that you don't need when working with simple C++ language examples like in this module. The precompiled headers option chosen by default resulted in the "stdafx.h" file being created in the project. This is a mechanism for making the compilation process more efficient when there are a lot of files in a program but won't be necessary for our examples. Furthermore, by default the project options will be set to use "unicode libraries" which results in a non-standard name "_tmain" for the main function in the program.

In order to avoid the extra baggage and write standard native C++ code I would propose to always create an empty Win32 Console Application project and change the character set. Here is what you have to do. Start Visual Studio. Choose [FILE > New > Project] from the menu bar. In the appearing {New Project} window choose [Installed > Templates > Visual C++] on the left and pick [Win32 Console Application] on the right. Provide a name for the project at the bottom of the window and click [OK]. In the appearing {Win32 Application Wizard} window choose [Application Settings] on the left and then activate [Empty project] under [Additional options] on the right. Click [Finish] and your project will be created. Now we change the character set. Choose [PROJECT > Properties] from the main menu bar. In the appearing pop-up window choose [Configuration Properties > Character Set] and pick [Not Set] from the pull down menu that appears on the right. That's it. You now have an empty project folder set up for write standard native C++ code and you can start adding classes.

IMPORTANT: When you create projects or save files, please do not use the network drive as your destination. There are known issues with this. Instead use the temporary folder on the local hard drive (c:\temp) or your USB stick.

"Hello World" in Visual Studio

Let's start developing our first application. It is of course the famous "Hello World" program. Create an empty Win32 Console Application project as described above with the name "Listing1.1". Then choose [Project > Add New Item...] and in the appearing {Add New Item} window choose [Installed > Visual C++] on the left and pick [C++ File (.cpp)] on the right. At the bottom of the window leave "Source.cpp" as filename and click [Add]. This will open an empty class in the EDITOR window. Add the following code to the class.



```

Source.cpp
(Global Scope)
#include <iostream>

int main(){
    std::cout << "Hello World" << std::endl;
    return 0;
}

```

Listing 1.1

The easiest way to compile and run your application is to choose [DEBUG > Start without debugging] from the main menu or use the shortcut (Ctrl+F5). This will build and run your application. If you have made changes to the code since the last time you run your application you will be asked if you would like to build it. Click [Yes] and pick [Do not show this dialog again].

Notice the main() function. This is the function that your machine always calls first upon running your program. Every program must have exactly ONE main function and this function must return an integer. It can have input parameters which are normally used to pass command line arguments to the program. For a good tutorial on this topic see www.cplusplus.com/forum/articles/13355/.

Exercise 1.1: In Visual C++ there are two different sets of options available: options that apply to the tools provided by Visual C++ ([TOOLS > Options...]) and options that apply to an individual project ([PROJECT > \$project properties...]). Have a look through both sets of options. Find out how to change the default path for projects. Find out how to show line numbers in the editor.

Exercise 1.2: Extend the "hello world" example to ask for a user's name. Provide a means for the user to input their name using "cin". After the user gives their name print the text "Hello \$user!" where \$user is the name provided using "cin". Search the internet for more information about "cin".

"Hello World" in Code::Blocks

Creating the Hello World program in Code::Blocks is quite easy. When you run Code::Blocks for the first time you will have to select a compiler. If you are using windows choose "MinGW" or "GNU GCC" Compiler. If you are on Mac or Linux "g++" is your best choice. You can always change your compiler later through [Settings > Compiler...]. The Code::Blocks installation page provides more information on this topic.

To create a new project open Code::Blocks and click on [File > New > Project...]. Choose "Console Application" in the appearing pop-up window and click [Go]. Make it a "C++" application in the next pop-up window and then provide a name for your project in the next pop-up window. When you provide a name Code::Blocks automatically fills in the other fields in the window. Make sure that they are all filled in correctly before pressing the [Next] button. In the next pop-up window you do not have to change anything. Just press [Finish] and your new project becomes visible in the project browser at the left.

If you browse the "Sources" folder in your project browser you will find a file named "main.cpp". Double-click it and it opens it in the editor pane. It should show the "Hello World" source code. Whenever you create a new project in this way the "Hello World" code will appear. You can change this later in the [Settings > Editor... > Default Code].

Now you have two different options: You can "build" it and then "run" it (using the first two buttons of the panel shown in Figure 1.1) or you can "build and run" it (using the third button of the panel

shown in Figure 1.1). The last button in Figure 1.1 allows you to "rebuild" a project. Make sure that when you have made changes to your source code that you compile the code before running it. I would recommend to always use the third button "build and run" to avoid mistakes.



Figure 1.1

NB: It is possible to run single C++ files without creating a project. Close all open projects. Then drag the C++ file(s) you want to run into the project browser pane on the left. The file will open up in the editor pane and you can build and run it. If you have multiple C++ files open in the editor you can choose which one you want to compile by clicking on the tab at the top of the editor window. Although this is an option I would recommend to always creating a project and past the content of the provided C++ file(s) into the "main.cpp" file.

Operators

Let's do some maths. Create a new project with the name "Listing1.2" and add the code below.

```
Source.cpp  [X]
(Global Scope)
1  #include <iostream>
2  using namespace std;
3
4  // Compute point on a line (b=angle, c=offset)
5  int main(){
6      float b = 5, c = 3, x, y;
7      cout << "give value for x: ";
8      cin >> x;
9      cout << "corresponding value on the line " << b << "x+" << c << " is ";
10     y = b*x + c;
11     cout << y << endl;
12     return 0;
13 }
```

Listing 1.2

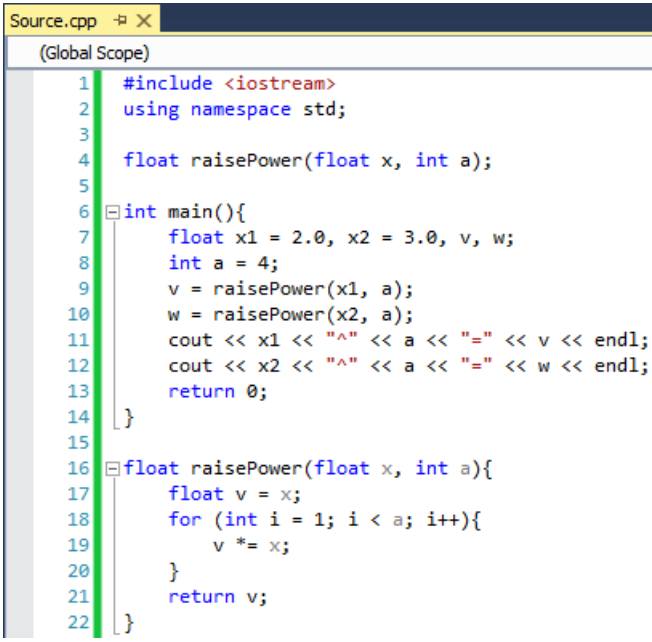
Note that the full namespace declaration of "cout" would be "std::cout" but this time we left out the "std::" part of the statements. We can do this as in line 3 we declare to use that namespace as a default. This often saves us a lot of work typing in the explicit namespaces that functions and classes originate from but it generates the risk of ambiguity if the same class or function name is used in two different namespaces. In general it is safe to use the standard library "std" namespace but more care should be taken with third-party libraries. See the www.cplusplus.com tutorial on namespaces for more information.

Exercise 1.3: Extend the program to compute the y-value of a quadratic function $ax^2 + bx + c$.

Functions

Using functions is usually your first taste of code re-use and encapsulation of functionality. But note that it does not allow for encapsulation of data. Unless you make use of unsavoury tricks, all data generated within a function is limited to the scope of that function. It is thus only a partial step towards full encapsulation.

