Polymorphism and Virtual Functions

Chapter 15

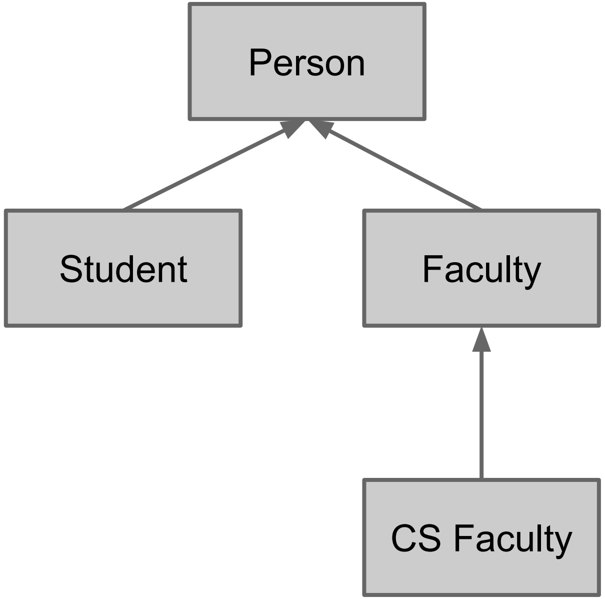
Type Compatibility In Inheritance

- Derived classes can serve as the base class for other classes
  - This results in inheritance hierarchy

Type Compatibility In Inheritance

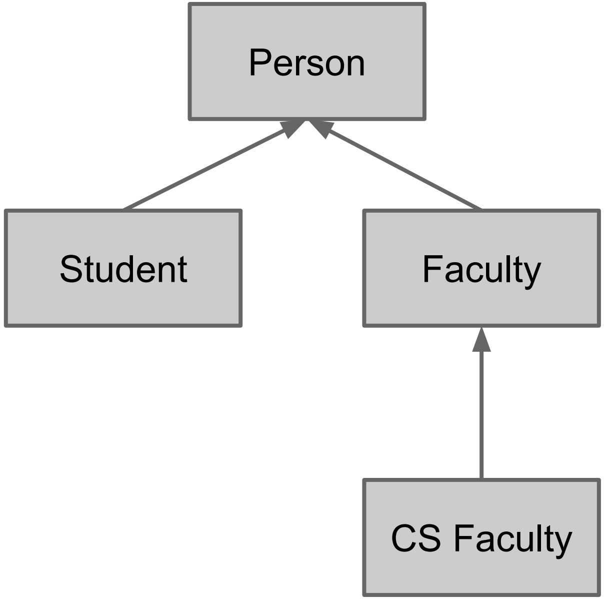
- An example of inheritance hierarchy would be
    - A class CSFaculty which inherits from class Faculty which inherits from class Person

Type Compatibility In Inheritance



Type Compatibility In Inheritance

Classes lower in the hierarchy are special cases of those above



Type Compatibility In Inheritance

- With the hierarchy comes some pointer fun
  - Pointers to derived classes can be assigned to a pointer of a base class

Type Compatibility In Inheritance

- With the hierarchy comes some pointer fun
    - Pointers to derived classes can be assigned to a pointer of a base class

```
Person* pFacultyMember = new CSFaculty();
```

Type Compatibility In Inheritance

- With the hierarchy comes some pointer fun
  - Pointers to derived classes can be assigned to a pointer of a base class

```
Person* pFacultyMember = new CSFaculty();
```

This opens up some cool features like
a list of Persons can contain a person,
faculty, or csfaculty object

Type Compatibility In Inheritance

- With the hierarchy comes some pointer fun
    - Pointers to derived classes can be assigned to a pointer of a base class
    - Moving back from a base class to a derived class pointer is possible but requires a cast

```
Person* pFacultyMember = new CSFaculty();
```

Type Compatibility In Inheritance

- With the hierarchy comes some pointer fun
  - Pointers to derived classes can be assigned to a pointer of a base class
  - Moving back from a base class to a derived class pointer is possible but requires a cast

```
Person* pFacultyMember = new CSFaculty();
CSFaculty* pOldFacMember = static_cast<CSFaculty*>(pFacultyMember);
```

Type Compatibility In Inheritance

- With the hierarchy comes some pointer fun
  - Pointers to derived classes can be assigned to a pointer of a base class
  - Moving back from a base class to a derived class pointer is possible but requires a cast

```
Person* pFacultyMember = new CSFaculty();
CSFaculty* pOldFacMember = static_cast<CSFaculty*>(pFacultyMember);
```

