CIS 190: C/C++ Programming

Lecture 11
Polymorphism
Outline

• Review of Inheritance
• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors
• Application of Polymorphism
• Project Alphas
Review of Inheritance

• specialization through sub classes

• child class has direct access to
  – parent member functions and variables that are
    • ???

Review of Inheritance

• specialization through sub classes

• child class has direct access to
  – parent member functions and variables that are
    • public
    • protected
Review of Inheritance

• specialization through sub classes

• child class has direct access to
  – parent member functions and variables that are:
    • public
    • protected

• parent class has direct access to:
  – ??? in the child class
Review of Inheritance

- specialization through sub classes

- child class has direct access to
  - parent member functions and variables that are:
    - public
    - protected

- parent class has direct access to:
  - nothing in the child class
What is Inherited

Parent Class

- public members
- protected members
- private variables

- private functions
- copy constructor
- assignment operator
- constructor
- destructor
What is Inherited

Child Class
- child class members (functions & variables)
- public members
- protected members
- private variables

Parent Class
- private functions
- copy constructor
- assignment operator
- constructor
- destructor

Diagram:
- Venn diagram showing the overlap of child class members and parent class members.
Outline

• Review of Inheritance

• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors

• Application of Polymorphism

• Project Alphas
class SUV: public Car {/*etc*/};
class Sedan: public Car {/*etc*/};
class Van: public Car {/*etc*/};
class Jeep: public Car {/*etc*/};
Car Rental Example

• we want to implement a catalog of different types of cars available for rental

• how could we do this?
Car Rental Example

• we want to implement a catalog of different types of cars available for rental

• how could we do this?

• can accomplish this with a single vector
  – using *polymorphism*
What is Polymorphism?

• ability to manipulate objects in a **type-independent** way
What is Polymorphism?

• ability to manipulate objects in a type-independent way

• already done to an extent via overriding
  – child class overrides a parent class function
What is Polymorphism?

• ability to manipulate objects in a type-independent way

• already done to an extent via overriding
  – child class overrides a parent class function

• can take it further using subtyping, AKA inclusion polymorphism
Using Polymorphism

• a pointer of a parent class type can point to an object of a child class type
  \texttt{Vehicle *vehiclePtr = \&myCar;}

• why is this valid?
Using Polymorphism

• a pointer of a parent class type can point to an object of a child class type
  
  ```
  Vehicle *vehiclePtr = &myCar;
  ```

• why is this valid?
  – because `myCar` is-a `Vehicle`
Polymorphism: Car Rental

```
vector <Car*> rentalList;
```

vector of *Car* objects
Polymorphism: Car Rental

```cpp
vector <Car*> rentalList;
```

vector of `Car*` objects

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SUV</td>
<td>SUV</td>
<td>Jeep</td>
<td>Van</td>
<td>Jeep</td>
<td>Sedan</td>
<td>Sedan</td>
<td>SUV</td>
<td></td>
</tr>
</tbody>
</table>

- can populate the vector with any of `Car`’s child classes
Outline

• Review of Inheritance
• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors
• Application of Polymorphism
• Project Alphas
Limitations of Polymorphism

• parent classes do not inherit from child classes
  — what about public member variables and functions?
Limitations of Polymorphism

• parent classes **do not** inherit from child classes
  – **not even** public member variables and functions
Limitations of Polymorphism

• parent classes **do not** inherit from child classes
  – **not even** public member variables and functions

```c++
Vehicle *vehiclePtr = &myCar;
```
Limitations of Polymorphism

- parent classes **do not** inherit from child classes
  - not even public member variables and functions

```cpp
Vehicle *vehiclePtr = &myCar;
```

- which version of `PrintSpecs()` does this call?
  ```cpp
  vehiclePtr->PrintSpecs();
  ```
Limitations of Polymorphism

• parent classes **do not** inherit from child classes
  – **not even** public member variables and functions

  ```
  Vehicle *vehiclePtr = &myCar;
  ```

• which version of **PrintSpecs()** does this call?

  ```
  vehiclePtr->PrintSpecs();
  ```

  ```
  Vehicle::PrintSpecs()
  ```
Limitations of Polymorphism

• parent classes do not inherit from child classes
  – not even public member variables and functions

  `Vehicle *vehiclePtr = &myCar;`

• will this work?

