Structures and Objects in C++

This background information document will introduce the idea of a structure in C/C++ before briefly outlining the idea of object classes in C++.

1 Structures

We saw a couple of weeks back that C++ vectors are great at bundling together related variables in dynamically sized arrays, but they were limited because each member had to be the same data type. Also, even if all our data of the same type organising it into a single array may not make sense conceptually. For example, imagine[1] you are writing a program to solve Newton's equations for the sun, earth and moon. You might proceed by defining all the quantities of interest separately:

```
    double sunPos[3], moonPos[3], earthPos[3]; // Position vectors
    double sunVel[3], moonVel[3], earthVel[3]; // Velocity vectors
    double sunMass, earthMass, moonMass; // Masses
```

Logically our problem comprises three bodies so our code would be much more elegant and readable if we collected the variables together in a way that mimicked this. Defining a single array for each body like this:

```
    double sun[7], moon[7], earth[7]; // Position vector + velocity vector + mass.
```

does collect all the information together and is cleaner, but it is also very obtuse[2]. A better way is to use a *structure*. If a variable is a box, then you can think of a structure as a bigger box, with smaller boxes inside it. This is best illustrated by defining a structure for the information about a planetary body as:

```
struct body {
    double mass; // Mass
    double pos[3]; // Position vector
    double vel[3]; // Velocity vector
};
```

Here the `struct <name>` indicates you are creating a structure called `<name>` and then between `{ ... };` is a one-by-one declaration of the member variables of this structure. By declaring a structure we are effectively defining a new data type (called an abstract data type) which we can use in our code just like a primitive data type, e.g. `int` or `double`. In our code we can instantiate three `Body` structures for our problem as:

```
    body sun, moon, earth; // Declare structures for all three bodies
```

analogous to how we would define three variables with a primitive data type:

```
    int myVar1, myVar2, myVar3; // Declare structures for all three bodies
```

[1] Imagining will very soon become doing in Assessment 3!
[2] You would need to remember that `sun[4]` is in fact the y-component of its velocity!

Structure are declared outside and before the `main()` function at the top of a source file. We don't access members of a structure using an index, rather we use a field name with dot `.`, for example `sun.Mass` gives us access to the mass of the sun. Note that names of members of structures (`mass`, `pos`,`vel`) are the same for every structure of the `body` type, but the name of the structure is different for each instance, `sun`, `earth` and `moon`. Putting this together our program using the `Body` structure might begin like this:

```
#include <iostream>

// Define the structure outside and before the main() function
struct body {
    double mass; // Mass
    double pos[3]; // Position vector
    double vel[3]; // Velocity vector
};

int main()
{
    Body sun, moon, earth; // Declare structures for all three bodies

    earth.mass = 6.0e24; // Mass of earth [kg]
    // Set the sun as the origin of our coordinate system:
    sun.pos[0] = 0.0;
    sun.pos[1] = 0.0;
    sun.pos[2] = 0.0;

    // and so on ...

    return EXIT_SUCCESS;
}
```

Like any data type we can define an array of structures with the usual notation:

```
    body neptune_moons[14]; // Declare an array of 14 body structures
```

We access the elements of a given structure in this array using an index as usual, e.g. `neptune_moons[3].mass` to access the 4th moon's mass, and similarly elements of an array inside a structure such as `neptune_moons[2].pos[0]` to access the 3rd moon's x position coordinate.

Naturally we can pass structures as arguments to functions. Consider the following program using a slightly modified version of our structure `body`:

```
#include <iostream>
#include <cmath>

using std::cout, std::endl;

struct body {
    double mass; // Mass
    double pos[3]; // Position vector
    double vel[3]; // Velocity vector
    double radius; // Radius -- added this new member here
};

double density(body planet); // Declaration of the function

int main()
{
    body earth;

    earth.mass = 6.0e24; // Mass of earth [kg]
    earth.radius = 6.3781e6; // Radius of earth [m]

    double rho = density(earth); // Declare and call density function to define rho

    cout << "Density of Earth = " << rho << " kgm^-3" << endl;

    return EXIT_SUCCESS;
}

```

```
29 double density(body planet) // Definition of the function
30 {
31     double volume = (4.0/3.0)*M_PI*pow(planet.radius,3);
32     double rho = planet.mass/volume;
33     return rho;
34 }
```

Here the entire `body` structure, including all the arrays inside it, are *deep-copied* into a new local variable `planet` that is accessible only inside the `density()` function. We can remedy this potential inefficiency by using pass-by-reference discussed last week.

The philosophy of designing a structure to encapsulate logically related variables was essentially the first embryonic step (in C) towards fully object orientated programming (OOP) (in C++). A fully OOP approach would define object classes which group together in a logical way both data, e.g. as in `body`, and functions for manipulating that data, e.g. like `density()`.

