SCIENTIFIC COMPUTING CLASSES

Dr. Johnson

School of Mathematics

Semester 1 2012
Topics:

- Initial value problems for ODEs
- Euler’s Method for ODEs
- Stability

Aims - week 3:

- Be able to solve an initial value ODE problem
- Understand where truncation errors come from
- Be aware of the concept of stability in a numerical method
- Convert mathematical algorithms into code
Topics:
- Objects, data and functions together;
- Constructors and destructors;
- Access qualifiers;
- Operator overloading.

Aims - week 4:
- Understand concept of data and functions as members of an object;
- Create and use simple objects (Points, Intergrand function);
- Overload operators to enable easy code writing.
The C++ **class** is used to define an object.
An object is collection of data, along with functions that act on that data.
The data and functions may have restricted access.
We declare classes in the following way:

```cpp
class class_name {
private data and functions
access specifier: 
data and functions :

access specifier: 
data and functions
} object list;
```
class Point{ // class name
    public:
        int x, y; // public access data
        double distance_to_origin(void)
            { return sqrt(x*x + y*y); } // finish a class with a semi-colon
**EXAMPLE: POINT**

- We can declare a new `Point` in the main code with the command
  
  ```
  Point a, b, c;
  ```

- and access data or functions with a dot:
  
  ```
  a.x = 1; a.y = 1;
  cout << " Distance to origin ";
  cout << a.distance_to_origin() << endl;
  ```
In the previous example there is no restrictions on access to data, everything is declared as **public**.

There are three types of access-specifiers:
- private - no access outside the class;
- protected - access inside derived classes;
- public - full access outside the class;
In the previous example there is no restrictions on access to data, everything is declared as **public**.

There are three types of access-specifiers:
- private - no access outside the class;
- protected - access inside derived classes;
- public - full access outside the class;

We can think of this as read and write access to data.
PROTECTING YOUR DATA

- There are many situations where you will not want your data to be changed accidentally.
- Protecting data can help stop bugs in your code.

```cpp
class Point{  // class name
    int x,y;  // private access data

public:
    void set_x(int x_){x=x_;}  // write to x
    void set_y(int y_){y=y_;}  // write to y

    // const function members read only
    int get_x(void) const {return x;}
    int get_y(void) const {return y;}
    double distance_to_origin(void) const
    { return sqrt(x*x + y*y); }
};  // finish a class with a semi-colon
```
When a new piece of data is created, C++ allows for defaults commands to be carried out.

- This allows us to initialise the data default values...
- or pass in parameters as arguments...
- or run functions to act on the data.
CONSTRUCTING YOUR DATA

- When a new piece of data is created, C++ allows for defaults commands to be carried out.
- This allows us to initialise the data default values...
- or pass in parameters as arguments...
- or run functions to act on the data.
- The function run on initialisation is a constructor.
- The function run on deletion is a deconstructor.

CONSTRUCTOR SYNTAX:

class_name(arguments):initialisation list
{ function calls }
CONSTRUCTING YOUR DATA

class Point{ // class name
    int x,y; // private access data
public:
    void set_x(int x_){x=x_;}
    void set_y(int y_){y=y_;}
    int get_x(void) const {return x;}
    int get_y(void) const {return y;}
    Point():x(0),y(0) {} // default constructor
    Point(int x_,int y_):x(x_),y(y_) {} 
    ~Point() {} 
    double distance_to_origin(void) const 
    {return sqrt(x*x + y*y);}
}; // finish a class with a semi-colon
Deconstructors are used to prevent memory leaks.
This shouldn’t be a problem unless you use pointers for dynamic storage.
Using std libraries to store data should mean that you don’t need to use deconstructors.
POINTING TO AN OBJECT

- We can create a pointer to an object in the same way as standard data types.

```
Point a(2,5),*ptr_a;    // we have a Point and a pointer to Point
```

- There is special syntax `->` to access data via the pointer

```
ptr_a=&a;    // ptr_a is the location of a
ptr_a->get_x();  // access data from a
```

- It is best to always pass an object to a function as a reference:

```
a_function(const Point& a,const Point& b)
{    // some function on a and b    }
```
OVERLOADING A FUNCTION

- We use overloading to enable a function with the same name to act differently in different situations.
- One example is a simple add function:
  1. Take in two ints and return an int

```cpp
int add(int a, int b) { return (a + b); }
```

  2. Take in two Points and return a Point

```cpp
Point add(const Point& a, const Point& b) {
    Point temp;
    temp.set_x(a.get_x() + b.get_x());
    temp.set_y(a.get_y() + b.get_y());
    return (temp); // Note we must return a Point
}
```
**Overloading an Operator**

- For some of your classes, addition, multiplication etc may mean something.
- For instance, we may want to be able to write:

```
Point a(2,1), b(3,4), c;
c = a + b;
```
For some of your classes, addition, multiplication etc may mean something.
For instance, we may want to be able to write:

```cpp
Point a(2,1), b(3,4), c;
c = a + b;
```

C++ allows us to **overload** operators.
So for the above to work we must include the function:

```cpp
Point operator+(const Point& a, const Point& b) {
    Point temp;
    temp.set_x(a.get_x() + b.get_x());
    temp.set_y(a.get_y() + b.get_y());
    return temp;
}
```
Another useful operator is the () operator
This can be used to make function calls, for example:

```cpp
class Intergrand{
   public:
      double operator()(double x) const {
         return 1 + x*x;
      }
};

Can be used like this

double u,v=1.;
Intergrand f;
u = 1. + f(v) + f(10.);
```
We saw that there were problems if the arguments to a function need to be changed.
Good codes will provide **generic** interfaces.
For example, the trapezium rule may be written:

```c
double trapezium(double a, double b,
                  int n, const Integrand& f) {
    double h = (b - a) / n;
    double sum = f(a) / 2.;
    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        sum = sum + f(x);
    }
    return sum * h;
}
```
Topics:
- Objects, data and functions together;
- Constructors and destructors;
- Access qualifiers;
- Operator overloading.

Aims - week 4:
- Understand concept of data and functions as members of an object;
- Create and use simple objects (Points, Intergrand function);
- Overload operators to enable easy code writing.