Introduction to Scientific Programming with C++

Session 5: Advanced object oriented programming

Martin Uhrin

UCL

February 11-13th 2013

Table of Contents

Separating class definition and declaration

Often it's a pain to write the body of a function within the class, consider:

```
class GameObject {
public:
  void draw()  {
  // many hundreds of lines
  // of graphics code here
  }
};
```

the class interface gets lost in all the graphics code.

Separating class definition and declaration

Often it's a pain to write the body of a function within the class, consider:

```
class GameObject {
public:
  void draw()  {
  // many hundreds of lines
  // of graphics code here
  }
};
```

the class interface gets lost in all the graphics code.

To help with this we can write:

```
class GameObject {
public:
  void draw();
};

// many lines or a
// different file later

GameObject::draw() {
  // many hundreds of lines
  // of graphics code here
}
```

Separating class definition and declaration

Often it's a pain to write the body of a function within the class, consider:

```
class GameObject {
public:
  void draw()  {
  // many hundreds of lines
  // of graphics code here
  }
};
```

the class interface gets lost in all the graphics code.

To help with this we can write:

```
class GameObject {
public:
  void draw();
};

// many lines or a
// different file later

GameObject::draw() {
  // many hundreds of lines
  // of graphics code here
}
```

This supports encapsulation by separating the interface from the implementation. You'll often find declaration in a `.h` file and the definition in a `.cpp` file.

Class inheritance

Not compatible with egalitarianism

Consider the following class:

```
class Mammal {
public:
  Mammal(const unsigned int age, const unsigned int weight);

  unsigned int getAge();
  void  setAge(const unsigned int age);

  unsigned int getWeight();
  void setWeight(const unsigned int weight);

  void speak();

protected:
  unsigned int myAge;
  unsigned int myWeight;
};
```

Class inheritance

Not compatible with egalitarianism

Consider the following class:

```
class Mammal {
public:
  Mammal(const unsigned int age, const unsigned int weight);

  unsigned int getAge();
  void  setAge(const unsigned int age);

  unsigned int getWeight();
  void setWeight(const unsigned int weight);

  void speak();

protected:
  unsigned int myAge;
  unsigned int myWeight;
};
```

What if we want to create a dog class. It can do everything a `Mammal` can. It would be a shame to have to copy everything over. With inheritance we don't have to!

Class inheritance

We can make a `Dog` inherit from `Mammal` like so:

```
class Dog : public Mammal {
public:
  Dog(const unsigned int age,
    const unsigned int weight,
    const std::string & breed);

  void wagTail();
  void goFetch(const std::string & fetchWhat);

private:
  std::string myBreed;
};
```

../code/5_advanced_oop/lectures/inheritance.cpp

Class inheritance

We can make a Dog inherit from Mammal like so:

```
class Dog : public Mammal {
public:
  Dog(const unsigned int age,
    const unsigned int weight,
    const std::string & breed);

  void wagTail();
  void goFetch(const std::string & fetchWhat);

private:
  std::string myBreed;
};
```

../code/5_advanced_oop/lectures/inheritance.cpp

Now Dog has inherited members from Mammal so we don't have to rewrite them. Dog is said to be *derived* from Mammal. Makes sense, right?

Class inheritance

Sit, boo-boo, sit. Good dog.

This is what happens when we use the Dog class:

```
int main()
{
  Dog fido(2, 10,
    "Boston terrier");

  std::cout << "Fido is : "
    << fido.getAge()
    << " years old\n";
  std::cout << "and weighs "
    << fido.getWeight()
    << " kg.\n\n";

  fido.speak();
  fido.wagTail();
  fido.goFetch("frisbee");

  return 0;
}
```

../code/5_advanced_oop/lectures/inheritance.cpp

Class inheritance

Sit, boo-boo, sit. Good dog.

