3F6 - Software Engineering and Design

Handout 2

Classes and C++ (I)

With Markup

Ed Rosten

Contents

1. Program Design Using Objects
2. Classes in C++
3. Constructors and Destructors
4. Operator Overloading
5. Data Types
6. Class Derivation

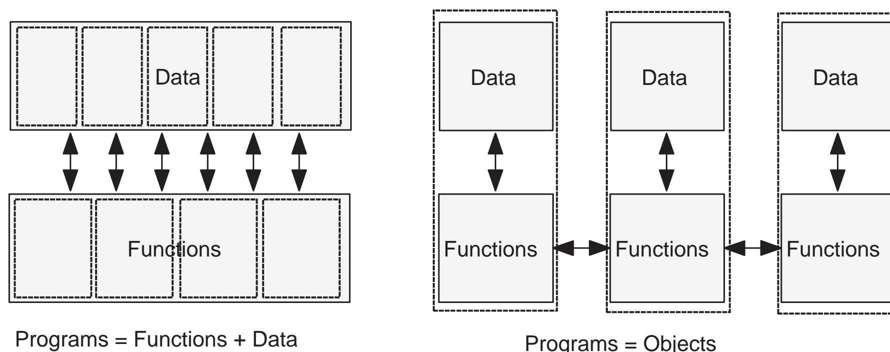
Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`

Program Design Using Objects

Object-oriented programming requires a new way of thinking about program design:

1. What classes of objects are present in the problem?
2. What services does each class have to provide?
3. What should happen when a service is requested of an object?

Compare this to procedural programming: instead of breaking the problem up into data + algorithms, we first break it up into objects (which contain data and algorithms and use these to supply services).



Objects provide abstractions. An object can be used without any knowledge of how it works. This allows software to be built from components - just like other forms of engineering system.

Implementing Classes in C++

Earlier we gave the definition of the Date class as:

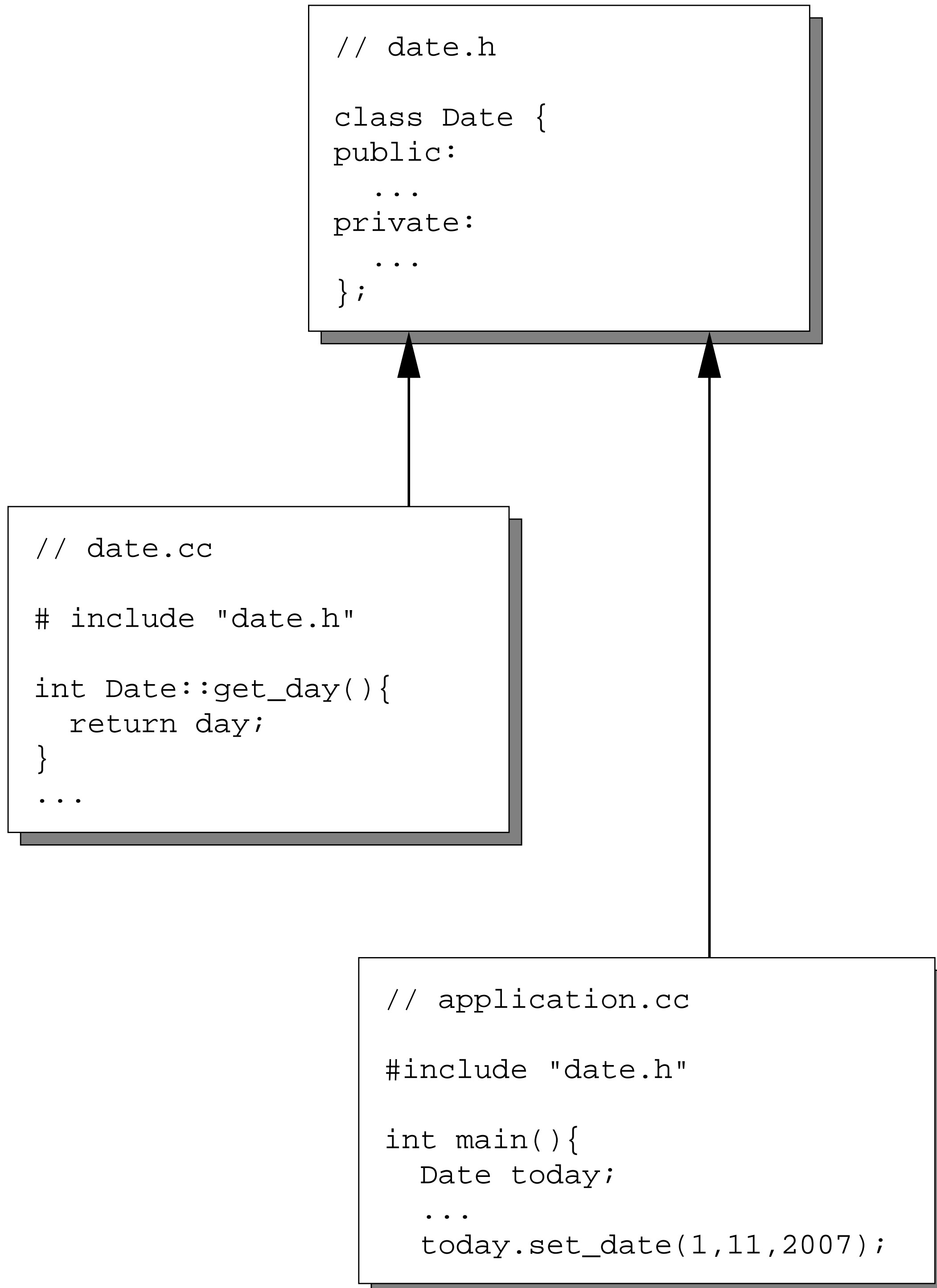
```
class Date {
public:
  // access functions
  int get_day();
  int get_month();
  int get_year();

  // function to set the date
  void set_date(int d, int m, int y);

private:
  int  day;
  int  month;
  int  year;
};
```

