Object-Oriented Programming: Polymorphism

Chapter 12

TOPICS

INTRODUCTION AND DEFINITIONS

PROGRAMMING IN THE GENERAL

To “program in the general” means to write code in such a way that it can accommodate many situations

One very basic example is to use variables that can store values, rather than hard-coding those values into your program

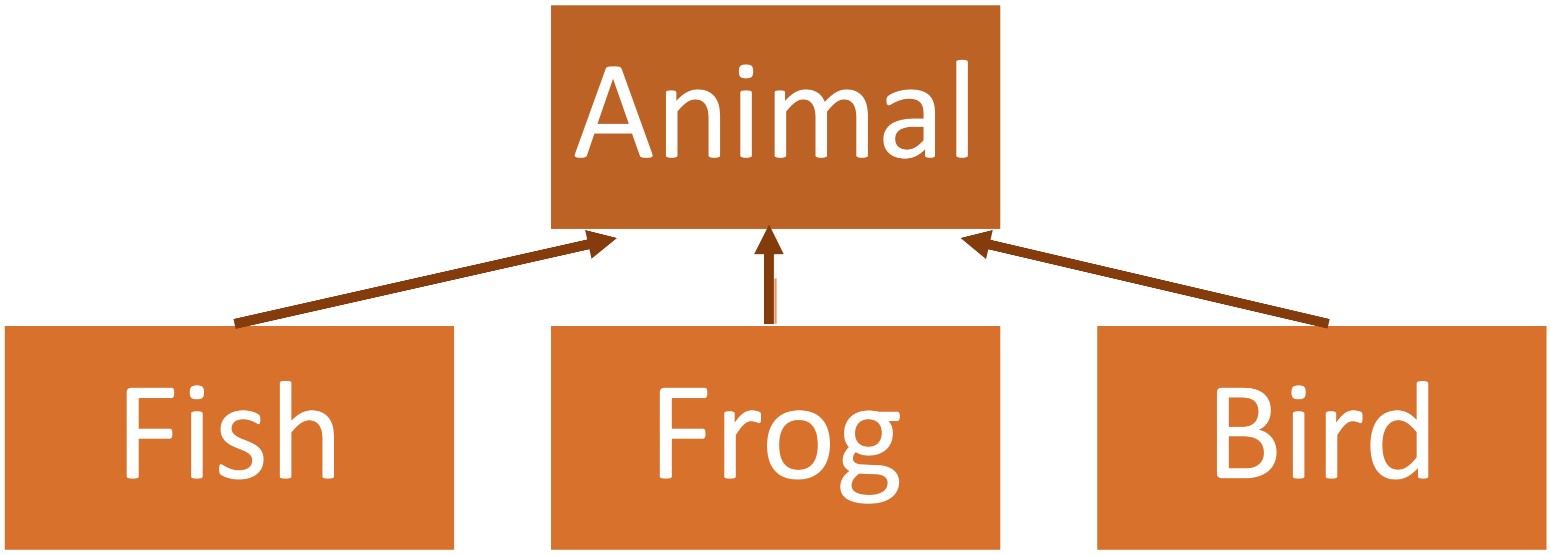
Putting code that will be reused into functions, and into classes, are also examples of this idea

In general, programming in the general is a better approach than “programming in the specific”, because it can simplify programming

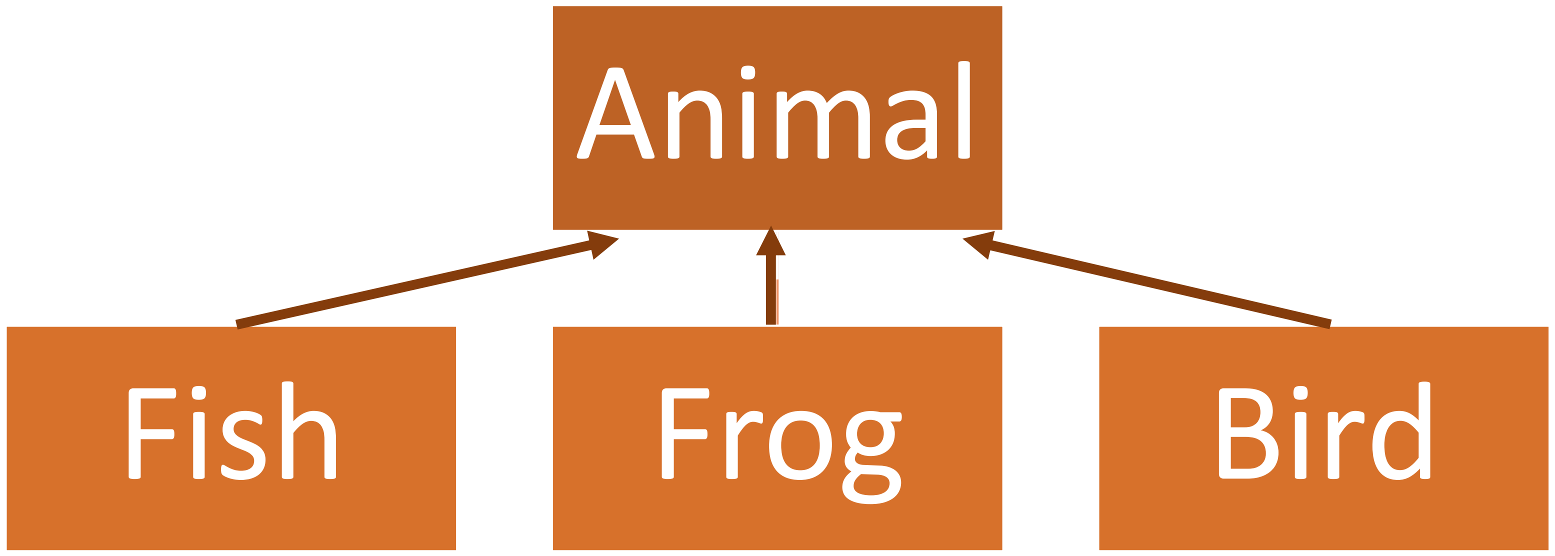
Inheritance is a Form of reuse

Remember that inheritance is a way to reuse the code in classes

If classes are similar enough, we can arrange them in an inheritance hierarchy:

