3F6 - Software Engineering and Design

Handout 3

Classes and C++ (II)

With Markup

Ed Rosten

Contents

1. A Drawing Editor Example
2. Polymorphism
3. Virtual Functions
4. Exception Handling
5. Templates
6. The Standard Template Library

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

A Drawing Editor Example

Imagine a very simple drawing editor where drawings consist only of rectangles represented by `class Rectangle`.

```
Rectangle * my_rectangles[N];

Ellipse * my_ellipses[N];

// display the drawing
for(int i=0; i<num_rectangles; i++){
  my_rectangles[i]->draw();
}

for(int i=0; i<num_ellipses; i++){
  my_ellipses[i]->draw();
}
```

If we now want to extend our drawing editor to allow ellipses as well, we could create `class Ellipse` and add a list of ellipses to the list of rectangles.

Because `Ellipse` has the same interface as `Rectangle`, we can easily extend the program by cutting and pasting.

But this quickly becomes unmanageable. (eg think of all the shapes in Powerpoint).

```
class Rectangle {
public:
  void draw();
  void move(int dx, int dy);
  void fill(int colour);

private:
  int left;
  int right;
  int top;
  int bottom;
};


class Ellipse {
public:
  void draw();
  void move(int dx, int dy);
  void fill(int colour);

private:
  int x_centre;
  int y_centre;
  int width;
  int height;
};
```

Polymorphism

When we created `class Ellipse` to look like `class Rectangle`, we were using a weak kind of **is-a** relationship. Both `Rectangle` and `Ellipse` provide the same interface for drawing, moving and filling shapes.

We can formalise this by using *class derivation* as in the `VideoFrame` example. However, here we go further.

First, we define an abstract data type to represent the abstract concept `Shape`.

```
class Shape {
public:
  virtual void draw()=0;
  virtual void move(int dx, int dy)=0;
  virtual void fill(int colour)=0;
};
```

This is called the *base* class and its functions are *pure virtual functions*. They have no implementation bodies, instead they are placeholders for the concrete functions that will be defined in each class derived from `Shape`.

We then derive `Rectangle`, `Ellipse` and any other shape that we want from this base class. Note "virtual" means that the function can be redefined in a derived class. "=0" means that no implementation body will be provided.

```
class Rectangle : public Shape {
public:
  virtual void draw();
  virtual void move(int dx, int dy);
  virtual void fill(int colour);
private:
  // as before
};
class Ellipse : public Shape {
public:
  virtual void draw();
  virtual void move(int dx, int dy);
  virtual void fill(int colour);
private:
  // as before
};
```

These are called *subclasses*, *derived types* or *derived classes*.

We can now declare pointers of type `Shape` and use them to point to these derived classes.

```
Shape *p;
Rectangle *r = new Rectangle();
Ellipse   *e = new Ellipse();

// assign either r or e to p

p->draw();   // draw whatever p points to
```

This is called *polymorphism*.

Using Virtual Functions

We can now simplify the code for drawing the rectangles and ellipses into a single loop:

