# Inheritance

Michael R. Nowak
Texas A&M University

# Truck

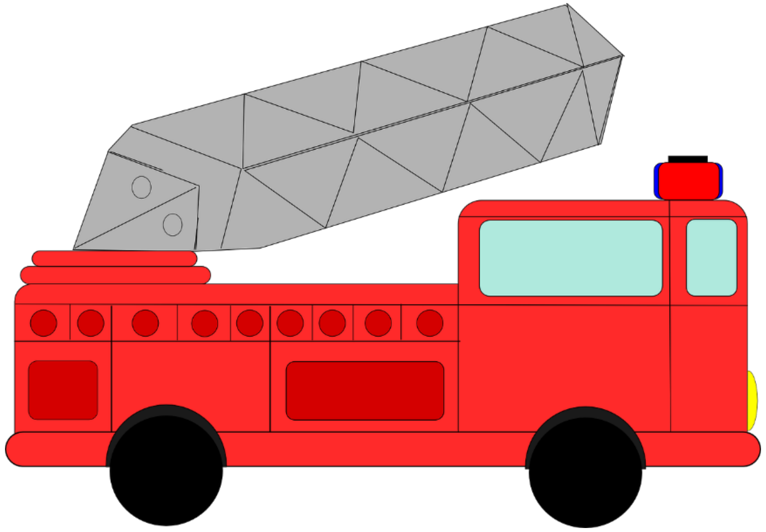
- Classes can model things that can be concrete or abstract.
- Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()
  - Etc.

# Fire Truck

- Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()

  - WaterCapacity
  - startSiren()
  - stopSiren()

Add to truck class?



# Concrete Truck

So we include members for all types of trucks?

So every truck could “startSiren()”???

- Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()

~~Add to truck class?~~

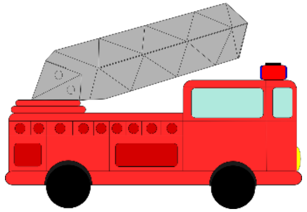
  - WaterCapacity
  - startSiren()
  - stopSiren()

- cubicFeetConcrete
- Pour()

# Separate Classes?

And more…

- Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()



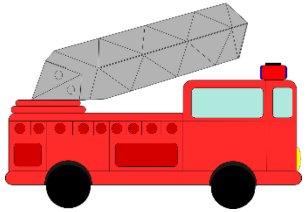
- Fire Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()
  - WaterCapacity
  - startSiren()
  - stopSiren()

- Concrete Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()
  - cubicFeetConcrete
  - Pour()

# Separate Classes?

How many updates???

- Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Width
  - Drive()
  - Stop()



- Fire Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()
  - WaterCapacity
  - startSiren()
  - stopSiren()

- Concrete Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Drive()
  - Stop()
  - cubicFeetConcrete
  - Pour()

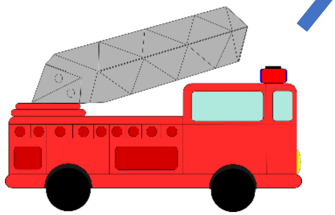
Share what's common?

# Separate Classes!

- Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Width
  - Drive()
  - Stop()

Share what's common!

- Fire Truck
  - WaterCapacity
  - startSiren()
  - stopSiren()



- Concrete Truck
  - cubicFeetConcrete
  - Pour()

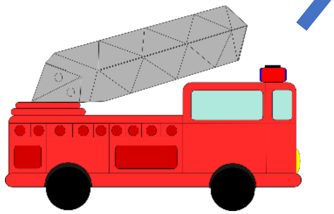
# Inheritance

Add to an existing class!

- Truck
  - Weight
  - Fuel type
  - Length
  - Height
  - Width
  - Drive()
  - Stop()

REUSE!!!