*This only works if the object truly is of the appropriate type. ie it really is a CSFaculty object

Using Type Casts with Base Class Pointers

- When a Base Class pointer is used to point to a derived class, C++ determines access to members based on the pointer type

Using Type Casts with Base Class Pointers

- When a Base Class pointer is used to point to a derived class, C++ determines access to members based on the pointer type
  - IE a Person* pointer would only have access to Person class members
    - This temporarily hides derived class members

Using Type Casts with Base Class Pointers

- When a Base Class pointer is used to point to a derived class, C++ determines access to members based on the pointer type
  - IE a Person* pointer would only have access to Person class members
    - This temporarily hides derived class members
  - This can be reversed by type casting
    - static_cast

Polymorphism and Virtual Member Functions

- Polymorphic code is code that behaves differently with different types of data

Polymorphism and Virtual Member Functions

- Polymorphic code is code that behaves differently with different types of data
  - For example, if we implement a getName() function to each class and call the function using the base class pointer, what happens?

Polymorphism and Virtual Member Functions

- Polymorphic code is code that behaves differently with different types of data
    - For example, if we implement a getName() function to each class and call the function using the base class pointer, what happens?
        - The base class function is called (not polymorphic)
        - What if we do want it to act polymorphic?

Polymorphism and Virtual Member Functions

- Virtual Member Function
  - A mechanism for achieving polymorphic functions in C++

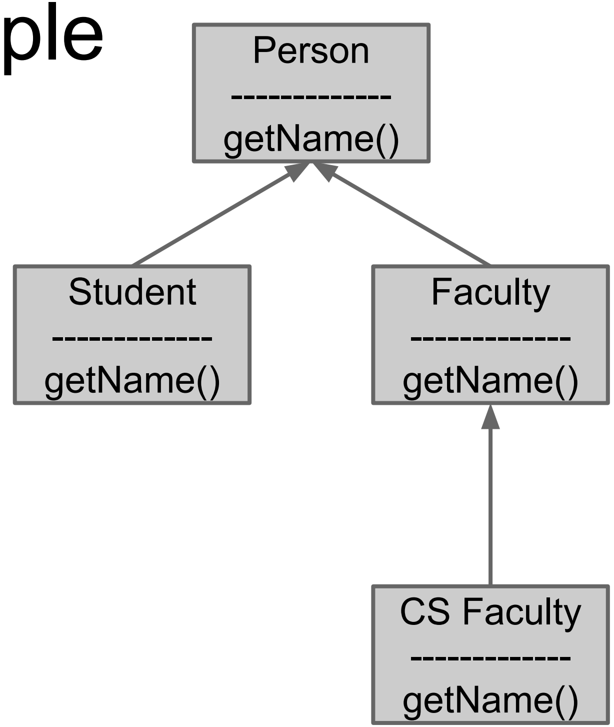
Polymorphism and Virtual Member Functions

- Virtual Member Function
  - A mechanism for achieving polymorphic functions in C++

- In our example
  - If getName was implemented polymorphically, then calling getName on Person, Faculty, and CSFaculty could respond differently

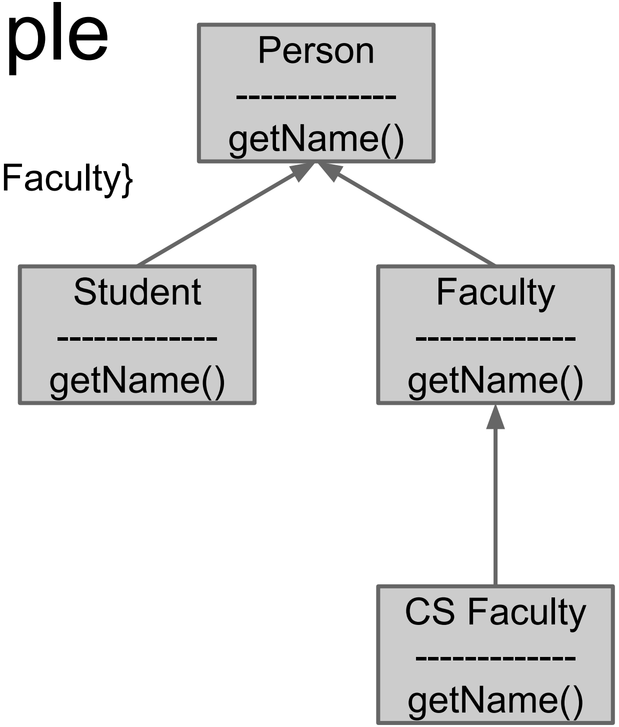
Polymorphism and Virtual Member Functions