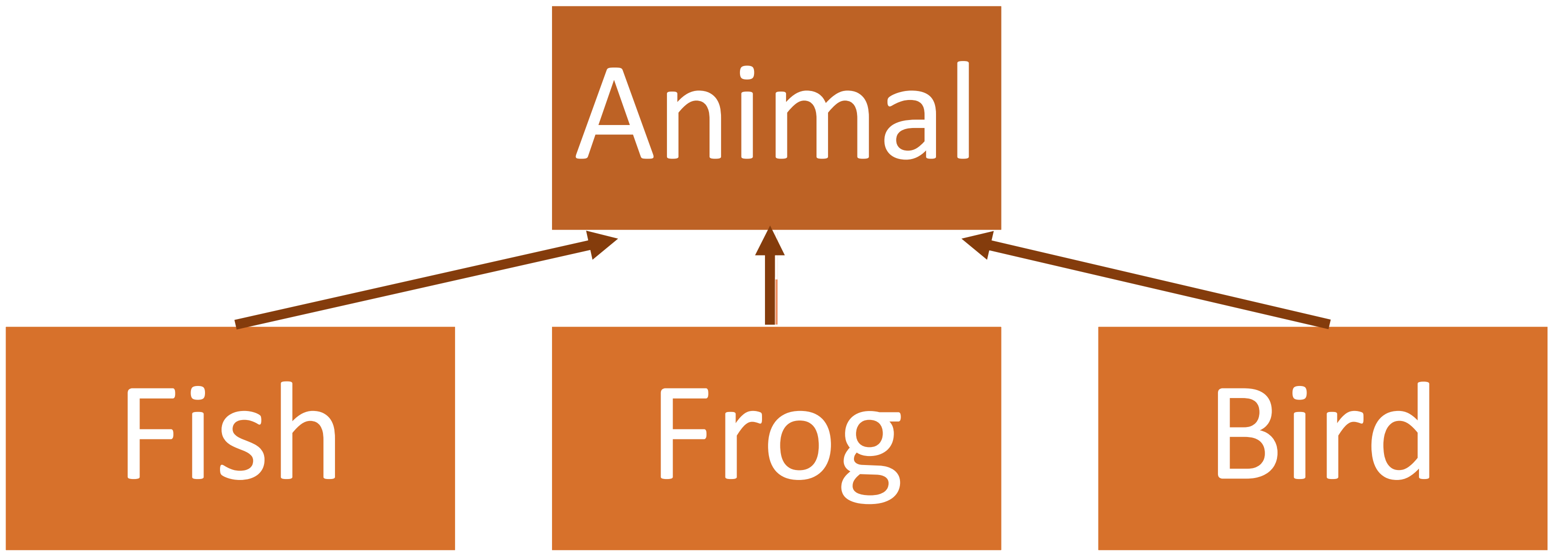


Which of these are Animals?



Which of these are Animals?

EACH CLASS DERIVES FROM BASE CLASS ANIMAL



They all are Animals.

Example – Animal Inheritance Hierarchy

Suppose a program simulates the movement of several types of animals for a biological study.

The classes Fish, Frog and Bird represent three types of animals

We may need to add classes later, if other animals are added to the study

In general, an animal moves, and has a location

So a class in this Animal hierarchy would contain a **move** function and a **location** data member (x, y coordinates)

Where should the move function, and the location data be located?

Example – Animal Inheritance Hierarchy – Move should be in Base Class

Since all animals move and have a location, that functionality is in common, and it makes sense to put it in one place, so all Animals in the hierarchy can use it

So move() and the location should go in the Animal base class, so that the Fish, Frog and Bird classes could inherit that functionality.

But there's a problem.

Example – Animal Inheritance Hierarchy – But Each Moves Differently

Each kind of animal in the hierarchy **moves** in a different way

- A Fish swims
- A Frog jumps
- A Bird flies

So, do we take advantage of the reusability of code that inheritance gives us, or do we write separate move() functions for each class in the hierarchy, thus defeating the purpose of the inheritance hierarchy in the first place?

The answer is we do both…

Polymorphism

“poly” = many

“morph” = form or structure

In a driver program, we create a variable that may refer to any of the classes in the inheritance hierarchy

This is the idea of **polymorphism**

Polymorphism Definition

Creating and using a variable that may refer to objects:

1. whose class is **not known at compile time**, and

2. which **respond at run time according to the actual class** of the object to which they refer

So polymorphism gives us a way to process Animal objects, rather than Fish, Frog, or Bird objects.

Back to the Animal Inheritance Hierarchy Example

A driver program that handles the simulation code for our study maintains an array or vector that can hold Animal objects:

Animal listOfAnimals[10] –or–

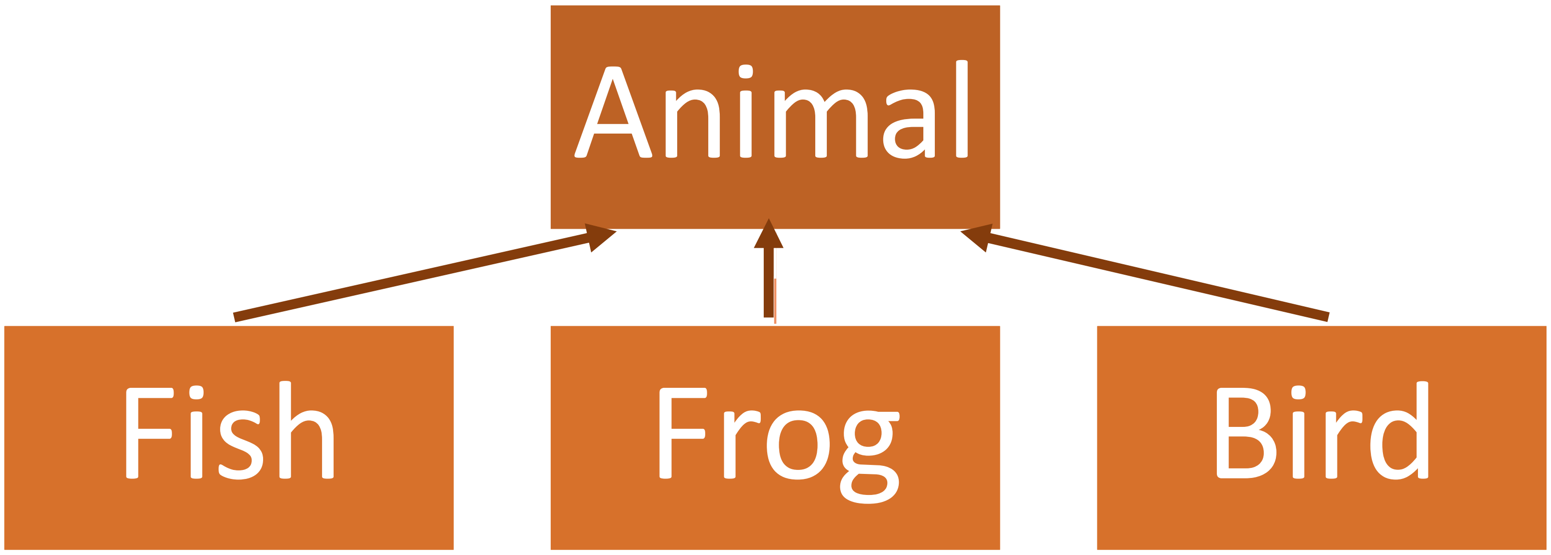
array<Animal, 10> listOfAnimals –or–

vector<Animal> listOfAnimals(10)

What kinds of objects can be stored in these data structures?