In Exercise 1.3 you had to compute the square of a number by doing something like "int y = x*x;". If you would like to raise x to the power of 3 you would have to do something like "int y = x*x*x;" as the C++ list of operators is limited to "+", "-", "*", "/", and "%" (for addition, subtraction, multiplication, division, and computing the remainder of a division). Note that there is no operator for raising powers. Yet writing powers the way we showed above is very tedious, and would change depending on the power to which we raise it. This is an ideal example of where using a function would be very helpful. Consider the code in Listing 1.3. In line 4 we declare the prototype of our function raisePower() which takes a floating point base number x and raises it to the power of a, where a is an integer. The actual definition of the function is given after the main() function in lines 16-22. The declaration of raisePower() has to appear in the code before its use in the main() function. Yet for clarity, we choose to put the implementation details at the end of the listing. Later you will learn to do this in a more structured manner, where you keep separate header files that contain all class and function prototypes.



```

Source.cpp
(Global Scope)
1  #include <iostream>
2  using namespace std;
3
4  float raisePower(float x, int a);
5
6  int main(){
7      float x1 = 2.0, x2 = 3.0, v, w;
8      int a = 4;
9      v = raisePower(x1, a);
10     w = raisePower(x2, a);
11     cout << x1 << "^" << a << "=" << v << endl;
12     cout << x2 << "^" << a << "=" << w << endl;
13     return 0;
14 }
15
16 float raisePower(float x, int a){
17     float v = x;
18     for (int i = 1; i < a; i++){
19         v *= x;
20     }
21     return v;
22 }

```

Listing 1.3

Note the use of the mixed assignment/operator "!=" in line 19. This multiplies the value of v by x and assigns the result to v. All operators can be combined with assignment in this way. Try to run this code.

Question 1.1: What happens if you provide a negative value for the power? What if you provide 0? Hint: Look at the way v and i are initialised. Could you get the same or better results by initialising them differently?

Exercise 1.4: Create a new project with the name "Listing1.3" and add the code shown in Listing 1.3. Fix the raisePower() code so it can deal with negative powers and 0.

Exercise 1.5: Create a new project with the name "Exercis1.5". Copy the code from Listing 1.2 and rewrite the quadratic equation solver so it has a function with prototype "float quadratic(float a, float b, float c, float x);". The function should in turn use your new function raisePower().

Classes

We mentioned in the previous section that functions only partly cover encapsulation as they do not store any data. Classes on the other hand can store data and thus give us the opportunity to deliver full encapsulation. Consider the implementation of a class shown in Listing 1.4 that generates quadratic equations. By default all members of a class are considered private and are thus not directly accessible by non-member functions. This means that the "private:" declaration in line 7 is redundant. We leave it there for clarity purposes only.

```
Source.cpp -# X
(Global Scope)
1  #include <iostream>
2  using namespace std;
3
4  float raisePower(float x, int a);
5
6  class Quadratic{
7  private:
8      float a, b, c; // defining the quadratic function  $ax^2+bx+c$ 
9  public:
10     Quadratic(float x, float y, float z); // constructor
11     void set_a(float x);
12     void set_b(float x);
13     void set_c(float x);
14     float compute(float x) const;
15 };
16
17 Quadratic::Quadratic(float x, float y, float z){
18     a = x;
19     b = y;
20     c = z;
21 }
22
23 void Quadratic::set_a(float x){a = x;}
24
25 void Quadratic::set_b(float x){b = x;}
26
27 void Quadratic::set_c(float x){c = x;}
28
29 float Quadratic::compute(float x) const {
30     return a*raisePower(x, 2) + b*x + c;
31 }
32
33 float raisePower(float x, int a) {
34     float v = x;
35     for (int i = 1; i < a; i++){
36         v *= x;
37     }
38     return v;
39 }
40
41 int main() {
42     Quadratic Q = Quadratic(1, 2, 3);
43     float x = 0;
44     cout << "Give value for x: ";
45     cin >> x;
46     float y = Q.compute(x);
47     cout << "Corresponding value of quadratic function is: " << y << endl;
48 }
```

Listing 1.4

Line 14 shows the use of the so-called const correctness best practice. It declares that the function compute() does not change any data within the class. Because compute() is never intended to do so the use of the "const" keyword will instruct the compiler to throw an error when we inadvertently do change data in our implementation of compute(). Thus, using const correctness diligently reduces

the bugs in our software and improves maintainability. Const correctness can take some time to get used to. For many useful tips on coding standards (including const correctness) have a look at the following website: www.possibility.com/Cpp/CppCodingStandard.html.

Question 1.2: Why can you not make the function `set_a()` const?

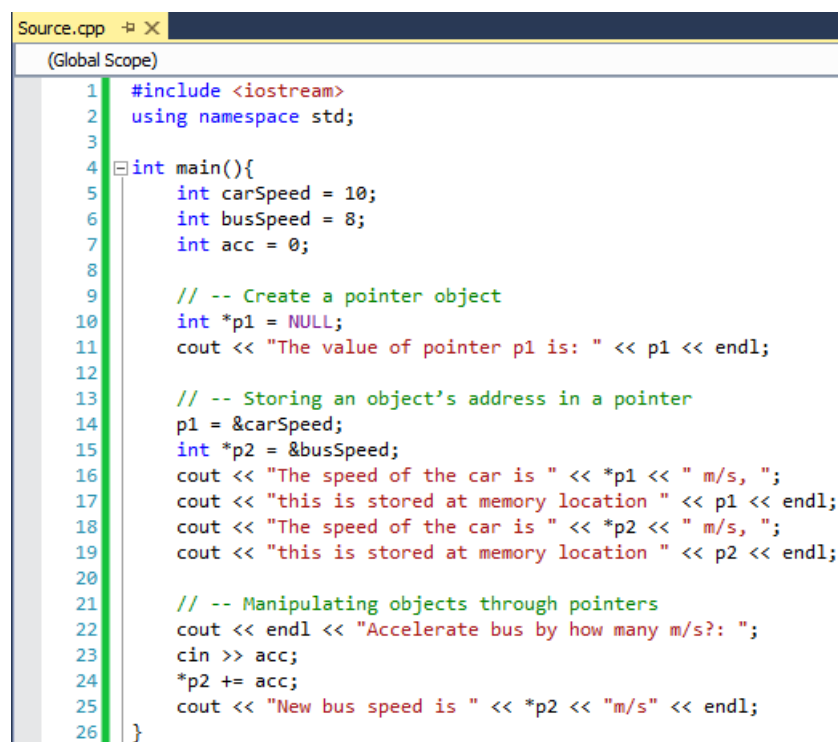
Question 1.3: Why would it not make sense to make `raisePower()` a member function of the Quadratic class?

Lab 2: Introduction to C++ - Pointers and Arrays

This week's lab is designed to help you understand two important aspects of C++: pointers and inheritance. A pointer is used to directly address and access a computer's memory. As such it is an incredibly powerful tool, but because it works with the machine at such a low level, it can be complicated to use. Still, having pointers allows you to create fast and small programs, and it is for this reason that it is still the industry standard in areas that require a lot of fast data manipulation, such as computer vision or machine learning.

Pointers

Ever wondered exactly where in memory a variable, object, or function was stored? In C++ you can know exactly that by using pointers. Pointers allow direct access to a memory location, and form an alternative way of addressing a variable. A very good tutorial about pointers can be found at the following web address: www.cplusplus.com/doc/tutorial/pointers/.



```
Source.cpp
(Global Scope)
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int carSpeed = 10;
6      int busSpeed = 8;
7      int acc = 0;
8
9      // -- Create a pointer object
10     int *p1 = NULL;
11     cout << "The value of pointer p1 is: " << p1 << endl;
12
13     // -- Storing an object's address in a pointer
14     p1 = &carSpeed;
15     int *p2 = &busSpeed;
16     cout << "The speed of the car is " << *p1 << " m/s, ";
17     cout << "this is stored at memory location " << p1 << endl;
18     cout << "The speed of the car is " << *p2 << " m/s, ";
19     cout << "this is stored at memory location " << p2 << endl;
20
21     // -- Manipulating objects through pointers
22     cout << endl << "Accelerate bus by how many m/s?: ";
23     cin >> acc;
24     *p2 += acc;
25     cout << "New bus speed is " << *p2 << "m/s" << endl;
26 }
```

Listing 2.1: Pointers

Now consider Listing 2.1. Notice what happened to the actual value of p1. The first time it is printed (line 11) the value is 0. The second time it is printed (line 17) it has a very different value. The exact value depends on the state of your machine at the time of execution.

Exercise 2.1: Check that p1 holds the same address as the variable carSpeed by directly accessing the address of carSpeed using the "address of" operator "&".

Exercise 2.2: Check that busSpeed indeed holds the new value printed in line 25 by printing it directly instead of via pointer p2.

Question 2.1: Why can't you tell beforehand what the values of p1 and p2 are, unless you set them to 0 or NULL?