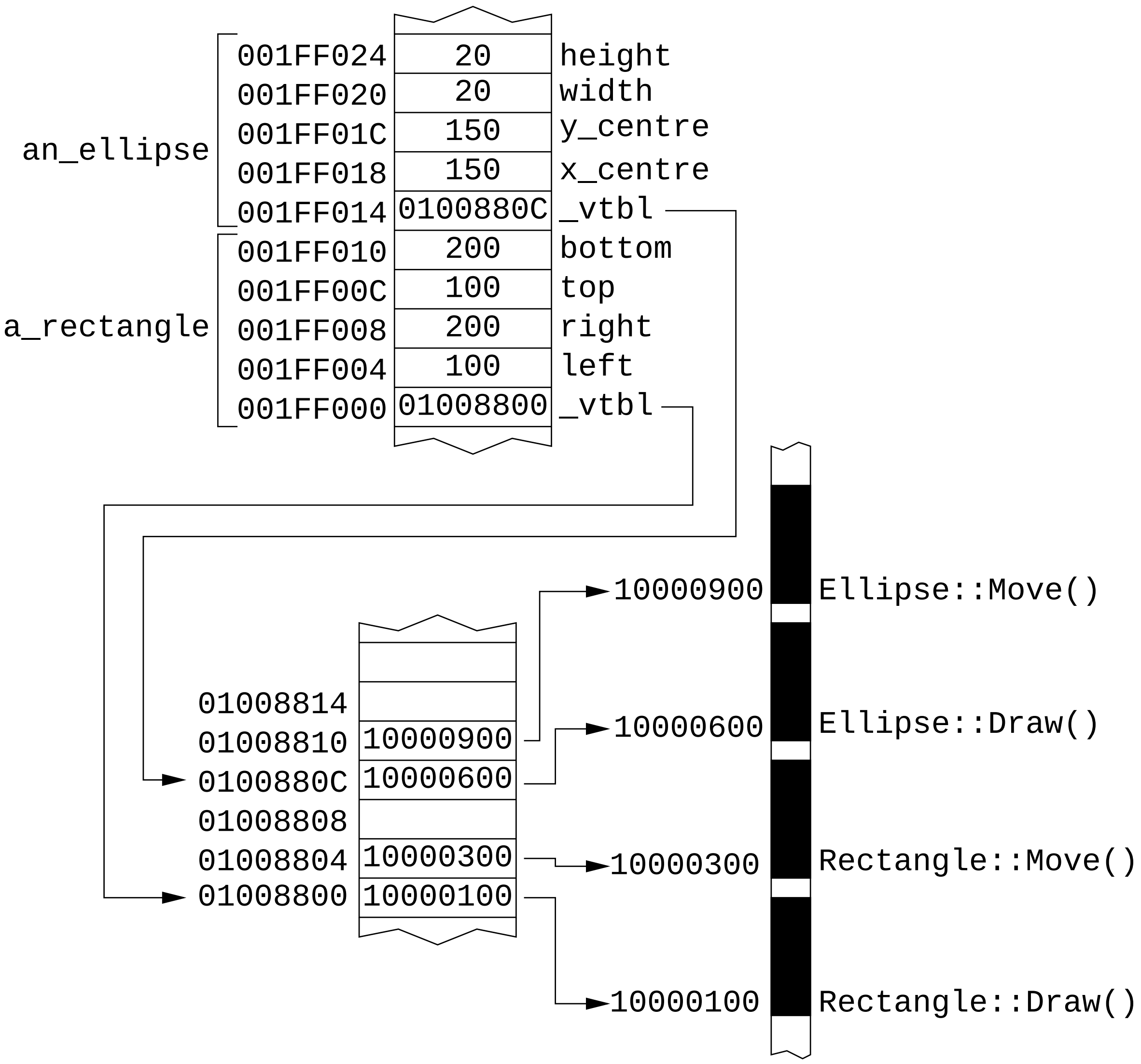
```
Shape* my_shapes[MAX_NUM_SHAPES];
int num_shapes = 0;

void AddShape(Shape *s) {
  my_shapes[num_shapes++] = s;
}

void DrawShapes() {
  for(int i=0; i<num_shapes; i++){
    my_shapes[i]->draw();
  }
}
```

A further advantage of this approach is that this code doesn't change when we introduce further shapes into our drawing editor with their own virtual functions. The line of code

```
my_shapes[i]->draw();
```

will automatically detect new subclasses of `Shape` and call the draw function that has been supplied by the programmer of the subclass.

How Virtual Functions are Implemented



Note that we use a double indirection to avoid duplicating a large set of pointers for every instance of the object.

Reporting Errors

Errors in the operation of a program are inevitable and robust software must be able to detect and handle them appropriately.

Consider the following:

```
int main() {
   ...
   int traderr = MyTrader();
   if(traderr != OKAY) {
      // Sort out the error
   }
   ...
}
//-------------------------------------------------
int MyTrader() {
   ...
   float price;
   int ret = TE_GetPrice(day, price);
   if(ret != OKAY)  return ret; // Pass the error back
   ...
   return OKAY; // Signal success
}
//-------------------------------------------------
int TE_GetPrice(int day, float& price) {
   ...
   if (!Valid(day)) return BAD_DAY;
   ...
   return OKAY;
}
```

Here return codes are used to signal errors.

Exceptions

Errors such as the above represent *exceptions* to the normal program flow.

Handling exceptions via return codes has a number of disadvantages:

- Extra code needs to be inserted in each function to pass the errors back.
- If one function fails to check for errors and pass them back, the errors will not get handled.
- The extra error checking obscures the main function of the code, making it difficult to understand.
- Error recovery code becomes intertwined with the normal operation code.
- Functions cannot use return values for normal purposes.

Fortunately, there is a better way.

Exception handling

In a structured approach to exception handling, exceptions are represented by class objects. The fields of the class can be used to record all relevant error information.

For example,

```
class TradingErr {
   TradingErr(ErrType ee, Time tt) {e=ee; t=tt;}
   ErrType e;
   Time t;
};
```

Exception handling now consists of two stages:

a) Raising the Error

```
throw TradingErr(BAD_DAY,TimeNow());
```

The effect of a **`throw`** is to exit the current procedure. If the calling procedure has a handler, it is invoked. Otherwise, the process repeats.

Note that the class object is constructed before it is thrown.

b) Handling the Error

At an appropriate point in the procedure call hiearchy, a `catch` statement is inserted to catch and handle a specific exception.

```
void SomeFunction () {
try {
   // regular code
}
// exception handler
catch (TradingErr x) {
    if (x.e == ... ) ....
}
```

Note

- using exception handlers, the code is much cleaner because the error handling parts are clearly separated from the regular code.
- a handler can throw the exception again allowing some errors to be trapped and repaired and others to be propagated.