Back to the Animal Inheritance Hierarchy Example – Hierarchy

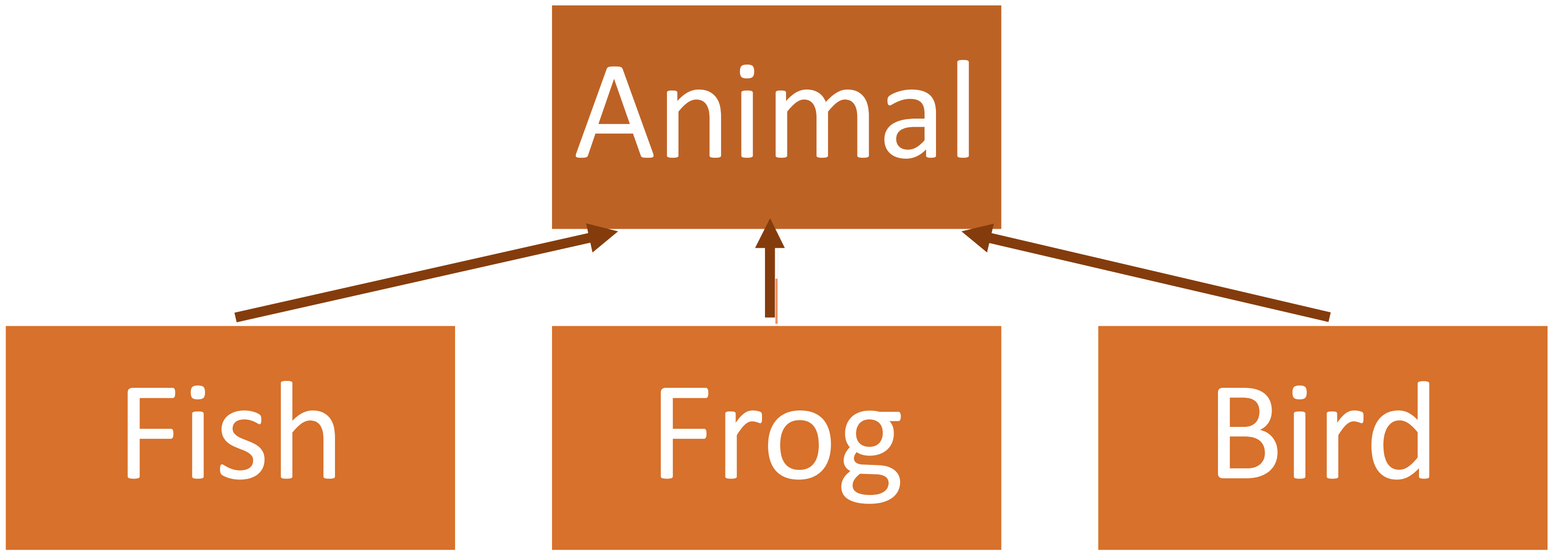
Any class that is part of the Animal inheritance hierarchy:



Why?

Back to the Animal Inheritance Hierarchy Example – Remember "Is-A"

Any class that is part of the Animal inheritance hierarchy:



Because Fish, Frog, and Bird **ARE ALSO Animals**.
Remember the "is-a" relationship?

Back to the Animal Inheritance Hierarchy Example – Another Problem

So as the driver creates Fish, Frog, or Bird objects, it can store them in the data structure created for Animal objects.

But then there's a problem.

What happens when the code to make these objects move needs to execute?

It would be convenient (and more reusable) if we could loop through the data structure, and call each object's move() function

Back to the Animal Inheritance Hierarchy Example – Move Again

But each type of animal moves in a different way:

- a Fish swims three feet
- a Frog jumps five feet
- a Bird flies ten feet

We can't write one move function that can accommodate all these movements

So we write a separate move function for each class, according to what's needed in that class to execute movement

ANIMAL DERIVED CLASSES

Each class in the inheritance hierarchy inherits a generic move function from the Animal base class

But each derived class **implements** that function in a different way

The concept of providing different implementations of function that exists in a base class is called **redefining** (and also **overriding)**

Driver class issues the same message (“move”) to each animal object in the data structure

Each object knows how to modify its x-y coordinates appropriately for its specific type of movement

Each Object Knows its Own Move Logic

Each object in a hierarchy knows how to respond to the same function call

This is the **key concept of polymorphism**

Same message “move” sent to a variety of objects has “many forms” of results

Easily Extensible Programs

With polymorphism, systems are easily extensible

- New class can be added with little change,
- If the new class is part of the inheritance hierarchy that the program processes generically

The only parts of program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

Quick Review of Pointers

But first, a review

What is a variable?

Variable

A name for a memory cell (or groups of cells)

What kind of information can be stored in a variable?

Variables Store Data

In C++, we can store three types of data in a variable:

1. Fundamental (primitive) data types
2. Class data types
3. Addresses of any of the above

Fundamental (Primitive) Data Type Variables

bool, char, int, long, float, etc.

So, variables that are **primitive data types** store true/false, single characters, and various kinds of numbers

Class Data Type Variables

Variables that are **class data types** store information about objects

Remember, an object is one instance of a class

Examples:

```
Car myCar;

Car myCar{“Honda”};

Car myCar{“Honda”, 5, “SUV”};
```

Pointer Type Variables

Variables that are **pointer types** store addresses of other memory cells

Those other memory cells can be of any data type

How is a pointer variable created?

Creating a Pointer Type Variable

The same way any kind of variable is created, with a declaration statement:

```
datatype* variableName;
```

Examples:

```
Car* carPtr;

int* numPtr;
```

Storing Data in a Pointer Type Variable

How does data get into a pointer type variable?

What IS the data that will be stored in a pointer type variable?

Storing Data in a Pointer Type Variable – Data is an Address

With an **assignment statement** or **initialization with brackets**, just like any other kind of variable

The actual data stored is the **address** of a memory cell

Assignment:

```
datatype* variableName = &anotherVariable;
```

Initialization w/brackets:

```
datatype* variableName{&anotherVariable};
```

Storing Data in a Pointer Type Variable Examples

Examples:

Assignment:

```
Car* carPtr = &alreadyCreatedCarObject;
```

Initialization w/brackets:

```
Car* carPtr{&alreadyCreatedCarObject};
```

Storing Data in a Pointer Type Variable – Object is Not The Variable

Important to distinguish between the **object**, and a **pointer variable used to refer to the object**

What is the difference between this approach:

```
Car* carPtr = &alreadyCreatedCarObject;
```

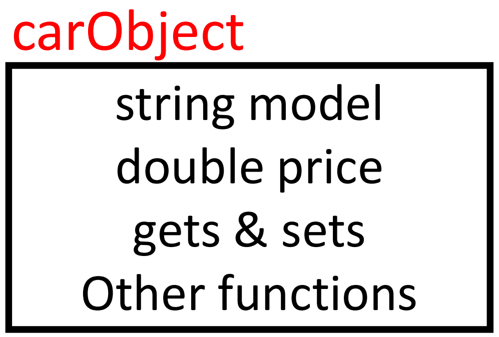
And this approach:

```
Car* carPtr;
carPtr = &alreadyCreatedCarObject;
```

Pointer Variables are not the Same as Objects – Step 1

In this example, when carPtr is declared, it is immediately assigned a value:

Step 1: **Car carObject;**
Step 2: **Car* carPtr = &carObject;**

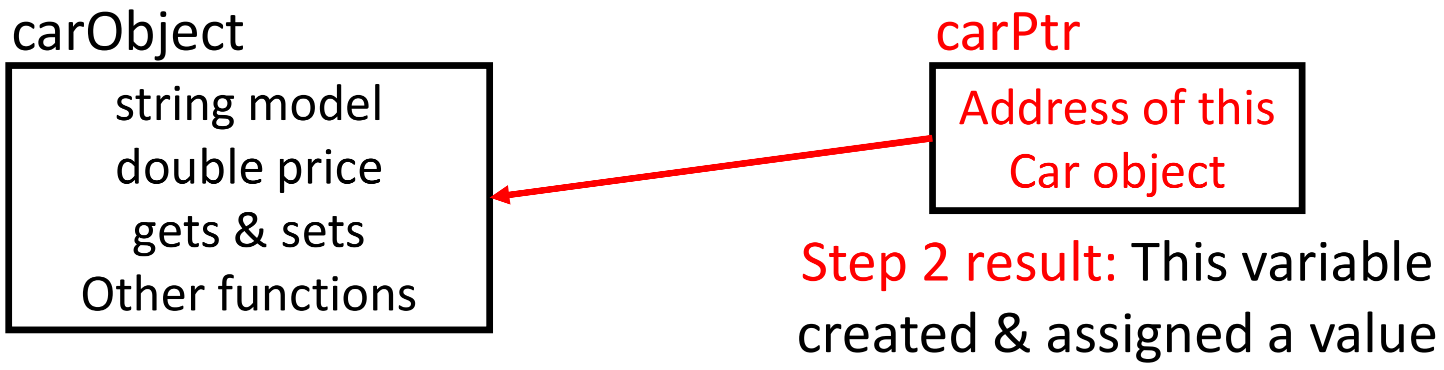


Step 1 result: This object created

Pointer Variables are not the Same as Objects – Step 2

In this example, when carPtr is declared, it is immediately assigned a value:

Step 1: **Car carObject;**
Step 2: **Car* carPtr = &carObject;**



carObject
string model
double price
gets & sets
Other functions
carPtr
Address of this Car object
Step 2 result: This variable created & assigned a value

Pointer Variables are not the Same as Objects – Step 1 Again

But in this example, when carPtr is declared, it is NOT immediately assigned a value. It is assigned a value in another step.