- Fire Truck
  - WaterCapacity
  - startSiren()
  - stopSiren()

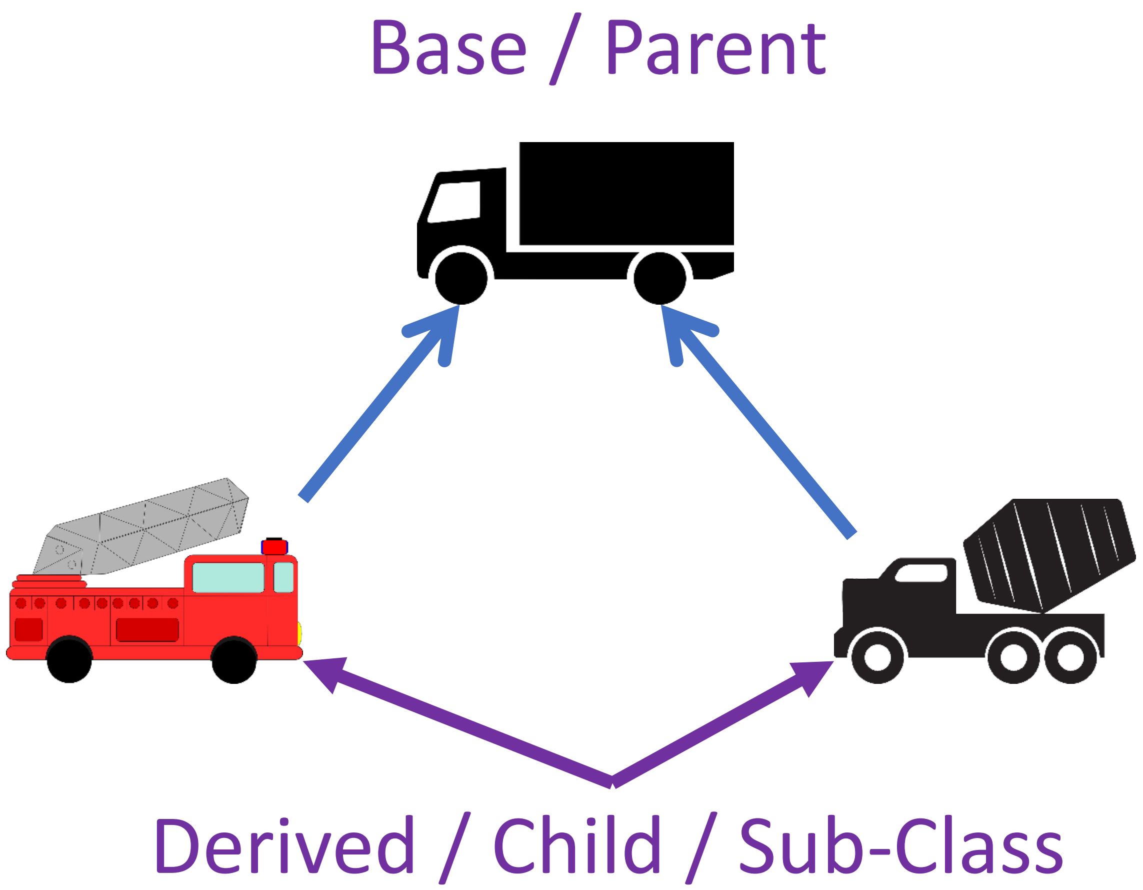


- Concrete Truck
  - cubicFeetConcrete
  - Pour()