The Trading example again:

```
int main() {

  try {
     ...
     MyTrader();
     ...
  }
  catch (TradingError x) {
     ReportError(x.e, x.t);
  }

}
//--------------------------------------------------
void MyTrader() {
   ....
   float price = TE_GetPrice(day);
   ....
}
//--------------------------------------------------
float TE_GetPrice(int day) {


   ...
   if (!Valid(day))
      throw TradingErr(BAD_DAY,TimeNow());
   ...


}
```

Templates

The `Image` class we defined earlier can only store greyscale images because

- The `pixels` data member is of type `char *`.
- The `get_pixel()` function returns a `char`.
- The `set_pixel()` function takes a `char` as its third argument.

If we want to handle colour images (where every pixel is of type `Colour`) or images where every pixel is an `int`, we have to either

(a) create new classes for each specific pixel type or

(b) define a polymorphic class for pixels.

but (a) is tedious and (b) is very inefficient.

What is needed is a specific mechanism to parameterise types.

In C++ this is achieved using *Templates*

- Templates allows a type to vary without virtual functions
- Templates provide *compile-time* polymorphism
- You can also template with numeric constants e.g. fixed sized matrices.
- Templates are Turing complete!

```
template <class T>     // T is a generic type
class Image {
public:
  Image(int w, int h);
  ~Image();

  int get_width();
  int get_height();

  T get_pixel(int x, int y);
  void set_pixel(int x, int y, T val);

  void load(char *filename);
  void save(char *filename);

private:
  int width;
  int height;
  T *pixels;          // array of T's
};
```

We can use template classes in the following way:

```
// create a greyscale image where T = char
Image<char> grey_im(200,200);}

// create a colour image  where T = Colour
Image<Colour> colour_im(200,200);}

...
grey_im.set_pixel(100,100,255);

Colour col = colour_im.get_pixel(100,100);
...
etc
```

The C++ Standard Template Library

The most common use for template programming is to create container classes. The standard template library (STL) contains many of these. For example

```
template<class T>
class vector {...};
```

allows the user to create arrays that work in a similar way to C style arrays but with many extra features such as dynamic sizing, array bound checking, etc.

```
vector<int> primes(100);
primes[0]=2;
primes[1]=3;
```

This code creates an array of 100 integers and stores the integer values 2 and 3 into the first two slots.

`vector` supports other functions as well, for example:

```
primes.push_back(547);
```

increases the size of the array by one and sets the last entry to 547.

Lists and Iterators

An alternative to the `vector` container is the `list`:

```
template<class T>
class list {...};
```

which allows the user to create linked lists:

```
list<int> lprimes;
lprimes.push_back(2);
lprimes.push_back(3);
```

This creates an initially empty linked list of integers and then pushes the integers 2 and 3 onto the list.

It is possible to iterate through the elements of a linked list by declaring an *iterator* type.

An iterator is a generalised form of index:

```
list<int>::iterator it;
for(it=lprimes.begin(); it!=lprimes.end(); it++){
    cout << (*it) << endl;
}
```

Note that similar code could be used to scan the elements of a vector

```
// scan vector using iterators
vector<int>::iterator it;
for(it = primes.begin(); it!=primes.end(); it++){
    cout << (*it) << endl;
```

However, in the case of vectors only, conventional indexing is supported

```
// scan vector using an integer index
int i;
for(i=0; i < primes.size(); i++){
    cout << primes[i] << endl;
}
```

where `primes.size()` returns the number of elements in the container (vector in this case).

One day, the syntax will get better. The new C++ standard is due to be finalised in March.

The STL Containers

The STL provides a variety of container types which allow complex program structures to be built with very little effort.

1. lists - linked list
2. vectors - array
3. strings - character array
4. sets - set of values
5. map - associative map, values are accessed using a key
6. mmap - a map which supports duplicate keys

plus a large set of built-in algorithms for manipulating these containers such as find, insert, replace, sort, reverse, ... , etc.

A further advantage of using STL and similar libraries is that they are robust and they have built-in memory management which helps to avoid memory leaks.

OO Programming

The object-oriented programming mechanisms described in the previous two lectures change the way we go about designing software. The process is something like:

1. What kinds of objects are present in the problem domain?

   - list of potential classes

2. Are there any classes with similar functionality and common purpose?

   - introduce abstract base classes
   - find the is-a relationships
   - define class hierarchies

3. What are the relationships between classes?

   - find the has-a relationships.
   - define class compositions

4. What services (functions) must each class provide?

   - define interfaces

5. How are these services going to be implemented?

   -implement interfaces

6. Iterate and refine 1–5.