  `vehiclePtr->RepaintCar();`
Limitations of Polymorphism

• parent classes **do not** inherit from child classes
  – **not even** public member variables and functions

```
Vehicle *vehiclePtr = &myCar;
```

• will this work?
  ```
  vehiclePtr->RepaintCar();
  ```

  – NO! **RepaintCar()** is a function of the Car child class, not the Vehicle class
• Review of Inheritance
• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors
• Application of Polymorphism
• Project Alphas
Virtual Functions

• can grant access to child methods by using *virtual functions*

• virtual functions are how C++ implements *late binding*
  – used when the child class implementation is unknown or variable at parent class creation time
Late Binding

• simply put, binding is determined at run time
  – as opposed to at compile time

• in the context of polymorphism, you’re saying
  I don’t know for sure how this function is going to
  be implemented, so wait until it’s used and then get
  the implementation from the object instance.
Using Virtual Functions

• declare the function in the parent class with the keyword `virtual` in front

```cpp
virtual void Drive();
```
Using Virtual Functions

• declare the function in the parent class with the keyword `virtual` in front
  
  ```
  virtual void Drive();
  ```

• only use `virtual` with the prototype
  
  ```
  // don't do this
  virtual void Vehicle::Drive();
  ```
Using Virtual Functions

• the corresponding child class function does not require the `virtual` keyword

• but...
Using Virtual Functions

• the corresponding child class function does not require the `virtual` keyword

• should still include it, for clarity’s sake
  – makes it obvious the function is virtual, even without looking at the parent class

```cpp
// inside the Car class
virtual void Drive();
```
Outline

• Review of Inheritance
• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors
• Application of Polymorphism
• Project Alphas
Function Types – Virtual

`virtual void Drive();`

- parent class **must** have an implementation  
  – even if it’s trivial or empty

- child classes may override if they choose to  
  – if not overridden, parent class definition used
Function Types – Pure Virtual

virtual void Drive() = 0;

• denote pure virtual by the “ = 0” at the end
Function Types – Pure Virtual

```cpp
virtual void Drive() = 0;
```

• denote pure virtual by the “` = 0`” at the end

• the parent class has **no implementation** of this function
  – child classes **must** have an implementation
Function Types – Pure Virtual

```c++
virtual void Drive() = 0;
```

• denote pure virtual by the " = 0" at the end

• the parent class has no implementation of this function
  – child classes **must** have an implementation
  – parent class is now an **abstract class**
Abstract Classes

• an *abstract class* is one that contains a function that is *pure virtual*
Abstract Classes

• an *abstract class* is one that contains a function that is *pure virtual*

• cannot declare abstract class objects
  – why?
Abstract Classes

- an **abstract class** is one that contains a function that is **pure virtual**

- cannot declare abstract class objects
  – why?

- this means abstract classes can only be used as base classes
Applying Virtual to Shape, etc.

• how should we label the following functions? (virtual, pure virtual, or leave alone)

```cpp
CalculateArea();
CalculatePerimeter();
Print();
SetColor();
```
Outline

• Review of Inheritance

• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors

• Application of Polymorphism

• Project Alphas
Behind the Scenes

- if our `Drive()` function is virtual, how does the compiler know which child class’s version of the function to call?

vector of Car* objects

<table>
<thead>
<tr>
<th>SUV</th>
<th>SUV</th>
<th>Jeep</th>
<th>Van</th>
<th>Jeep</th>
<th>Sedan</th>
<th>Sedan</th>
<th>SUV</th>
</tr>
</thead>
</table>
Virtual Function Tables

• the compiler uses *virtual function tables* whenever we use polymorphism

• virtual function tables are created for:
  – what types of classes?
Virtual Function Tables

• the compiler uses *virtual function tables* whenever we use polymorphism

• virtual function tables are created for:
  – classes with virtual functions
  – child classes of those classes
### Virtual Table Pointer

<table>
<thead>
<tr>
<th></th>
<th>SUV</th>
<th>SUV</th>
<th>Jeep</th>
<th>Van</th>
<th>Jeep</th>
<th>Sedan</th>
<th>Sedan</th>
<th>Van</th>
</tr>
</thead>
</table>
Virtual Table Pointer

• the compiler adds a hidden variable

<table>
<thead>
<tr>
<th>SUV</th>
<th>SUV</th>
<th>Jeep</th>
<th>Van</th>
<th>Jeep</th>
<th>Sedan</th>
<th>Sedan</th>
<th>Van</th>
</tr>
</thead>
<tbody>
<tr>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
</tr>
</tbody>
</table>
Virtual Table Pointer