Truck **AND** Fire Truck

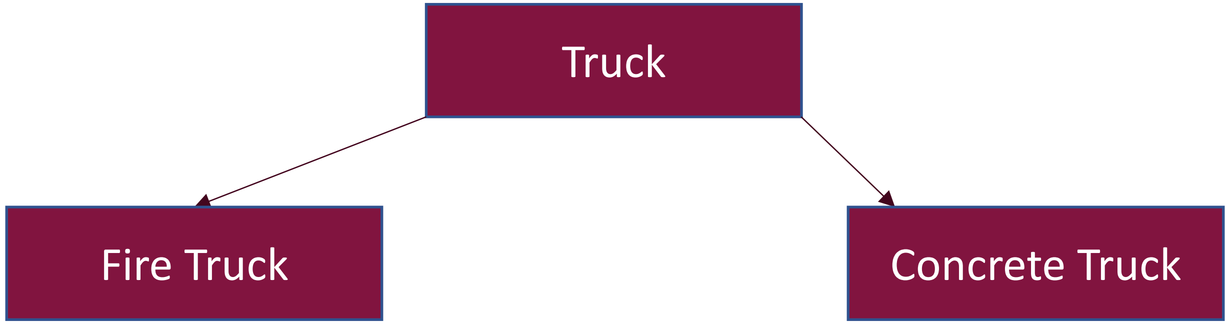
Truck **AND** Concrete Truck

# Some Terminology



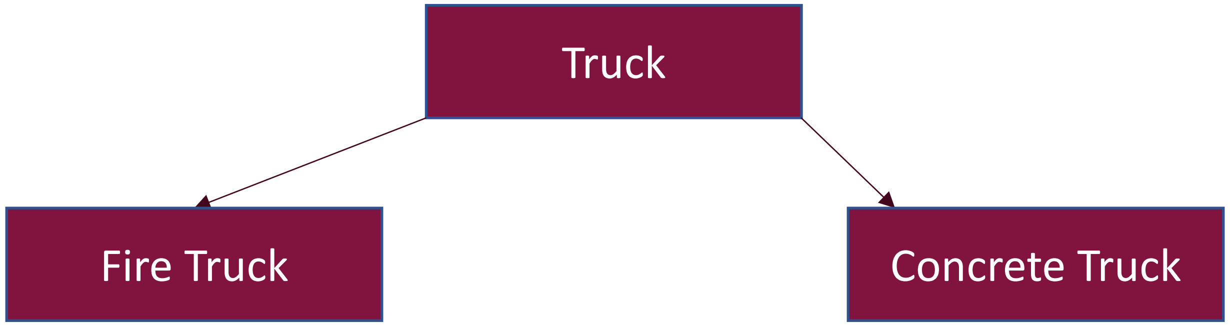
# Inheritance

- Many *things* share common features with other *things*, the extent to which is dependent on the level of abstraction from which we reason about them
- We can use the process of abstraction to encapsulate the commonality of those *things* into a base class
- Lower-level abstractions of the *things* comprising this base can be derived specialization and complexification



# Inheritance

- As we implement these representations using inheritance in C++, two fundamental but related functions of inheritance become apparent
  - We can say that a Fire Truck is derived from Truck
    - Our abstraction of a Fire Truck can automatically reuse our interface and/or implementation of Truck [**Interface Inheritance**]
  - Likewise, that a Fire Truck is a kind of Truck
    - Our abstraction of Fire Truck allows us to take advantage of the inherited facilities (i.e., attributes and behaviors) of Truck [**Implementation Inheritance**]



# Visibility of data members wrt inheritance

- Consider the following base class :

```
class Truck {
public:
    // If something knows where Truck lives, that thing can access these members…
    int x;
protected:
    // Only Truck children (and their children) can access the protected members…
    int y;
private:
    // Only this Truck can directly access the private members…
    int z;
};
```

# Visibility of inheritance

- **Public inheritance**
  - This is the traditional style of inheritance modeling an "is-a" relationship
  - FireTruck inherits the attributes and behaviors of Truck
    - A FireTruck is thus a Truck, with added specialization to make it a FireTruck
    - Therefore, when a FireTruck is upcast to an Truck, it can act like an Truck

```
class FireTruck: public Truck {
// FireTruck inherits from Truck with public visibility; if Truck and FireTruck
are known, then it is also known that FireTruck inherits from Truck.
     // x stays public
    // y stays protected
    // z stays private (in Truck) and is thus not accessible from FireTruck
};
```