2 A first look at Objects in C++

As we have mentioned many times C++ is an OOP language built on C. For the most part we have really only used the the OOP power of C++ very tangentially in this course. For our final topic it is therefore fitting that we discuss briefly how we can build objects in C++. What we will see is that C++ upgrades structures from C into fully fledged classes.

2.1 Declaring a class

Structures allow us to create sophisticated new abstract data types with many different members potentially of different primitive types or other structures. The purpose of a class is to embed functions that manipulate data with data itself creating a logical unit. This is best illustrated by an example and consistent with the flavour of this course it will be based on numbers. Let's create a C++ class for rational numbers, so numbers which can be expressed as a fraction a/b of two integers. Here is what a basic class declaration would look like:

```
1  /* --- Declaration of rational --- */
2
3  class rational {
4      public:
5          // Constructors
6          rational();
7          rational(int a, int b);
8          // Responsibilities
9          int GetNumerator() const;
10         int GetDenominator() const;
11     private:
12         int numerator; // The numerator
13         int denominator; // The denominator
14 };
```

In contrast to a `struct` a `class` has `public` methods by which you can interact with the object and `private` data that cannot be accessed directly (this is so-called data encapsulation). For rational numbers the data is simply the numerator and denominator integers. The public methods include *constructors*, these are functions that initialise an instance of the class, and additional functions for getting the numerator and denominator (since we don't have access to those variables directly).

2.2 Implementing a class

Having declared our class we now need to implement its methods. Here is what this might look like:

```
/* --- Implementation of rational --- */

// Default construction:
rational::rational()
{
    numerator = 0; // Set rational number to 0/1 = 0
    denominator = 1;
}

// Parameterised construction:
rational::rational(int a, int b)
{
    numerator = a; // Set rational number to a/b
    denominator = b;
}

// Access methods:
int rational::GetNumerator() const
{
    return numerator;
}

int rational::GetDenominator() const
{
    return denominator;
}
```

Notice that we define the name of these functions as `rational::<function>`, where `::` is the scope resolution operator. We have used this already numerous times for e.g. `std::cout`. Here it is saying that these functions are methods belonging to the class `rational`. The methods themselves are extremely trivial in this example.

2.3 Using a class

Assuming we put the class declaration and the class implementation in a single source file `rational.cpp`, then a program using this class could be:

```
#include <iostream> // Usual library header

/* --- Declaration of rational --- */

int main()
{
    rational A; // Declare a rational number
    rational B(231,983); // Declare and initialise a rational number
    std::cout << "A = " << A.GetNumerator() << "/" << A.GetDenominator() << std::endl;
    std::cout << "B = " << B.GetNumerator() << "/" << B.GetDenominator() << std::endl;

    return EXIT_SUCCESS;
}

/* --- Implementation rational --- */
```

The output from this program just confirms everything works as expected:

```
A = 1/1
B = 231/983
```

While the `rational` class illustrates objects it could clearly be improved with more functionality. Let's modify the class so it has some intuitive properties expected for numbers in a programming language. Consider the following declaration:

```
/* --- Declaration of rational --- */

class rational {
```

```
    public:
        // Constructors
        rational();
        rational(int a, int b);
        // Responsibilities
        void print() const;
        rational operator+(const rational& other_rational) const;
    private:
        int numerator; // The numerator
        int denominator; // The denominator
};
```

and associated implementation:

```
/* --- Implementation of rational --- */

// Default construction:
rational::rational()
{
    numerator = 0; // Set rational number to 0/1 = 0
    denominator = 1;
}

// Parameterised construction:
rational::rational(int a, int b)
{
    numerator = a; // Set rational number to a/b
    denominator = b;
}

// Access methods:
void rational::print() const
{
    std::cout << numerator << "/" << denominator << std::endl; // Print out number
}

rational rational::operator+(const rational& other_rational) const
{
    // Compute the addition of two rational numbers:
    int a = numerator*other_rational.denominator + other_rational.numerator*denominator;
    int b = denominator*other_rational.denominator;
    rational sum(a,b); // Assign a new rational number
    return sum; // Return this rational number
}
```

Here is a program demonstrating how this new version works:

```
#include <iostream> // Usual library header

/* --- Declaration of rational --- */

int main()
{
    rational A(13,727); // Declare and initialise a rational number
    rational B(231,983); // Declare and initialise a rational number
    // Display the numbers:
    A.print();
    B.print();
    // Declare and initialise a rational number by adding the first two together:
    rational C = A + B; // Can use + operation
    C.print();

    return EXIT_SUCCESS;
}

/* --- Implementation rational --- */
```

What is interest here is that we have overloaded the addition operator + for this class and thereby defined what `A + B` would be for two instances of `rational`. This is exactly the behaviour we would want from a number. Running this program confirms that $13/727 \approx 0.0179$ added to $231/983 \approx 0.235$ gives $180716/714641 \approx 0.2529$. We can overload many other operators such as $-$, $/$ and even $++$ so they are defined for this new data type. Hopefully the power of OOP design approach is now becoming apparent and exploring it in much more detail will be a major task for next year. □