Computational Physics Lectures: Programming aspects, object orientation in C++ and Fortran

Morten Hjorth-Jensen[1,2]

Department of Physics, University of Oslo[1]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University[2]

Mar 21, 2018

© 1999-2018, Morten Hjorth-Jensen. Released under CC Attribution-NonCommercial 4.0 license

Object orientation

Why object orientation?

- Three main topics: objects, class hierarchies and polymorphism
- The aim here is to be to be able to write a more general code which can easily be tailored to new situations.
- **Polymorphism** is a term used in software development to describe a variety of techniques employed by programmers to create flexible and reusable software components. The term is Greek and it loosely translates to "many forms". Strategy: try to single out the variables needed to describe a given system and those needed to describe a given solver.

Object orientation

In programming languages, a polymorphic object is an entity, such as a variable or a procedure, that can hold or operate on values of differing types during the program's execution. Because a polymorphic object can operate on a variety of values and types, it can also be used in a variety of programs, sometimes with little or no change by the programmer. The idea of write once, run many, also known as code reusability, is an important characteristic to the programming paradigm known as Object-Oriented Programming (OOP).

OOP describes an approach to programming where a program is viewed as a collection of interacting, but mostly independent software components. These software components are known as objects in OOP and they are typically implemented in a programming language as an entity that encapsulates both data and procedures.

Programming classes

In Fortran a vector or matrix start with 1, but it is easy to change a vector so that it starts with zero or even a negative number. If we have a double precision Fortran vector which starts at -10 and ends at 10, we could declare it as `REAL(KIND=8) :: vector(-10:10)`. Similarly, if we want to start at zero and end at 10 we could write `REAL(KIND=8) :: vector(0:10)`. We have also seen that Fortran allows us to write a matrix addition $\mathbf{A} = \mathbf{B} + \mathbf{C}$ as `A = B + C`. This means that we have overloaded the addition operator so that it translates this operation into two loops and an addition of two matrix elements $a_{ij} = b_{ij} + c_{ij}$.

Programming classes

The way the matrix addition is written is very close to the way we express this relation mathematically. The benefit for the programmer is that our code is easier to read. Furthermore, such a way of coding makes it more likely to spot eventual errors as well.

In Ansi C and C++ arrays start by default from $i = 0$. Moreover, if we wish to add two matrices we need to explicitely write out the two loops as

```
for(i=0 ; i < n ; i++) {
   for(j=0 ; j < n ; j++) {
      a[i][j]=b[i][j]+c[i][j]
```

Programming classes

However, the strength of C++ is the possibility to define new data types, tailored to some particular problem. Via new data types and overloading of operations such as addition and subtraction, we can easily define sets of operations and data types which allow us to write a matrix addition in exactly the same way as we would do in Fortran. We could also change the way we declare a C++ matrix elements a_{ij}, from $a[i][j]$ to say $a(i,j)$, as we would do in Fortran. Similarly, we could also change the default range from $0 : n-1$ to $1 : n$.

To achieve this we need to introduce two important entities in C++ programming, classes and templates.

Programming classes

The function and class declarations are fundamental concepts within C++. Functions are abstractions which encapsulate an algorithm or parts of it and perform specific tasks in a program. We have already met several examples on how to use functions. Classes can be defined as abstractions which encapsulate data and operations on these data. The data can be very complex data structures and the class can contain particular functions which operate on these data. Classes allow therefore for a higher level of abstraction in computing. The elements (or components) of the data type are the class data members, and the procedures are the class member functions.

Programming classes

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.

A simple example is that of a point in two dimensions. The data could be the x and y coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

Programming classes

C++ has a class complex in its standard template library (STL). The standard usage in a given function could then look like

```
// Program to calculate addition and multiplication of two complex num
using namespace std;
#include <iostream>
#include <cmath>
#include <complex>
int main()
{
  complex<double> x(6.1,8.2), y(0.5,1.3);
  // write out x+y
  cout << x + y << x*y  << endl;
  return 0;
```

where we add and multiply two complex numbers $x = 6.1 + \imath 8.2$ and $y = 0.5 + \imath 1.3$ with the obvious results $z = x + y = 6.6 + \imath 9.5$ and $z = x \cdot y = -7.61 + \imath 12.03$.

Programming classes

We proceed by splitting our task in three files.

