COSC230 Practical 1: Programming in C++ and a Simple Linux IDE (Practical for Week 1 and 2)

The purpose of this practical is: 1) to introduce a simple Linux-based IDE (integrated development environment) that allows you to edit, compile, build, and debug C++ programs, and 2) to apply the C++ language features useful for implementing your own data structures, including those for encapsulation; operator overloading; memory management; and exception handling.

1 Compiling a C++ program

To start, log into Turing (using a remote desktop connection called x2go) and pull up a terminal from the tool bar at the bottom of the desktop. Open your favourite text editor (gedit is a simple, light-weight, editor available on Turing) and type:

```
#include <iostream>

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

    return 0;
}
```

Conventional style is to indent C++ code using a tab of 4 spaces to help make it human readable. Save this as a file named hello.cc, for example, and enter on the terminal command line: g++ hello.cc -o hello. This command compiles the source file hello.cc and generates the executable file hello. Typing ./hello on the command line should then execute the program, yielding the console output: "Hello World". The structure and syntax of this program has already been discussed in lectures.

2 Encapsulation in C++

Next, consider the following class `Bank_Account` that maintains customer details for a bank account:

```
// A class for storing bank account details.
#include <string>
#include <iostream>

class Bank_Account {
public:
    // default constructor
    Bank_Account(): account_number(), balance() {  }

    // constructor
    Bank_Account(std::string a, int b, double c = 0.0)
    {
        customer_name = a;
        account_number = b;
        balance = c;
    }

    // modification member function
    void set_balance(double a) { balance = a; }

    // constant (read-only) member functions
    std::string get_customer_name() const { return customer_name; }
    int get_account_number() const {return account_number; }
    double get_balance() const { return balance; }
private:
    // data members (variables for storing data)
    std::string customer_name;
    int account_number;
    double balance;
};
```

The main point to note here is that the default class constructor delegates its work to the default constructors for each of the data types. Also, there is a default argument giving `balance` a default initial value of 0.0 in the constructor for `Bank_Account`. Now let's test this class:

```
// testing the class

int main()
{
    // object instantiation
    Bank_Account customer1;
    Bank_Account customer2("David", 123, 100.0);

    // output values for customer 1
    std::cout << "Name: " << customer1.get_customer_name() << std::endl;
    std::cout << "Account number: " << customer1.get_account_number() << std::endl;
    std::cout << "Balance: " << customer1.get_balance() << std::endl;

    // output values for customer 2
    std::cout << "Name: " << customer2.get_customer_name() << std::endl;
    std::cout << "Account number: " << customer2.get_account_number() << std::endl;
    std::cout << "Balance: " << customer2.get_balance() << std::endl;

    // testing the "set" member
    customer2.set_balance(200.0);
    std::cout << "New Balance: " << customer2.get_balance() << std::endl;

    return 0;
}
```

Saving all code in a file `Bank_Account_first.cc`, then compiling and running the code, generates the output:

```
Name:
Account number: 0
Balance: 0
Name: David
Account number: 123
Balance: 100
New Balance: 200
```

3 Using Makefile

As seen in the previous example, `Class` is the main language feature for encapsulation in `C++`. However, the code would be more human readable if we could separate the class *interface* from its *implementation*. In addition, separating the source code into different files and compiling

them independently would make code debugging much easier. Fortunately, a nice utility called `makefile` facilitates building `C++` code from multiple source files.