This specifies the private data contained in the Date class and the public interface that it provides. The latter consists of a set of class member functions (aka *operations* or *methods*) that may be called to operate on instantiated class objects.

Member functions are defined like any other function except that their names are prepended with the class name and within the function, all of the class's data elements are directly accessible.

```
int Date::get_day(){
  return day;
}

int Date::get_month(){
  return month;
}

void Date::set_date(int d, int m, int y){
  // check month is 1..12
  if(m < 1 || m > 12) Raise_Error();

  // check day is 1..31
  if(d < 1 || d > 31)  Raise_Error();

  // if April, May, September, November
  if(m==4 || m==6 || m==9 || m==11){
    if(d > 30) Raise_Error();
  }

  // if February use leap year rules
  if(m==2){
    if((y%4==0)^(y%100==0)^(y%400==0)){
      if(d > 29)  Raise_Error();
    } else {
      if(d > 28) Raise_Error();
    }
  }
  // if we got here d,m,y are OK
  day   = d;
  month = m;
  year  = y;
}
```

Putting an application together



Changing the Implementation

Classes allow the interface and implementation of a system component to be decoupled.

Debugging object-oriented programs should be easier. If something goes wrong with the data in a class, the culprit must lie in the class member functions.

There is also another benefit: since the implementation details are hidden, we can change the internal data representation of a class while maintaining the same public interface to the rest of the program.

Eg. we might want to change the implementation of the date class:

```
class Date {
public:
  // access functions
  int get_day();
  etc as before

private:
  int  num_days_since_1_1_1900;
};
```

Programs which use this class would never know that the implementation had been changed.

Also, additional interface functions can be added to enhance functionality *without* affecting existing programs.

Constructors and Destructors

Not all of our problems have been solved yet. What happens if a programmer writes:

```
Image img;                    // img is not initialised
img.set_pixel(100,100,255); // write to a random area of memory
```

The programmer has used the object `img` before its values have been initialised.

Constructors provide a mechanism to make sure that objects have appropriate values that meet the invariants from the moment they are created.

