C and C++

6. Operators — Inheritance — Virtual

Alastair R. Beresford

University of Cambridge

Lent Term 2008

Operators

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

```
bool operator==(Complex a, Complex b) {
  return a.real()==b.real()
         && a.imag()==b.imag();
}
```

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

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

- Almost all operators can be overloaded

Streams

- Overloaded operators also work with built-in types
- Overloading is used to define a C++ "`printf`"; for example:

```
#include <iostream>

int main() {
  const char* s = "char array";

  std::cout << s << std::endl;

  //Unexpected output; prints &s[0]
  std::cout.operator<<(s).operator<<(std::endl);

  //Expected output; prints s
  std::operator<<(std::cout,s);
  std::cout.operator<<(std::endl);
  return 0;
}
```


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:

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

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:

```
class X {
  Complex c;
  Complex d;
  X(double a, double b): c(a,b), d(b) {
    ...
  }
};
```

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

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*
- Example: the `string` class has a function `c_str()` which returns a pointer to a C representation of a string:

```
string a("A "), b("string");
const char *s = (a+b).c_str(); //Wrong
 ...
//s still in scope here, but the temporary holding
//"a+b" has been deallocated
 ...
```


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:

```
class Matrix {
  ...
  friend Vector operator*(const Matrix&, \
                          const Vector&);
   ...
  };
}
```


Inheritance

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

```
class vehicle {
  int wheels;
public:
  vehicle(int w=4):wheels(w) {}
};

class bicycle : public vehicle {
  bool panniers;
public:
  bicycle(bool p):vehicle(2),panniers(p) {}
};

int main() {
  bicycle(false);
}
```

Derived member function call

- Default derived member function call semantics differ from Java:

```
class vehicle {
  int wheels;
public:
  vehicle(int w=4):wheels(w) {}
  int maxSpeed() {return 60;}
};

class bicycle : public vehicle {
  int panniers;
public:
  bicycle(bool p=true):vehicle(2),panniers(p) {}
  int maxSpeed() {return panniers ? 12 : 15;}
};
```


Example

```
#include <iostream>
#include "example13.hh"

void print_speed(vehicle &v, bicycle &b) {
  std::cout << v.maxSpeed() << " ";
  std::cout << b.maxSpeed() << std::endl;
}

int main() {
  bicycle b = bicycle(true);
  print_speed(b,b); //prints "60 12"
}
```


Virtual functions

- Non-virtual member functions are called depending on the *static type* of the variable, pointer or reference
- Since a derived class can be cast to a base class, this prevents a derived class from overloading a function
- To get polymorphic behaviour, declare the function `virtual` in the superclass:

```
class vehicle {
  int wheels;
 public:
  vehicle(int w=4):wheels(w) {}
  virtual int maxSpeed() {return 60;}
};
```


Virtual functions

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

```
bicycle b;
vehicle v;
vehicle* pv;

user_input() ? pv = &b : pv = &v;

std::cout << pv->maxSpeed() << std::endl;
}
```

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

Abstract classes

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

```
class shape {
 public:
  virtual void draw() = 0;
}
```

- 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

Example

```
class shape {
public:
  virtual void draw() = 0;
};

class circle : public shape {
public:
  //...
  void draw() { /* impl */ }
};
```


Multiple inheritance

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

```
class ShapelyVehicle: public vehicle, public shape {
 ...
}
```

- 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();`

Multiple instances of a base class

- With multiple inheritance, we can build:

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

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


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

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


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 two dimensional 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`?