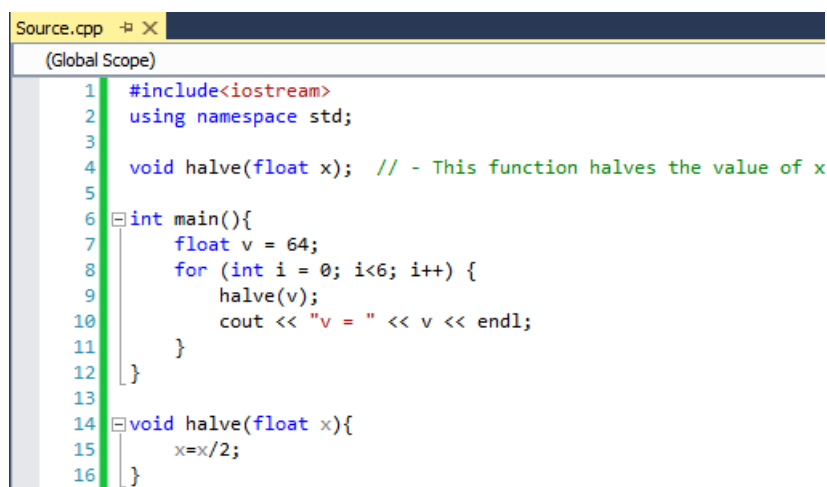
Question 2.2: What could happen if you do not initialise a pointer to 0 or NULL?

Question 2.3: What is the difference between setting a pointer to 0 or NULL?

Parameter Passing

One of the reasons why pointers exist is so you can pass arguments by reference instead of by value, the latter being the default way.

Exercise 2.3: Look at Listing 2.2 below. What do you expect will be printed on standard output? Now enter and run the code. Chances are that the output is not at all what you expected, and even if you did expect this output, it probably isn't what the poor programmer who wrote this code intended to happen.



```
1  #include<iostream>
2  using namespace std;
3
4  void halve(float x); // - This function halves the value of x
5
6  int main(){
7      float v = 64;
8      for (int i = 0; i<6; i++) {
9          halve(v);
10         cout << "v = " << v << endl;
11     }
12 }
13
14 void halve(float x){
15     x=x/2;
16 }
```

Listing 2.2: Parameter passing

Exercise 2.4: Fix the code of Listing 2.2 to let function `halve()` modify the value of `x` as intended. Do this by changing the function to take a pointer to a variable instead of the actual variable.

Exercise 2.5: Do the same as in Exercise 2.4 but use references instead of pointers (hint: a reference to `x` would be `&x`).

Question 2.4: What's the difference between a reference to a variable and a pointer to a variable?

Question 2.5: The function using references looks much cleaner. Why would you ever want to use pointers?

Arrays

Arrays are contiguous portions in memory, sequentially storing a number of elements of the same type. The number of elements to store has to be known in advance (if you don't know this, you probably want to use Vectors or another container class from the standard library).

```

Source.cpp
(Global Scope)
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      const int n = 5;
6      int odd[n];
7      for (int i = 0; i<n; i++){
8          cout << odd[i] << endl;
9      }
10     cout << endl;
11     int even[n] = { 0, 2, 4, 6, 8 };
12     for (int i = 0; i<n; i++){
13         cout << even[i] << endl;
14     }
15 }

```

Listing 2.3: Arrays

Question 2.6: Consider Listing 2.3. What's the difference in output of the arrays odd and even?

Question 2.7: What happens if you write a value in odd[5]?

Question 2.8: How can you extend an array?

Exercise 2.6: Write a function that asks for 7 inputs, and prints all odd numbers, and returns all input numbers that were odd in a new array.

Exercise 2.7: Write a function that takes an array as input, and returns the sum of all the values

Exercise 2.8: Combine the code of the previous two exercises to create a new function that asks for 7 inputs, prints all odd numbers, their sum, and returns the array of odd numbers. Advanced option: modify this function so that instead of always requesting 7 inputs, it first asks how many numbers to input.

Lab 3: UML Practice - Object Oriented Analysis and Design Group Activity (1/2)

Your task: Development of a Greenhouse Appliances Control System Proposal

You are working for a software company that wants to bid for a "Greenhouse Appliances Control System" (GACS) development contract. The goal of such a system is to maximise plant growth and plant quality while minimising production costs. Currently the greenhouse keeper controls everything manually. In future the new system is supposed to control all technical functions: heating, water, light, plant feeding, and ventilation.

For the initial bid you are asked to build a GACS for light and one other mechanism that supports plant growth (from the above list) to allow the greenhouse keeper to control the quality of the plants and the costs of growth. The software you are going to develop should be easy to maintain and extend. Therefore an Object Oriented approach to system Analysis and Design (OOA/D) seems to be the right choice. After an initial discussion with the greenhouse keeper you and your colleagues start working on the initial design. At the end your boss expects a presentation that allows him to market your design and win the contract to implement the system.



Your presentation should include at least the following technical elements:

- Use case diagram + a top level use case specification (including base and alternative paths)
- CRC cards (use the internet to find out what they are and how they are used)
- Class diagram
 - Classes including attributes and operations
 - Relationships
 - Multiplicity indicators
- Sequence diagrams (perhaps you want to break sequences down into sub diagrams)

For creating the UML diagrams you can either use pen and paper (and scan in your results) or Visual Studio (if you are able to save your diagrams on "c:\temp") or Visual Paradigm (see Appendix A for the registration process). Visual Paradigm also offers a community edition free of charge.

A guide for using Visual Studio's UML capabilities is available here:

<http://msdn.microsoft.com/en-us/library/dd409445.aspx>

A quick guide for using Visual Paradigm is available here:

http://d1dlalugb0z2hd.cloudfront.net/quickstart/quickstart_vpuml.pdf

A good source of information for any kind of queries about UML diagrams is:

<http://www.uml-diagrams.org/>

Homework 3.1: Work through the Visual Studio or Visual Paradigm guides. You might also find some useful information on YouTube.

Lab 4: UML Practice - Object Oriented Analysis and Design Group Activity (2/2)

Use the first 40 minutes to finish preparing your presentation. After that each group has a 10 minute slot to present their ideas and receive some feedback from their peers (7 minutes for presentation and 3 minutes for feedback).

At the end of Lab please email me a copy of your group presentation.

Lab 5: More Advanced C++ Concepts – Inheritance and Polymorphism

I am sure you agree that understanding how to correctly apply inheritance and polymorphism in C++ seems to be quite difficult. I believe that the best way to learn such things is by experimenting with existing code and seeing the consequences of small changes or extensions. This works best when you do it in small groups (perhaps with your neighbour).

Exercise 5.1: Download the slides from Lecture 5 together with the source code from the module website. Study the slides and programs together. Make small changes to the programs or extend them and discuss the consequences of these changes with your neighbour. If your code does not compile have a look at the error message and try to understand from this information what is going wrong. Then have a look on the internet to find a solution to the problems.

Question 5.1: What happens if you remove the "public" keyword located in front of the base class when defining inheritance (i.e. "class Derived: public Base {};" becomes "class Derived: Base {};"). Use the internet to help answering this question.

Inheritance

A good example of inheritance is the idea of shapes. A rectangle is a shape, and so is a circle. A square could be defined as a special type of rectangle, where the width is always equal to the height of the rectangle. Such shapes would probably share similar variables, such as their centre of mass, and you would probably define similar operations on them, such as `area()` or `circumference()`, but the way they would be implemented would be completely different. Figure 5.1 gives an example class diagram for this family of objects, and Listing 5.1 gives an example implementation of the base class `Shape` and the derived class `Rectangle`.

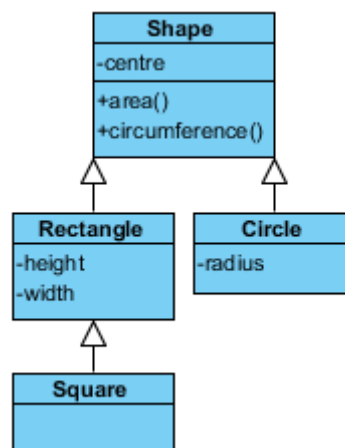


Figure 5.1: Class diagram for Shape class and derived classes

Exercise 5.2: Find some information about "abstract base classes" and "pure virtual functions" on the internet. A good source is <http://www.cplusplus.com/doc/tutorial/polymorphism/> (look at sub section "Abstract base classes").

