

C and C++

6. Operators — Inheritance — Virtual

Alan Mycroft

University of Cambridge

(heavily based on previous years' notes – thanks to Alastair Beresford and Andrew Moore)

Michaelmas Term 2012–2013

1 / 20

From last lecture ...

```
1 class Complex {
2     double re,im;
3     public:
4         Complex(double r=0.0L, double i=0.0L);
5 };
6
7 Complex::Complex(double r,double i) {
8     re=r,im=i; // deprecated initialisation-by-assignment
9 }
10
11 int main() {
12     Complex c(2.0), d(), e(1,5.0L);
13     return 0;
14 }
```

2 / 20

Operators

- ▶ C++ allows the programmer to overload the built-in operators
- ▶ For example, a new test for equality:

```
1 bool operator==(Complex a, Complex b) {
2     return a.real()==b.real() && a.imag()==b.imag();
3     // presume real() is an accessor for field 're', etc.
4 }
```

- ▶ An operator can be defined or declared within the body of a class, and in this case one fewer argument is required; for example:

```
1 bool Complex::operator==(Complex b) {
2     return re==b.real() && im==b.imag();
3 }
```

- ▶ Almost all operators can be overloaded

3 / 20

Streams

- ▶ Overloaded operators also work with built-in types
- ▶ Overloading is used to define << (C++'s "printf"); for example:

```
1 #include <iostream>
2
3 int main() {
4     const char* s = "char array";
5
6     std::cout << s << std::endl;
7
8     //Unexpected output; prints &s[0]
9     std::cout.operator<<(s).operator<<(std::endl);
10
11    //Expected output; prints s
12    std::operator<<(std::cout,s);
13    std::cout.operator<<(std::endl);
14    return 0;
15 }
```

- ▶ Note `std::cin`, `std::cout`, `std::cerr`

4 / 20

The 'this' pointer

- ▶ If an operator is defined in the body of a class, it may need to return a reference to the current object
 - ▶ The keyword `this` can be used

- ▶ For example:

```
1 Complex& Complex::operator+=(Complex b) {
2     re += b.real();
3     this->im += b.imag();
4     return *this;
5 }
```

- ▶ In C (or assembler) terms `this` is an implicit argument to a method when seen as a function.

5 / 20

Class instances as member variables

- ▶ A class can have an instance of another class as a member variable
- ▶ How can we pass arguments to the class constructor?
- ▶ New notation for a constructor:

```
1 class X {
2     Complex c;
3     Complex d;
4     X(double a, double b): c(a,b), d(b) {
5         ...
6     }
7 };
```

- ▶ This notation must be used to initialise `const` and reference members
- ▶ It can also be more efficient

6 / 20

Temporary objects

- ▶ Temporary objects are often created during execution
- ▶ A temporary which is not bound to a reference or named object exists only during evaluation of a full expression (BUGS BUGS BUGS!)
- ▶ Example: the `string` class has a function `c_str()` which returns a pointer to a C representation of a string:

```
1 string a("A "), b("string");
2 const char *s1 = a.c_str(); //Okay
3 const char *s2 = (a+b).c_str(); //Wrong
4 ...
5 //s still in scope here, but the temporary holding
6 //"a+b" has been deallocated
7 ...
8 string tmp = a+b;
9 const char *s3 = tmp.c_str(); //Okay
```

7 / 20

Friends

- ▶ A (non-member) `friend` function can access the private members of a class instance it befriends
- ▶ This can be done by placing the function declaration inside the class definition and prefixing it with the keyword `friend`; for example:

```
1 class Matrix {
2     ...
3     friend Vector operator*(const Matrix&, \
4                             const Vector&);
5     ...
6 };
7 }
```

8 / 20

Inheritance

- ▶ C++ allows a class to inherit features of another:

```
1 class vehicle {
2     int wheels;
3 public:
4     vehicle(int w=4):wheels(w) {}
5 };
6
7 class bicycle : public vehicle {
8     bool panniers;
9 public:
10    bicycle(bool p):vehicle(2),panniers(p) {}
11 };
12
13 int main() {
14     bicycle(false);
15 }
```

9 / 20

Derived member function call

i.e. when we call a function overridden in a subclass.