Non Polymorphic Example



Polymorphism and Virtual Member Functions

Non Polymorphic Example



```
Person* pA[] = {new Person, new Faculty, new CSFaculty}
then
for (int i = 0; i < 3; i++)
{
        pA[i]->getName();
}
```

Would return

“Person”, “Person”, “Person”

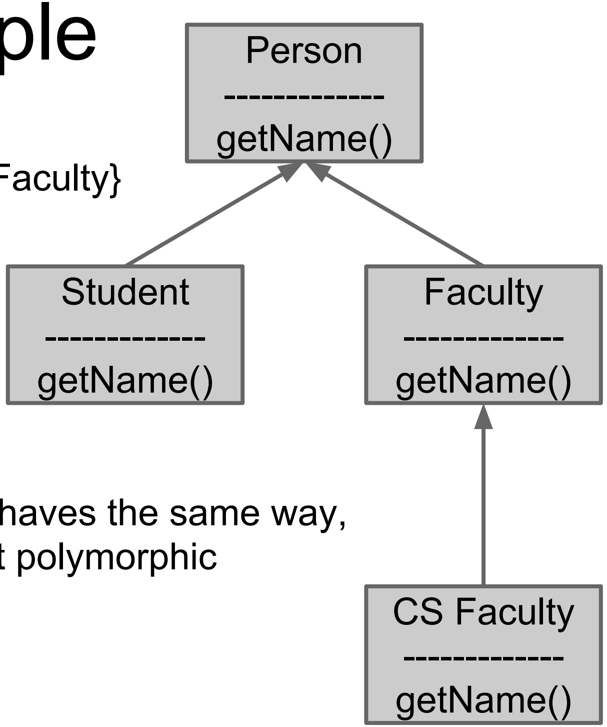
Polymorphism and Virtual Member Functions

Non Polymorphic Example

```
Person* pA[] = {new Person, new Faculty, new CSFaculty}
then
for (int i = 0; i < 3; i++)
{
        pA[i]->getName();
}
```

Would return

“Person”, “Person”, “Person”



Person
----
getName()

Student
----
getName()

Faculty
----
getName()

CS Faculty
----
getName()

*The code behaves the same way, hence it is not polymorphic

Polymorphism and Virtual Member Functions

- The example can be made polymorphic using Virtual Functions

Polymorphism and Virtual Member Functions

- The example can be made polymorphic using Virtual Functions
  - In C++ a virtual function is prefixed with ‘virtual’
  - This forces the compiler to check the type of each object to see if it defines a more specific version of the virtual function

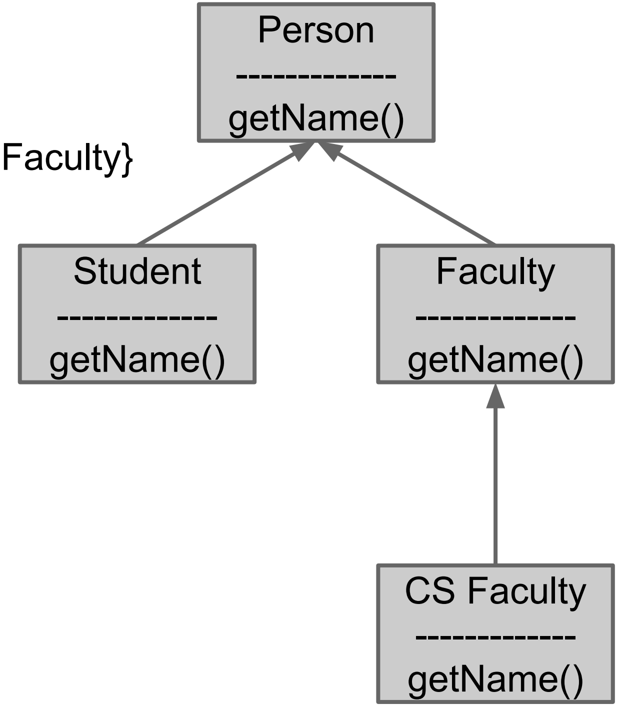
Polymorphism and Virtual Member Functions

Polymorphic Example

```
Person* pA[] = {new Person, new Faculty, new CSFaculty}
then
for (int i = 0; i < 3; i++)
{
        pA[i]->getName();
}
```

Would return

“Person”, “Faculty”, “CS Faculty”



