OBJECT-ORIENTED LANGUAGES: INHERITANCE

Abstract Data Types

Most object-oriented languages make the class encapsulation mechanism synonymous with the introduction of a new type. Smalltalk, C++ and Java are all like this. This means that, in C++:

```
class C {
  ...
};
```

declares a type C, so that the creation of an object can be typed:

```
C obj1;
```

We can then write programs with a strong flavor of abstract data types; first declare types and their operations and then create objects from these declarations and use the operations on them. A simple example is a partial definition of a type for times:

```
class Time {
private:
  int hrs, mins, secs;
public:
  Time() { hrs = mins = secs = 0; }
  Time(h, m, s) {
    hrs = h;
    mins = m;
    secs = s;
  }
  Time operator+(Time t) {
    Time temp;
    temp.secs = secs + t.secs;
    if (temp.secs > 60) {
        temp.secs -= 60;
        temp.mins = mins + t.mins + 1;
    }
    else temp.mins = mins + t.mins;
    if (temp.mins > 60) {
        temp.mins -= 60;
        temp.hrs = hrs + t.hrs + 1;
    }
    else temp.hrs = hrs + t.hrs
    return temp;
  }
};
```

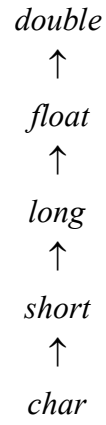
In the example, the only operation allowed is the addition of two Times, and we can overload the + operator to do this:

```
Time t1(1,15,42), t2(0,5,0), t3;
t3 = t1 + t2; // compiled as t3 = t1.operator+(t2)
```

Abstract data types of some complexity can be built following this model, such as rational numbers, complex numbers, etc.

Relationships between types

Using the ADT model, a program is a number of type declarations, and operations on objects of those types. However, their can exist types with relationships between them. A simple example is the hierarchy of numbers in C (and C++ and Java):



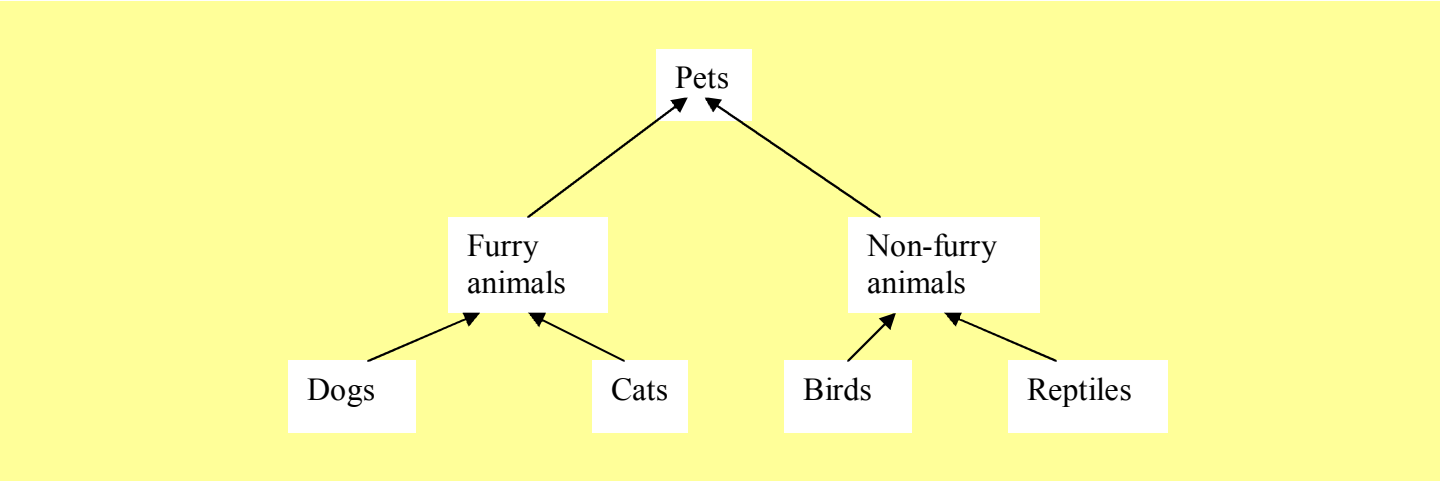
Each type defines a finite set of numbers that is a proper subset of the one above it (except *double*, which is at the top). This hierarchy is made use of in the coercion of one number type to another. Unfortunately, C does not care whether information is lost, so coercing a long to a short integer is done silently. Actually, if we were more careful, we would disallow downward coercions, or casts, and allow the upward ones. So, we could take a *char* value like 42 and 'upcast' it to 42 *short*, and then to 42 *long*. Then add a zero fractional part to yield 42.0 *float* and then *double*.