- ▶ Default derived member function call semantics differ from Java:

```
1 class vehicle {
2     int wheels;
3 public:
4     vehicle(int w=4):wheels(w) {}
5     int maxSpeed() {return 60;}
6 };
7
8 class bicycle : public vehicle {
9     int panniers;
10 public:
11    bicycle(bool p=true):vehicle(2),panniers(p) {}
12    int maxSpeed() {return panniers ? 12 : 15;}
13};
```

10 / 20

Example

```
1 #include <iostream>
2 #include "example13.hh"
3
4 void print_speed(vehicle &v, bicycle &b) {
5     std::cout << v.maxSpeed() << " ";
6     std::cout << b.maxSpeed() << std::endl;
7 }
8
9 int main() {
10    bicycle b = bicycle(true);
11    print_speed(b,b); //prints "60 12"
12 }
```

11 / 20

Virtual functions

- ▶ Non-virtual member functions are called depending on the static type of the variable, pointer or reference
- ▶ Since a pointer to a derived class can be cast to a pointer to a base class, calls at base class do not see the overridden function.
- ▶ To get polymorphic behaviour, declare the function virtual in the superclass:

```
1 class vehicle {
2     int wheels;
3 public:
4     vehicle(int w=4):wheels(w) {}
5     virtual int maxSpeed() {return 60;}
6};
```

12 / 20

Virtual functions

- ▶ In general, for a virtual function, selecting the right function has to be run-time decision; for example:

```
1 bicycle b(true);
2 vehicle v;
3 vehicle* pv;
4
5 user_input() ? pv = &b : pv = &v;
6
7 std::cout << pv->maxSpeed() << std::endl;
8 }
```

13 / 20

Enabling virtual functions

- ▶ To enable virtual functions, the compiler generates a virtual function table or vtable
- ▶ A vtable contains a pointer to the correct function for each object instance
- ▶ The vtable is an example of indirection
- ▶ The vtable introduces run-time overhead (this is compulsory in Java; contemplate whether C++'s additional choice is good for efficiency or bad for being an additional source of bugs)

14 / 20

Abstract classes

- ▶ Sometimes a base class is an un-implementable concept
- ▶ In this case we can create an abstract class:

```
1 class shape {
2 public:
3     virtual void draw() = 0;
4 }
```

- ▶ It is not possible to instantiate an abstract class:
`shape s; //Wrong`
- ▶ A derived class can provide an implementation for some (or all) the abstract functions
- ▶ A derived class with no abstract functions can be instantiated

15 / 20

Example

```
1 class shape {
2 public:
3     virtual void draw() = 0;
4 };
5
6 class circle : public shape {
7 public:
8     //...
9     void draw() { /* impl */ }
10};
```

16 / 20

Multiple inheritance

- ▶ It is possible to inherit from multiple base classes; for example:

```
1 class ShapelyVehicle: public vehicle, public shape {
2   ...
3 }
```

- ▶ Members from both base classes exist in the derived class
- ▶ If there is a name clash, explicit naming is required
- ▶ This is done by specifying the class name; for example:

```
ShapelyVehicle sv;
sv.vehicle::maxSpeed();
```

17 / 20

Multiple instances of a base class

- ▶ With multiple inheritance, we can build:

```
1 class A {};
```

```
2 class B : public A {};
```

```
3 class C : public A {};
```

```
4 class D : public B, public C {};
```

- ▶ This means we have two instances of A even though we only have a single instance of D
- ▶ This is legal C++, but means all references to A must be stated explicitly:

```
1 D d;
2 d.B::var=3;
3 d.C::var=4;
```

18 / 20

Virtual base classes

- ▶ Alternatively, we can have a single instance of the base class
- ▶ Such a "virtual" base class is shared amongst all those deriving from it

```
1 class Vehicle {int VIN;};
2 class Boat : public virtual Vehicle { ... };
3 class Car : public virtual Vehicle { ... };
4 class JamesBondCar : public Boat, public Car { ... };
```

19 / 20

Exercises

1. If a function f has a static instance of a class as a local variable, when might the class constructor be called?
2. Write a class `Matrix` which allows a programmer to define 2×2 matrices. Overload the common operators (e.g. $+$, $-$, $*$, and $/$)
3. Write a class `Vector` which allows a programmer to define a vector of length two. Modify your `Matrix` and `Vector` classes so that they interoperate correctly (e.g. $v2 = m*v1$ should work as expected)
4. Why should destructors in an abstract class almost always be declared `virtual`?

20 / 20