Let's split the code from the previous example into a header file called `Bank_Account.h`, a source file called `Bank_Account.cc`, and a main file called `main.cc`. Here is what they look like:

```
// Bank_Account.h (the class interface)
// A class for storing bank account details.
#include <string>

class Bank_Account {
public:
    Bank_Account(): account_number(), balance() {  }
    Bank_Account(std::string, int, double);
    void set_balance(double);
    std::string get_customer_name() const;
    int get_account_number() const;
    double get_balance() const;
private:
    std::string customer_name;
    int account_number;
    double balance;
};

// Bank_Account.cc (the member function implementations)
#include <string>
#include "Bank_Account.h"

Bank_Account::Bank_Account(std::string a, int b, double c = 0.0)
{
    customer_name = a;
    account_number = b;
    balance = c;
}

void Bank_Account::set_balance(double a)
{
    balance = a;
}
```

```
std::string Bank_Account::get_customer_name() const
{
    return customer_name;
}

int Bank_Account::get_account_number() const
{
    return account_number;
}

double Bank_Account::get_balance() const
{
    return balance;
}

// main.cc (testing the class)
#include <iostream>
#include "Bank_Account.h"
using namespace std;

int main()
{
    Bank_Account customer1;
    Bank_Account customer2("David", 123, 100.0);

    // output values for customer 1
    cout << "Name: " << customer1.get_customer_name() << endl;
    cout << "Account number: " << customer1.get_account_number() << endl;
    cout << "Balance: " << customer1.get_balance() << endl;

    // output values for customer 2
    cout << "Name: " << customer2.get_customer_name() << endl;
    cout << "Account number: " << customer2.get_account_number() << endl;
    cout << "Balance: " << customer2.get_balance() << endl;

    // testing set member
    customer2.set_balance(200.0);
    cout << "Balance: " << customer2.get_balance() << endl;

    return 0;
}
```

To compile and build the three source files `Bank_Account.h`, `Bank_Account.cc`, and `main.cc` we now write a makefile routine:

```
main: main.o Bank_Account.o
    g++ main.o Bank_Account.o -o main

main.o: main.cc Bank_Account.h
    g++ -c main.cc -o main.o

Bank_Account.o: Bank_Account.cc Bank_Account.h
    g++ -c Bank_Account.cc -o Bank_Account.o

clean:
    rm Bank_Account.o main.o main
```

and save it as `makefile`. Notice the structure of `makefile` here. Each first line begins with the name of the file that will be generated, followed by a list of files it depends on. If one of these files is edited following compilation, then calling `make` will cause only those files that depend upon it to be re-compiled. The second line (**indented by a tab space** – very important to note) gives the command that will be executed upon calling a file name prefixed with `make`. For example, calling `make main.o` will cause `g++ -c main.cc` to be executed if `main.cc` or `Bank_Account.h` have been modified since `main.cc` was previous compiled.

Notice also the different stages of compilation. Roughly speaking, a source file (i.e., `main.cc`) is compiled into an object file (i.e., `main.o`), which is then linked together with other object files to produce an executable file (i.e., `main`). There is also a preprocessor step before the object file is generated where any `#include` or `#define` statements are processed to generate macro code. This is the stage where header file code is copied into source code.

Typing `make` on the command line will now build your code. Typing `./main` will run it.

4 Debugging C++ Code

Now that we have some C++ basics it is time to look at debugging software. This software will allow you to step through your code while it is running, and print the values of any variables.

The open source *Data Display Debugger*, or `ddd`, is what will be used here. It is a graphical front-end for the well-known command line debugger `gdb`. To carry out debugging with `ddd` the flag `-g` needs to be included in the compile commands. For example, `g++ -c Bank_Account.cc` becomes `g++ -g -c Bank_Account.cc`. If you are adding the flag `-g` to your `makefile`, make sure you call `make clean`, followed by `make`. This will ensure your code is recompiled with the newly added flag. Now entering

```
ddd main
```

will bring up the debugger. Double-clicking the mouse in the space to the left of a line of code in the debugger window will set a break point where execution will stop. Right clicking will allow the break point to be deleted. Clicking the `Run` button on the tool bar over on the right will run the code up until the break point is reached. Then you can click either `Next` or `Step` to increment execution to the next line of code. Double-clicking on a variable or object will display that variable or objects' value(s) in the data display pane. For example, after the `customer2` object has been instantiated, double-clicking on `customer2` will display the values of all data members for `customer2`. You can also type `print customer2` on the command line in the debugger window to achieve this.

5 Operator Overloading in C++

Operator overloading refers to an operator that has different meanings for operands of different types. For example, working with type `int` and evaluating `1/2` yields 0. Working with type `double` will yield 0.5. So the `/` operator does something different for integers and double precision numbers.