Exercise 5.3: Implement the classes "Circle: public Shape" and "Square: public Rectangle".

Exercise 5.4: Try to create an object of class `Shape`. Can you? Think of implementing the functions `area()` and `circumference()`. How would you do that? Does all this make sense to you?

```

1  #include <iostream>
2  using namespace std;
3
4  class Shape {
5  protected:
6      int centre[2];
7  public:
8      // -- Constructor with Centre given, c must be a 2-element array
9      Shape(int c[]);
10     void moveTo(int d[]);
11     virtual float area()=0;
12     virtual float circumference()=0;
13 };
14
15 class Rectangle: public Shape {
16 protected:
17     int height;
18     int width;
19 public:
20     // -- Constructor with Centre and side given
21     Rectangle(int c[], int h, int w);
22 };
23
24 int main(){
25 }
26
27 Shape::Shape(int c[]){
28     centre[0] = c[0];
29     centre[1] = c[1];
30 }
31
32 void Shape::moveTo(int d[]){
33     centre[0] += d[0];
34     centre[1] += d[1];
35 }
36
37 Rectangle::Rectangle(int c[], int h, int w): Shape(c), height(h), width(w) {}

```

Listing 5.1: Code fragments for Shape class and derived classes

Question 5.2: Do you need to make both `area()` and `circumference()` pure virtual to make Shape an abstract base class? Does it make any difference whether you make both pure virtual?

Exercise 5.5: Implement the overriding of the (pure) virtual functions `area()` and `circumference()` for the classes Rectangle, Square, and Circle.

Exercise 5.6: Implement the function `max x()`, that returns the maximum x-value of the space occupied by a shape. For example, in case of a 4 by 4 square, centred at position $[x,y] = [1,1]$, `max x` would return the value 3. Think carefully in what class(es) this should be defined, and how.

Lab 6: Relationships

In this lab you start putting together a part of a real world application. Your consultancy has been winning a bid for developing an office building management system for a company that rents out office space with or without secretarial support. You are asked to work on a first prototype based on some code fragments that you have inherited from a colleague.

Here is the user story: The client owns offices in office buildings in different cities all over the world. Depending on their size offices can have a number of work spaces. Each work space is equipped with a desk, a chair, and a computer. Secretarial support for an office can also be booked. There can be several secretaries working in one office. A secretary might also work in different offices at different times of the day.

Exercise 6.1: Download the slides and the source code provided at the module website (the project files are for Code::Blocks) and have a closer look at them. Perhaps discuss it with your neighbour. Then start improving the source code and also implement the missing bits. Below you find some suggestions for how to improve it.

Composition

- Consider good coding practice (in particular const correctness and file organisation)
- Add names to your offices
- Create the objects on the heap
- Use vectors to store your objects
- ...

Aggregation

- Consider good coding practice (in particular const correctness and file organisation)
- Implement the aggregation as presented in the class diagram

Many-to-Many Relationship

- Consider good coding practice (in particular const correctness and file organisation)
- Currently associations can only be initiated from the "office object"; this should also be possible from the "secretary object"
- Improve reusability of code
- ...

Exercise 6.2: Put it all together in one program including a test class that demonstrates the joint functionality of the individual bits.

Lab 7: Release and Discussion of Coursework

For more details see Chapter 3.

Lab 8: Test Driven Development and Pair Programming

Today we practice Test Driven Development and Pair Programming. For this you need to find a colleague (choose someone next to you). We have to do this exercise in Visual Studio as there are some problems with Code::Blocks. If you like a challenge you can do everything from scratch as described below. If you like it smooth then simply download the zip file from the module website. It contains a compiled version of the framework as well as the project file for the test. You can then just use this as a base project for your own developments.

The Hard Way

Download UnitTest++ (<http://sourceforge.net/projects/unittest-cpp/files/>) and unzip it onto your local hard drive under "c:\temp\". In Visual Studio open "UnitTest++.vsnet2005.sln" which is inside "c:\temp\UnitTest++" and agree to the update and then build it. The output of this build will be "c:\temp\UnitTest++\Debug\UnitTest++.vsnet2005.lib" which you will need to reference later in your unit test application.

Now create an empty project named "TDDTest" (as described in the beginning of the lab manual) and add a C++ file. Then go to {PROJECT\TDDTest Properties...} and add the text below in the positions shown in Figures 8.1-8.3.

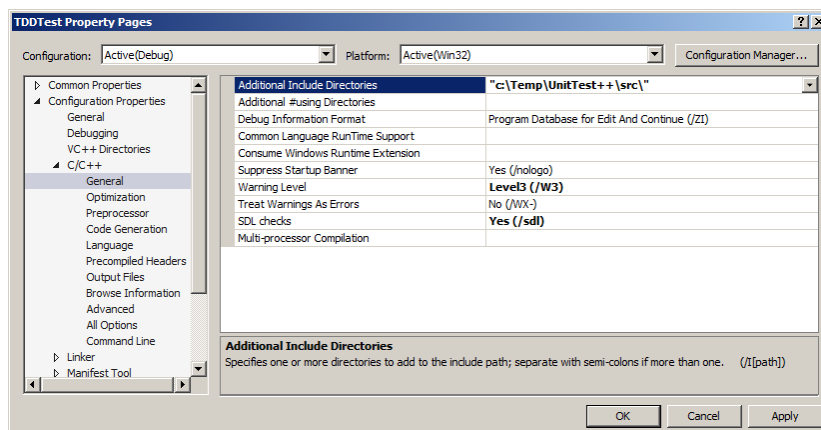


Figure 8.1: Text = "c:\Temp\UnitTest++\src\"

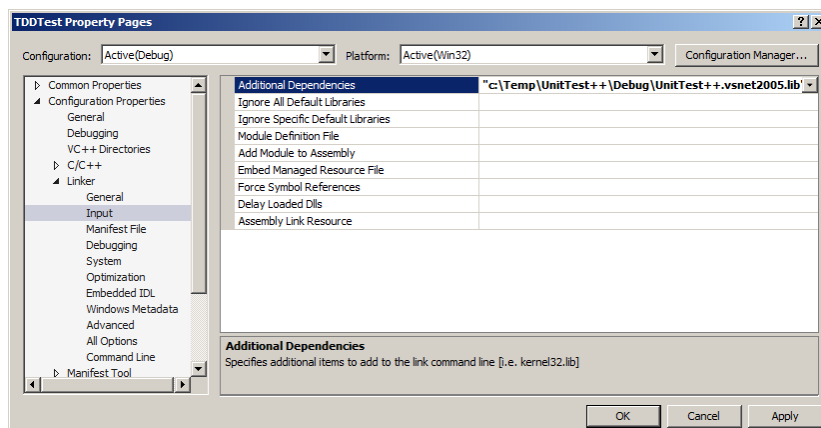


Figure 8.2: Text = "c:\Temp\UnitTest++\Debug\UnitTest++.vsnet2005.lib";

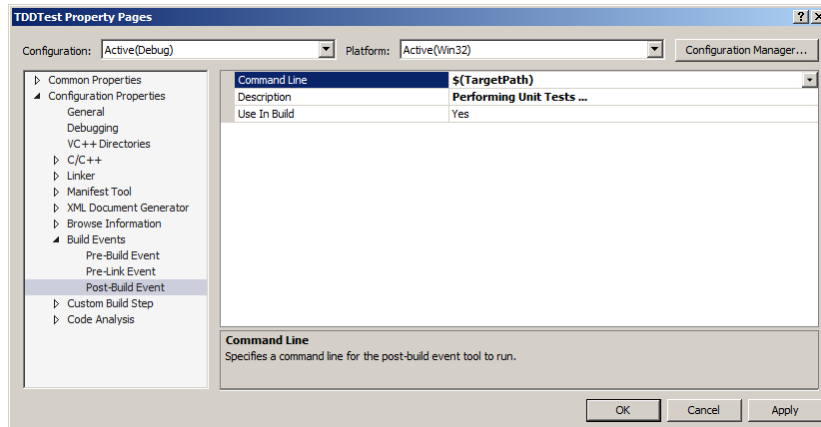


Figure 8.3: Text1 = \$(TargetPath) Text2 = Performing Unit Tests ...