Step 1: **Car carObject;**
Step 2: **Car* carPtr;**
Step 3: **carPtr = &carObject;**

carObject

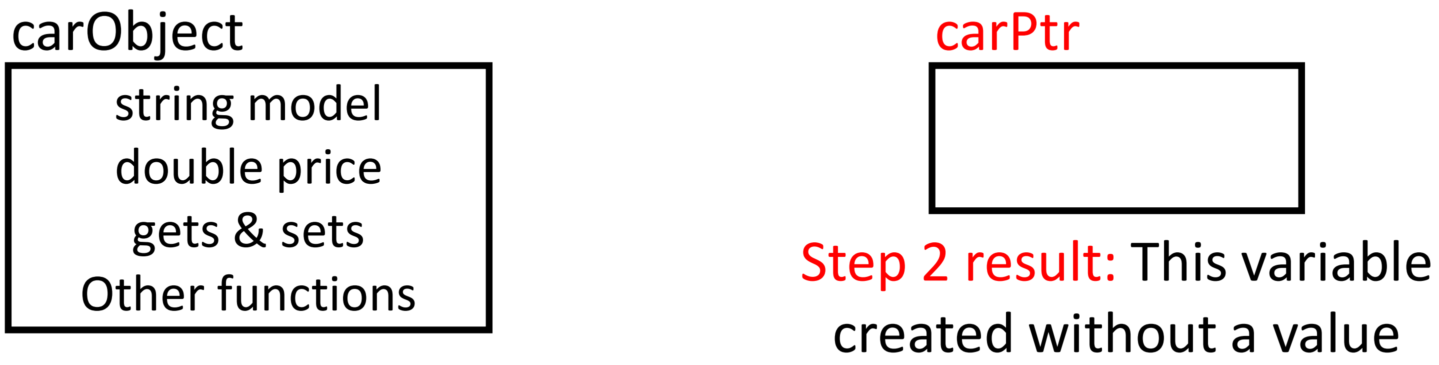
string model
double price
gets & sets
Other functions

Step 1 result: This object created

Pointer Variables are not the Same as Objects – Step 2 Again

But in this example, when carPtr is declared, it is NOT immediately assigned a value. It is assigned a value in another step.

Step 1: **Car carObject;**
Step 2: **Car* carPtr;**
Step 3: **carPtr = &carObject;**

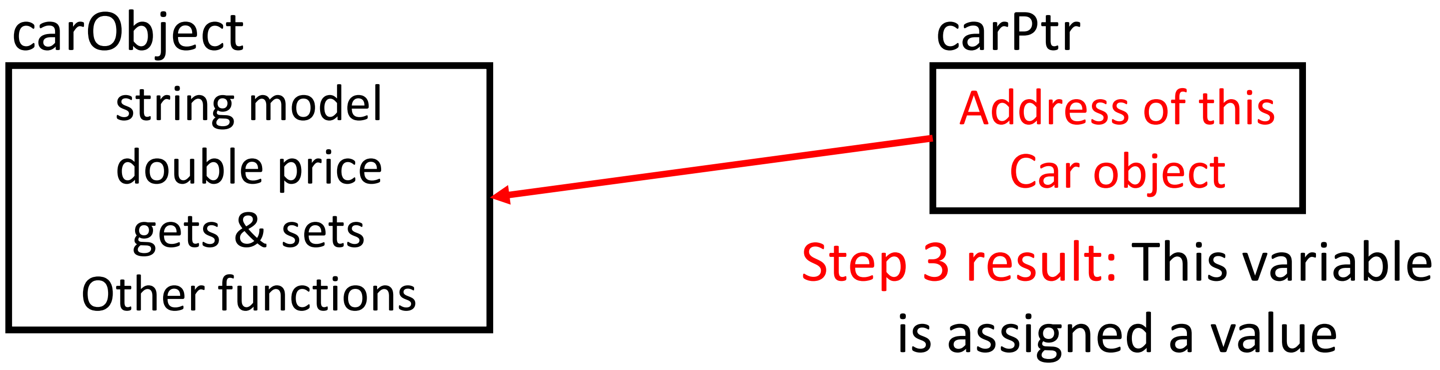


Step 2 result: This variable created without a value

Pointer Variables are not the Same as Objects – Step 3

But in this example, when carPtr is declared, it is NOT immediately assigned a value. It is assigned a value in another step.

```
Step 1:   Car carObject;
Step 2:   Car* carPtr;
Step 3:   carPtr = &carObject;
```


carObject
string model
double price
gets & sets
Other functions
carPtr
Address of this Car object
Step 3 result: This variable is assigned a value

Pointer Variables are not the Same as Objects

This review of pointers has a point, which we will see in a bit…

Sorry…

Demonstrating Polymorphic Behavior

Demonstrating Polymorphic Behavior – Next Example

Next example uses a **base class pointer variable** to manipulate a **derived class object**

- Demonstrates that an object of a derived class can be treated as an object of its base class

A program can create an array of base class pointer variables that refer to objects of many derived class types

- Allowed because each derived class object is … what?

Demonstrating Polymorphic Behavior – Next Example uses array

Next example uses a **base class pointer variable** to manipulate a **derived class object**

- Demonstrates that an object of a derived class can be treated as an object of its base class

A program can create an array of base class pointer variables that refer to objects of many derived class types.

- Allowed because each derived class object is … what? **Also an object of its base class**

Base Classes are Not Derived classes

A base class object cannot be treated as a derived class object

Is-a relationship applies only **up** the hierarchy from a derived class to its direct (and indirect) base classes

