Chapter Ten: Inheritance, Part I

Slides by Evan Gallagher & Nikolay Kirov

Chapter Goals

- To understand the concepts of inheritance and polymorphism
- To learn how to inherit and override member functions

Inheritance Hierarchies

Maybe this will convince you...

I did some research, yes, *on the web*

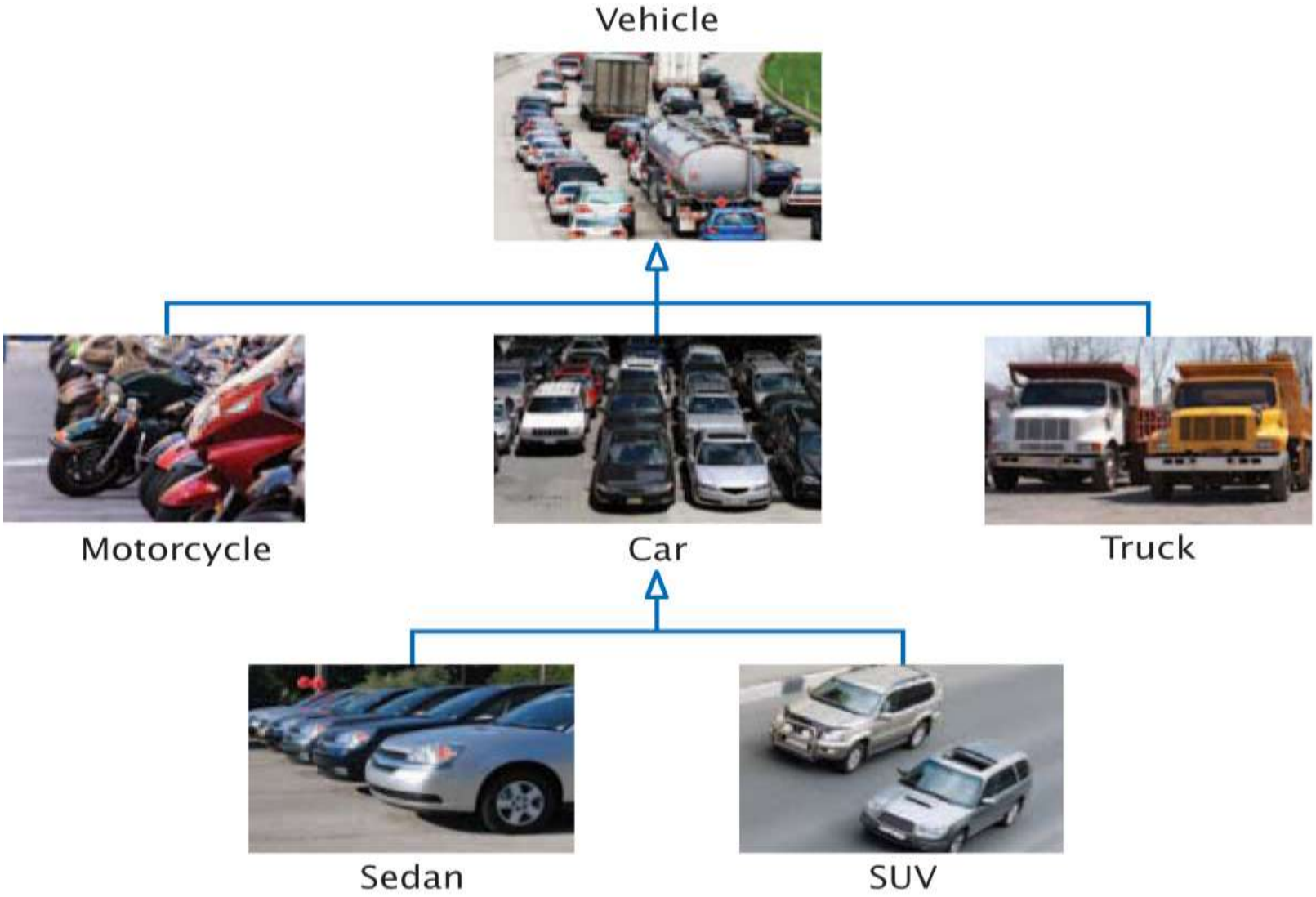
(I *told* you I have an onboard computer.)

