Polymorphism, Cont.

Review, Discussion, and Details

Review

- Polymorphism
  - When a call to a member function executes different code depending on the type of object that invokes the function.

- Virtual function
  `virtual void example();`
  - A base-class function that is declared as **virtual**, indicating to the compiler that it should wait until run-time to determine which version of the function should run.
  - A virtual function can be overridden if it is re-defined in a child class.

Review

- Pure virtual function (also known as abstract function)

  `virtual void example() = 0;`
  - A virtual function that has no definition in the base class.
  - Used when you are intending for child classes to implement the function.

- Abstract class
  - Any class that has one or more pure virtual functions.
  - An abstract class cannot be instantiated (i.e. you cannot create an object out of an abstract class).

General Observations

- Polymorphism & Inheritance often go hand-in-hand
- Important notes:
  - A derived class does **NOT** inherit:
    - The base class's contructors or destructor
    - Friends of the base class
  - If you implement a destructor in a class that utilizes inheritance, **make sure it is marked as virtual**!
    - The compiler will often remind you
    - If you ignore this advice, your code might run the wrong destructor ☹

    ```
    virtual ~BaseClass();
    virtual ~DerivedClass();
    ```

New Vocabulary

- override specifier
  `virtual void example() override;`
  - Used when you want to tell the compiler that this function is intended to override some function in the base class.
  - Not required but good to use because you lower the chance of bugs

- final specifier (for a function)
  `virtual void example() final;`
  - Used when you want to tell the compiler that no child class is allowed to override this function.

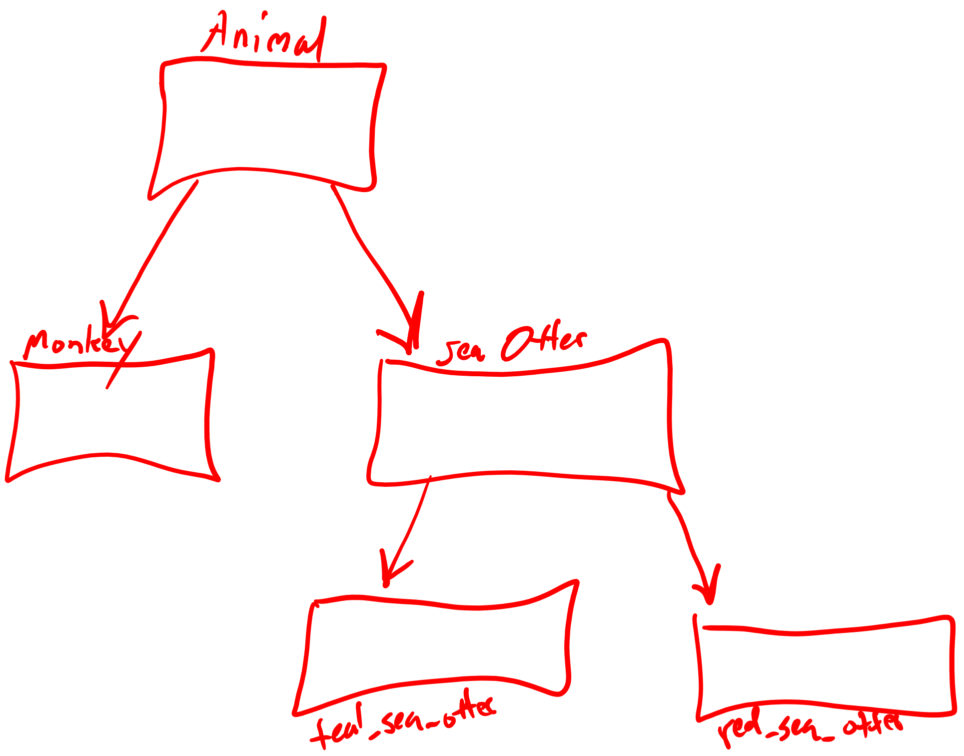
- final specifier can also be applied to an entire class:
  `class Elephant final : public Animal {}`
  - In this context, **final** means that no child class can exist for Elephant. In other words, no class can inherit from Elephant.

New Vocabulary

- Late binding / dynamic binding
  - Used when the type of object <u>is evaluated at runtime</u>. The compiler generated code will check to determine the object type and then execute the correct version of code.
  - This is what allows C++ to support polymorphism.
  - We do this using the **virtual** keyword
- Early binding / Static binding
  - The default behavior in C++. A function call always executes the same version of code.

A Discussion of Objects

- Recall that inheritance involves base classes and derived classes.
- In some cases you might want to convert objects into different types
- Consider the inheritance hierarchy shown in this image:



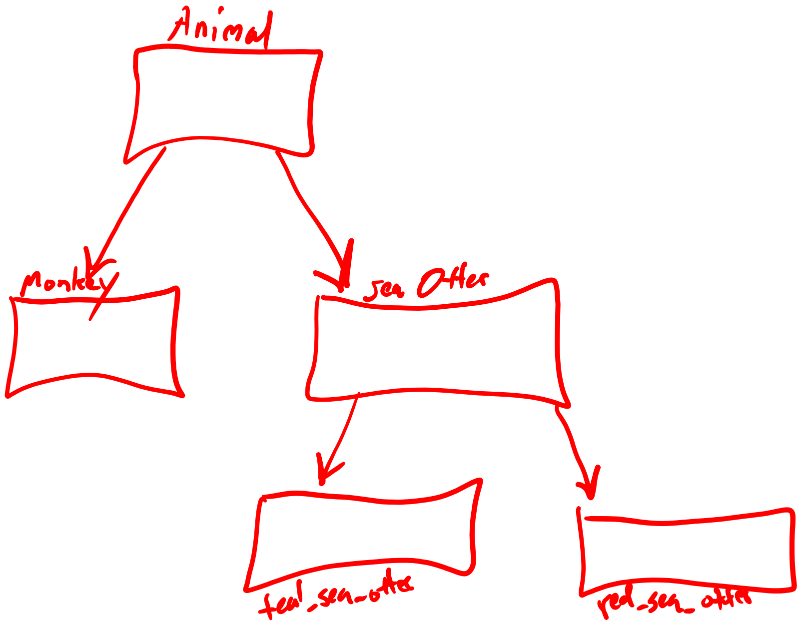
Object Slicing

- Imagine that we want to create an array of animals
- Perhaps we want to convert a Red_Sea_Otter object into an Animal so that we can place this object into the array
- The naïve approach is to use object slicing (probably not what you wanted!)

```
// an example of "object slicing"

Red_Sea_Otter rso;
Animal a1 = rso; // <-- object slicing

// a1 has now "lost" any member variables
// that were specific to Red_Sea_Otter
```



The Pointer Approach

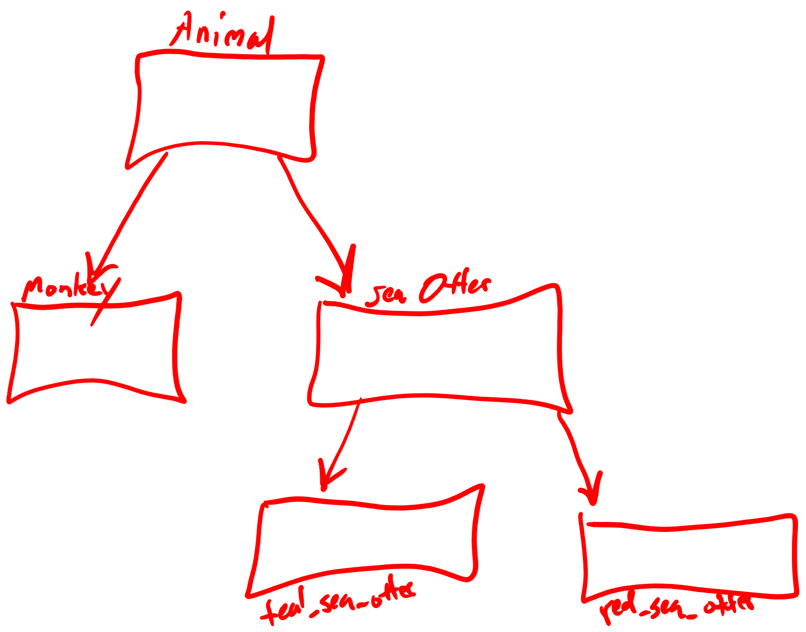
- Pointers allow us to treat an object as a different type without permanently losing data
- Create an array of **Animal pointers** instead of storing **Animal objects**

```
Animal* array[2];
Red_Sea_Otter rso;
Monkey m1;


array[0] = &rso;
array[1] = &m1;

// now a for loop can be used to
// process the array

for (int i = 0; i < 2; i++;)
  array[i]->make_noise();
```



One detail…

- It was easy to convert a Red_Sea_Otter pointer into an Animal pointer
- What about going in the opposite direction?

```
Red_Sea_Otter rso;
Animal* a_ptr = &rso;

// a_ptr is an Animal pointer that points
// to a Red_Sea_Otter

// now attempt to convert this into a
// Red_Sea_Otter pointer…
Red_Sea_Otter* r_ptr = a_ptr;
```

This does not work!

```
prog.cpp: In function 'int main()':
prog.cpp:24:25: error: invalid conversion from 'Animal*' to 'Red_Sea_Otter*' [-fpermissive]
  Red_Sea_Otter* r_ptr = a_ptr;
                         ^
```

Static Object Casting

- How could we convert an Animal* into Red_Sea_Otter* ?

```
Red_Sea_Otter rso;
Animal* a_ptr = &rso;

// a_ptr is an Animal pointer that points
// to a Red_Sea_Otter

// use a static cast to convert this into a
// Red_Sea_Otter pointer…
Red_Sea_Otter* r_ptr = static_cast<Red_Sea_Otter*>(a_ptr);

// r_ptr can now be used as normal
r_ptr->swim_forward();

// if a_ptr was not pointing to a Red_Sea_Otter, the cast
// will fail silently! (Often causing a SEGFAULT later!)
```

This only works if a_ptr actually points to a Red_Sea_Otter and not some other type of Animal.

Problems with a Static Cast

- The static cast does not allow us to determine if the pointer is actually valid
  - Doesn’t work if you have an array of pointers to random animals
  - E.g. if you inadvertently try to cast a Monkey* into a Red_Sea_Otter* bad things will happen

Introducing the Dynamic Cast

- Useful when you don’t know what the type of the object is
  - e.g. You have an animal pointer, but you don’t know what kind
- Cast is verified at runtime (rather than blindly proceeding)
  - If the cast was successful, you get a valid pointer
  - If the cast failed, you get a nullptr
- Dynamic casts also work with references
  - If you try to cast an invalid reference, an exception will be thrown

Example of Dynamic Object Casting

- Demonstrates how to verify whether the resulting pointer is valid
- Allows your code to handle multiple types of objects

```
Red_Sea_Otter rso;
Red_Sea_Otter* tmp;
Animal* a_ptr = &rso;

tmp = dynamic_cast<Red_Sea_Otter*>(a_ptr);
if (tmp != nullptr) {
   tmp->swim(12);
} else {
   cout << "Couldn’t cast a_ptr into a Red_Sea_Otter*" << endl;
}
```