CS2141 – Software Development using C/C++

Polymorphism

Polymorphism in C+

- All polymorphism in C++ is done using inheritance; there is no concept of an interface
- A subclass is declared using the name of the class, a colon, the visibility of the parent class, and the name of the parent class:

```
class 2DObject
{
   public:
   int x_pos;
   int y_pos;
};
```

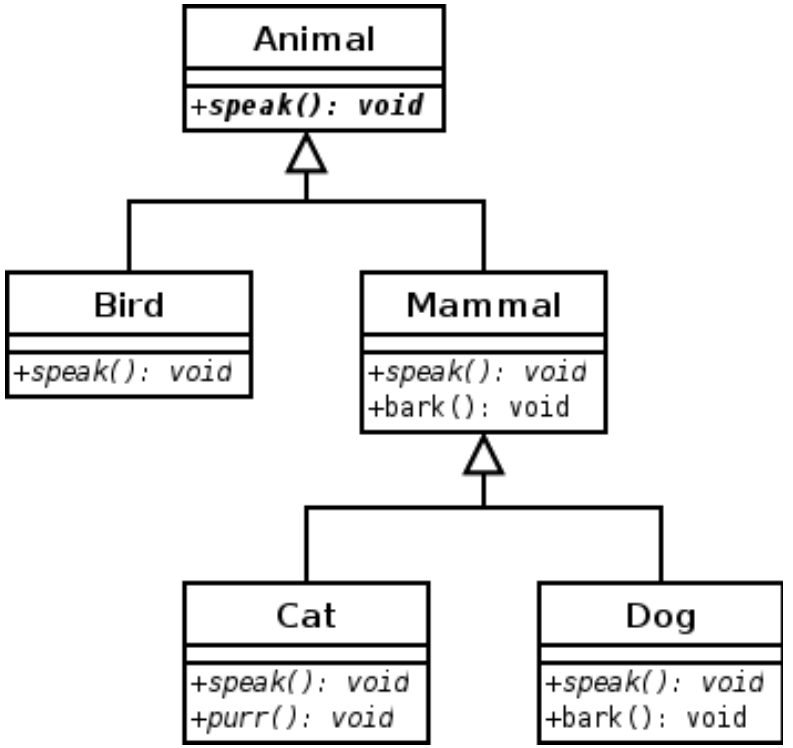
```
class Circle : public 2DObject
{
   public:
   int radius;
};
```

A Class Hierarchy

- Consider the following class hierarchy:

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

class Bird : public Animal {
public:
virtual void speak( )
{ cout << “twitter”; }
};
```



A Class Hierarchy cont.

```
class Mammal : public Animal {
public:
virtual void speak( ) { cout << “can't speak”; }
void bark( ) { cout << “can't bark”; }
};

class Cat : public Mammal {
public:
void speak( ) { cout << “meow!”; }
virtual void purr( ) { cout << “purrrrr”; }
};

class Dog : public Mammal {
public:
virtual void speak( ) { cout << “woof!”; }
void bark( ) { cout << “woof!”; }
};
```

Virtual and Non-Virtual Overriding

- Overriding occurs when a child class has a method with the exact same type signature as one of the parent class methods
- *Binding* is the process of deciding whether to execute the parent's version or the child's version of a method
- The keyword **`virtual`** determines whether *static binding* or *dynamic binding* is used
- **`virtual`** only appears in the class definition

Static Binding

- `virtual` is not used when declaring the method:

```
void bark( )
{ cout << “can't bark”; }
```

- The decision is made at compile time based on the type of the variable:

```
Dog * d = new Dog( );
Mammal * m = d;
d->bark( );  // woof!
m->bark( );  // can't bark.
```

Dynamic Binding

- **`virtual`** is used to declare the method:

```
virtual void speak( )
{ cout << “woof!”; }
```

- The binding decision is made at run-time based on the type of the object:

```
d->speak( );  // woof!
m->speak( );  // woof!
Animal * a = d;
a->speak( );  // woof!
```

Limitations

- The validity of calling a method is always static. If a method is not defined in a class or inherited from a parent class, it cannot be called:

```
Dog * d = new Dog( );
Animal * a = d;
d->bark( );  // woof!
a->bark( );  // Compile error, not allowed.
```

- Overriding only works with heap-resident values:

```
Mammal m = *d;
m.speak( );  // can't speak
```

More Limitations

- Child classes cannot change the type of binding
  - A method that is declared **`virtual`** in a parent class will always be **`virtual`** in a child class, even if **`virtual`** is not used in the child class
  - Similarly, a method that is not declared **`virtual`** in the parent class can never be made **`virtual`** in the child class
- Any method that is called from a constructor cannot be overridden
- Virtual methods are never inlined

Abstract Classes

- An *abstract class* (or *abstract base class*) is a class that contains *pure virtual methods*.
  - A pure virtual method does not have a body.
  - It is instead assigned a null value:

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

- Abstract base classes can only be used through inheritance
- It is impossible to create an instance of an abstract class

Downcasting

- C++ does not perform run time type checking
- If a pointer to a parent is type casted to point to a child the behavior can be unpredictable:

```
Animal * a = new Dog( );
Cat * c = (Cat *) a;
c->purr( );    // behavior is undefined
```

  - Note that only the data type associated with the pointer is being changed - the object the pointer points at is **not** changed in any way.

Downcasting cont.

- The RTTI (run-time type information system) provides a mechanism to protect against this:

```
Animal * a = new Dog( );
Cat * c = dynamic_cast<Cat *>( a );
if( c )
   cout << “Variable was a Cat” << endl;
else
   cout << “It was not a Cat” << endl;
```

- A **dynamic_cast** will return a valid pointer if the cast was successful, and 0 if not successful.

Name Resolution

- The following code will not compile:

```
Holstein * betty = new Holstein( );

betty->moo( 5 );
```

```
class Cow {
   public:
   void moo( int i );
};

class Holstein :
public Cow {
   public:
   void moo( string s );
   void moo( Cow & c );
};
```

Name Resolution cont.

- The compiler could not find **`moo( int i )`**
  - There are three name scopes
    - One for each class
    - the global scope
  - The scopes are nested inside each other
    - **`Holstein`** is in **`Cow's`** scope
    - **`Cow`** is in the global scope
  - The compiler first looks for the innermost scope that has the function **`moo`**, which will be **`Holstein`**
  - It then looks for a **`moo`** function that takes a single integer, but **`Holstein`** does not have one

Name Resolution

- The problem can be fixed by adding **`moo( int i )`** to the Holstein class:

```
class Holstein : public Cow {
  public:
  void moo( int i ) { Cow::moo( i ); }
  void moo( string s );
  void moo( Cow & c );
};
```

- The new method will simply call the same method in the parent class

A Forest, Not a Tree

- No C++ class is the ancestor of all classes
- A void pointer can be used as a generic pointer:

```
Animal * snoopy = new Dog( );
void * v = snoopy;

Dog * spike = dynamic_cast<Dog *>( v );
```

- A **`dynamic_cast`** is needed to safely change the void pointer to the original type

Private and Protected Inheritance

- Usually inheritance is public
- Protected inheritance changes public members in the parent to protected in the child
- Private inheritance changes public and protected members to private

```
class Pig : protected Mammal
{
   public:
   void oink( ) { cout << “Oink!”; }
   // The speak and bark methods can only be
   // accessed by child classes.
};
```

Virtual Destructors

- If any virtual methods are used, the destructor should be virtual to ensure that both the parent and child destructors are called

```
class Bird : public Animal {
   public:
   virtual ~Bird( ) { cout << “bird killed”; }
};

class Duck : public Bird {
   public:
   virtual void speak( ) { cout << “quack!”; }
   virtual ~Duck( ) { cout << “duck killed”; }
};
```