Inheritance Hierarchies

Not only am I beautiful, shiny and new …

– I am part of a grand Hierarchy!

Inheritance Hierarchies



Inheritance Hierarchies

I have an *ancestry*!

I AM ROYALTY.

Inheritance Hierarchies

Shovels, rakes, and clippers all perform gardening tasks.

They can be considered as *specialized* versions of the *general* 'gardening tool' type.

Inheritance Hierarchies

In object-oriented design,
inheritance is a relationship between
a more general class (called the **base class**)
and a more specialized class (called the **derived class**).

The derived class *inherits* data and
behavior from the base class.

Inheritance Hierarchies

Just as I inherited my ROYALNESS.

Inheritance

IS-A

Inheritance: The IS-A Relationship

All cars are vehicles.

(This is correct and good English.)

Inheritance: The IS-A Relationship

All cars IS-A vehicles.

(Correct and … um … English?)

Inheritance: The IS-A Relationship

You may recall the UML notation HAS-A for *containment*.

IS-A

denotes *inheritance*.

Inheritance: The IS-A Relationship

All cars IS-A vehicles.

(Correct…

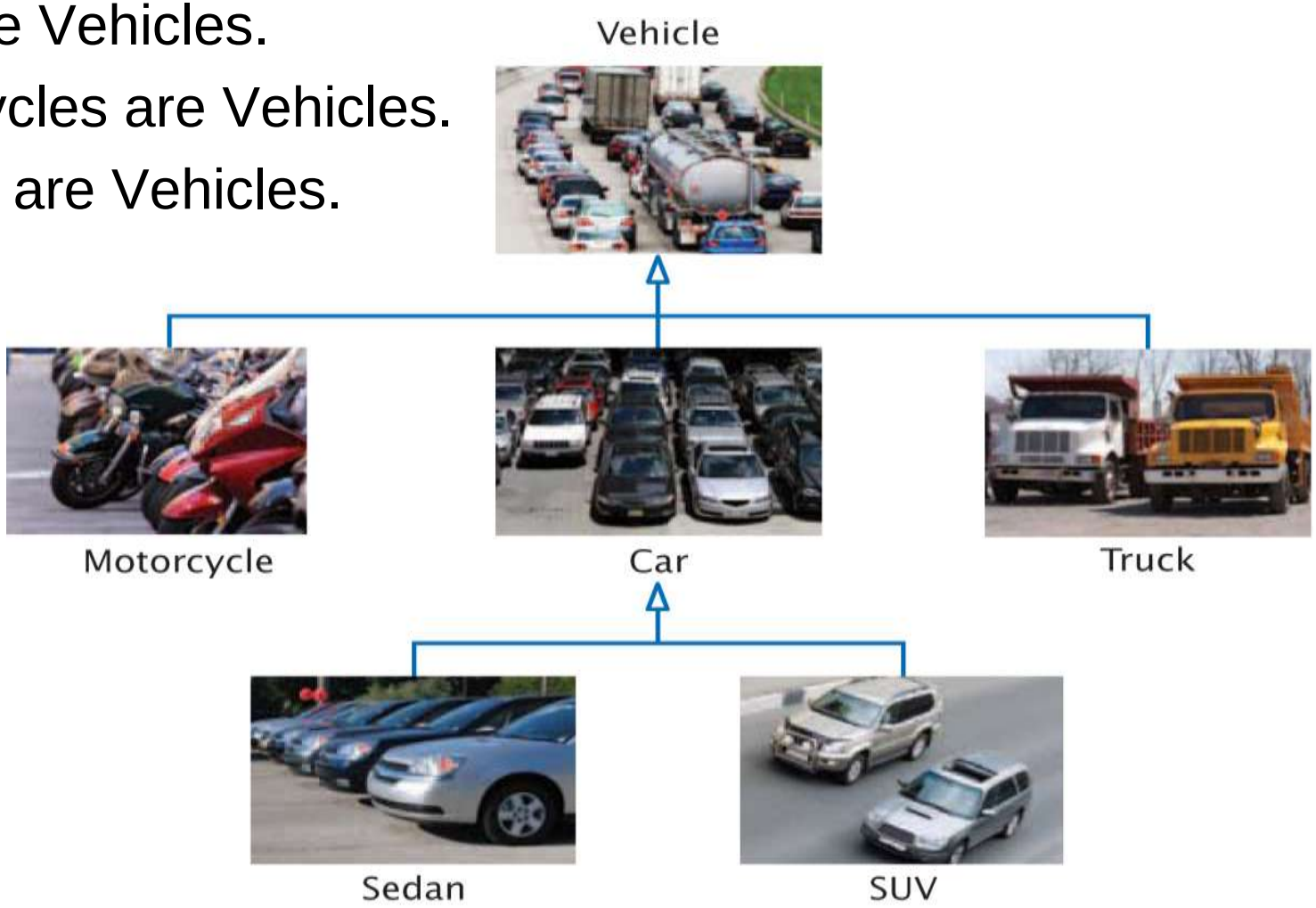
…when speaking *object*ively)