Polymorphism and Virtual Member Functions

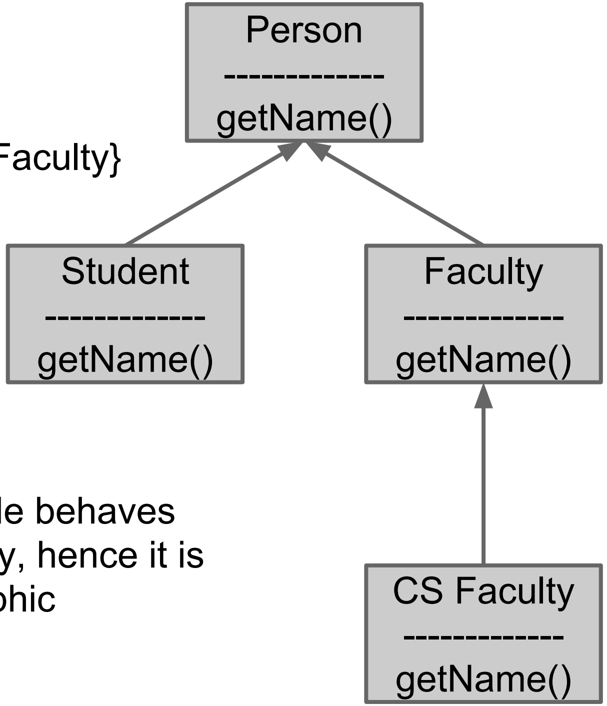
Polymorphic Example

```
Person* pA[] = {new Person, new Faculty, new CSFaculty}
then
for (int i = 0; i < 3; i++)
{
        pA[i]->getName();
}
```

Would return

“Person”, “Faculty”, “CS Faculty”

*The code behaves differently, hence it is polymorphic



Polymorphism and Virtual Member Functions

- Function Binding
  - The process by which the compiler determines which function definition to use for a specific function call is called

Polymorphism and Virtual Member Functions

- Function Binding
  - The process by which the compiler determines which function definition to use for a specific function call is called
    - The alternatives are static and dynamic binding

Polymorphism and Virtual Member Functions

- Static Binding
  - The compiler chooses the function in the base class pointer and ignores any versions in the object class
    - Done at compile time

Polymorphism and Virtual Member Functions

- Static Binding
    - The compiler chooses the function in the base class pointer and ignores any versions in the object class
        - Done at compile time
        - An example of this is our non polymorphic getName example

Polymorphism and Virtual Member Functions

- Dynamic Binding
  - The function to be invoked is determined at execution time
    - Looks at the actual class of the object and chooses the most specific version
    - Used to bind virtual functions

Abstract Base Classes and Pure Virtual Functions

- An abstract class is a class that can not be instantiated by itself

Abstract Base Classes and Pure Virtual Functions

- An abstract class is a class that can not be instantiated by itself
    - IE The class must be subclassed to be used
    - For example
        - There is no Animal that is not a dog, or cat, or …
        - The Animal class is an abstract class

Abstract Base Classes and Pure Virtual Functions

- Abstract classes are used as organizational tools

Abstract Base Classes and Pure Virtual Functions

- Abstract classes are used as organizational tools

- Abstract classes can be used to specify an interface that MUST be implemented by all derived classes

Abstract Base Classes and Pure Virtual Functions

- Abstract classes
  - Not all functions have to be implemented
    - It can be left up to the subclasses

Abstract Base Classes and Pure Virtual Functions

- Abstract classes
    - Not all functions have to be implemented
        - It can be left up to the subclasses

- In C++, an abstract class is a class with at least one abstract member function

Abstract Base Classes and Pure Virtual Functions

- An abstract function is defined by
  - marking it virtual
  - replacing the body with ‘ = 0;’

Abstract Base Classes and Pure Virtual Functions

- An abstract function is defined by
  - marking it virtual
  - replacing the body with ‘ = 0;’

- This is called a *pure virtual function* or an *abstract function*

Abstract Base Classes and Pure Virtual Functions

- An abstract class
  - Can not be instantiated
  - Can only be inherited from
  - All pure virtual functions must be implemented in the derived classes

Composition VS Inheritance

- As we talked about in Chapter 11
  - Inheritance has an ‘is-a’ relation between classes

Composition VS Inheritance

- As we talked about in Chapter 11
  - Inheritance has an ‘is-a’ relation between classes

- Example
  - Cow is an Animal
  - Poodle is a Dog
  - Faculty is a Person

Composition VS Inheritance

- Composition should be used when a new class needs to use an object of an existing class

Composition VS Inheritance

- Composition should be used when a new class needs to use an object of an existing class
- Inheritance should be used when
  - The new class is a subset of an existing class
  - The new class will be used in the same ways as the objects of an existing class