Now consider a class for rational numbers. Remember, these are numbers that are ratios of integers. By defining a class for rational numbers we can add two fractions and get a result with infinite precision, as opposed to approximating those fractions using `double` and getting a result of finite precision. Here is the class:

```
// Rational.h
// A rational number class with operator overloading.
#include <iostream>

class Rational {
friend std::ostream& operator<<(std::ostream&, const Rational&);
public:
    Rational(): numerator(), denominator(1) {  }
    Rational(int, int);
    void reduce();
    Rational operator+(const Rational&) const;
private:
    int numerator;
    int denominator;
};
```

```
// Rational.cc
#include <cstdlib>
#include "Rational.h"

// Overloading the stream insertion operator to print a rational number.
std::ostream& operator<<(std::ostream& output, const Rational& r)
{
    output << r.numerator << "/" << r.denominator;

    return output;
}

// Non-member function for recursive Euclidean Algorithm to find gcd.
int greatest_common_divisor(int a, int b)
{
    a = abs(a), b = abs(b); // positive and negative integers have same divisor.

    if (b == 0)
    {
        return a;
    }
    else
    {
        return greatest_common_divisor(b, a % b);
    }
}

// Constructor
Rational::Rational(int a, int b)
{
    numerator = a;
    denominator = b; // assuming a non-zero denominator.

    reduce();
}
```

```
// Member function to reduce fractions to lowest terms.
void Rational::reduce()
{
    int gcd = greatest_common_divisor(numerator, denominator);

    numerator = numerator/gcd;
    denominator = denominator/gcd;

    if (denominator < 0)
        numerator = -numerator, denominator = -denominator;
}

// Overloading the + operator to add two rational numbers
Rational Rational::operator+(const Rational& a) const
{
    Rational result;

    // cross-multiply, then add together.
    result.numerator = numerator*a.denominator + a.numerator*denominator;
    result.denominator = denominator*a.denominator;

    return result;
}
```

Now let's test this class using the following code:

```
// main.cc
#include "Rational.h"
using namespace std;

int main()
{
    // Testing the rational number class.

    // Testing the constructors.
    Rational a;
    Rational b(4,3);
    Rational c(5,10);

    // Testing the overloaded stream insertion operator.
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;

    // Testing the overloaded + operator.
    Rational d = b + c;

    cout << d << endl;

    return 0;
}
```

This code returns the output:

```
0/1
4/3
1/2
11/6
```

Try writing and testing overloaded operators for subtraction, multiplication, and division of rational numbers.

6 Pointers, Arrays, and Memory Management

The C/C++ language makes extensive use of pointers. Pointers also allow for the most flexible implementation of linked data structures. A pointer variable has a value that corresponds to the *address* (memory location) of a variable or an object. That is, a pointer refers to a block of memory. Pointers are more versatile than references but also require more management. An important aspect for data structures is that pointers facilitate dynamic allocation and de-allocation of memory. This allows a data structure to grow or shrink during run-time, as the situation requires.

While there are many things that can be done with pointers, here we focus only on those related to the memory management of data structures. Firstly, let's show how a pointer can indirectly reference a value (called *indirection*). Remembering that `int* p` declares `p` to be a variable of type `int*` (that is, a pointer to type `int`), `&` is the *address operator*, and `*` is the *dereference operator*, consider the following:

```
int x = 2;

// declaring a pointer p that points to x
int* p = &x;
cout << x << endl;

// change the value of x through p
*p = 1;
cout << x << endl;
```

What would you expect the output to be? Run it to make sure.

An array data structure has the defining property that array values are randomly accessible, meaning that *any* array value can be accessed efficiently. This is achieved through indexing. If blocks of memory are allocated contiguously (adjacent to one another) then memory locations can easily be indexed in a simple linear manner. In C++, provided the size of an array is known in advance (at compile time) then the array can be allocated statically. For example, an array of `10` items of type `int` can be declared statically as

```
int p[10];
```

In this case the size of the array remains fixed for the duration of the program. To allocate an array dynamically pointers are used. The dynamic allocation of an array is given by

```
int* p = new int[n];
```

where the value of `n` can now be provided at run time; i.e., this code will compile fine without a value for `n`. Note that in both examples `p` is a pointer to the first element of the array. This means the value of the first element of the array can be found using either pointer dereferencing

`*p`, or index notation `p[0]`. Because pointers support indexing, it is possible to use *pointer arithmetic* to find the value of the nth element of the array as `*(p + n)`. This corresponds to `p[n]` in index notation. Pointer arithmetic can also be used to show that memory locations are contiguous. For example, type `int` takes 4 bytes. So `p + i` and `p + i + 1` will give hexadecimal numbers (corresponding to memory locations) that differ by 4. Try this out yourself!