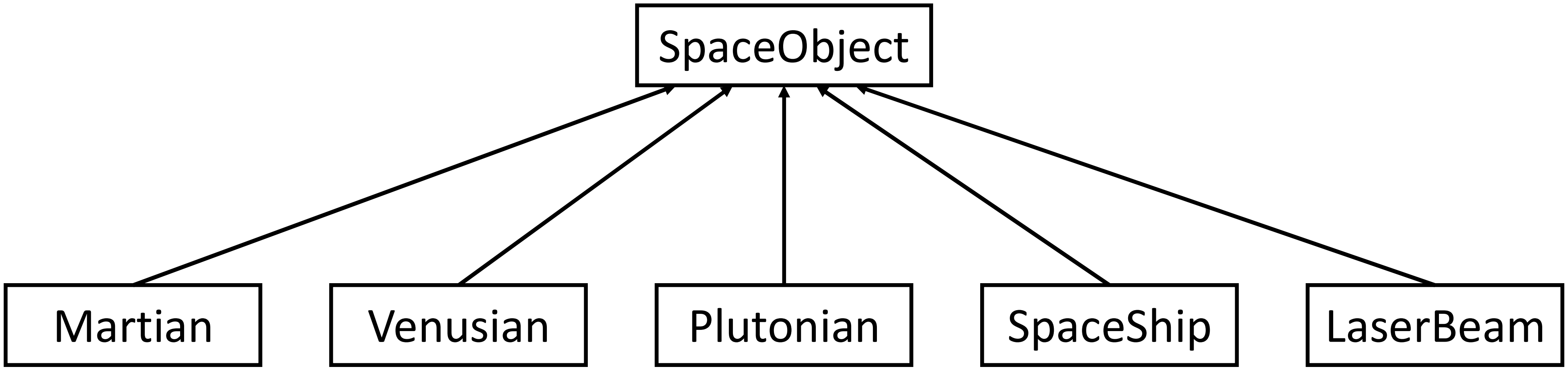
Downcasting is an exception to this rule

- Enables a program to invoke derived class functions that are not in the base class

SpaceObject Hierarchy Example

A game manipulates objects of classes Martian, Venusian, Plutonian, SpaceShip and LaserBeam

Each derives from SpaceObject

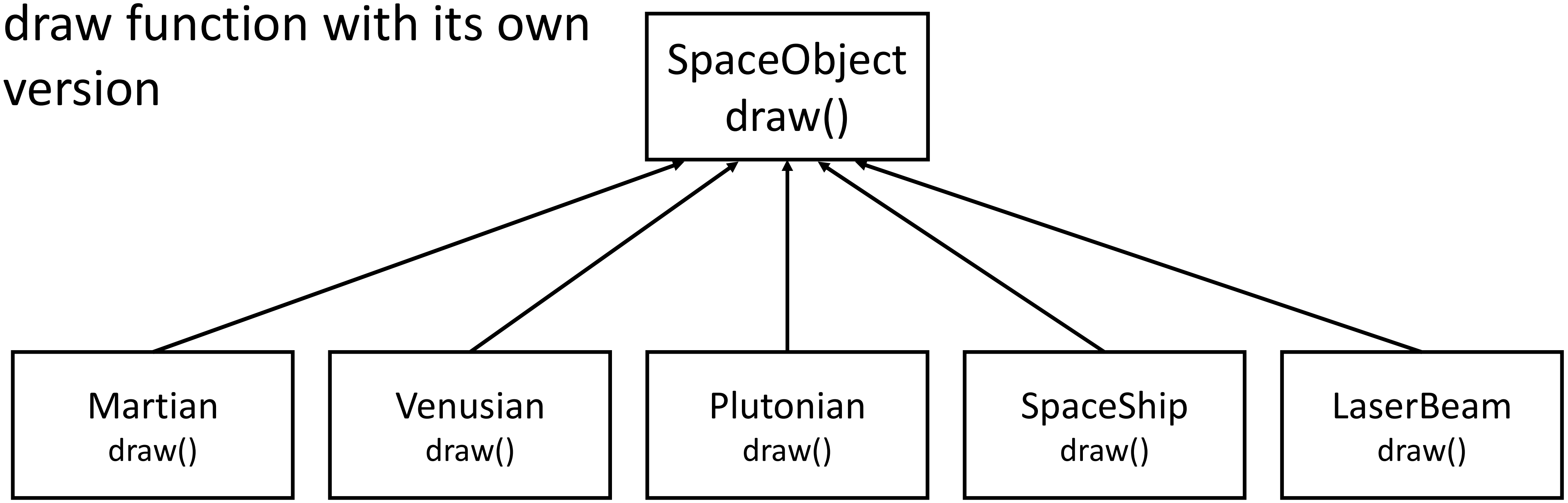


So all of these are SpaceObjects…

SpaceObject Hierarchy Example – Part 1

SpaceObject contains a draw function

Each derived class **redefines** that draw function with its own version



SpaceObject Hierarchy Example – Part 2

Game's screen manager code needs to maintain pointers to objects in this hierarchy

So that it can tell each one to draw itself when necessary

SpaceObject Hierarchy Example – Part 3

What if we wanted a list of pointers to objects of these classes?

What do we call a list of data?

SpaceObject Hierarchy Example – Part 4

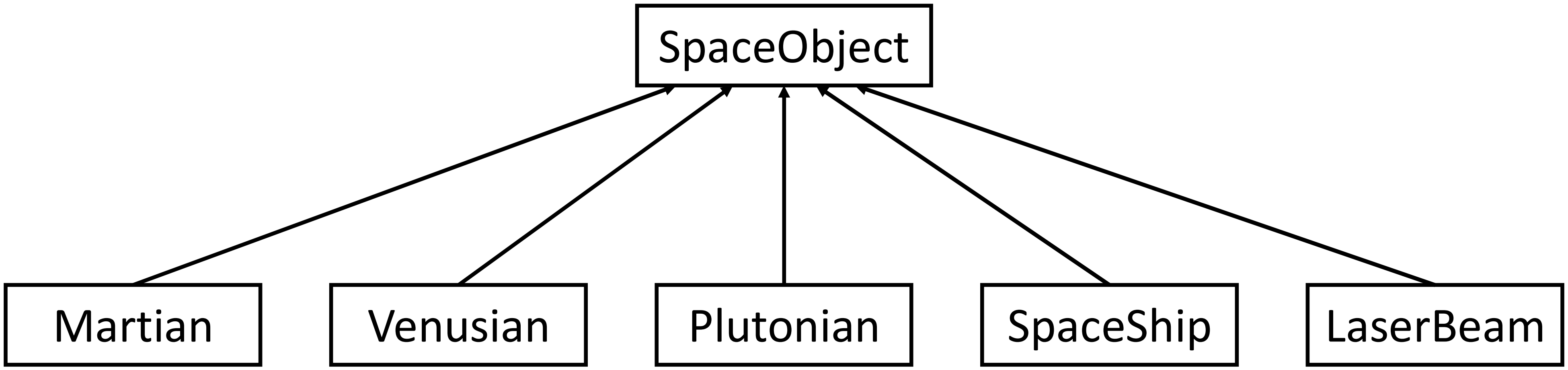
A list of data is an array (or a vector)

What do we know about the kind of data that can be stored in an array (or a vector)?

SpaceObject Hierarchy Example – Part 5

An array (or vector) can only store **one type of data**, for example, only integers or only Car objects

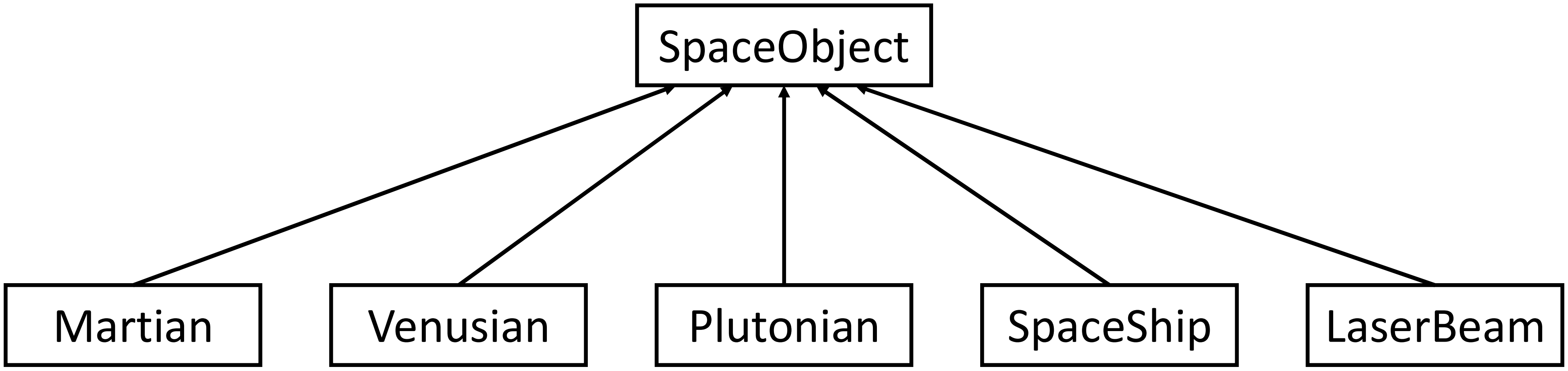
If we wanted to create an array that could hold **any** of these objects, what type of data would it hold?



