Overview of C++ Polymorphism

- Two main kinds of types in C++: native and user-defined
  - "User" defined types: declared classes, structs, unions
    - including types provided by the C++ standard libraries
  - Native types are "built in" to the C++ language itself: int, long, float, …
  - A typedef creates a new type *name* for another type (type aliasing)
- Public inheritance creates sub-types
  - Inheritance only applies to user-defined classes (and structs)
  - A publicly derived class is-a subtype of its base class
  - Known as "inheritance polymorphism"
- Template parameters also induce a subtype relation
  - Known as "interface polymorphism"
  - We'll cover how this works in depth, in later sessions
- Liskov Substitution Principle (for both kinds of polymorphism)
  - if S is a subtype of T, then wherever you need a T you can use an S

C++ Polymorphism, Continued

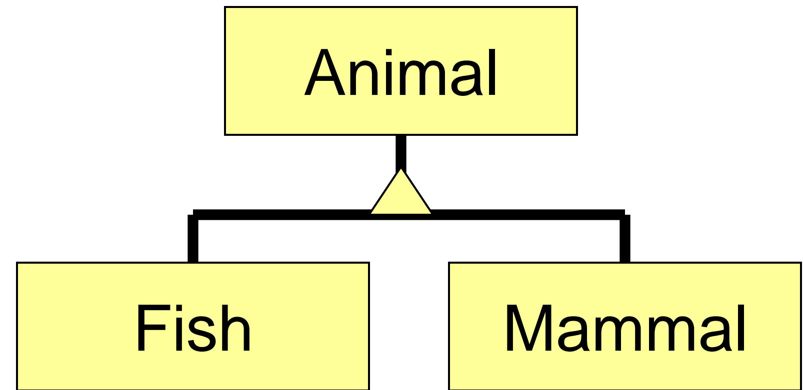
- Inheritance polymorphism depends on public virtual member functions in C++
  - Base class declares a member function virtual
  - Derived class overrides the base class's definition of the function

- Private or protected inheritance creates a form of encapsulation
  - Does not create a substitutable sub-type
  - A privately derived class wraps its base class
  - The class form of the Adapter Pattern uses this technique

Static vs. Dynamic Type

- The type of a variable is known statically (at compile time), based on its declaration

```
int i; int * p;
Fish f; Mammal m;
Fish * fp = &f;
```



- However, actual types of objects aliased by references & pointers to base classes vary dynamically (at run-time)

```
Fish f; Mammal m;
Animal * ap = &f;
ap = &m;
Animal & ar =
  get_animal();
```

- A base class and its derived classes form a *set* of types

  type(`*ap`) ∈ {`Animal`, `Fish`, `Mammal`}

  typeset(`*fp`) ⊂ typeset(`*ap`)

- Each type set is *open*
  - More subclasses can be added

Forms of Inheritance

- Derived class inherits from base class
- Public Inheritance (“is a”)
  - Public part of base class remains public
  - Protected part of base class remains protected
- Protected Inheritance (“contains a”)
  - Public part of base class becomes protected
  - Protected part of base class remains protected
- Private Inheritance (“contains a”)
  - Public part of base class becomes private
  - Protected part of base class becomes private

Public, Protected, Private Inheritance

```
class A {
public:
  int i;
protected:
  int j;
private:
  int k;
};

Class B : public A {
// ...
};
Class C : protected A {
// ...
};
Class D : private A {
// ...
};
```

- Class A declares 3 variables
  - i is **`public`** to all users of class A
  - j is **`protected`**. It can only be used by methods in class A or its derived classes (+ friends)
  - k is **`private`**. It can only be used by methods in class A (+ friends)
- Class B uses public inheritance from A
  - i remains **`public`** to all users of class B
  - j remains **`protected`**. It can be used by methods in class B or its derived classes
- Class C uses protected inheritance from A
  - i becomes **`protected`** in C, so the only users of class C that can access i are the methods of class C
  - j remains **`protected`**. It can be used by methods in class C or its derived classes
- Class D uses private inheritance from A
  - i and j become **`private`** in D, so only methods of class D can access them.