Then try it out by writing and running the following source code

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

TEST(HelloUnitTestPP){
    CHECK(true);
}

int main(){
    return UnitTest::RunAllTests();
}
```

Now you can use this as a basis for the other exercises

The Easy Way

Simply download the zip file from the module website. It contains a compiled version of the framework as well as the project file for the test. You can then just use this as a base project for your own developments.

Pair Programming Exercises

Exercise 8.1: Implement the TDD example from Lecture 8. Here are some reminders:

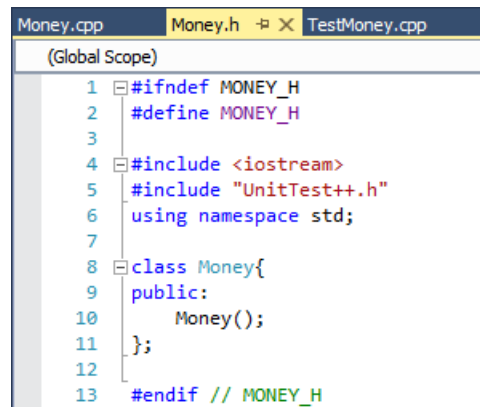
- Task: Create a function that checks if a given year is a leap year or not
- Logic: A year is a leap year when it can be divided by 4, unless it can be divided by 100. But in case it can be divided by 400, then again it is a leap year.
- How to get started ...
 - You need three files: LeapYear.h and LeapYear.cpp for the implementation and Source.cpp for the testing.
 - You need to think of a test that will help you in the design ...


```
TEST(OurFirstTest){
    const bool Result=isLeapYear(1972);
    CHECK_EQUAL(true, Result);
}
```

Only look at the slides from Lecture 8 if both of you get stuck!

Exercise 8.2: Read the "Money Tutorial" (http://unittest-cpp.sourceforge.net/money_tutorial/) and try to implement it but follow good coding practice (i.e. having separate header and implementation files). To get started quickly you can reuse the test project provided in the zip file on the module website. Then start working with the Money Tutorial where it says "Go ahead and permanently delete the Money.cpp implementation file" but do not follow this advice. Remember that you following "good coding practice". Use this tutorial only as inspiration for your own implementation.

Below are some screenshots (Figures 8.4-8.6) how your project should look like once it is prepared for the first test.

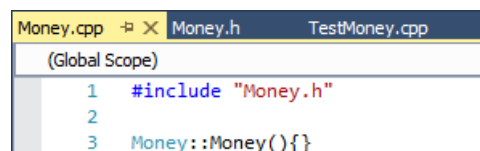


```

Money.cpp  Money.h  TestMoney.cpp
(Global Scope)
1  #ifndef MONEY_H
2  #define MONEY_H
3
4  #include <iostream>
5  #include "UnitTest++.h"
6  using namespace std;
7
8  class Money{
9  public:
10     Money();
11 };
12
13 #endif // MONEY_H

```

Figure 8.4: Money.h

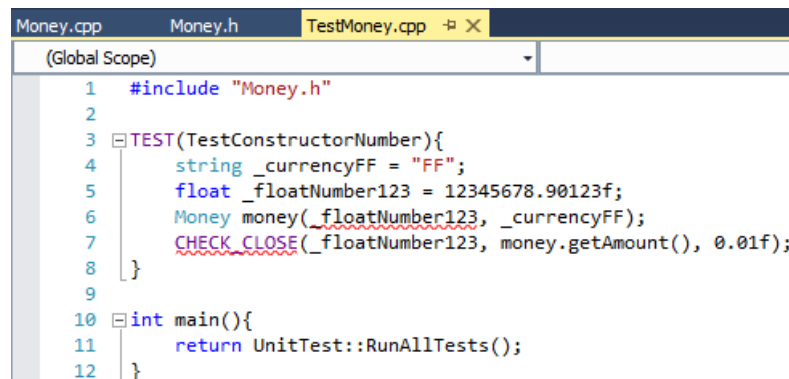


```

Money.cpp  Money.h  TestMoney.cpp
(Global Scope)
1  #include "Money.h"
2
3  Money::Money(){}

```

Figure 8.5: Money.cpp



```

Money.cpp  Money.h  TestMoney.cpp
(Global Scope)
1  #include "Money.h"
2
3  TEST(TestConstructorNumber){
4      string _currencyFF = "FF";
5      float _floatNumber123 = 12345678.90123f;
6      Money money(_floatNumber123, _currencyFF);
7      CHECK_CLOSE(_floatNumber123, money.getAmount(), 0.01f);
8  }
9
10 int main(){
11     return UnitTest::RunAllTests();
12 }

```

Figure 8.6: TestMoney.cpp (or Source.cpp if you do not want to rename anything)

Additional Exercises

Object containers

As you learned in Lecture 5 it is possible to mix objects of different classes in a container such as an array, as long as they have the same base class. An example of this is given in Listing 6.1.

```
1  #include <iostream>
2
3  int main() {
4      // -- Create some locations
5      int L0[2] = {0, 0};
6      int L1[2] = {1, 1};
7      int L2[2] = {3, 5};
8      // -- Create some shape objects
9      Rectangle *R1 = new Rectangle(L0, 2, 4);
10     Rectangle *R2 = new Rectangle(L1, 4, 2);
11     Circle *C = new Circle(L2, 1);
12     // -- Group the shape objects in an array
13     Shape* S[3] = {R1, C, R2};
14     // -- Tidy up
15     delete R1;
16     delete R2;
17     delete C;
18 }
```

Listing a.1: Possible implementation of main() for Listing 5.1

Exercise a.1: Write a (set of) function(s) that ask(s) for user input to create a number of shapes. It should first ask how many shapes the user wants to create, and then for each shape what type of shape, followed by asking the user to input the variables relevant to that shape type. You should return an array of Shape pointers.

- Hint 1: Look at slide 23 of Lecture 5 for inspiration.
- Hint 2: You may want to store the length of the array somewhere. How to do that? One way is to split the user input operation in two functions, first asking for the number of shapes, then the details of each shape. The second way would be to use pointers, but that is more complicated.

Exercise a.2: Write the function `void moveShapes(Shape* S, int* d)` that takes an array of shapes of different types and moves each by the same displacement `d`. Check that this works by printing the locations of the centres of all shapes in `S`.

Exercise a.3: Write the function `float mixedShapesArea(Shape* S)` that takes an array of shapes of different types and returns the sum of their individual areas.

Exercise a.4: Write the function `void sortShapesByArea(Shape* S, int n)` that sorts the elements in an array `S` of length `n` by area, with the smallest area first.

Exercise a.5: Write the function `void lineUpShapes(Shape* S, int n)` that lines up all shapes left to right, with the y-value of the centre of each shape having the same value, and the minimum x-value of each next shape being one greater than the maximum x-value of the previous shape. See Figure 6.1 for an example of this. The figure also contains triangles, which you don't need to implement. You will have to implement the member function `Shape::min x()` to do this.



Figure a.1: Number of 2-dimensional geodesic shapes on a line

Performing operations on sets of related objects

This week's lab will let you perform operations on sets of related objects. The general principle is to access the concrete objects through an interface defined by an Abstract Base Class (ABC). You will be practising good coding practices as well. Write all class and function definitions in a header file, all class and function implementations in a ".cpp" file, and write the main function of the program you create in a separate file called "main.cpp". You should also comment your code, and use const correctness.

You will be writing a component of a robot's AI that deals with sets of 3-dimensional wooden blocks. There are three types of blocks: cubes, cylinders, and cones. The blocks come in four colours: red, green, yellow, and blue. The robot is to perform tasks on a set of blocks such as sorting them, or picking the brightest coloured block.

Listing a.2 gives a first version of the abstract base class Shape3D. Create a new file called Shape3D.h and copy the code from Listing 6.2 into it.