Inheritance: The IS-A Relationship

All Cars are Vehicles.

All Motorcycles are Vehicles.

All Sedans are Vehicles.

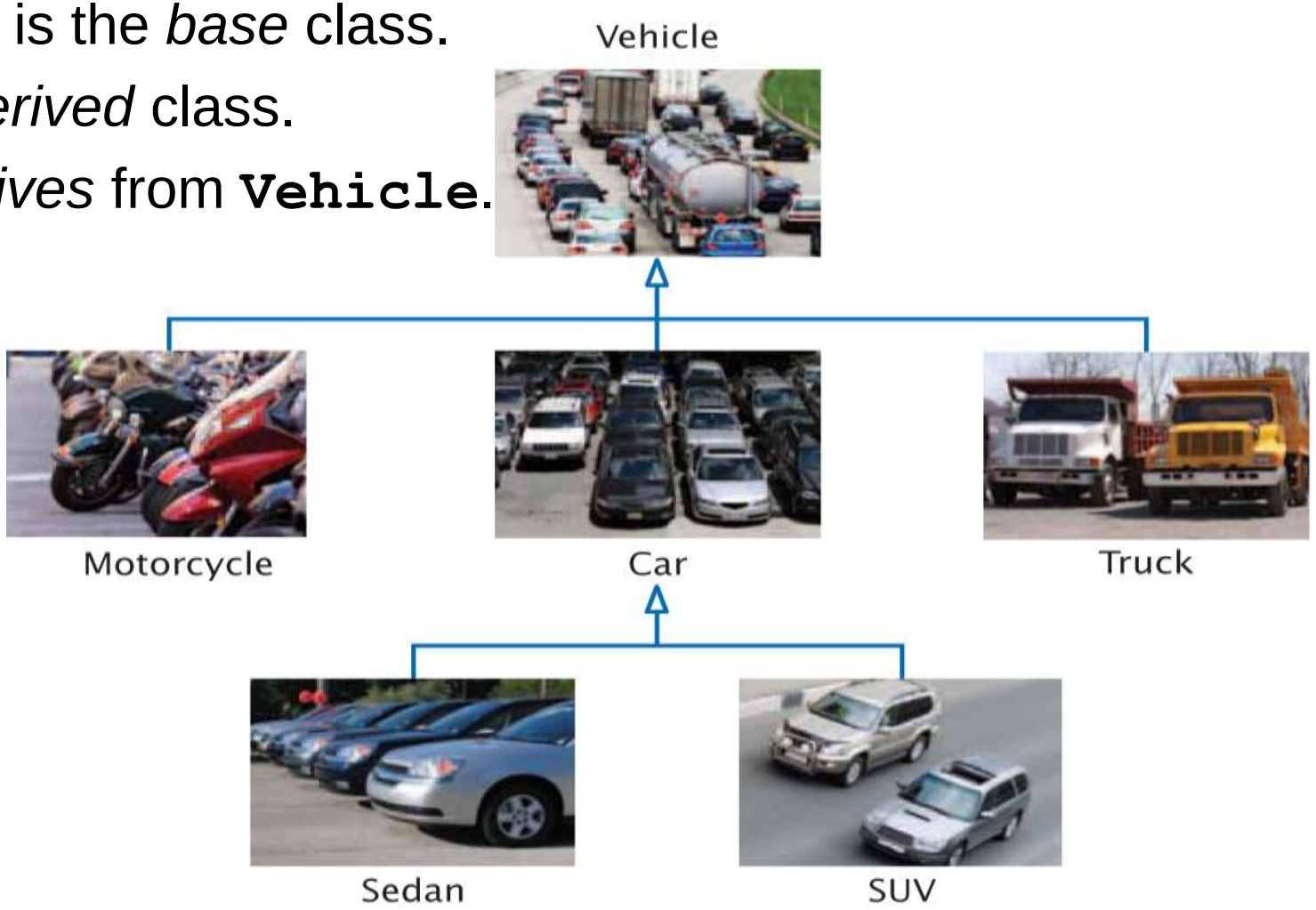