SpaceObject Hierarchy Example – Part 6

It would be an array of SpaceObject pointers

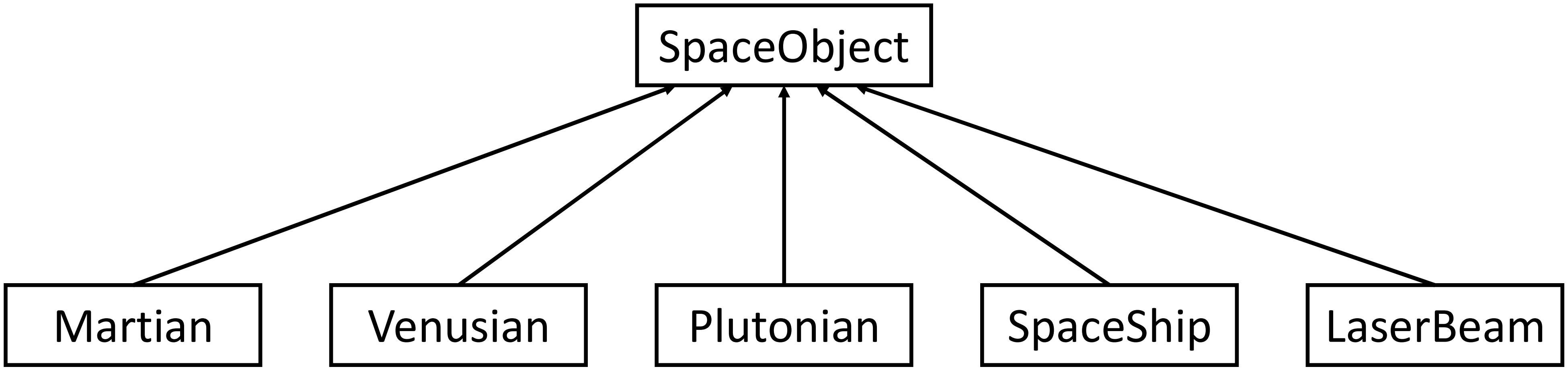
Why?



SpaceObject Hierarchy Example – Part 7

Because every class in this hierarchy **IS A** SpaceObject

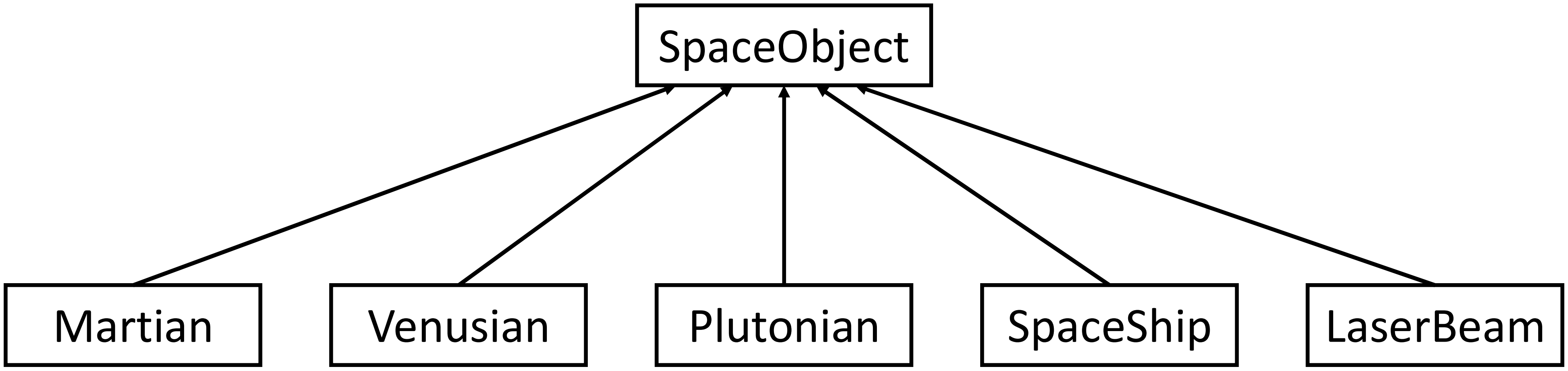
Due to ??



SpaceObject Hierarchy Example – Part 8

Inheritance, which is the “is a” relationship between classes

So a Martian **is a** SpaceObject, and a SpaceShip **is a** SpaceObject, etc.



SpaceObject Hierarchy Example – Part 9

If we create an array of SpaceObject pointers:

```
SpaceObject* spaceObjectPtrs[5];
```

What kinds of **variables** are we creating?

What kinds of **data** can be stored in those variables?

What kind of data is stored in these variables right now?

When will data get put into these variables?

SpaceObject Hierarchy Example – Part 10

SpaceObject* spaceObjectPtrs[5];

What kinds of **variables** are we creating?

- pointer variables of type SpaceObject* (or “pointer to a SpaceObject”)

What kind of **data** can be stored in those variables?

- Address of any kind of SpaceObject (which are?)

What kind of data is stored in these variables right now?

- No data yet

When will data get put into these variables?

- At run time

SpaceObject Hierarchy Example – Part 11

Remember, there are two steps to working with variables (even variables that are part of an array, <array> object, or <vector> object)

1. Create the variable

2. Store the appropriate kind of data in the variable

SpaceObject Hierarchy Example – Part 12

What would happen if we wrote these statements:

```
SpaceObject* spaceObjectPtr;
spaceObjectPtr = &someSpaceObject;
```

What about these?

```
int* x;
x = &someSpaceObject;
```

SpaceObject Hierarchy Example – Part 13

This is fine:

```
SpaceObject* spaceObjectPtr;
spaceObjectPtr = &someSpaceObject;
```

This is an error. When would the error be flagged?

```
int* x;
x = &someSpaceObject;
```

SpaceObject Hierarchy Example – Part 14

This is an invalid conversion error and is flagged at **compile time**.

```
int* x;

x = &someSpaceObject;
```

The compiler checks if the items on either side of the assignment operator are compatible types.

x’s data type is the address of an int, but we’re attempting to assign it the address of a SpaceObject

SpaceObject Hierarchy Example – Part 15

What about these statements:

```
SpaceObject* spacePtr;
Martian martianObject;
spacePtr = &martianObject;
```

What about these?

```
Martian* martianPtr;
SpaceObject ;
martianPtr = &spaceObject;
```

SpaceObject Hierarchy Example – Part 16

This is ok. Why?

```
SpaceObject* spacePtr;
Martian martianObject;
spacePtr = &martianObject;
```