This is what happens when we use the Dog class:

```
int main()
{
  Dog fido(2, 10,
    "Boston terrier");

  std::cout << "Fido is : "
    << fido.getAge()
    << " years old\n";
  std::cout << "and weighs "
    << fido.getWeight()
    << " kg.\n\n";

  fido.speak();
  fido.wagTail();
  fido.goFetch("frisbee");

  return 0;
}
```

../code/5_advanced_oop/lectures/inheritance.cpp

Output:

```
Fido is :  2 years old
and weighs 10 kg.

Grrr, mammal noise!
\/\/\/ <- That's my tail wagging!
Fetching frisbee...Here you go.
```

Passing arguments to base constructors

`Mammal` has only one constructor:

```
Mammal(const unsigned int age, const unsigned int weight);
```

So we need to give an age and a weight to build a `Mammal`. Because `Dog` inherits from `Mammal` it too must provide these.

Passing arguments to base constructors

`Mammal` has only one constructor:

```
  Mammal(const unsigned int age, const unsigned int weight);
```

So we need to give an age and a weight to build a `Mammal`. Because `Dog` inherits from `Mammal` it too must provide these. Here's how:

```
Dog::Dog(const unsigned int age,
    const unsigned int weight,
    const std::string & breed):
Mammal(age, weight)
{
  myBreed = breed;
}
```

The `Dog` class calls the `Mammal` constructor in the initialiser list as part of its constructor to initialise the `Mammal` part of itself.

Initialiser lists

Definition

initialiser list a list used to initialise base class(es), class constants, member constants and (optionally) member variables as part of a class's constructor.

The format is:

```
constructor_name(paramters...):
  initialise_item1,
  initialise_item2,
        ...
        { /* constructor body */ }
```

So we could also initialise a Dog's breed in this list:

```
Dog(const unsigned int age,
  const unsigned int weight,
  const std::string & breed):
  myBreed(breed) {}
```

Overriding functions

When Fido spoke he said "`Grrr, mammal noise!`", not really what a Dog says. That's because I didn't provide a custom `speak` method. Let's try again:

Overriding functions

When Fido spoke he said "`Grrr, mammal noise!`", not really what a Dog says. That's because I didn't provide a custom `speak` method. Let's try again:

```
class Dog : public Mammal {
public:
  // Override Mammal's speak
  void speak();
};
void Dog::speak()
{ std::cout << "Woof!\n"; }

int main() {
  Dog fido(2, 10,
    "Boston terrier");
  fido.speak();

  return 0;
}
```

Output: `Woof!`

../code/5_advanced_oop/lectures/inheritance2.cpp

Overriding functions

When Fido spoke he said "`Grrr, mammal noise!`", not really what a Dog says. That's because I didn't provide a custom `speak` method. Let's try again:

```
class Dog : public Mammal {
public:
  // Override Mammal's speak
  void speak();
};
void Dog::speak()
{ std::cout << "Woof!\n"; }

int main() {
  Dog fido(2, 10,
    "Boston terrier");
  fido.speak();

  return 0;
}
```

Output: `Woof!`

../code/5_advanced_oop/lectures/inheritance2.cpp

The new `speak` method is said to *override* the one in `Mammal`.

Derived class member access

Finally we get to see what the `protected` access specifier is all about. Access types can be summarised as follows:

Access	`public`	`protected`	`private`
members of same class	yes	yes	yes
members of derived class	yes	yes	no
non members	yes	no	no

Derived class member access

Finally we get to see what the `protected` access specifier is all about. Access types can be summarised as follows:

Access	`public`	`protected`	`private`
members of same class	yes	yes	yes
members of derived class	yes	yes	no
non members	yes	no	no

Do

Use the most restrictive access possible. Member variables should rarely be anything but `private`, even derived classes should use getters/setters to access these. It's fine to make methods protected though.

What *exactly* is inherited?

Pretty much everything is inherited by the derived class, except:

- Constructors and destructors
- `operator =()` members
- friends (more on this later)

Multiple inheritance

Let's say we have the following classes:

```
class Mammal {
  double furLength;
public:
  void feedYoungWithMilk();
};
```

```
class Bird {
  int beakType;
public:
  void fly();
  void layEgg();
};

class Amphibian {
  bool webbedFeet;
};
```

Multiple inheritance

Let's say we have the following classes:

```
class Mammal {
  double furLength;
public:
  void feedYoungWithMilk();
};
```

```
class Bird {
  int beakType;
public:
  void fly();
  void layEgg();
};

class Amphibian {
  bool webbedFeet;
};
```

Pretty awesome.