```
1  #include <iostream>
2  using namespace std;
3  enum BlockColour{RED, GREEN, YELLOW, BLUE};
4
5  class Shape3D {
6  protected:
7      BlockColour colour;
8  public:
9      Shape3D(BlockColour c);
10     string getColour() const;
11     void setColour(BlockColour c);
12     virtual float volume() const = 0;
13 };
```

Listing a.2: Abstract base class Shape3D

Exercise a.6: Extend Shape3D.h to include definitions for the Cube, Cylinder, and Cone classes. Think about what the minimum set of variables is that each class needs. Next, create a new file called Shape3D.cpp. This is where you will implement the Shape3D family of classes.

Exercise a.7: Implement the constructors and methods for Shape3D, Cube, Cylinder, and Cone.

Exercise a.8: Extend Shape3D to include the function void print() const;. The function should output the type of an object, the colour, and the volume. Decide whether you should implement that as a single function of Shape3D, or as a polymorph function.

Exercise a.9: Write a (set of) function(s) that ask(s) for user input to create a number of shapes. It should ask for each shape what type of shape, its colour, followed by asking the user to input the variables relevant to that shape type. You should return either a vector or a list of Shape3D pointers. Look at slide 21 of Lecture 5 for inspiration. Adapt the code to suit your definition, if needed, and edit the main.cpp file you have to let the user create a set of blocks. Use either the <vector> or <list> container to store the blocks.

Exercise a.10: Add a void printBlockSet(vector<Shape3D*>) or void printBlockSet(list<Shape3D*>) function to the Shape3D.h. This function prints a set of blocks. Before printing each block, it states what position in the container it occupies. Implement the function in Shape3D.cpp and modify the main.cpp file to print the set of blocks provided by the user. We now want the robot to be able to sort the blocks in various ways. In particular, we want the robot to sort blocks by volume and by colour.

Exercise a.11: Write a function that sorts either a list of Shape3D pointers or a vector of Shape3D pointers by volume. Edit the main.cpp to print the Shape3D objects sorted by volume after a user provides details of the shapes.

Exercise a.12: Write a function that sorts either a list of Shape3D pointers or a vector of Shape3D pointers by the brightness of the colour. As a definition, we declare that yellow is the brightest colour, followed by red, then green, and finally blue. Edit the main.cpp to print the Shape3D objects sorted by colour after you have printed the list sorted by volume.

3. Individual Coursework

3.1 Synopsis

It is the year 2269 and Earth is too crowded. To date space travel has been limited to our own solar system but a recent invention allows entire space fleets to be transported with near-light speed velocity. A number of Earth's biggest corporations have decided to grasp this opportunity and colonise the closest known habitable planet "Gaia" which is a mere 33 light-years away from earth. A race is gathering pace. To keep competition fair the United Nations have specified a date on which all corporations will launch their fleets together and have set up rules to determine who will eventually own Gaia.

The design of such a fleet is complex: corporations have access to a variety of spaceships; each with different characteristics. At the core of the fleet there are the colonisation ships (CS) that carry the colonists. Then there are the solar sail ships (SSS) which provide power converted from starlight to the fleet, and medical ships that prevent disease. Protection from alien attacks is provided by the United Nations via their network of star bases (one of the most famous is Babylon 5). Therefore there is no need to have any military ships in the fleet.

Thanks to the recently invented faster-than-light (FTL) bubble ship, individual ships don't need their own FTL drive. There's a catch though: The heavier the fleet - the slower the bubble moves. The technology behind the FTL-bubble ship is kept secret by the United Nations Space Agency (UNSA) and UNSA is providing each corporation with a single FTL-bubble ship.

Under UN law - now universally accepted - war is illegal. To resolve rival corporations lay claim to the same planet they have to abide to a simple set of rules: If a planet is uninhabited (i.e. the corporation is the first to arrive) the corporation is free to colonise it. If a planet is inhabited the arriving corporation can colonise it only if it arrives with more colonists than are living on the planet upon arrival. When this happens the current inhabitants pack their bags and use the ships of the rival corporation that just claimed their planet to start searching for another place to live. No colonists of the arriving corporation are lost in this process. So - the more people in your fleet - the greater the chance you win the planet Gaia! There is a catch though: due to space limitations, a fleet's population cannot grow in transit. On the other hand, a population on a planet grows by 5% year on year.

3.2 Coursework

It is your task to implement a space fleet that adheres by the rules specified in sections 3.1 and 3.3, implementing ships with specifications provided in section 3.4, and an interface to the game simulator specified in section 3.5. You are largely free to implement this however you like as long as you adhere to the rules specified in this coursework.

A few constraints do apply: the program has to be written in iso-standard C++. You can only use standard library (STL) classes besides the classes you define yourself. The source code should compile on our machine, that is, it should compile on mersey using g++. To be very specific, the command 'g++ Fleet.cpp -o Fleet' should work on the linux machine mersey.cs.nott.ac.uk. You have to submit your coursework via Moodle in a single zip file with a filename that contains your name and student number. See section 3.6 for full details about the deliverables.

3.3 Game Rules

Below are the rules to which the game, and thus your code, must and will adhere:

Money: Each corporation (that's you, the student!) has the same fixed amount of money to spend on ships, to wit, 10,000.- UNP (United Nations Pounds). The price of ships is fixed and provided in section 3.4.

Energy: Ships require energy. All ships in your fleet must have power, or else the entire fleet won't move.

Disease: Exactly halfway during the space voyage a disease breaks out on one of the colony ships (randomly chosen). If the fleet does not have a medic ship all colonists on that ship die. The ship (now empty) will continue to travel with the fleet though and you should assume its weight is unmodified by the event.

Colonisation: As stated in the synopsis, what happens when a fleet arrives at Gaia is governed by two rules: if you are the first to arrive, your colonists settle on the planet and your colonisation population starts to grow. If you arrive and the planet is already inhabited, two things can happen: either your fleet is carrying more colonists than the current planet population, and you take over, settling down with all your colonists, or if you arrive with fewer colonists in your fleet than there are people on the planet, you lose.

Planet population growth: Every year the population of Gaia grows by 5%. This happens instantly, after exactly one year has passed.

Distance to planet: The planet is 33 light years away from Earth.

Fleet speed: The speed of the FTL-Bubble is determined as follows:

$$v = \frac{\alpha c}{\sqrt{w}}$$

where "v" is the fleet speed in meters per second, "c" is the speed of light, and "omega" is the fleet weight, i.e. the sum of the weights of all ships in your fleet. "alpha" is set to 10 for this coursework.

Winner: The corporation (student) populating Gaia in the end has won.

3.4 The Ships

From the synopsis and rules sections, you probably gather that there are quite a few ships involved in this game. You will have to implement them all. Below is a list of all ships, per category. It also lists for every ship the values for their respective attributes. **Note that you don't have to implement the Bubble Ship.**

Colony Ships: There are three different colonisation ships, each with different costs, weight, and energy consumption. The properties of each ship are listed in Table 3.1.

Name	Colonists	Cost	Weight	Energy Consumption
Ferry	100	500	10	5
Liner	250	1000	20	7
Cloud	750	2000	40	10

Table 3.1: Colony ship attributes

Solar Sail Ships: There are two types of Solar Sail Ships, the standard ship Radiant, and the massive Ebullient. The Ebullient is so big, that it can often provide energy for an entire fleet. See Table 3.2 for details.

Name	Energy Generation	Cost	Weight	Energy Consumption
Radiant	50	50	3	5
Ebullient	500	350	50	5

Table 3.2: Solar Sail Ship attributes

Medic Ship: There is only one type of Medical ship, called the Medic. It is very small and efficient but rather expensive. The benefit of this ship type is that if you choose to include one in your fleet, it will render your fleet immune to disease. The medic ship is **composed** of a hospital and a regeneration platform. Details of the ship are provided in Table 3.3.

Name	Cost	Weight	Energy Consumption
Medic	1000	1	1

Table 3.3: Medical Ship attributes

Supply Ship: There is only one type of supply ship, called the Supplier. Supply ships are very simple (they carry containers with the supplies for the travel period) and therefore relatively cheap. Each Colony ship requires a certain amount of Suppliers, depending on their own size. During the travel these are **aggregated** with the Colony Ship they belong to. The supply ships store containers with fluids (for example water and milk) and solids (for example rice and peas). Details of the ship are provided in Table 3.4.

Name	Cost	Weight	Required by colony ships	Energy Consumption
Supplier	100	2	Ferry: 1; Liner: 2; Cloud: 4	3

Table 3.3: Supply Ship attributes

3.5 Interface with Game Environment

You must implement the elements of the interface shown in Table 3.4 so that your fleet can interact with the simulation environment. **Please note that the interface might change slightly throughout the first half of the project. This is common if you code with multiple parties. It should be finalised for the second half of the project.** Besides the interface with the simulation environment, your code should implement the following functionality: (1) a user interface that lets a user compose a fleet, and returns a pointer to that fleet; call the function "Fleet* userInterfaceCreateFleet();" the user interface should be interactive, and use the keyboard and screen only; (2) functions that allow to save and load data; implement fleet persistence by creating a function that takes a (pointer to a) Fleet object and saves it to a fleet data file "void writeFleet(Fleet* f, string x)", as well as a function that takes a filename to a fleet data file and returns a (pointer to a) Fleet object "Fleet* readFleet(string x)". The format of the fleet data file is described in Listing 3.2 (p22).

Operation	Explanation
int Fleet::getWeight() const;	Returns cumulative weight of fleet
int Fleet::getEnergyConsumption() const;	Returns cumulative energy consumption of fleet
int Fleet::getColonistCount() const;	Returns cumulative colonist count of fleet
int Fleet::getCost() const;	Returns cumulative fleet cost
int Fleet::getEnergyProduction() const;	Returns cumulative energy production of fleet
bool Fleet::hasMedic() const;	Returns true if the fleet has a medic ship, false otherwise
string Fleet::getCorporationName() const;	Returns your chosen name of your corporation
vector<Ship*> Fleet::colonyShips() const;	Returns a vector with ship pointers of all ships that are a colony ship
vector<Ship*> Fleet::shipList() const;	Returns a vector with all ships in the fleet
int Ship::getEnergyConsumption() const;	Returns energy consumption of a ship
int Ship::getWeight() const;	Returns weight of a ship
int Ship::getCost() const;	Returns cost of a ship
string Ship::getTypeName() const;	Returns the ship type (e.g. Ferry or Radiant); make sure you use the correct spelling!
int ColonyShip::getColonistCount() const;	Returns nr of colonists of a ship
void ColonyShip::infect();	Infects a colony ship
bool ColonyShip::isInfected() const;	Returns "true" if the ship is infected with a disease, "false" otherwise
int SolarSailShip::getEnergyProduction const;	Returns energy production of Solar Sail Ship

Table 3.4: Interface

3.6 Deliverables

You need to hand in the following four files:

1. Fleet.h, containing the definition of your classes and functions.
2. Fleet.cpp, containing the implementation of your classes and functions.
3. \$studentid\$ fleet.dat, your fleet data file containing the fleet you want to enter in the competition
4. \$studentid\$ report.pdf, which is your report containing the class diagram and a description of your design as well as a list of class specifications

where \$studentid\$ is your student number.

The format of the fleet data file should be as follows: Each line contains the information of the number of ships of each type, by putting the ship type name first, followed by whitespace, followed by the number of ships of that type. Note that if you don't have any ships of a particular type, you may simply omit it from your fleet data file. See Listing 3.2 for an example of a small fleet.

Listing 3.2: Fleet Data Example