We define first a header file complex.h which contains the declarations of the class. The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

```
#ifndef Complex_H
#define Complex_H
//   various include statements and definitions
#include <iostream>          // Standard ANSI-C++ include files
#include <new>
#include ....

class Complex
{...
definition of variables and their character
};
//   declarations of various functions used by the class
...
#endif
```

Programming classes

Next we provide a file complex.cpp where the code and algorithms of different functions (except inlined functions) declared within the class are written. The files `complex.h` and `complex.cpp` are normally placed in a directory with other classes and libraries we have defined.

Finally, we discuss here an example of a main program which uses this particular class. An example of a program which uses our complex class is given below. In particular we would like our class to perform tasks like declaring complex variables, writing out the real and imaginary part and performing algebraic operations such as adding or multiplying two complex numbers.

Programming classes

```
#include <iostream>
#include <cmath>
#include "mycomplex.h"
using namespace std;
int main()
{
  // we declare a complex variable a
  Complex a(0.1,1.3);
  // we declare  complex variables b and c
  Complex b(3.0), c(5.0,-2.3);
  // using the copy constructor to define a new complex variable z=c
  Complex z(c);   // Could use C++11 as z{c}
  // C++11 way of declaring compile with c++ -std=c++11
  Complex g{3,4};
  cout << g.Re() << " " << g.Im() << endl;
  //  we declare  a new complex variable d using the assignment operat
  Complex d = z;
  //  we declare  a new complex variable e using the assignment operat
  Complex e = d;
  d = a +c;
  e = a*c - d/b;
  // write out of the real and imaginary parts
  cout << "Re(d)=" << d.Re() << ", Im(d)=" << d.Im() << endl;
  cout << "Re(d)=" << e.Re() << ", Im(d)=" << e.Im() << endl;
  // write out absolute value
  cout << "Abs(d)=" << d.abs() << endl;
  cout << "Abs(e)=" << e.abs() << endl;
  return 0;
```

Programming classes

We include the header file complex.h and define four different complex variables. These are $a = 0.1 + i1.3$, $b = 3.0 + i0$ (note that if you don't define a value for the imaginary part this is set to zero), $c = 5.0 - i2.3$ and $d = b$. Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

Programming classes, the header file mycomplex.h

```
#ifndef Complex_H
#define Complex_H
//   various include statements and definitions
#include <iostream>          // Standard ANSI-C++ include files
#include <new>

//  My own Complex class
class Complex
{
private:
  double re, im; // real and imaginary part
public:
  // Constructors, default, extended and copy constructor
  Complex ();
  Complex (double re = 0.0, double im = 0.0);
  // copy constructor
  Complex(const Complex& c) : re(c.re), im(c.im) {}
  // destructor
  ~Complex () {}
  double   Re () const;        // T real_part = a.Re();
  double   Im () const;        // T imag_part = a.Im();
  double   abs () const;       // T m = a.abs(); // modulus
  // assignment operator c = a;
  Complex& operator= (const Complex& c);
  friend Complex operator+ (const Complex& a, const Complex& b);
  friend Complex  operator- (const Complex& a, const Complex& b);
  friend Complex operator* (const Complex& a, const Complex& b);
  friend Complex operator/ (const Complex& a, const Complex& b);
```

Programming classes, the cpp file mycomplex.cpp

```
#include <cmath>
#include "mycomplex.h"
// Constructors
Complex:: Complex () { re = im = 0.0; }
Complex:: Complex (double re_a, double im_a) {re = re_a; im = im_a; }
double Complex::  Re () const { return re; } //  getting the real part
double Complex::  Im () const { return im; }  //    and the imaginary p
double Complex:: abs () const { return sqrt(re*re + im*im); }
// All member functions not declared using the keyword static have a s
// called this that points to the instantiating object. When accessing
// the class where they are defined, the this pointer is implied and c
Complex& Complex:: operator= (const Complex& c)
{
  re = c.re;
  im = c.im;
  return *this;
}
Complex operator+ (const Complex& a, const Complex& b) { return Comple
Complex operator- (const Complex& a, const Complex& b) { return Comple

Complex operator* (const Complex& a, const Complex& b) {return Complex
Complex operator/ (const Complex& a, const Complex& b) {return Complex
```

Programming classes

The class is defined via the statement `class Complex`. We must first use the key word `class`, which in turn is followed by the user-defined variable name `Complex`. The body of the class, data and functions, is encapsulated within the parentheses `{...}`.