Multiple inheritance

Let's say we have the following classes:

```
class Mammal {
  double furLength;
public:
  void feedYoungWithMilk();
};
```

```
class Bird {
  int beakType;
public:
  void fly();
  void layEgg();
};

class Amphibian {
  bool webbedFeet;
};
```

Pretty awesome. But what do we do with a Platypus:

✓ Has fur

Multiple inheritance

Let's say we have the following classes:

```
class Mammal {
  double furLength;
public:
  void feedYoungWithMilk();
};
```

```
class Bird {
  int beakType;
public:
  void fly();
  void layEgg();
};

class Amphibian {
  bool webbedFeet;
};
```

Pretty awesome. But what do we do with a Platypus:

- ✓ Has fur
- ✓ Has beak

Multiple inheritance

Let's say we have the following classes:

```
class Mammal {
  double furLength;
public:
  void feedYoungWithMilk();
};
```

```
class Bird {
  int beakType;
public:
  void fly();
  void layEgg();
};

class Amphibian {
  bool webbedFeet;
};
```

Pretty awesome. But what do we do with a Platypus:

- ✓ Has fur
- ✓ Has beak
- ✓ Lays eggs

Multiple inheritance

Let's say we have the following classes:

```
class Mammal {
  double furLength;
public:
  void feedYoungWithMilk();
};
```

```
class Bird {
  int beakType;
public:
  void fly();
  void layEgg();
};

class Amphibian {
  bool webbedFeet;
};
```

Pretty awesome. But what do we do with a Platypus:

- ✓ Has fur
- ✓ Has beak
- ✓ Lays eggs
- ✓ Webbed feet

Multiple inheritance

Let's say we have the following classes:

```
class Mammal {
  double furLength;
public:
  void feedYoungWithMilk();
};
```

```
class Bird {
  int beakType;
public:
  void fly();
  void layEgg();
};

class Amphibian {
  bool webbedFeet;
};
```

Pretty awesome. But what do we do with a Platypus:

- ✓ Has fur
- ✓ Has beak
- ✓ Lays eggs
- ✓ Webbed feet
- ✓ Feeds young with milk

Multiple inheritance

Simple, just use multiple inheritance:

```
class Platypus :
  public Mammal, public Bird,
  public Amphibian
{
public:
  void stingWithSpur();
};
```

So `Platypus` gets everything from `Mammal`, `Bird` and `Amphibian`.

Multiple inheritance

Simple, just use multiple inheritance:

```
class Platypus :
  public Mammal, public Bird,
  public Amphibian
{
public:
  void stingWithSpur();
};
```

So `Platypus` gets everything from `Mammal`, `Bird` and `Amphibian`.

But wait. A platypus can't fly!

Multiple inheritance

Simple, just use multiple inheritance:

```
class Platypus :
  public Mammal, public Bird,
  public Amphibian
{
public:
  void stingWithSpur();
};
```

So `Platypus` gets everything from `Mammal`, `Bird` and `Amphibian`.

But wait. A platypus can't fly! Okay, so let's override `fly` and make it private:

```
class Platypus :
  public Mammal, public Bird,
  public Amphibian
{
public:
  void stingWithSpur();
private:
  void fly();
};
```

Multiple inheritance

Simple, just use multiple inheritance:

```
class Platypus :
  public Mammal, public Bird,
  public Amphibian
{
public:
  void stingWithSpur();
};
```

So `Platypus` gets everything from `Mammal`, `Bird` and `Amphibian`.

But wait. A platypus can't fly! Okay, so let's override `fly` and make it private:

```
class Platypus :
  public Mammal, public Bird,
  public Amphibian
{
public:
  void stingWithSpur();
private:
  void fly();
};
```

Now if you try to do this:

```
int main()
{
  Platypus platypus;
  platypus.fly(); // Compiler error: can't access private!
  return 0;
}
```

Polymorphism

The solution to all your animal taxonomy needs

Let's bring Fido back out. What happens when we try:

```
int main()
{
  Dog fido(2, 10,
    "Boston terrier");

  // Perfectly valid, after
  // all a Dog is a mammal:
  Mammal * fidoPtr = &fido;
  fidoPtr->speak();

  return 0;
}
```