This is an invalid conversion error. Why?

```
Martian* martianPtr;
SpaceObject spaceObject;
martianPtr = &spaceObject;
```

SpaceObject Hierarchy Example – Part 17

A Martian **IS A** SpaceObject, so the types match.

```
SpaceObject* spacePtr;
Martian martianObject;
spacePtr = &martianObject;
```

A SpaceObject is not a Martian, so invalid conversion error.

```
Martian* martianPtr;
SpaceObject spaceObject;
martianPtr = &spaceObject;
```

Types Are Also Matched Up at Run Time

If, at compile time, it’s not clear yet what type of object will be stored, the types will have to be checked at **run time**

Types Are Also Matched Up at Run Time – Part 1

Example pseudocode:

```
Create an array of pointers to SpaceObjects (array is empty at this point)

Fill up the array using a loop:

    Flip a coin

    If heads, create a Venusian object, & store its address in the array

    If tails, create a Martian object, & store its address in the array
```

Types Are Also Matched Up at Run Time – Part 2

Example result:

Address of a Venusian object	Address of a Venusian object	Address of a Martian object	Address of a Martian object	Address of a Venusian object

After creating the array, we want to read through it, and call the draw() function of each object, so that our game screen will refresh

So which version of the draw() function will be called?

Types Are Also Matched Up at Run Time – Part 3

Example result:

Address of a Venusian object	Address of a Venusian object	Address of a Martian object	Address of a Martian object	Address of a Venusian object

Remember, the variables in the array may be of a different type than the object the variable points to

What **type** is each variable in this array?

Types Are Also Matched Up at Run Time – Part 4

Example result:

Address of a Venusian object	Address of a Venusian object	Address of a Martian object	Address of a Martian object	Address of a Venusian object

Each variable in this array holds the address of a SpaceObject object

What type of object is stored in each variable?

Types Are Also Matched Up at Run Time – Part 5

Example result:

Address of a Venusian object	Address of a Venusian object	Address of a Martian object	Address of a Martian object	Address of a Venusian object

SpaceObject[0], SpaceObject[1], and SpaceObject[4] point to Venusian objects

SpaceObject[2] and SpaceObject[3] point to Martian objects

But they are all also SpaceObjects

TYPES ARE ALSO MATCHED UP AT RUN TIME – PART 6

Example result:

Address of a Venusian object	Address of a Venusian object	Address of a Martian object	Address of a Martian object	Address of a Venusian object

So which version of the draw() function will be called?

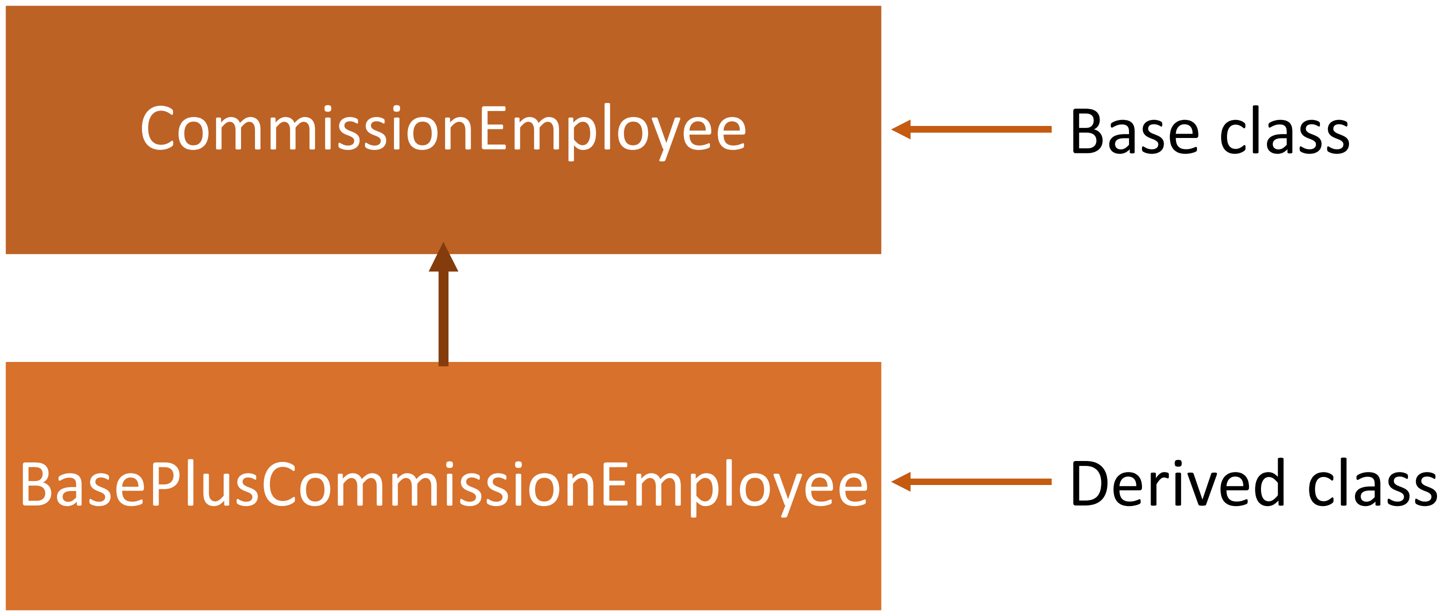
The one associated with the object **at run time**

Relationships Among Objects in an Inheritance Hierarchy

Remember the Employee Classes from Chapter 11

Code from Fig11_14_15

Inheritance hierarchy:



Two Different Pointer Variables Can be Created

With these two classes, there are two different kinds of pointer variables we can create:

Base class pointer: ComEmp* comEmpPtr;		
Derived class pointer: BPComEmp* BPComEmpPtr;		

For the sake of saving space, the names are abbreviated.

Two Different Objects Can be Created

And two different kinds of objects we can create:

	Base class object: ComEmp ComEmpObj;	**Derived class object:** BPComEmp BPComEmpObj;
Base class pointer: ComEmp* comEmpPtr;		
Derived class pointer: BPComEmp* BPComEmpPtr;		

For the sake of saving space, the names are abbreviated.

Can Both Types of Pointers Be Used?

Can both types of pointers be pointed at both types of objects?

	Base class object: ComEmp ComEmpObj;	**Derived class object:** BPComEmp BPComEmpObj;
Base class pointer: ComEmp* comEmpPtr;	Base pointer → base object?	Base pointer → derived object?
Derived class pointer: BPComEmp* BPComEmpPtr;	Derived pointer → base object?	Derived pointer → derived object?

For the sake of saving space, the names are abbreviated.

Yes and No

Can both types of pointers be pointed at both types of objects?

	Base class object: ComEmp ComEmpObj;	**Derived class object:** BPComEmp BPComEmpObj;
Base class pointer: ComEmp* comEmpPtr;	Base pointer → base object? **YES**	Base pointer → derived object? **YES, BUT…**
Derived class pointer: BPComEmp* BPComEmpPtr;	Derived pointer → base object? **NO**	Derived pointer → derived object? **YES**

For the sake of saving space, the names are abbreviated.