# Visibility of inheritance

```
class Truck {
public:
    int x;
protected:
    int y;
private:
    int z;
};

class FireTruck: public Truck {
// FireTruck inherits from Truck with public visibility; if Truck and FireTruck
are known, then it is also known that FireTruck inherits from Truck.
     // x stays public
    // y stays protected
    // z stays private (in Truck) and is thus not accessible from FireTruck
};
```

# Inheritance Class Example

* We’re going to build both the parent and child class up over the next series of slides

```
class Parent {
public:
    Parent()
    ~Parent()
    std::string get_str() const { return str; }
private:
    std::string str;
};
```

If you do not provide a default constructor, then the compiler create one for you. As we note in a moment, if you do not specify otherwise in the derived class, the default constructor of the base will be called implicitly.

```
Parent::Parent()
{
    std::cout <<  "[" << this << "] Parent::Parent()" << std::endl;
}
Parent::~Parent()
{
    std::cout << "[" << this << "] Parent::~Parent()" << std::endl;
}
```

# Inheritance Class Example

* We're going to build both the parent and child class up over the next series of slides

```
class Child : public Parent {
public:

private:
};
```

If you do not provide a default constructor, then the compiler create one for you. As we note in a moment, if you do not specify otherwise in the derived class, the default constructor of the base will be called implicitly.

In main, let's do the following for now:

```
int main ( int argc, char **argv )
{
    Child p{};
    return 0;
}
```

We observe the following output once we executed the compiled program

```
~/Desktop
% ./a.out
[0x7fff54cb03a0] Parent::Parent()
[0x7fff54cb03a0] Parent::~Parent()
```

Parent::Parent() was called automatically

# Inheritance Class Example

* We're going to build both the parent and child class up over the next series of slides

We declared and then defined our own default constructor for the child class

```
Child::Child()
{
    std::cout << "[" << this << "] Child::Child()" << std::endl;
}
Child::~Child()
{
    std::cout << "[" << this << "] Child::~Child()" << std::endl;
}
```

And observed the following output once we executed the compiled program

Parent::Parent() was still called automatically and was first!

```
~/Desktop
% ./a.out
[0x7fff59f833a0] Parent::Parent()
[0x7fff59f833a0] Child::Child()
[0x7fff59f833a0] Child::~Child()
[0x7fff59f833a0] Parent::~Parent()
```

# Inheritance Class Example

* We're going to build both the parent and child class up over the next series of slides

We would like to initialize std::string str with a value passed to the base's constructor (want the base constructor to set-up the base; the child constructor to set-up the child)

```
class Parent {
public:
    Parent()
    ~Parent()
    std::string get_str() const { return str; }
private:
    std::string str;
};
```

We declare and define an additional constructor Parent::Parent(std::string str) that will initialize this->str with str through the initialization list

```
Parent::Parent(std::string str) : str(str) {
    std::cout << "[" << this << "] Parent::Parent
        (std::string)" << std::endl;
}
```

The question now is how to call this constructor during the initialization of the derived class. We can do this by creating a new constructor that takes a std::string as an argument, and calls the base constructor in the initializer list as follows:

```
Child::Child(std::string str)
     : Parent(str)
{
    std::cout << "[" << this << "]
       Child::Child(std::string)" <<
       std::endl;
}
```

# Inheritance Class Example

* We’re going to build both the parent and child class up over the next series of slides

We update main to include the initialization of an the Child p object, with the std::string argument “Hello, World!”

```
int main ( int argc, char **argv )
{
    Child p{"Hello, World!"};
    std::cout << p.get_str() << std::endl;
    return 0;
}
```

After compiling and running the program, we observed the following output:

```
~/Desktop
% ./a.out
[0x7fff57d34350] Parent::Parent(std::string)
[0x7fff57d34350] Child::Child(std::string)
Hello, World!
[0x7fff57d34350] Child::~Child()
[0x7fff57d34350] Parent::~Parent()
```

The value “Hello, World!” is getting stored in the std::string str object that we created in the Parent class!