Polymorphism

The solution to all your animal taxonomy needs

Let's bring Fido back out. What happens when we try:

```
int main()
{
  Dog fido(2, 10,
    "Boston terrier");

  // Perfectly valid, after
  // all a Dog is a mammal:
  Mammal * fidoPtr = &fido;
  fidoPtr->speak();

  return 0;
}
```

Output: `Grrrr, mammal noise!`

Polymorphism

The solution to all your animal taxonomy needs

Let's bring Fido back out. What happens when we try:

```
int main()
{
  Dog fido(2, 10,
    "Boston terrier");

  // Perfectly valid, after
  // all a Dog is a mammal:
  Mammal * fidoPtr = &fido;
  fidoPtr->speak();

  return 0;
}
```

Output: `Grrrr, mammal noise!`

Huh? Why is Fido speaking like a mammal again?

Virtual functions

Virtual functions provide the solution:

```
class Mammal {
public:
  virtual void speak();
};
class Dog : public Mammal {
public:
  virtual void speak();
};

// Notice, I don't need
// virtual here:
void Dog::speak()
{ std::cout << "Woof!\n"; }
```

../code/5_advanced_oop/lectures/virtual_functions.cpp

Virtual functions

Virtual functions provide the solution:

```
class Mammal {
public:
  virtual void speak();
};
class Dog : public Mammal {
public:
  virtual void speak();
};

// Notice, I don't need
// virtual here:
void Dog::speak()
{ std::cout << "Woof!\n"; }
```

```
int main() {
  Dog fido(2, 10,
    "Boston terrier");

  Mammal * fidoPtr = &fido;
  fidoPtr->speak();

  return 0;
}
```

Output: `Woof!`

../code/5_advanced_oop/lectures/virtual_functions.cpp

Virtual functions

Virtual functions provide the solution:

```
class Mammal {
public:
  virtual void speak();
};
class Dog : public Mammal {
public:
  virtual void speak();
};

// Notice, I don't need
// virtual here:
void Dog::speak()
{ std::cout << "Woof!\n"; }
```

```
int main() {
  Dog fido(2, 10,
    "Boston terrier");

  Mammal * fidoPtr = &fido;
  fidoPtr->speak();

  return 0;
}
```

Output: `Woof!`

../code/5_advanced_oop/lectures/virtual_functions.cpp

As if by magic, the correct method in Dog gets called. This works for multiply derived types if they all use the `virtual` keyword.

Virtual functions

Health warning

Warning!

If you declare any of your methods to be `virtual` make sure to have a `virtual` destructor!
Otherwise your object will not be destructed properly!

Abstract base classes

I'll take one of your finest mammals please

At the moment we can do this:

```
int main()
{
  Mammal theFinest(27, 70);

  theFinest.speak();

  return 0;
}
```

Output: `Grrrr, mammal noise!`

But this doesn't make a lot of sense. A mammal is an abstraction, we shouldn't be able to instantiate a concrete mammal. We just want all derived types to be able to speak.

Abstract base classes

The solution: use a pure virtual function by adding `= 0`.

```
class Mammal {
public:
  virtual void speak() = 0;
};

int main() {
  Mammal theFinest(27, 70);
// Error: cannot instantiate abstract class

  return 0;
}
```

../code/5_advanced_oop/lectures/pure_virtual_functions.cpp

Abstract base classes

The solution: use a pure virtual function by adding `= 0`.

```
class Mammal {
public:
  virtual void speak() = 0;
};

int main() {
  Mammal theFinest(27, 70);
// Error: cannot instantiate abstract class

  return 0;
}
```

../code/5_advanced_oop/lectures/pure_virtual_functions.cpp

Now `Mammal` is deemed to be abstract and cannot be instantiated but any concrete derived type that we do want to instantiate needs to be able to speak or do whatever it is that mammals do. Perfect.