```
Ferry 3
Supplier 3
Medic 1
Radiant 1
```

The report should contain an image of the complete class diagram including all attributes and operations of the classes as well as the relationships between classes of your program. Make sure it includes all elements of the interface code you need to implement! With the class diagram should come a text with a description of the diagram providing sound explanations and justifications for design decisions made. Here you can also mention where you followed (beyond the basic) design principles or applied design patterns.

This should be followed by a list of class specifications following the template provided in Table 3.5. Please note that this is different to CRC cards. We want you to put the actual names of classes.

Class name:	
Base class:	
Derived class:	
Function:	
Variables	Description
Operations	Description

Table 3.5: Class specification template

Note that the Base class/Derived class fields should hold only the first generation up or down a class family tree. The Component of field should list classes that use this class as a component (usually as a member variable). The field Component classes on the other hand lists classes that are added as components or aggregates to this class. The Function: field should give a very short description of the function of this class (i.e. the reason for its existence). Variables and operations should have the same name as your variables and operations in the class diagram and in your C++ implementation, but you don't have to mention return types or whether they are public, private, or protected.

3.7 Submission Guidelines

The deadline for course work submission is the 09/04/2014 @ 4 pm. Late delivery of the coursework will result in a penalty of 5% per day. To reference other works please use the Harvard style. Please submit all your files via Moodle archived in a single zip file named "\$studentid\$ \$studentname\$.zip" where \$studentid\$ represents your student number and \$studentname\$ your name.

3.8 Plagiarism

No marks will be awarded for regurgitation of lecture notes or material from text books, or copying the code of your fellow students. You must complete this piece of work independently as an individual. Copying other students' work will incur severe penalties. Each submitted file will be automatically checked against every other file submitted using "TurnItIn" plagiarism checker.

3.9 Marking

The coursework will be marked on a number of things: correct use of Object Oriented Paradigms (e.g. encapsulation, inheritance, composition), non-technical quality of your code (e.g. symmetry, le organisation, whitespace, and comments), as well as technical quality of your code (e.g. efficiency of search/optimisation code, minimisation of data/functionality redundancy).

Below is a breakdown of exactly how marks will be assigned:

1. Documentation 30%

- Class diagram (INCLUDING RELATIONSHIPS) 10%
- Class specification (using the template provided in Table 3.5) 10%
- Description of Class diagram 10%

2. Implementation 70%

- Compilation 15%
- Adheres to Ship/Fleet specification 5%
- Adheres to Interface specification 10%
- Coding good practice 10%
- Consistency with class diagram 10%
- Code elegance 20%

Your code and fleet data file will be checked for correctness (e.g. that you didn't overspend on the fleet, or that all ships have power). If your fleet is not 'launch ready', you will not get full points for the Ship/Fleet specification.

The winner of the space race will be immortalised on the course's web-site.

3.10 Late Additions to the Coursework

Please check this section frequently for the latest notifications.

28/03/2014

The simulator class (v0.98) is now available on the module website. It still needs some testing!

Here are the final changes to the specs. To make the space race more interesting I have added an alien attack. During the alien attack a random colony ship will be destroyed. Therefore you have to provide one more function in your Fleet class with the signature "void ColonyShip::destroy()". The function is very similar to the "void ColonyShip::infect()" function but while an infected colony ship still contributes to the total weight of the fleet a destroyed colony ship (including aggregated suppliers) does not. There is also a small change to the "userInterfaceCreateFleet()" function. When you create your "fleet.dat" file you have to write your corporate name in the first line. Your corporate name should be composed of your surname followed by your first name - both with first letter capitalised. Below is an example of my new "fleet.dat" file.

```
SiebersPeer
Ferry 3
Supplier 3
Medic 1
Radiant 1
```

It is up to you how you implement the user interface. Figure 3.1 shows an example. Please note that the user should not be able to choose Supplier ships. These should be automatically added to the fleet (and charged for) when a ColonyShip is bought.

```

G:\Workspace\C++\testui2\bin\Debug\g64oos_coursework2013.exe
Enter Coporation name: SiebersPeer

COLONY SHIPS
Name          Cost          No. of Colonists
1. Ferry      500 (<+100)      100
2. Liner      1000 (<+200)     250
3. Cloud      2000 (<+400)     750

SOLAR SAIL SHIP
Name          Cost          Energy Production
4. Radiant    50           50
5. Ebullient  350          500

MEDIC SHIP
Name          Cost          Energy Consumption
6. Medic      1000         1

=====
Total budget available to ***SiebersPeer*** is 10000
Enter a ship number to select it or press 0 to stop or exit:

```

Figure 3.1: User interface example

Here are some more screenshots that might help you. Figure3.2 shows the main() class that initiates the simulation and provide the output report. Figure 3.3 shows a sample output report.