The object-oriented languages allow declaration of the subtype relationship between types by specifying a subclass relationship between classes. E.g. in C++:

```
class B {
...
};

class D : public B {
...
};
```

This creates two classes, the *base* class B, and the *derived* class D. Now, if we think through the subtype relationship a little more, we can see that there is also a relationship between the properties of the classes. To see this, we turn to a hierarchy of natural types, such as pets. We will allow two kinds of pet; a furry animal and a non-furry animal. Furry animals can be cats or dogs; non-furry animals can be birds and reptiles. If we draw this it forms a tree of types and subtypes:



The direction of the arrows gives the subtype directionality. I.e. all Cats are Furry animals, all Birds are Non-furry animals. If we now consider the properties of, say, dogs, then we can see that these are either special to dogs, or they are in common with cats, i.e. the property can be *inherited* from the class of all Furry animals. Such a property is "walks on four legs". All Furry animals walk on four legs, therefore all dogs walk on four legs, *and* all cats walk on four legs; the property has been inherited. An example of a difference is "rough tongue". All cats have rough tongues, and no dog has a rough tongue. This property cannot be inherited from Furry animals. In fact, it is these differences that define how a type is to be split. If there is no difference, then there is no need for a subtype. If there is at least one difference, then there are grounds for at least two subtypes.

The same set of principles can be followed in the object-oriented languages. Properties are really the members of the class, and these can be inherited from bases classes by derived classes. Consider:

```
class B {
private:
  int a, b, c;
public:
  void f1() { ... }
};

class D : public B {
private:
  int x, y;
public:
  void f2() { ... }
};
```

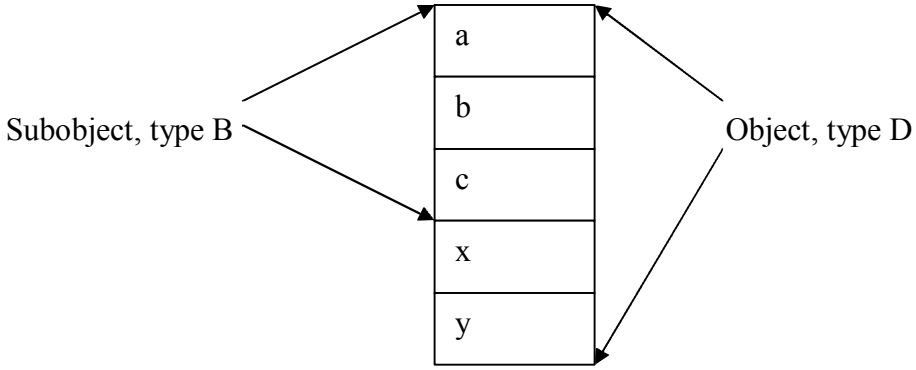
If we declare two objects we can then look at what function members are callable:

```
B obj1;
obj1.f1(); // f1 is a member of B

D obj2;
obj2.f2(); // f2 is a member of D
obj2.f1(); // f1 is inherited from B
```

This last line is an example of the subtype principle: *where two classes define a subtype relationship, an object of a derived type can substitute where an object of the base type is expected.* B and D are in a subtype relationship, and D inherits from B. f1 expects to be called with an object of type B, but we can substitute an object of type D and still call f1.

In fact, when obj2 is constructed, it contains an object of type B as a sub-object:



These data member that make up the allocated space for the object are also inherited. Thus, class D inherits variables a, b and c. Unfortunately, function f2 cannot directly access these variables, as we would want, because they are declared private in class B. We could add accessor function, as is done in Smalltalk:

```
class B {
private:
  int a, b, c;
public:
  void f1() { ... }
  int geta() ( return a; }
  int getb() ( return b; }
  int getc() ( return c; }
};

class D : public B {
private:
  int x, y;
public:
  void f2() { ... }
};
```

Then we could write 'geta()' instead of 'a' in the body of f2. geta is a member of B, so it has direct access to a, and it is inherited by D.