Programming classes

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use `protected` instead of `private`, then data and functions can be inherited outside the class.

Programming classes

The key word `public` means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class. The declaration `friend` means that stand-alone functions can work on privately declared variables of the type `(re, im)`. Data members of a class should be declared as private variables.

Programming classes

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type `Complex` and how this variable is initialized. We have chose three possibilities in the example above:

A declaration like `Complex c;` calls the member function `Complex()` which can have the following implementation

```
Complex:: Complex () { re = im = 0.0; }
```

meaning that it sets the real and imaginary parts to zero. Note the way a member function is defined. The constructor is the first function that is called when an object is instantiated.

Programming classes

Another possibility is

```
Complex:: Complex () {}
```

which means that there is no initialization of the real and imaginary parts. The drawback is that a given compiler can then assign random values to a given variable.

A call like `Complex a(0.1,1.3);` means that we could call the member function 'Complex(double, double)'as

```
Complex:: Complex (double re_a, double im_a) {
    re = re_a; im = im_a; }
```

Programming classes

The simplest member function are those we defined to extract the real and imaginary part of a variable. Here you have to recall that these are private data, that is they invisible for users of the class. We obtain a copy of these variables by defining the functions

```
double Complex:: Re () const { return re; }} //  getting the real part
double Complex:: Im () const { return im; }  //   and the imaginary pa
```

Note that we have introduced the declaration `const`. What does it mean? This declaration means that a variabale cannot be changed within a called function.

Programming classes

If we define a variable as `const double p = 3;` and then try to change its value, we will get an error when we compile our program. This means that constant arguments in functions cannot be changed.

```
// const arguments (in functions) cannot be changed:
void myfunc (const Complex& c)
{ c.re = 0.2; /* ILLEGAL!! compiler error... */  }
```

If we declare the function and try to change the value to 0.2, the compiler will complain by sending an error message.

Programming classes

If we define a function to compute the absolute value of complex variable like

```
double Complex:: abs ()  { return sqrt(re*re + im*im);}
```

without the constant declaration and define thereafter a function `myabs` as

```
double myabs (const Complex& c)
{ return c.abs(); }   // Not ok because c.abs() is not a const func.
```

the compiler would not allow the c.abs() call in myabs since `Complex::abs` is not a constant member function.

Programming classes

Constant functions cannot change the object's state. To avoid this we declare the function abs as

```
double Complex:: abs () const { return sqrt(re*re + im*im); }
```

Programming classes

C++ (and Fortran) allow for overloading of operators. That means we can define algebraic operations on for example vectors or any arbitrary object. As an example, a vector addition of the type $\mathbf{c} = \mathbf{a} + \mathbf{b}$ means that we need to write a small part of code with a for-loop over the dimension of the array. We would rather like to write this statement as `c = a+b;` as this makes the code much more readable and close to eventual equations we want to code. To achieve this we need to extend the definition of operators.

Programming classes

Let us study the declarations in our complex class. In our main function we have a statement like `d = b;`, which means that we call `d.operator= (b)` and we have defined a so-called assignment operator as a part of the class defined as

```
Complex& Complex:: operator= (const Complex& c)
{
   re = c.re;
   im = c.im;
   return *this;
}
```

Note: All member functions not declared using the keyword static have a special pointer called **this** that points to the instantiating object. When accessing members within the class where they are defined, the **this** pointer is implied and can be omitted

Programming classes

With this function, statements like `Complex d = b;` or `Complex d(b);` make a new object *d*, which becomes a copy of *b*. We can make simple implementations in terms of the assignment

```
Complex:: Complex (const Complex& c)
{ *this = c; }
```

which is a pointer to "this object", `*this` is the present object, so `*this = c;` means setting the present object equal to *c*, that is

```
this->operator= (c);.
```

Programming classes

The meaning of
the addition operator + for Complex objects is defined in the function `Complex operator+ (const Complex& a, const Complex& b); // a`
The compiler translates `c = a + b;` into `c = operator+ (a, b);`. Since this implies the call to function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop.

Programming classes

The solution to this is to inline functions. We discussed inlining in chapter 2 of the lecture notes. Inlining means that the function body is copied directly into the calling code, thus avoiding calling the function. Inlining is enabled by the inline keyword

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Inline functions, with complete bodies must be written in the header file complex.h. If they are included in the header file, we can skip the **inline** keyword.