A constructor is executed when the object is created and it can be given parameters if required. For example

```
Image img(200,200);         // img is 200 x 200
img.set_pixel(100,100,255); // now safe
```

Similarly, when we are finished with an image, we have to make sure that the allocated memory is released. This is achieved using a *destructor*.

```
class Image {
public:

  Image(int w, int h);
  ~Image();

  // functions to access the data
  int get_width();
  int get_height();
  char get_pixel(int x, int y);
  void set_pixel(int x, int y, char value);
  void save(char * filename);
  etc

private:
  int width;
  int height;
  char *pixels;
};
```

```
Image::Image(int w, int h) {
  width = w; height = h;
  pixels = new char[w*h];
}
Image::~Image() {
  delete[] pixels;
}
```

A Complex Number Class

In engineering applications, it is often useful to deal with complex numbers. These could be implemented in a class as follows:

```
class Complex {
public:
  // constructor
  Complex (float r, float i);

  // arithmetic
  Complex add(Complex rhs);
  Complex subtract(Complex rhs);
  Complex multiply(Complex rhs);
  Complex divide(Complex rhs);

  void set_value(Complex c);

  // leave the data members public in this example
  float real;
  float imaj;
};
```

This provides all the basic operations and we can use this to write code like:

```
Complex c1(1.5,0);
Complex c2(1.7,1.9);
Complex c3(0,0);
Complex c4(0,0);

c3.set_value(c1.multiply(c2));  // c3=c1*c2
c4.set_value(c1.add(c3));       // c4=c1+c3

cout << "answer is" << c4.real << "+"
     << c4.imag << "i" << endl;
```

Operator Overloading

It would be much more convenient if we could write the code using standard arithmetic symbols with complex numbers so that the code looks just the same as if we had been using `float`s.

This can be done in C++ using a feature called *operator overloading.*

```
class Complex {
public:
  // Constructor
  Complex(float r, float i);

  //arithmetic
  Complex operator+(Complex rhs);
  Complex operator-(Complex rhs);
  Complex operator*(Complex rhs);
  Complex operator/(Complex rhs);

  // to set the values
  Complex operator=(Complex rhs);

  // data members
  float real;
  float imag;
};
```

This new version of the Complex class means we can now write

```
c3 = c1*c2;
c4 = c1+c3;
```

instead of

```
c3.set_values(c1.multiply(c2));
c4.set_values(c1.add(c3));
```

It is also possible to write code to enable the use of

```
cout << c4 << endl;
```

This would require code to define the operator `<<` in the context "`ostream << complex`" such as

```
ostream & operator << (ostream& os, Complex& c) {
   os << c.real << "+" << c.imag << "i";
   return os;
}
```

User Defined Data Types

Superficially, all we have achieved by using operator overloading is that our programs are a bit easier to read. For that reason this is sometimes referred to as *syntactic sugar*.

But we have also achieved something further.

If we have some old code that uses floats, that code can be reused with Complex. For example recall the 1A exercise to solve simultaneous equations:

```
// coeffs of the eqns of form ax + by =c
float a1, b1, c1, a2, b2, c2, x, y;
...
// find the solution using Cramer's rule
x = (b2*c1-b1*c2)/(a1*b2-a2*b1);
y = (a1*c2-a2*c1)/(a1*b2-a2*b1);
```

This code can be converted to find solutions to equations with complex coefficients just by changing the word `float` to `Complex`.

This is possible because we made sure our improved class Complex has exactly the same interface as all other kinds of number in C++.

