5.3 Classes: a second approach

This section is motivated by a need for clarification of C++ codes and elementary operations. It is taken from Scott Meyer’s book, *Effective C++ second edition*.

5.3.1 C++ elementary operations

Definition vs Declaration

There is a fundamental difference between *declarations* and *definition*. The declaration often precedes the definition by telling the compiler the *type* of an object, for instance:

**Declaration**

```cpp
double MidPointRule(double, double, int, double (f*)(double));
class Timer;
ostream& operator<<(ostream& output, const Matrix& mat)
```

A definition specifies the details of the object, for instance when writing a class or a function.

**Definition**

```cpp
double MidPointRule(double a, double b, int N, double (f*)(double)){
    double integral_value = 0;
    double delta = (b-a)/N;
    double x = 0;
    for(int i = 0; i<N; i++){
        x = a + (i+0.5)*delta;
        integral_value += f(x)*delta;
    }
    integral_value /= 2.0;
};
class Timer {
public:
    Timer();
    virtual ~Timer();
public:
    void start_timer();
    void stop_timer();
    double Get_time_elapsed();
private:
    double start, stop;
};
```

Declarations are the purpose of *header* files (.h), whereas definitions are done in *source* files (.cpp).

5.3.2 Class Design

Constructors

The *default* constructor is a constructor that is called without arguments. It is needed for the definition of arrays or 'empty' definitions.

**Timer default constructor**

```cpp
Class TimerA{
```
public:
    TimerA();
}
TimerA record;
TimerA record_array[10];

Class TimerB{
    public:
        TimerB(int start = 0);
    }
TimerB record_array[10];

Class TimerC{
    public:
        TimerC(int start); // not default constructor
    }
TimerC record_array[10]; // error!

This last example is *compiler dependent* since the creation of the array requires to call a default constructor, which here was not specified. The way around this is the use of pointers.

Timer default constructor

TimerC *record_array[10];
record_array[0] = TimerC((double)clock());

Note that this last declaration *allocates* and *constructs*.

**Initialization vs assignment**

Copy constructors are often used to initialize data. In the Timer example:

Copy constructor

```
TimerB record;
TimerB record1 = TimerB((double) clock());
TimerB record2 = record1;
record = record2;
```

These three operations are initialization and the two last call a *copy* constructor. The last operation is an assignment. The problem arises when the target of the assignment already has resources allocated to it! A proper allocation would need the operator= to be overloaded so that it destroys the target before creating a new block of memory.

Copy constructor

```
String a("Hello"); // define and construct a
{
    String b("Word"); // define and construct b
    ...
    b = a; // execute default op=, lose b's memory
    // close scope, calls b's destructor
}
String c = a; // a.data does not exist !
```
Member functions

This section explains the difference between member functions, friend functions and non-member functions. Consider the two multiplication operators defined for our Matrix class.

Copy constructor

Matrix operator* (const double& a);
friend Matrix operator* (const Matrix& mat1, const double& a);

These operators multiply a Matrix by a constant. You may easily use these operations as

Copy constructor

Matrix A;
Matrix B = A*2.0;
B = B * 10.0;

The problems lies in making the multiplication operator commutative, since we would like to perform operations such as \(A \times 2\) or \(2 \times A\). Now, looking at our previous operator, if we use the member function operator*, then we have two choices:

Copy constructor

Matrix A,B;
B = A.operator*(2.0);
B = 2.operator*(A);

Your compiler is smart though, and will look for a non-member function to be called so that

Copy constructor

B = operator*(2,A);

But obviously, the search will fail. A possible solution is to create a non-member function, i.e. outside of the scope of the class.

Copy constructor

Matrix operator* (const Matrix& mat1, const double& a);
Matrix operator* (const double& a, const Matrix& mat1);

5.3.3 Public vs Private

Avoid data members in the public interface!! If your data is public, anybody can read and write the data, possibly modifying your values throughout your code run.

Copy constructor

```cpp
class Timer {
public:
    Timer();
    ~Timer();

public:
    void set_start() { start = (double) clock(); };
    void set_stop() { stop = (double) clock(); };
    double Get_time_elapsed() { compute_ellapsed(); return elapsed; }

private:
    void compute_ellapsed() { elapsed = (stop - start)/(double) CLOCKS_PER_SEC; }
private:
    double start, stop, elapsed;
};
```
5.3.4 Usage of const

const is a key word to specify to compilers that a certain variable is not to be modified, i.e. it is invariant. There are tons of reason for why you would want to restrict the usage of your variables to read only. Const do not solely apply to variables, but also to pointer.

```cpp
char *p;
const char *p; // pointer to constant data
char *const p; // constant pointer to non-constant data
const char *const p;
```

Think about the lines drawn at the asterisk. What is to the left is what is pointed to, what is to the right is the pointer itself.

```cpp
const Matrix& operator* (const Matrix& mat1, const Matrix& mat2);
Matrix A, B, C;
(A*B) = C;
```