Programming classes

Consider the case `c = a + b;` that is, `c.operator= (operator+ (a,b));` If `operator+`, `operator=` and the constructor `Complex(r,i)` all are inline functions, this transforms to

```
c.re = a.re + b.re;
c.im = a.im + b.im;
```

by the compiler, i.e., no function calls

Programming classes

The stand-alone function operator+ is a friend of the Complex class

```
class Complex
{
   ...
   friend Complex operator+ (const Complex& a, const Complex& b);
   ...
};
```

so it can read (and manipulate) the private data parts *re* and *im* via

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Programming classes

Since we do not need to alter the re and im variables, we can get the values by Re() and Im(), and there is no need to be a friend function

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.Re() + b.Re(), a.Im() + b.Im()); }
```

Programming classes

The multiplication functionality can now be extended to imaginary numbers by the following code

```
inline Complex operator* (const Complex& a, const Complex& b)
{
  return Complex(a.re*b.re - a.im*b.im, a.im*b.re + a.re*b.im);
```

It will be convenient to inline all functions used by this operator.

Programming classes

To inline the complete expression a*b;, the constructors and operator= must also be inlined. This can be achieved via the following piece of code

```
inline Complex:: Complex () { re = im = 0.0; }
inline Complex:: Complex (double re_, double im_)
{ ... }
inline Complex:: Complex (const Complex& c)
{ ... }
inline Complex:: operator= (const Complex& c)
{ ... }
```

Programming classes

```
// e, c, d are complex
e = c*d;
// first compiler translation:
e.operator= (operator* (c,d));
// result of nested inline functions
// operator=, operator*, Complex(double,double=0):
e.re = c.re*d.re - c.im*d.im;
e.im = c.im*d.re + c.re*d.im;
```

The definitions operator- and operator/ follow the same set up.

Programming classes

Finally, if we wish to write to file or another device a complex number using the simple syntax cout << c;, we obtain this by defining the effect of << for a Complex object as

```
ostream& operator<< (ostream& o, const Complex& c)
{ o << "(" << c.Re() << "," << c.Im() << ") "; return o;}
```

Programming classes, templates

What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace `double` with for example `int`.

C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword `template` followed by a list of template arguments enclosed in brackets.

Programming classes, the same class but now with templates

We can therefore make a more general class by rewriting our original example as

```
template<class T>
class Complex
{
private:
  T re, im; // real and imaginary part
public:
  Complex ();                                  // Complex c;
  Complex (T re_a, T im_a) {re = re_a; im = im_a; }; // Definition of
  // copy constructor
  Complex(const Complex& c) : re(c.re), im(c.im) {}
  // destructor
  ~Complex () {}                               // destructor
  Complex& operator= (const Complex& c)
    {
      re = c.re;
      im = c.im;
      return *this;
    }
  T   Re () const { return re; } //  getting the real part
  T   Im () const { return im; }          // T imag_part = a.Im();
  T   abs () const { return sqrt(re*re + im*im); }       // T m = a.ab
  friend Complex operator+ (const Complex& a, const Complex& b) { retu
  friend Complex  operator- (const Complex& a, const Complex& b) { ret
  friend Complex operator* (const Complex& a, const Complex& b) {retur
```

Programming classes

What it says is that `Complex` is a parameterized type with T as a parameter and T has to be a type such as double or float. The class complex is now a class template and we would define variables in a code as

```
Complex<double> a(10.0,5.1);
Complex<int> b(1,0);
```

Programming classes

Member functions of our class are defined by preceding the name of the function with the `template` keyword. Consider the function we defined as `Complex:: Complex (double re_a, double im_a)`. We would rewrite this function as

```
template<class T>
Complex<T>:: Complex (T re_a, T im_a)
{ re = re_a; im = im_a; }
```

The member functions are otherwise defined following ordinary member function definitions.

A matrix-vector class, first its usage

```
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "/Users/hjensen/Teaching/fys4411/programs/cgm/vectorclass.h"

using namespace  std;