Pointers, dynamic memory allocation, classes, and memory management all come together when considering container classes. A container is a data structure that holds a collection of values. Let's now consider a very simple container class called `Vec` (short for vector) that holds a collection of double precision numbers. To start with, `Vec` is specified by just one member function given by the class constructor:

```
// A minimal 1D array class to demonstrate memory management.
class Vec {
public:
    Vec();
    Vec(int);
private:
    int array_limit;
    double* pointer_to_data;
};

// Constructors
Vec::Vec()
{
    array_limit = 0;
    pointer_to_data = 0;
}

Vec::Vec(int a)
{
    array_limit = a;

    // Allocate new memory for array.
    pointer_to_data = new double[array_limit];

    // Initialize array.
    for (int i = 0; i != array_limit; ++i) {
        pointer_to_data[i] = 0.0;
    }
}
```

If we now try to instantiate an object by compiling and executing the following code in `main`:

```
Vec v(5);
```

we see that everything appears to be fine. However, there is actually a *memory leak* present. This is a fairly common beginner problem with programming languages such as `C/C++` that require explicit memory management. To see this, type

```
valgrind ./test
```

on the command line. As part of the console output you should now see:

```
LEAK SUMMARY:
    definitely lost: 40 bytes in 1 blocks
```

`Valgrind` is an open source memory debugging program that is very useful for detecting memory leaks. The problem here is that every call to `new` needs to be paired with a call to `delete` in order to correctly allocate and de-allocate memory resources in a program. This problem can be fixed by including a *destructor* in the class:

```
class Vec {
public:
    ~Vec();
```

which is implemented as:

```
// Destructor
Vec::~Vec()
{
    delete[] pointer_to_data;
    // Set pointer to null
    pointer_to_data = 0;
}
```

The destructor is called when a `Vec` object goes out of scope. The `Vec` object is then correctly cleaned up, and the memory associated with it is de-allocated. Recompiling, and calling `valgrind`, now indicates the leak has been plugged:

```
All heap blocks were freed -- no leaks are possible
```

It is always important to check your code for memory leaks.

When writing classes that use pointers, there are a couple of other pitfalls to be aware of. To see these we will need to overload the index operator so that we can access the values stored in our container:

```
class Vec {
public:
    double& operator[](int);
```

which is implemented as:

```
// Overloaded Index Operator
double& Vec::operator[](int i)
{
    return pointer_to_data[i];
}
```

Now consider the following code segment where we declare a `Vec v1` to be of `array_limit` 1 and initialize its value to 2.0, then make a copy of `v1` called `v2`, before re-setting `v2` to 4.0:

```
// in main.cc
Vec v1(1);
v1[0] = 2.0;

Vec v2(v1);
cout << v1[0] << " " << v2[0] << endl;

v2[0] = 4.0;
cout << v1[0] << " " << v2[0] << endl;
```

This code will probably crash when it is run. To see what happens before the crash, use `ddd` to run the program until just before it exits from `main`. You will see that the output gives:

```
2 2
4 4
```

Resetting `v2` seems to have also reset `v1`, i.e. these container objects are not independent. The reason is that the pointer address has been copied from `v1` to `v2` so that both pointers `v1.pointer_to_data` and `v2.pointer_to_data` now point to the same block of memory! Because we have not provided a *copy constructor* that defines what should happen when a `Vec` object is copied, the compiler has automatically generated one for us. And in this case it does not do what we would like. Here is what we need:

```
class Vec {
public:
    Vec(const Vec&);
```

which is implemented as:

```
// Overloaded Copy Constructor
Vec::Vec(const Vec& vec_to_copy)
{
    array_limit = vec_to_copy.array_limit;

    // Allocate new memory for array.
    pointer_to_data = new double[array_limit];

    // Copy data across.
    for (int i = 0; i != array_limit; ++i) {
        pointer_to_data[i] = vec_to_copy.pointer_to_data[i];
    }
}
```