```

1 // v0.98 of the Simulator
2
3 #include <iostream>
4 #include "Fleet.h"
5 #include "Simulator.h"
6
7 int main(){
8     Fleet* fleet1 = new Fleet();
9     Fleet* fleet2 = new Fleet();
10    fleet1=FileIOFuncs::loadSavedFleet("4177615_fleet.dat");
11    fleet2=FileIOFuncs::loadSavedFleet("4119207_fleet.dat");
12    Simulator sim = Simulator();
13    sim.addFleet(fleet1);
14    sim.addFleet(fleet2);
15    sim.launchFleets();
16    sim.report();
17    return 0;
18 }

```

Figure 3.2: RunSimulation.cpp

```

G:\Workspace\C++\g64cw2013new\bin\Debug\g64oos_coursework2013.exe
Loaded the following ships into fleet:
Cloud, Cloud, Cloud, Radiant, Radiant, Cruiser, Cruiser, Cruiser, Medic

Loaded the following ships into fleet:
Cloud, Cloud, Destroyer, Ebullient, Ferry, Liner, Liner, Medic, Radiant, Radiant
=====
Now launching 2 fleets
=====
Simulating travel of GonzalesSpeedy
Colonists travelling: 2250; Fleet weight: 103
First half of journey takes 16.7457 lightyears
Fleet attacked by alien!
Cloud attacked - 750 colonists died
Fleet infected by disease!
Disease is prevented by the Medic!
Colonists travelling: 1500; Fleet weight: 73
Second half of journey takes 14.0976 lightyears
Total journey duration: 30 lightyears

Simulating travel of BunnyBugs
Colonists travelling: 2100; Fleet weight: 186
First half of journey takes 22.503 lightyears
Fleet attacked by alien!
Cloud attacked - 750 colonists died
Fleet infected by disease!
Disease is prevented by the Medic!
Colonists travelling: 1350; Fleet weight: 156
Second half of journey takes 20.6085 lightyears
Total journey duration: 43 lightyears

=====
Resolving fleet arrivals:
=====
GonzalesSpeedy arrives at planet:
- Arrival year: 30
- Colonists living on board: 1500
- Population growth since last fleet arrival: 0
- Current planet owner: Unclaimed
- Current population: 0
- GonzalesSpeedy is the new owner of Gaia

BunnyBugs arrives at planet:
- Arrival year: 43
- Colonists living on board: 1350
- Population growth since last fleet arrival: 1328
- Current planet owner: GonzalesSpeedy
- Current population: 2828
- BunnyBugs arrives with fewer colonists
- GonzalesSpeedy remains the owner of Gaia

Final owner of Gaia is: GonzalesSpeedy
- Year at which spacerace is resolved: 43
- Final population: 2828

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.

```

Figure 3.3: Sample report produced by the report() function of Simulator.cpp

31/03/2014

An extended simulator class (v0.99) is now available on the module website. The extension also allows interface testing.

As mentioned earlier there is one addition to the interface which is the function "destroy". This also requires a simple function to test if a ship is destroyed. Please add the following function to your fleet source code:

```

bool ColonyShip::isDestroyed() const {
    return destroyed;
}

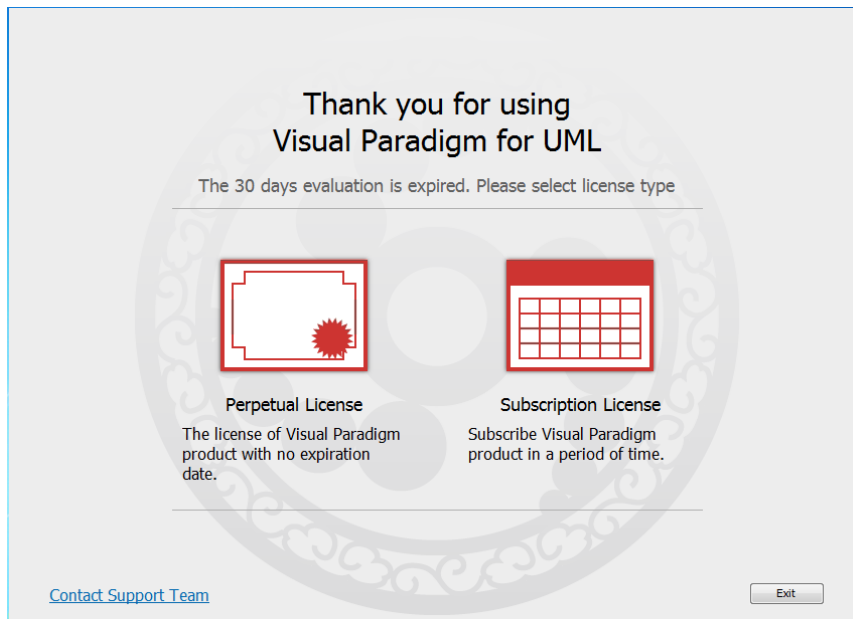
```

Finally, to make the interface more consistent please note the change in Row 4 of Table 3.4:

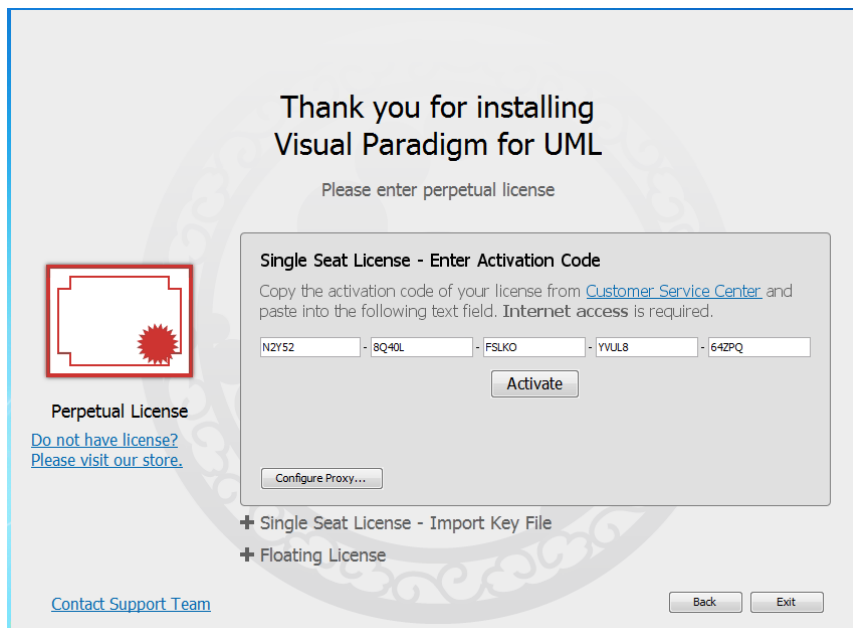
- "int Fleet::EnergyProduction() const;" has become "int Fleet::getEnergyProduction() const;"

Appendix A: Visual Paradigm Registration

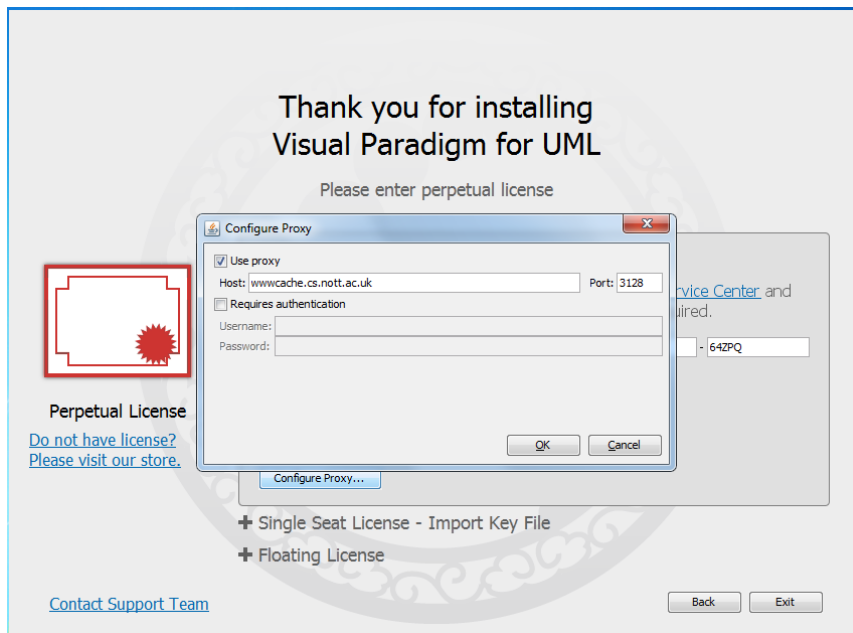
Run Visual Paradigm



Select Perpetual License



Copy and paste key: DP420-8M564-U2S1M-UQ09E-492EF



Select "Configure Proxy" and enter host "wwwcache.cs.nott.ac.uk" and port "3128". Press "OK" and select "Activate"