SCIENTIFIC COMPUTING
INHERITANCE

Dr. Johnson
School of Mathematics
Semester 1 2012
Last Week...

- **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.
This Week…

Topics:
- Inheritance - runtime polymorphism
- Templates - efficient coding
- Error handling and analysis

Aims - week 5:
- Be able to use inheritance to overload function operators
- Use templates to create efficient codes and minimise code reuse
- Error checking within your code
C++ allows the creation of new classes that inherit from existing classes.

We say the new class is derived from a base class.

The general syntax for creating derived classes is:

```cpp
class class_name : access-specifier base_class_name;
```
Here Circle inherits all data and functions from Point.

class Circle: public Point{
    double radius; // inherits x and y
public:
    void set_r(int r){radius=r;}
    int get_r(void){return radius;}
    Circle(int x_,int y_,int r_):Point(x_,y_),r(r_){} // finish a class with a semi-colon
Does the function `distance_to_origin` still apply?
OVERLOADING FUNCTIONS

- Does the function `distance_to_origin` still apply?
- We can declare a new function in `Circle`, with the same name, to override the function in `Point`. 

```c
double distance_to_origin(void)
{return max(0.,sqrt(x*x + y*y)-radius);};
```
Does the function `distance_to_origin` still apply?

We can declare a new function in `Circle`, with the same name, to override the function in `Point`.

We add the function in `Circle`:

```c
double distance_to_origin(void)
{
    return max(0., sqrt(x*x + y*y) - radius);
}
```
**OVERLOADING FUNCTIONS**

- Does the function `distance_to_origin` still apply?
- We can declare a new function in `Circle`, with the same name, to override the function in `Point`.
- We add the function in `Circle`:

  ```
  double distance_to_origin(void)
  {return max(0.,sqrt(x*x + y*y)-radius);};
  ```

- We need to change the access to `x` and `y` in `Point` to protected, to allow read/write access for `Circle`.
Now because Circle is a Point, a pointer to Point will also be a pointer to Circle
Now because Circle is a Point, a pointer to Point will also be a pointer to Circle

Then we may write

```cpp
Point *ptr_a; // a pointer to Point
Circle b(2,5,2); // a circle
ptr_a = &b; // set pointer to circle b
cout << " distance to origin is ";
ptr_a->distance_to_origin();
```
**Pointers and Inheritance**

- Now because *Circle* is a *Point*, a pointer to *Point* will also be a pointer to *Circle*
- Then we may write

```cpp
Point *ptr_a;  // a pointer to Point
Circle b(2, 5, 2); // a circle
ptr_a = &b;  // set pointer to circle b
cout « " distance to origin is ";
ptr_a->distance_to_origin();
```

- The output is
  
  distance to origin is 5.38516
Because we use a pointer, the original distance function is called.

Get round this with a *virtual* function.
Because we use a pointer, the original distance function is called.

Get round this with a virtual function.

Simply change the appropriate line in the class Point to

```cpp
virtual double distance_to_origin(void) {
    return sqrt(x*x + y*y);
};
```

and the output will now be
distance to origin is 3.38516
Sometimes we may wish to declare a base class containing prototypes for a function, with no implementation.

The base class can then be used as a generic interface between different methods.

A virtual function with no implementation is called a pure virtual function.

This means we cannot:

- Create an instance of the base class
- Create an instance of a derived class unless the pure function is implemented
The syntax is:

```c
virtual return_type function
  (arg_type arg,...)=0;
```
An Example

- Lets go back to the Integrand, and make it virtual:

```cpp
class Intergrand{
public:
    virtual double operator()(double x) = 0;
};
```

- Nothing else in your codes needs to be changed.
**An Example**

- Lets go back to the `Integrand`, and make it virtual:

```cpp
class Intergrand{
public:
    virtual double operator()(double x)=0;
};
```

- Nothing else in your codes needs to be changed.
- Need a derived class to implement the function call...
- This could be:

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

Then in the main code we can have numerous calls to the integrating function,

and pass any class that inherits Intergrand as an argument

OnePlusXX f;
AnotherFunc g;
cout << " I = " << I.trapezium(f) << endl;
cout << " I = " << I.trapezium(g) << endl;

This is runtime polymorphism.
AN EXAMPLE

- Then in the main code we can have numerous calls to the integrating function,
- and pass any class that inherits Intergrand as an argument

OnePlusXX f;
AnotherFunc g;
cout << " I = " << I.trapezium(f) << endl;
cout << " I = " << I.trapezium(g) << endl;

- This is runtime polymorphism.
- It can be very useful...
- but also very inefficient!
The most efficient way to calculate short (one line) functions is to write them out in full.

This means that the integrate function from examples 3 would look like:
// store running sum
double sum = \exp(-(x*x)/(4.*\kappa)) / \sqrt(2.*\Pi*(1+x))/2.;

for(int i=1;i<n;i++)
{
    x = i*h;
    sum = sum + \exp(-(x*x)/(4.*\kappa)) / \sqrt(2.*\Pi*(1+x));
}

sum = sum + \exp(-(x*x)/(4.*\kappa)) / \sqrt(2.*\Pi*(1+x))/2.;

return sum*h;
The C++ compiler is able to **rewrite your code** exchanging function calls to write them out in full.

This is called inlining a function.

To tell the compiler to inline a function, write **inline** before the prototype.

Only do this if the function can be written on one line.

Optimisations may do this automatically anyway...
Function Templates

- Using virtual function causes a small overhead while finding the function at run time
- An alternative is to use templates, which can be written like

```cpp
// integrate function
template<class T>
double trapezium(double a, double b, int n, T f) {
    // stepsize
    double h = (b-a)/n;
    // store running sum
    double sum = (f(a) + f(b))/2.;
    for(int i=1; i<n; i++) sum += f(a + i*h);
    return sum*h;
}
```
Function Templates

- The C++ compiler generates new code for each function call with a different class.
- This allows the compiler to rewrite the functions inline.
- This is most useful when the algorithms in the function are short and generic.
- They must not require different behaviour for different classes.

You don't want it to generate too much code!!! You also have to be very careful about which bits of code go into a .cpp file if you split your project into different files...
The C++ compiler generate new code for each function call with a different class.

This allows the compiler to rewrite the functions inline.

This is most useful when the algorithms in the function are short and generic.

They must not require different behaviour for different classes.

You don’t want it to generate too much code!!!

You also have to be very careful about which bits of code go into a .cpp file if you split your project into different files...
We can also use the compiler's ability to generate code to write generic code for different data types.

For instance

```cpp
// generic Point
template<class T>
class Point
{
    public:
        T x, y;
        T distance_to_origin(){sqrt(x*x + y*y)}
};
```
We declare an instance with:

```cpp
// a double Point
Point<double> a;
```
We declare an instance with:

```cpp
// a double Point
Point<double> a;
```

- This is useful if we want to run codes in single or double precision (examples 2)
- This code will only compile if the operators + and * are defined for the class T
Error checking is something that we may wish to turn on and off
Checking for errors takes time, making programs slow
One way to turn it on and off is to use preprocessor flags...

```c
// check if denominator is zero
#ifdef DEBUG
if(g(x)==0.) // g=0 then throw exception
{ cout << " denominator = 0 "; throw; }
#endif
return f(x)/g(x);
```

The two lines of code here will only be compiled if `DEBUG` has been defined in the code...

```c
#define DEBUG
```
Topics:
- Inheritance - runtime polymorphism
- Templates - efficient coding
- Error handling and analysis

Aims - week 5:
- Be able to use inheritance to overload function operators
- Use templates to create efficient codes and minimise code reuse
- Error checking within your code