Class and Member Construction Order

```
class A {
public:
  A(int i) :m_i(i) {
    cout << "A“ << endl;}
  ~A() {cout<<"~A"<<endl;}
private:
  int m_i;
};
class B : public A {
public:
  B(int i, int j)
    : A(i), m_j(j) {
    cout << “B” << endl;}
  ~B() {cout << “~B” << endl;}
private:
   int m_j;
};
int main (int, char *[]) {
   B b(2,3);
   return 0;
};
```

- In the main function, the B constructor is called on object b
  - Passes in integer values 2 and 3
- B constructor calls A constructor
  - passes value 2 to A constructor via base/member initialization list
- A constructor initializes **`m_i`**
  - with the passed value 2
- Body of A constructor runs
  - Outputs “A”
- B constructor initializes **`m_j`**
  - with passed value 3
- Body of B constructor runs
  - outputs “B”

Class and Member Destruction Order

```
class A {
public:
  A(int i) :m_i(i) {
    cout << "A“ << endl;}
  ~A() {cout<<"~A"<<endl;}
private:
  int m_i;
};
class B : public A {
public:
  B(int i, int j) :A(i), m_j(j) {
    cout << “B” << endl;}
  ~B() {cout << “~B” << endl;}
private:
   int m_j;
};
int main (int, char *[]) {
   B b(2,3);
   return 0;
};
```

- B destructor called on object b in main
- Body of B destructor runs
  - outputs “~B”
- B destructor calls “destructor” of **m_j**
  - **int** is a built-in type, so it’s a no-op
- B destructor calls A destructor
- Body of A destructor runs
  - outputs “~A”
- A destructor calls “destructor” of **m_i**
  - again a no-op
- Compare orders of construction and destruction of base, members, body
  - at the level of each class, order of steps is reversed in constructor vs. destructor
  - ctor: base class, members, body
  - dtor: body, members, base class

Virtual Functions

```
class A {
public:
  A () {cout<<" A";}
  virtual ~A () {cout<<" ~A";}
  virtual f(int);
};

class B : public A {
public:
  B () :A() {cout<<" B";}
  virtual ~B() {cout<<" ~B";}
  virtual f(int) override; //C++11
};

int main (int, char *[]) {
   // prints "A B"
   A *ap = new B;

   // prints "~B ~A" : would only
   // print "~A" if non-virtual
   delete ap;

   return 0;
};
```

- Used to support polymorphism with pointers and references
- Declared virtual in a base class
- Can override in derived class
  - Overriding only happens when signatures are the same
  - Otherwise it just *overloads* the function or operator name
    - More about overloading next lecture
- Ensures derived class function definition is resolved dynamically
  - E.g., that destructors farther down the hierarchy get called
- Use **`final`** (C++11) to prevent overriding of a virtual method
- Use **`override`** (C++11) in derived class to ensure that the signatures match (error if not)

Virtual Functions

```
class A {
public:
  void x() {cout<<"A::x";};
  virtual void y() {cout<<"A::y";};
};

class B : public A {
public:
  void x() {cout<<"B::x";};
  virtual void y() {cout<<"B::y";};
};

int main () {
   B b;
   A *ap = &b; B *bp = &b;
   b.x (); // prints "B::x"
   b.y (); // prints "B::y"
   bp->x (); // prints "B::x"
   bp->y (); // prints "B::y"
   ap->x (); // prints "A::x"
   ap->y (); // prints "B::y"
   return 0;
};
```

- Only matter with pointer or reference
  - Calls on object itself resolved statically
  - E.g., **`b.y();`**
- Look first at pointer/reference type
  - If non-virtual there, resolve statically
    - E.g., **`ap->x();`**
  - If virtual there, resolve dynamically
    - E.g., **`ap->y();`**
- Note that virtual keyword need not be repeated in derived classes
  - But it's good style to do so
- Caller can force static resolution of a virtual function via scope operator
  - E.g., **`ap->A::y();`** prints "**`A::y`**"

Potential Problem: Class Slicing

- Catch derived exception types by reference
- Also pass derived types by reference
- Otherwise a temporary variable is created
  - Loses original exception's "dynamic type"
  - Results in "the class slicing problem" where only the base class parts and not derived class parts copy