# Inheritance Class Example

* We’re going to build both the parent and child class up over the next series of slides

Even though we are not working with dynamic memory in this class, we are going to declare and define the Parent& Parent::operator=(const Parent& other).

```
Parent& operator=(const Parent& other)
{
    std::cout << "[" << this << "]
       Parent::operator=(const Parent&)"
<< std::endl;
    str = other.str;
    return *this;
}
```

We update main to include the creation of another Child object p2, which we then assigned to p.

```
int main ( int argc, char **argv )
{
    Child p{"Hello, World!"};
    Child p2{“Howdy!"};
    p = p2;
    std::cout << p.get_str() << '\t'
      << p2.get_str() << std::endl;
    return 0;
}
```

# Inheritance Class Example

* We're going to build both the parent and child class up over the next series of slides

Given that we overloaded operator= in the Parent class, and that Child inherits from Parent with Public visibility, Child inherited the parent's operator= (which was the best match given the argument to operator=)

```
~/Desktop
% ./a.out
[0x7fff50d8d300] Parent::Parent(std::string)
[0x7fff50d8d300] Child::Child(std::string)
[0x7fff50d8d2e0] Parent::Parent(std::string)
[0x7fff50d8d2e0] Child::Child(std::string)
[0x7fff50d8d300] Parent::operator=(const Parent&)
Howdy!  Howdy!
[0x7fff50d8d2e0] Child::~Child()
[0x7fff50d8d2e0] Parent::~Parent()
[0x7fff50d8d300] Child::~Child()
[0x7fff50d8d300] Parent::~Parent()
```

Child inherited Parent::operator=()

We really would like to write an overloaded operator= for our Child class, given that the Child should manage Child components (none at this point) and Parent the Parent components…

The definition and declaration that we came-up with follows:

```
Child& operator=(const Child& other)
{
    std::cout << "Child::operator=(const
        Child&)" << std::endl;
    return *this;
}
```

# Inheritance Class Example

* We're going to build both the parent and child class up over the next series of slides

After overloading Child::operator=, we see that the assignment (which is done in Parent::operator=) never occurred.

```
~/Desktop
% ./a.out
[0x7fff5f897300] Parent::Parent(std::string)
[0x7fff5f897300] Child::Child(std::string)
[0x7fff5f8972e0] Parent::Parent(std::string)
[0x7fff5f8972e0] Child::Child(std::string)
Child::operator=(const Child&)
Hello, World!   Howdy!
[0x7fff5f8972e0] Child::~Child()
[0x7fff5f8972e0] Parent::~Parent()
[0x7fff5f897300] Child::~Child()
[0x7fff5f897300] Parent::~Parent()
```

The reason for this is that by overloading Child::operator=(), we have a more suitable overloaded operator= provided the argument

To resolve this issue, allowing the Parent to initialize the components of the base class and Child to do the same for the derived class, we update our constructor to explicitly call Parent::operator=

```
Child& operator=(const Child& other)
{
    std::cout << "Child::operator=(const
        Child&)" << std::endl;
    Parent::operator=(other);
    return *this;
}
```

# Inheritance Class Example

After adding the explicit call to Parent::operator= in Child::operator=, the desired behavior was observed

```
~/Desktop
% ./a.out
[0x7fff59388300] Parent::Parent(std::string)
[0x7fff59388300] Child::Child(std::string)
[0x7fff593882e0] Parent::Parent(std::string)
[0x7fff593882e0] Child::Child(std::string)
Child::operator=(const Child&)
[0x7fff59388300] Parent::operator=(const Parent&)
Howdy!  Howdy!
[0x7fff593882e0] Child::~Child()
[0x7fff593882e0] Parent::~Parent()
[0x7fff59388300] Child::~Child()
[0x7fff59388300] Parent::~Parent()
```