However, a better solution exists, the use of the *protected* form of access control.

```
class B {
protected:
  int a, b, c;
public:
  void f1() { ... }
};

class D : public B {
private:
  int x, y;
public:
  void f2() { ... }
};
```

Protected means the members are still private, but are directly accessible in subclasses. So now f2 can access the inherited variables a, b and c, directly, without using accessor functions.

Up- and Down-casting

The subtype principle leads us to consider the following code:

```
B b1;
D d1;
b1 = d1;
```

Here the object of type D is being assigned to the object of type B. What would we expect? The object d1 contains a, b, and c inherited from B, and x and y. The object b1 only contains a, b, and c. We would expect the values of a, b and c to be copied from d1 to b1, and the remainder of d1, i.e. x and y to be thrown away. This is called *object slicing* because the object d1 is sliced into its parts and only the part inherited from B is copied. The other assignment:

```
d1 = b1;
```

has a problem because there is no x and y in b1 to copy into x and y inside d1. The OK assignment is called up-casting, and is allowed in object-oriented languages. The not-OK assignment is called down-casting and is not allowed. The former can be made explicit with an explicit cast:

```
b1 = (B)d1;
```

hence up-casting. The other way:

```
d1 = (D)b1;
```

is not allowed.

This situation can be slightly complicated where there are references or pointers involved. In Java, the following code is OK:

```
B b1 = new B();
D d1 = new D();
b1 = (B)d1;
```

Up-casting has been performed, but since b1 is a reference, no slicing has been done – it still is a reference to an object of type D. Since we know this, we could down-cast it back, and call f2:

```
((D)b1).f2();
```

Here the down-cast is allowed since we know b1 is a reference to an object of type D. The end-result is that down-casting where references are involved is an issue to be resolved at run-time. In C++, every object comes with Run-Time Type Information (RTTI), so the following code is OK:

```
B *pb1 = new B();
D *pd1 = new D();
pb1 = (B *)pd1;
((D *)pb1)->f2();
```

Without RTTI this situation would be impossible to detect. (Actually C++ has a templated function called dynamic_cast to perform this in a type safe manner).

Inheritance in Smalltalk

Smalltalk has a particularly simple model of inheritance determined at run-time. If a class B inherits from A, and class A has a method m1, the message m1 can still be sent to an object of type B:

```
| anObject |
anObject <- B new.
anObject m1
```

Here a variable anObject is declared between |…| (without a type since Smalltalk is dynamically typed) and assigned an object of type B. The expression B new sends a message new to the class B which responds by returning a new object of type B. The expression anObject m1 sends the message m1 to the object in anObject. A search is started to find a match between the message *selector*, i.e. m1, and a method with the same name. Since class B does not have such a method, the search is continued in the superclass of B (there can only be one in Smalltalk) and its superclasses if necessary. Since m1 is contained in class A, it is excecuted. In this way, m1 is effectively inherited from A. Data (variables) is also inherited, but since in Smalltalk all data is private (and all methods public), methods in a subclass cannot access the inherited variables – only inherited methods can. Thus in the two classes A, and B:

```
class A supertype Object
  instance variables: x
  instance methods:
        getX
          ^x

class B supertype A
   instance variables: y
   instance variables:
        getY
          ^y
```

sending the message getX to an object of type B will return the value of x which is inherited from A. However, the body of getY cannot access x directly (only through getX) and the body of getX clearly cannot access y.

IS-A and HAS-A

The subtype relationship is often called an IS-A relationship since, to use our hierarchy of natural types again, a Dog IS-A Furry animal and a Furry animal IS-A Pet. The data members of a class can be put in the role of properties of the type, and, since there is no distinction between them, apart from a name, the we can call this relationship HAS-A, which covers all data members. Thus, in the last example, above, a D IS-A B and B HAS-A int a and an int b and an int c. In general, data members can be of any type:

```
class C {
private:
   D x; E y; F z;
...
};
```

Here, C HAS-A D called x, C HAS-A E called y and C HAS-A F called z. This is the only other relationship between types that C++ can express.

Summary

In C++, if a class is declared to be a subclass of another (using the keyword public is important here), then a subtype relationship (IS-A) also exists between them. Derived classes can inherit data and function members from base classes, and the subtype principle is followed. Data members can be of any type, leading to a catch-all HAS-A relationship between the type corresponding to the class and the type of the enclosed data member.