Abstract bases classes example

Back to the real world

We want to write a log of how our simulation is progressing, but it may be convenient to be able to log to the screen, to file or to a database.

```
class Logger {
  virtual void logMessage(const std::string & message) = 0;
  virtual ~Logger() {}
};

class ScreenLogger : public Logger {
  virtual void logMessage(const std::string & message);
};

class FileLogger : public Logger {
  virtual void logMessage(const std::string & message);
};

class DatabaseLogger : public Logger {
  virtual void logMessage(const std::string & message);
};

int main()
{
  Logger * myLogger = new FileLogger("program.log");
  myLogger->writeMessage("Hello universe!")
  return 0;
}
```

Friend functions

We've seen that only a class can access its `private` members. Sometimes we may want to make an exception for a particular external function. We do this with the `friend` keyword followed by the function prototype:

Friend functions

We've seen that only a class can access its `private` members. Sometimes we may want to make an exception for a particular external function. We do this with the `friend` keyword followed by the function prototype:

```
class Rectangle {
public:
  Rectangle(unsigned int widthIn, unsigned int heightIn);

  unsigned int area();

  friend void drawRectangle(const Rectangle &);
private:
  unsigned int width, height;
};
void drawRectangle(const Rectangle & rect)
{
  std::cout << " ";      // Draw top
  printChars('-', rect.width);
```

../code/5_advanced_oop/lectures/friend_function.cpp

Friend functions

Not for every day

Warning!

- The use of `friend`s can undermine encapsulation, after all you're giving away access to parts of your class that were designed to be kept internal.

Friend functions

Not for every day

Warning!

- The use of `friend`s can undermine encapsulation, after all you're giving away access to parts of your class that were designed to be kept internal.
- Sometimes friendship can be used effectively to keep dependent code separated if there is a good design reason. For example the code to draw a rectangle may be very complicated and may fit more naturally in the graphics handling portion of the codebase. This enhances encapsulation.

Friend classes

Not a taxonomy of friends

We can grant `friend` status to an entire class:

```
class SolarSystem {
private:
  Vector2 planetPositions[NUM_PLANETS];
  Planet planets[NUM_PLANETS];
  friend class Planet;
};

class Planet {
public:
  Vector2 getPosition()
  { return mySystem.planetPositions[myIndex]; }
private:
  unsigned int myIndex;
  SolarSystem & mySystem;
};
```

Friend classes

Not a taxonomy of friends

We can grant `friend` status to an entire class:

```
class SolarSystem {
private:
  Vector2 planetPositions[NUM_PLANETS];
  Planet planets[NUM_PLANETS];
  friend class Planet;
};

class Planet {
public:
  Vector2 getPosition()
  { return mySystem.planetPositions[myIndex]; }
private:
  unsigned int myIndex;
  SolarSystem & mySystem;
};
```

Friend classes can be useful in highly coupled parts of code like this: a `Planet` cannot exist without a `SolarSystem`.

Friendship rules

Let's define some ground rules

The following transitivity and reciprocity rules apply to the `friend` property:

- Friendship isn't reciprocated: Just because a friend can access a classes `private` members doesn't mean the class can access the friend's. It has to be explicitly marked as a `friend`.

Friendship rules

Let's define some ground rules

The following transitivity and reciprocity rules apply to the `friend` property:

- Friendship isn't reciprocated: Just because a friend can access a classes `private` members doesn't mean the class can access the friend's. It has to be explicitly marked as a `friend`.
- Friendship isn't transitive: Friends of a friend can't access a classes `private` members.

Friendship rules

Let's define some ground rules

The following transitivity and reciprocity rules apply to the `friend` property:

- Friendship isn't reciprocated: Just because a friend can access a classes `private` members doesn't mean the class can access the friend's. It has to be explicitly marked as a `friend`.
- Friendship isn't transitive: Friends of a friend can't access a classes `private` members.
- Friendship isn't inherited: Friends can't access derived classes `private` members.

Friendship rules

Let's define some ground rules

The following transitivity and reciprocity rules apply to the `friend` property:

- Friendship isn't reciprocated: Just because a friend can access a classes `private` members doesn't mean the class can access the friend's. It has to be explicitly marked as a `friend`.
- Friendship isn't transitive: Friends of a friend can't access a classes `private` members.
- Friendship isn't inherited: Friends can't access derived classes `private` members.

Also note classes aren't automatically friends of their derived classes.

Thank You!