  Vector ConjugateGradient(Matrix A, Vector b, Vector x0){
  int dim = x0.Dimension();
  const double tolerance = 1.0e-14;
  Vector x(dim),r(dim),v(dim),z(dim);
  double c,t,d;

  x = x0;
  r = b - A*x;
  v = r;
  c = dot(r,r);
  for(int i=0;i<dim;i++){
    if(sqrt(dot(v,v))<tolerance){
      cerr << "An error has occurred in ConjugateGradient: execution o
      break;
    }
    z = A*v;
    t = c/dot(v,z);
    x = x + t*v;
    r = r - t*z;
    d = dot(r,r);
```

A matrix-vector class, the class definitions themselves

```
#ifndef _vectorclass
#define _vectorclass


#include <cmath>
#include <iostream>
using namespace  std;


class Point;
class Vector;
class Matrix;


/*******************************/
/*        Point Class          */
/*******************************/

class Point{
 private:
  int   dimension;
  double *data;

 public:
  Point(int dim);
  Point(const Point& v);
  ~Point();
```

A matrix-vector class, and finally all its functions

```
#include "vectorclass.h"

Point::Point(int dim){
  dimension = dim;
  data = new double[dimension];

  for(int i=0;i<dimension;i++)
    data[i] = 0.0;
}


Point::Point(const Point &v){
  dimension = v.Dimension();
  data = new double[dimension];

  for(int i=0;i<dimension;i++)
    data[i] = v.data[i];
}


Point::~Point(){
  dimension = 0;
  delete[] data;
  data = NULL;
}


int Point::Dimension() const{
```

Programming classes, the vector only class but now with templates

```
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "vectortemplates.h"

using namespace  std;


//   Main function begins here
int main(int  argc, char * argv[]){
  int dim = 2;
  Vector<double> x(dim), b(dim);
  // Set values for x and y
  x(0) = x(1) = 10.0;
  Vector<double> y(dim);
  y(0) = 2.;
  y(1) = -2.;
  b = x+y;
  cout << "The vector b: " << endl;
  b.Print();
  cout << endl;
  cout << "The norm of  b: " << endl;
  cout<<  b.VectorNorm2() << endl;
  Vector<int> z(dim);
  z(0) = 2;
```

Programming classes, the vector only class but now with templates

```
//  Vector class with templates

#ifndef _vectorclass
#define _vectorclass
#include <cmath>
#include <iostream>
#include <sstream>
#include <iomanip>
#include <cstdlib>
using namespace std;

template<class T>
class Vector{
 private:
  int   dimension;
  T *data;

 public:
  Vector();
  Vector(int dim);
  Vector(const Vector<T>& v);
  ~Vector();
  int    Dimension() const;
  void   Normalize();
  T VectorNorm1();
  T VectorNorm2();
```

Virtual classes

Inheritance plays a central role in OO programming. When you derive a new class from another existing class using the **virtual** keyword, the derived class inherits data members and member functions of the already defined classes. A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the **virtual keyword**.

Virtual classes

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
```

Unit Testing

Unit Testing is the practice of testing the smallest testable parts, called units, of an application individually and independently to determine if they behave exactly as expected. Unit tests (short code fragments) are usually written such that they can be preformed at any time during the development to continually verify the behavior of the code. In this way, possible bugs will be identified early in the development cycle, making the debugging at later stage much easier.

Unit Testing, benefits

There are many benefits associated with Unit Testing, such as

- It increases confidence in changing and maintaining code. Big changes can be made to the code quickly, since the tests will ensure that everything still is working properly.
- Since the code needs to be modular to make Unit Testing possible, the code will be easier to reuse. This improves the code design.
- Debugging is easier, since when a test fails, only the latest changes need to be debugged.
  - Different parts of a project can be tested without the need to wait for the other parts to be available.
- A unit test can serve as a documentation on the functionality of a unit of the code.

Simple example of unit test

Look up the guide on how to install unit tests for c++ at course webpage. This is the version with classes.

```
#include <unittest++/UnitTest++.h>

class MyMultiplyClass{
public:
    double multiply(double x, double y) {
        return x * y;
    }
};

TEST(MyMath) {
    MyMultiplyClass my;
    CHECK_EQUAL(56, my.multiply(7,8));
}

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

Simple example of unit test

And without classes

```
#include <unittest++/UnitTest++.h>


double multiply(double x, double y) {
    return x * y;
}

TEST(MyMath) {
    CHECK_EQUAL(56, multiply(7,8));
}

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

For Fortran users, the link at
http://sourceforge.net/projects/fortranxunit/ contains a similar software for unit testing.