Inheritance: The IS-A Relationship

`Vehicles` is the *base* class.

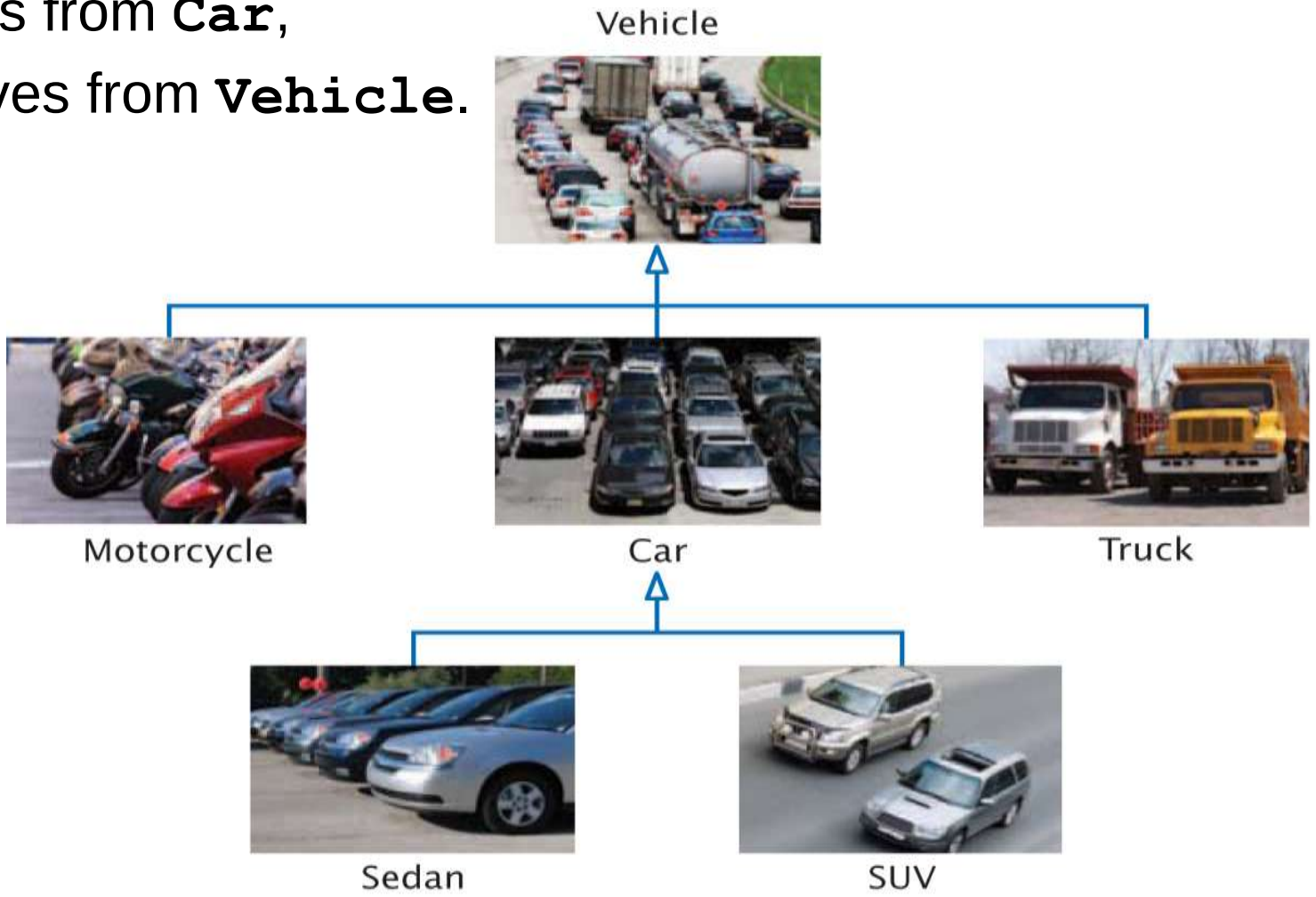
`Car` is a *derived* class.