Derived-Class Pointers are More Specific

Remember, derived classes are more specific than base class objects

Derived-class pointers are also more specific

Derived-class pointers can only point at derived-class objects

- Addresses of derived-class objects can always be assigned to derived-class pointers

Derived-class pointers **cannot point** at base-class objects

- Compilation error; verifying that the data types match is done at compile time

But Base-Class Pointers are More General

Base-class pointers can point at base-class objects

- Addresses of base-class objects can always be assigned to base-class pointers
- Verifying that the data types match is done at compile time

Base-class pointers can also point at derived-class objects

- Addresses of derived-class objects can be assigned to base-class pointers
- Verifying that the data types match is done at RUN time

However, there are some limitations to be aware of…

INVOKING FUNCTIONALITY WITH BASE-CLASS POINTERS

There are some limitations in how a base-class pointers can access functionality in the derived-class object

Base-class pointers to derived-class objects can only **invoke base-class functionality** in derived-class object

Attempting to invoke a **derived-class function via a base-class pointer is an error**

Using Employee Examples to Show Polymorphism

Fig12-01

- CommissionEmployee.h / CommissionEmployee.cpp (from ch 11)
- BasePlusCommissionEmployee.h / BasePlusCommissionEmployee.cpp (from ch 11)
- Fig12_01.cpp

Demonstrates three ways to use base and derived pointers at base and derived class objects:

- Base pointer pointed at base object, and invoke base functionality
- Derived pointer pointed at derived object, and invoke derived functionality
- Base pointer pointed at derived object, and invoke base functionality from derived object

Base-class pointers pointed at derived-class objects is polymorphism

REMEMBER:

A derived-class object can be treated as an object of its base-class

- Because a derived-class object IS-A base-class object

But not the other way around…

Invoking Base Functions from Derived Objects – Part 1

earnings function

- Base class version (CommissionEmployee.cpp, lines 70-72)
- Derived class redefines it (BasePlusCommissionEmployee.cpp, lines 32-34), and uses some of its functionality

Base and derived objects created

- In driver program fig12_01.cpp, lines 12-13 and 16-17

Base and derived objects values displayed, using the object names to invoke toString function specific to each class (fig12_01.cpp)

- Line 23 invokes the base class version of toString, using the base class object name
- Line 25 invokes the derived class version of toString, using the derived class object name

Invoking Base Functions from Derived Objects – Part 2

Create base pointer, point it at base object (line 28)

Use base pointer to invoke toString of base object (line 31)

Create derived pointer, point it at derived object (line 34-35)

Use derived pointer to invoke toString of derived object (line 39)

Invoking Base Functions from Derived Objects – Part 3

Point base pointer at derived object (line 42)

Use base pointer to invoke toString (line 46)

This invokes the **base class version of toString**, because it's called via a **base pointer**

INVOKING BASE FUNCTIONS FROM DERIVED OBJECTS

Summary:

Invoked functionality (base or derived) depends on the **type of pointer** used to invoke the function, **not the type of object**

Later we'll see how we can invoke object's functionality instead

Pointing Derived Pointers at Base Objects

Fig12_02.cpp

Attempt to point a derived pointer at a base object (lines 12-13)

Compiler error

DERIVED FUNCTION CALLS VIA BASE POINTERS

Fig12_03.cpp

Base-class functions can only be called via base-class pointers

Attempting to invoke derived-class function from base-class pointer is a compiler error

Line 26 – attempt to invoked derived class function via base pointer

- Creates a local variable baseSalary and attempts to assign it a value via initialization brackets
- Inside the brackets is a call to getBaseSalary function, which only exists in the derived class
- The pointer is a base pointer, assigned to a derived object
- Compiler error
- Line 27 would fail for the same reason, but compiler doesn't even get that far

Derived Function Calls via Base Pointers – Need to Downcast

If we really need to invoke derived class functionality via a base class pointer, there is a way…

Downcasting

- Casting a value to a type that is either smaller than the original type (in the case of the primitive data types) or lower in an inheritance hierarchy

If we want to do this, we need to explicitly cast the base-class pointer to a derived-class pointer

- Can be dangerous, so is not something we should regularly do

VIRTUAL FUNCTIONS AND VIRTUAL DESTRUCTORS

VIRTUAL FUNCTIONS

Remember, a base pointer will invoke base functions, even if the pointer is aimed at a derived object

We can alter this behavior by using **virtual functions**

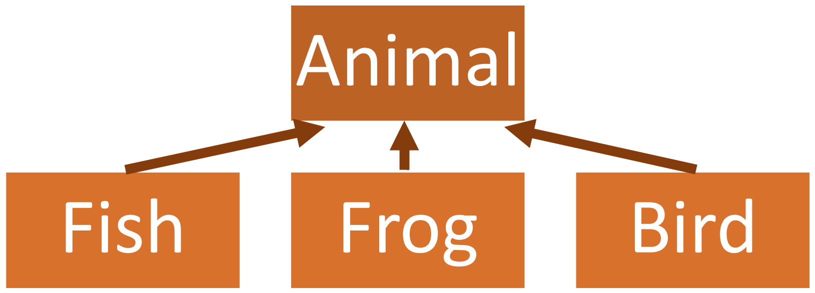
Virtual Functions with the Animal Inheritance Hierarchy

Remember our earlier example of the Animal inheritance hierarchy

If our case study of Animal movement used an array of pointers to Animal objects, then only the move function defined in the base class of the hierarchy (Animal) would ever be invoked

But a Fish moves differently from a Frog, which moves differently from a bird

So how can we invoke the right version of move?



Virtual Functions Need to be Used

We use a **virtual function:**

The type of the **object**, not the type of the pointer pointing to the object, determines which version of the function to invoke

Useful for implementing polymorphic behavior

- Treating a derived object (or usually, a group of derived objects) as its base object

Declaring a Virtual Function in the Base Class

In the **base class header file**, a virtual function is declared using the **virtual** keyword:

General format:

virtual *returnValue functionName(parameter list);*

So this is the prototype.

The base class implementation file may or may not provide an actual implementation. It depends on the program and what is needed.

Redefining Virtual Functions in Derived Class

A derived class **overrides** the virtual function

Overridden function has same signature and return type

- So, the same prototype

Plus, the word **override** is added

General format:

virtual *returnValue functionName(parameter list)* **override***;*

Declaring Virtual Functions

Virtual functions stay virtual all the way down the hierarchy

If derived-class does not redefine it, the derived-class inherits the base-class version

Invoking a Virtual Function

A virtual function can be invoked in 3 ways:

1. base pointer pointing at a derived object
2. base reference to derived object
   - We haven’t covered this in this chapter.
3. object name
   - base object name, if function exists in base class
   - derived object name, if function exists in derived class

INVOKING A VIRTUAL FUNCTION VIA STATIC BINDING

When a virtual function is invoked via the object’s name and the dot operator, the function call is resolved at **compile time**

The compiler checks if that function exists in that class

This is called **static binding**

Not polymorphic behavior

Invoking a Virtual Function via Late Binding

When a virtual function is invoked via a base pointer pointed at a derived object, the function call is resolved at **run time**

There is no way to know which object a pointer is pointing to until the program executes

This is called **dynamic or late binding binding**

This **IS** polymorphic behavior

Virtual Functions in the CommissionEmployee Hierarchy

Fig12_04_06

earnings and toString functions declared as virtual in the header files

- CommissionEmployee.h (lines 28 & 29)
- BasePlusCommissionEmployee.h (lines 17 & 18)
- Also notice “override” in BasePlusCommissionEmployee

No changes to implementations of earnings and toString in the implementation files

VIRTUAL FUNCTIONS IN THE CE HIERARCHY

Fig12_06.cpp

Base class pointer pointed at derived class object (line 47)

toString method invoked via the base pointer (line 54)

Because toString is declared **virtual**, line 54 invokes the **derived version**, not the base version