POLYMORPHISM

Today

- We will finish off our discussion of inheritance by talking more about the way that inheritance enables polymorphism.
- This will lead us into a few topics that we didn't yet cover:
  - Operator overloading
  - The relationship between friendship and inheritance
- It will also preview the idea of *templates* which we will cover properly in the next unit.
- The textbook says little about polymorphism explcitly (see pages 484-485) but does have a lot to say about the methods for achieving it in various places.
- For example there is a long discussion of operator overloading on pages 243-263.

Polymorphism

- The folk who know about these things have declared that C++ has four mechanisms that enable polymorphism:
  - Coercion
    This is considered to be *ad hoc* polymorphism.
  - Overloading
    Again, this is considered to be ad hoc.
  - Inclusion
    This is *pure* polymorphism.
  - Parametric polymorphism
    Again this is pure.
- We will look at each of these in turn in some detail.

Coercion

- Coercion is when we write code that forces type conversions.
- It is a limited way to deal with different types in a uniform way.
- For example

```
double x, d;
int i;
x = d + i;
```

- During the addition, the integer `i` is *coerced* into being a double so that it can be added to `d`.
- So you have been using polymorphism for ages without knowing it.

Overloading

- Overloading is the use of the same function or operator on different kinds of data to get different results.
- As we recall:

```
double x;
int a = 5;
int b = 5;
x = a/b;
```

gives us a different result than:

```
double x;
double a = 5;
double b = 2;
x = a/b;
```

- In the first case the division is integer division and x has value 2.
- In the second case the division is floating point, and x has value 2.5.
- (It should be clear that there is no coercion in this case — for coercion the arguments would be of different types).
- The difference is because there are different versions of \ for integers and for doubles.
- It is also possible to define new operators, through *operator overloading*.

Operator Overloading

- Many of the operators in C++ can be overloaded.
- We have already seen some — the `string` class for example overloads `+`, `+=` and `[]`..
- We can overload operators to work with classes we define.

- Let's overload an operator for the following class:

```
class person {

private:
  string first_name;
  string second_name;
  int    age;

public:

  person(string, string, int);
  void print();
};
```

- Let's imagine we need to sort `person` objects by age.
- We could access ages and compare them with >.
- We could also overload >, and that would reduce the amount of code we need to write if we do lots of comparisons.
- To overload > in class `person` we add:

  ```
  bool operator>(person);
  ```

  as a function member of the `person` class.
- We then define what we want the operator to do.
- Note that a binary operator becomes a function with one argument.

- This is because when we call:

  ```
  a > b
  ```

  the system processes this as:

  ```
  a.>(b);
  ```

  that is as calling the method > of the first object with the second object as its argument.

- The code for the operator then becomes:

```
bool person::operator>(person p){
  if (this->age > p.age){
    return true;
  }
  else {
    return false;
  }
}
```

- using the `this` pointer to refer to the relevant attribute of the first object.
- For the full code of an example using this overloaded operator, see the file `overload.cpp` on the course website.

Inclusion

- We have aready seen that inclusion — which is what inheritance gives us, the inclusion of the attributes of one class inside another class — helps us schieve polymorphism.
- We can define functions on `animal` and pass them a `rabbit`.
- When we set up functions to be virtual and make them have arguments that are pointers to `animal`, we can get the system to look through the class heirarchy to call exactly the most appropriate function.
- *Run-time determination of sub-type*.

- Note that this combination of inheritance and pointers to convert from one pointer to another is not limited to passing parameters.
- With:

```
living l;
living *lptr;
animal *aptr;
```

we are allowed to do this:

```
lptr = aptr;
```

converting from a pointer to animal to a pointer to living.

- The reverse:

  ```
  aptr = lptr;
  ```

  or

  ```
  aptr = &l;
  ```

  is not allowed.

A last thing about inheritance

- A derived class inherits every member of a base class except:
  - its constructor and its destructor
  - its operator members
  - its friends
- Note particularly the point about friends (and if it helps, think about how many of your parents' friends are your friends too :-)

- Recall that when:
  - $b2$ is a friend of $b1$
  - $d1$ is derived from $b1$
  - $d2$ is derived from $b2$
- It is the case that:
  - $b2$ has special access to private members of $b1$, as a friend
  - But $d2$ does not inherit this special access
  - Nor does $b2$ get special access to $d1$ (derived from friend $b1$)

Parametric polymorphism

- This means writing code where the type of thedata isn't specified.
  - It is determined at run-time.
- In C++ this means *templates*.
- For example:

```
template <class T>
bool greater(T a, T b){
  return (a > b);
}
```

- We will look at templates in detail in the next unit.

Summary

- In this lecture we finished up talking about inheritance by thinking about how it allows us to write polymorphic code.
- In doing this we looked at the various forms of polymorphism.
- We also spent some time talking about operator overloading and reminding ourselves of the relationship between inheritance and friendship.
- Finally we previewed templates.