Today

Today we will look at object-oriented programming in more detail.

In particular we will look at:

– Composition versus inheritance
– Access to base classes
– public, private and protected.
– Multiple inheritance and virtual classes
– UML and object-oriented design.

Much of this lecture refers to the program rabbit4.cpp which we developed in the previous lecture, and which can be downloaded from the course web-page.

This material is taken from Pohl, Chapters 8 and 11.
Composition and inheritance

- We use *composition* when one class contains a data member that is an object of another class.
- Thus in `rabbit4.cpp`, the class `living` contains a data member `location` which is an object of the class `point`.
- Thus `living` and `point` are related by composition.
- Any object of type `living` thus includes an object, called `location`, of type `point`.
- To access the private data members of `location` from within an object that contains it, we have to use the public function members of `point`.
• We use *inheritance* when one class extends another class, as in:

```cpp
class animal : public living
from rabbit4.cpp.
```

• Here *living* is called the *base class* or *super-class* and *animal* is called the *sub-class*.

• We can think of this as meaning that an object of class *animal* contains all the data and function members of class *living*.

• If we had an object `a` of class *animal*, we would refer to its member *location* by:

```cpp
    a.location
```
• And the data member \( x \) of \texttt{location} as:

\[
\texttt{a.location.x}
\]

• However, it is not quite as simple as that.
• The way that C++ implements inheritance is such that an object of class \texttt{animal} contains an object of class \texttt{living} (rather than the members of that object).
• Access to the members of this sub-object follow the usual access rules.
• Thus the \texttt{private} data members of \texttt{living} are not accessible from within \texttt{animal}.
• This is typically not what we want.
“public”, “private” and “protected”

• One way to handle the fact that a sub-class can’t access the private members of a base class is to write public methods that access them.

• Methods like set, getX and getY for point.

• Another approach is to redefine the private members as protected.

• Thus:

```cpp
class living {

protected:

    point location;
    bool eaten;

};
```
Using protected here means that the members are treated as public in classes derived from living (like animal).

However, for classes that are not derived from living, the protected data members are treated like they are private.

This is exactly what we want in rabbit4.cpp.

The general question of how sub-classes can access members of base classes is more complex than this, however.
Access to base class members

• Each member of a base class can be:
  – public
  – protected
  – private

• Classes can also be derived as:
  – class A : public B
  – class A : protected B
  – class A : private B

• These access levels interact.
• If we have class A : public B
  – public and protected members of B remain public and protected in A.
• If we have class A : protected B
  – public and protected members of B are protected in A.
• If we have class A : private B
  – public and protected members of B become private in A.
• Of course, even if base class members are private they can be accessed by friend classes.
• (Now would be a good time to go back and recap friend classes).
Multiple inheritance

• In statements of class derivation like

\[
\text{class A : public B}
\]

we are not limited to deriving from a single base class.

• We can have, for example:

\[
\text{class A : public B, private C}
\]

• This is called \textit{multiple inheritance}.

• In the latter case A has all of the members of B and C.
• As an example of multiple inheritance, consider a variation on the classes in `rabbit4.cpp`.

• We could have:

```cpp
class predator: public living{

public:
void eat();
};

class prey: public living{

public:
void beEaten();
};
```
• carrot is then a sub-class of prey, and fox is a sub-class of predator.

• rabbit is both predator and prey (it eats carrots but is eaten by foxes), so we would define:

```cpp
class rabbit: public predator, public prey
```

• This illustrates a common problem with multiple inheritance.

• We have the class hierarchy:

```
living
   /  \
  /    \
predator  prey
   |      |
  /       |
fox      rabbit       carrot
```
• rabbit now inherits from living twice, once through predator and once through prey.
• This means it has two copies of all the members that it inherits from living.
• If we have:

```java
rabbit peter;
peter.location.set(1, 2);
```

it is ambiguous which location this refers to.
• It is possible to get around this problem using virtual base classes.
• If we define:

```cpp
class predator: virtual public living {
    public:
    void eat();
};
```

```cpp
class prey: virtual public living {
    public:
    void beEaten();
};
```

```cpp
class rabbit: public predator, public prey {
};
```

then rabbit will only contain one copy of living.

• For more on virtual base classes, see the textbook.
The unified modelling language or UML is a method of designing and documenting object-oriented designs. We are already familiar with the idea of drawing the relationship between classes:

UML expands on this.
• UML uses the same notation as we have been using already to show inheritance between classes.

• UML adds a graphical representation of composition:

  living  △  point

indicates that living includes an object of type point

• UML also shows the data and function members that a class contains.

• The full UML representation of living and point from rabbit4.cpp is shown on the next slide.
<table>
<thead>
<tr>
<th>living</th>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>eaten location</td>
<td>x</td>
</tr>
<tr>
<td>print set</td>
<td>y</td>
</tr>
<tr>
<td>print set</td>
<td></td>
</tr>
<tr>
<td>print set</td>
<td></td>
</tr>
<tr>
<td>print set</td>
<td></td>
</tr>
<tr>
<td>beEaten</td>
<td></td>
</tr>
<tr>
<td>print set</td>
<td></td>
</tr>
<tr>
<td>print set</td>
<td></td>
</tr>
<tr>
<td>print set</td>
<td></td>
</tr>
</tbody>
</table>
• Clearly we could expand the rest of the class hierarchy with this additional information.

• The idea behind UML is to use this graphical notation to develop the class design before coding.

• The diagrams also serve as a form of documentation.

• Tools for drawing UML diagrams, tutorials and much more can be found at http://www.uml.org/.
Summary

• This class has looked at some of the finer points of object-oriented programming.

• We recapped the difference between inheritance and composition and covered:
  – Access to base class members.
  – public, private and protected.
  – Multiple inheritance
  – UML

• Next lecture we will go on to look at pointers.