`Truck` *derives* from **`Vehicle`**.



Inheritance: The IS-A Relationship

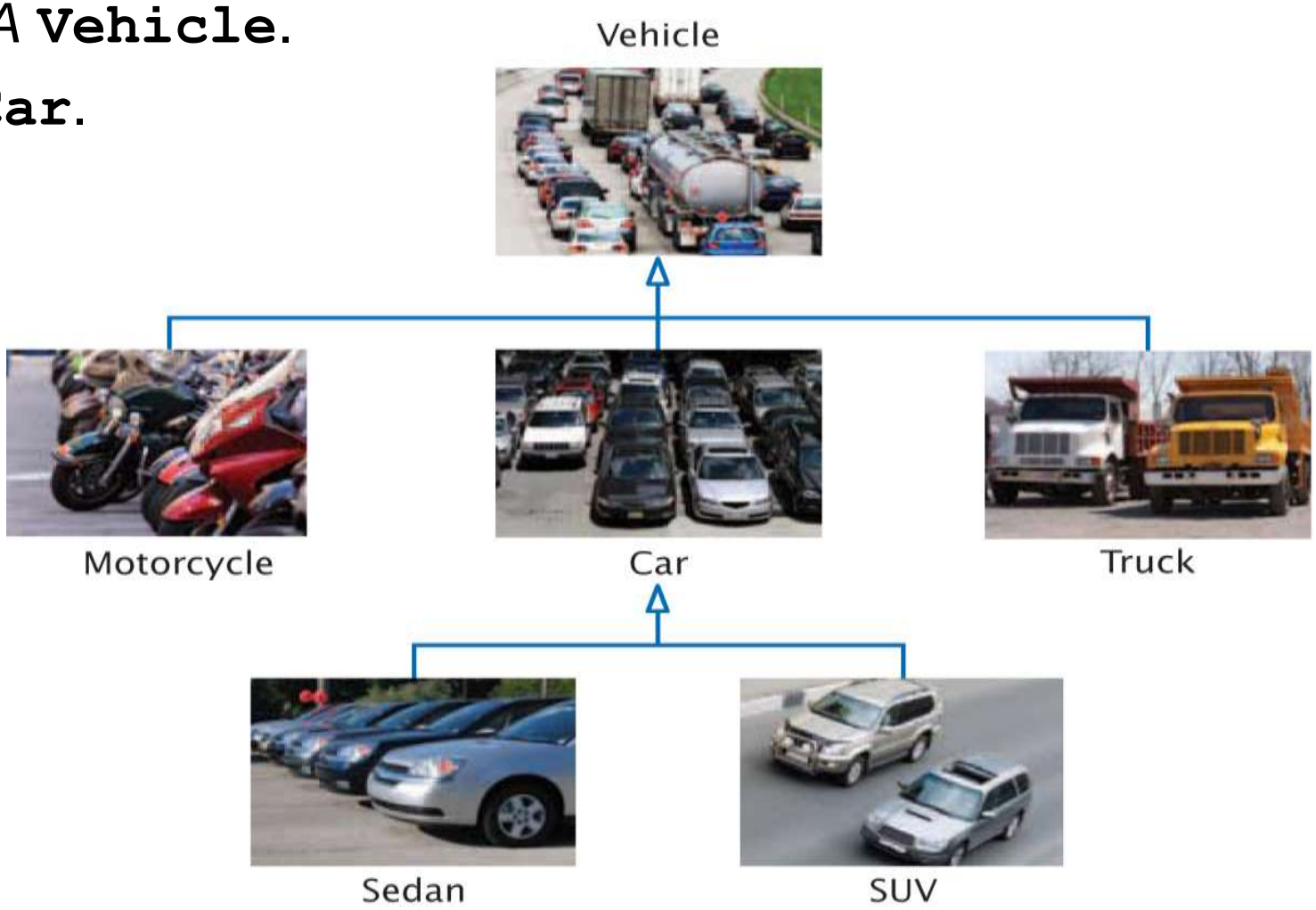
`SUV` derives from **`Car`**,
which derives from **`Vehicle`**.