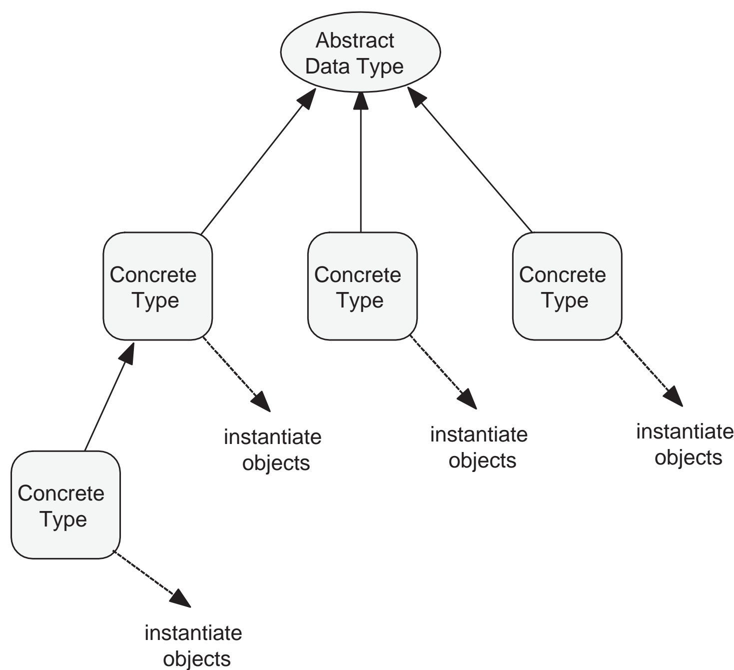
We have made Complex a fully-fledged data-type with an interface consistent with built-in numerical types.

Abstract Data Types

The concept *number* has a well defined interface that can be implemented by several different types of number (e.g. int, float, double, Complex).

Thus number is an *abstract data type* with many possible realisations. We can say that float **is-a** number and Complex **is-a** number.

This is a weak kind of **is-a** relationship. Object oriented languages like C++ also allow for a much stronger expression of this relationship using *class derivation*.



Class Derivation

Suppose we are programming a video processing application such as an MPEG4 codec for a video camera or an industrial inspection system.

Problem:
For each image captured from the video camera by the system, we need to know the time when each image was captured. Unfortunately, there is no place to store this information in our existing `Image` class.

Solution 1:
Add a timestamp data member to class `Image`.

BUT our image class would quickly become cluttered with data members which were only used in certain circumstances.

Solution 2:
Create a new class `VideoFrame` with a copy of all the `Image` data members and functions and with a timestamp as well.

BUT now we can't use existing code that works with `Image`.

Solution 3:
Derive class `VideoFrame` from class `Image`.

```
class Image {
public:
  Image(int w, int h);
  ~Image();
  int get_width();
  int get_height();
  char get_pixel(int x, int y);
  void set_pixel(int x, int y, char val);
    void save(char * filename);
  etc
private:
  int width;
  int height;
  char *pixels;
};

class VideoFrame : public Image {
public:
  VideoFrame(int w, int h, int t);
  int get_timestamp();
private:
  int timestamp;
};
```

Using Derived Classes

We can create a VideoFrame object:

```
VideoFrame frame(200,200,10);
```

And then use all the functions in the image class:

```
frame.save("image.pgm");
```

We can also use functions in the derived class:

```
int t = frame.get_timestamp();
```

We can supply a reference (or pointer) to the derived class when a reference (or pointer) to the base class is needed:

```
// function prototype
void blur (Image& im);

blur(frame);
```

Similarly if a function returns a reference (or pointer) to the derived class, it can be treated as a reference (or pointer) to the base class.

```
// function prototype
VideoFrame *get_frame_from_video_card();

Image* imptr = get_frame_from_video_card();
```

Class Derivation

By deriving `VideoFrame` from `Image`, the programmer is saying that:

1. A `VideoFrame` does everything an `Image` does
2. It can be used wherever an `Image` is expected
3. It can do some new things as well

Items 1 and 2 mean that `VideoFrame` **is-an** `Image`. This is the strong kind of is-a relationship because code designed to work with `Image`s can be reused for `VideoFrame`s without changing anything (or even recompiling).

We say that `Image` is the base class and `VideoFrame` is the derived class. These are also referred to as the superclass and the subclass respectively.

But note that you can not get a timestamp from an Image class.

```
VideoFrame *vfp = new VideoFrame(200,200,10);
Image *ip = vfp;   // safe since a VideoFrame
                   // can do anything that an image can do
ip->set_pixel(10,20,0);   // ok
t = ip->get_timestamp();  // illegal even though ip
                          // really points to a VideoFrame
```

How Class Derivation is Implemented