- the compiler also adds a virtual table of functions for each class

<table>
<thead>
<tr>
<th>SUV</th>
<th>SUV</th>
<th>Jeep</th>
<th>Van</th>
<th>Jeep</th>
<th>Sedan</th>
<th>Sedan</th>
<th>Van</th>
</tr>
</thead>
<tbody>
<tr>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
<td>*__vptr</td>
</tr>
</tbody>
</table>

SUV virtual table  |  Jeep virtual table  | Van virtual table  | Sedan virtual table
Virtual Table Pointer

- each virtual table has pointers to each of the virtual functions of that class

<table>
<thead>
<tr>
<th>SUV</th>
<th>SUV</th>
<th>Jeep</th>
<th>Van</th>
<th>Jeep</th>
<th>Sedan</th>
<th>Sedan</th>
<th>Van</th>
</tr>
</thead>
<tbody>
<tr>
<td>*___vptr</td>
<td>*___vptr</td>
<td>*___vptr</td>
<td>*___vptr</td>
<td>*___vptr</td>
<td>*___vptr</td>
<td>*___vptr</td>
<td>*___vptr</td>
</tr>
</tbody>
</table>

SUV virtual table
* to SUV::Drive();

Jeep virtual table
* to Jeep::Drive();

Van virtual table
* to Van::Drive();

Sedan virtual table
* to Sedan::Drive();
Virtual Table Pointer

- the hidden variable points to the appropriate virtual table of functions

<table>
<thead>
<tr>
<th>SUV</th>
<th>SUV</th>
<th>Jeep</th>
<th>Van</th>
<th>Jeep</th>
<th>Sedan</th>
<th>Sedan</th>
<th>Van</th>
</tr>
</thead>
<tbody>
<tr>
<td>__vptr</td>
<td>__vptr</td>
<td>__vptr</td>
<td>__vptr</td>
<td>__vptr</td>
<td>__vptr</td>
<td>__vptr</td>
<td>__vptr</td>
</tr>
</tbody>
</table>

- SUV virtual table: 
  * to SUV::Drive();
- Jeep virtual table: 
  * to Jeep::Drive();
- Van virtual table: 
  * to Van::Drive();
- Sedan virtual table: 
  * to Sedan::Drive();
Virtual Everything!

• in Java, all functions are virtual by default
  – everything seems to work fine for Java

• why don’t we make all our functions virtual in C++ classes?
  – ???
Virtual Everything!

• in Java, all functions are virtual by default
  – everything seems to work fine for Java

• why don’t we make all our functions virtual in C++ classes?
  – non-virtual functions can’t be overridden (in the context of parent class pointers)
  – creates unnecessary overhead
Outline

• Review of Inheritance
• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors
• Application of Polymorphism
• Project Alphas
Virtual Destructors

```cpp
Vehicle *vehicPtr = new Car;
delete vehicPtr;
```

- for any class with virtual functions, you must declare a virtual destructor as well
- why?
Virtual Destructors

Vehicle *vehicPtr = new Car;
delete vehicPtr;

• for any class with virtual functions, you must declare a virtual destructor as well

• non-virtual destructors will only invoke the base class’s destructor
Virtual Constructors

• not a thing... why?
Virtual Constructors

• not a thing... why?

• we use polymorphism and virtual functions to manipulate objects \textbf{without} knowing type or having complete information about the object

• when we construct an object, \textbf{we have} complete information
  – there’s no reason to have a virtual constructor
Outline

• Review of Inheritance
• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors
• Application of Polymorphism
• Project Alphas
Application of Polymorphism

• examine polymorphism and virtual functions

• using these classes:
  – Animal
    • Bird
    • Cat
    • Dog
Outline

• Review of Inheritance
• Polymorphism
  – Limitations
  – Virtual Functions
  – Abstract Classes & Function Types
  – Virtual Function Tables
  – Virtual Destructors/Constructors
• Application of Polymorphism
• Project Alphas
Project Alphas

• due next Sunday (November 23rd)

• doesn’t:
  – have to be working
  – a complete project

• in a folder named <your_team_name>
Next Time

- take an (anonymous) in-class survey for 1% overall extra credit
- receive feedback on your project proposal
- have some say in what we cover during our last *(gasp! sob!)* class together