The program should now run fine and generate the correct output:

```
2 2
2 4
```

If a class requires a destructor and a copy constructor, then it probably also requires an *assignment operator*. This is known as the *rule of three*. An assignment operator is defined as an operation that obliterates an existing left-hand side and replaces it with the right-hand side:

```
class Vec {
public:
    Vec& operator=(const Vec&);
```

which is implemented as:

```
// Overloaded Assignment Operator
Vec& Vec::operator=(const Vec& rhs)
{
    // Check for self-assignment.
    if (&rhs != this) {
        // Free lhs array memory.
        delete[] pointer_to_data;

        // Allocate new memory for lhs array.
        array_limit = rhs.array_limit;
        pointer_to_data = new double[array_limit];

        // Copy rhs data across.
```

```
        for (int i = 0; i != array_limit; ++i) {
            pointer_to_data[i] = rhs.pointer_to_data[i];
        }
    }

    return *this;
}
```

The `this` variable is a pointer to the `lhs` object.

The complete class header file is now:

```
// Vec.h
// A minimal 1D array class to demonstrate memory management.

class Vec {
public:
    // Constructors
    Vec();
    Vec(int);
    // Destructor
    ~Vec();
    // Overloaded copy constructor
    Vec(const Vec&);
    // Overloaded assignment operator
    Vec& operator=(const Vec&);
    // Overloaded index operator
    double& operator[](int);
private:
    int array_limit;
    double* pointer_to_data;
};
```

7 Inheritance in C++

It is assumed that you already know about object-oriented programming from previous programming units. Here, I only present the C++ syntax for inheritance. Consider a base class given by the `Bank_Account` class. We would like to write a derived class that maintains a credit card balance on top of the basic bank account details. This is what the derived class looks like:

```
// Add code to Bank_Account.h
// A class derived from Bank_Account.
class Credit_Account: public Bank_Account {
public:
    Credit_Account(): Bank_Account(), credit_balance() {  };
    Credit_Account(std::string, int, double, double);
    void set_credit_balance(double);
    double get_credit_balance() const;
private:
    double credit_balance;
};
```

This says that the class inherits the public interface of `Bank_Account`. It maintains an extra data member, `credit_balance`, as well as associated read-only and modification member functions. The default constructor delegates its work to the default constructors for class `Bank_Account` and type `double`. Constructor delegation is also possible for other (non-default) constructors since `C++11`. To use this feature, add the flag `-std=c++11` to your `makefile` code. The constructor can then be implemented as:

```
Credit_Account::Credit_Account(std::string a, int b, double c, double d)
: Bank_Account(a, b, c)
{
    credit_balance = d;
}
```

and delegates most of its work to the constructor for `Bank_Account`. Alternatively, you can write further *set* member functions for the `Bank_Account` class. These will then be inherited by `Credit_Account`, and can be used in its constructor to set the private data members in the base-class part of the `Credit_Account` class.

8 Exception Handling in `C++`

To throw an exception in `C++` use the keyword `throw`. The program will then crash when an exception is thrown unless a `try-catch` statement is used to handle the exception. However, it is the responsibility of the *user* of a class to handle the exception by supplying the `try` and `catch` blocks within the `main()` function. The *code developer* is only responsible for throwing the exception by supplying the `throw` block within the function or class implementation.

```
// A minimal example of exception handling.
#include <stdexcept> // library for "std::invalid_argument" class
#include <iostream>
using namespace std;

void some_function(int a)
{
    if (a < 0)
        throw invalid_argument("argument cannot be a negative number");
}

int main()
{
    int some_negative_integer = -1;

    // crashes program when exception is thrown.
    some_function(some_negative_integer);

    // does not crash program when exception is thrown:
    // restricts scope of "some_function" to try block,
    // handles exception in catch block.
    try {
        some_function(some_negative_integer);
    }
    catch (invalid_argument& exception) {
        cout << "Error: " << exception.what() << endl;
    }

    cout << "Program continues running..." << endl;

    return 0;
}
```

This concludes Practical 1.