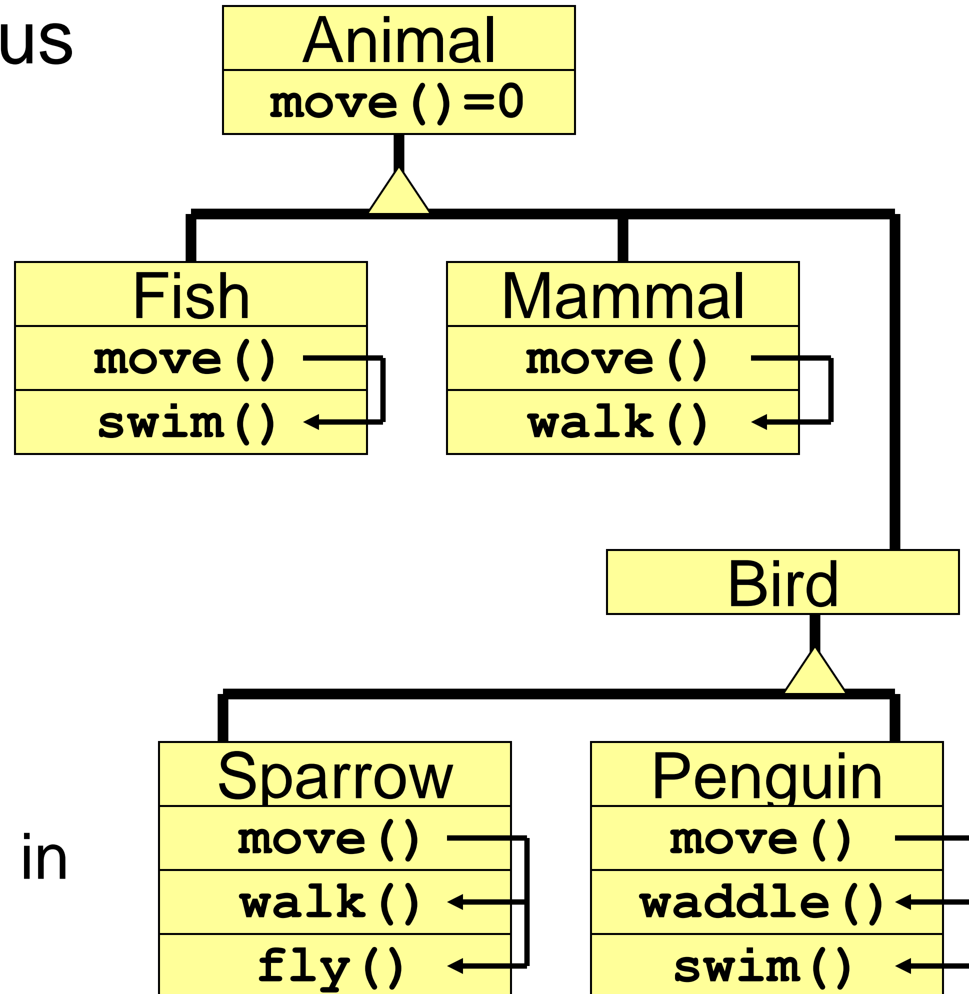
Pure Virtual Functions

```
class A {
public:
  virtual void x() = 0;
  virtual void y() = 0;
};

class B : public A {
public:
  virtual void x();
};

class C : public B {
public:
  virtual void y();
};

int main () {
  A * ap = new C;
  ap->x ();
  ap->y ();
  delete ap;
  return 0;
};
```

- A is an abstract (base) class
  - Similar to an interface in Java
  - Declares pure virtual functions (**=0**)
  - May also have non-virtual methods, as well as virtual methods that are not pure virtual
- Derived classes override pure virtual methods
  - B overrides **x()**, C overrides **y()**
- Can't instantiate an abstract class
  - class that declares pure virtual functions
  - or inherits ones that are not overridden
  - A and B are abstract, can create a C
- Can still have a pointer or reference to an abstract class type
  - Useful for polymorphism

Design with Pure Virtual Functions

- Pure virtual functions let us specify interfaces appropriately
  - But let us defer implementation decisions until later (subclasses)



- As the type hierarchy is extended, pure virtual functions are replaced
  - By virtual functions that fill in (and may override) the implementation details
  - Key idea: *refinement*

Summary: Tips on Inheritance Polymorphism

- A key tension
  - Push common code and variables up into base classes
  - Make base classes as general as possible
- Use abstract base classes to declare interfaces
- Use public inheritance to make sets of polymorphic types
- Use private or protected inheritance only for encapsulation
- Inheritance polymorphism depends on dynamic typing
  - Use a base-class pointer (or reference) if you want inheritance polymorphism of the objects pointed to (or referenced)
  - Use virtual member functions for dynamic overriding
- Even though you don't have to, label each *inherited* virtual (and pure virtual) method "virtual" in derived classes
- Use **`final`** (C++11) to prevent overriding of a virtual method
- Use **`override`** (C++11) to make sure signatures match