Inheritance: The IS-A Relationship

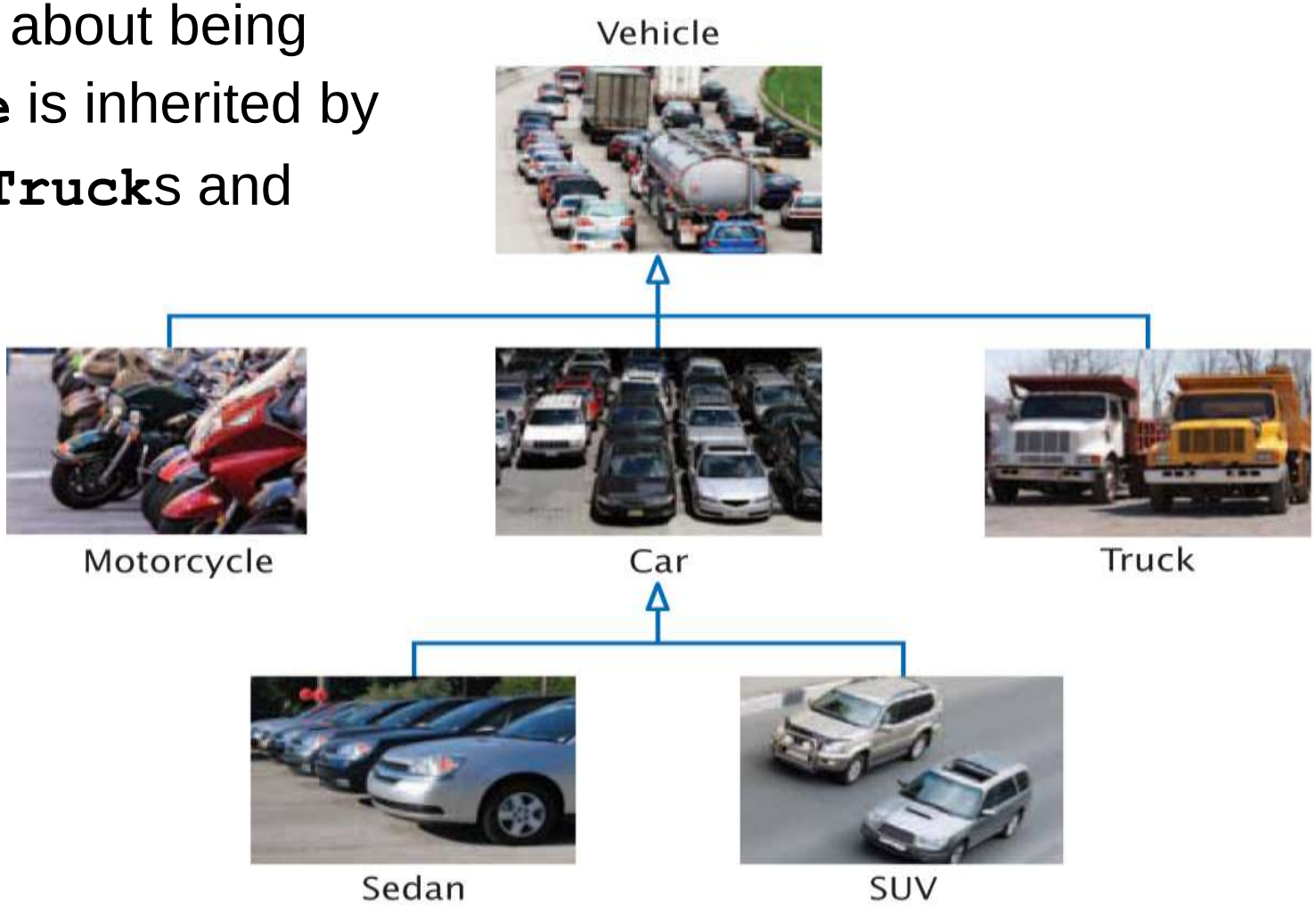
`Truck` *IS-A* **`Vehicle`**.

`SUV` *IS-A* **`Car`**.



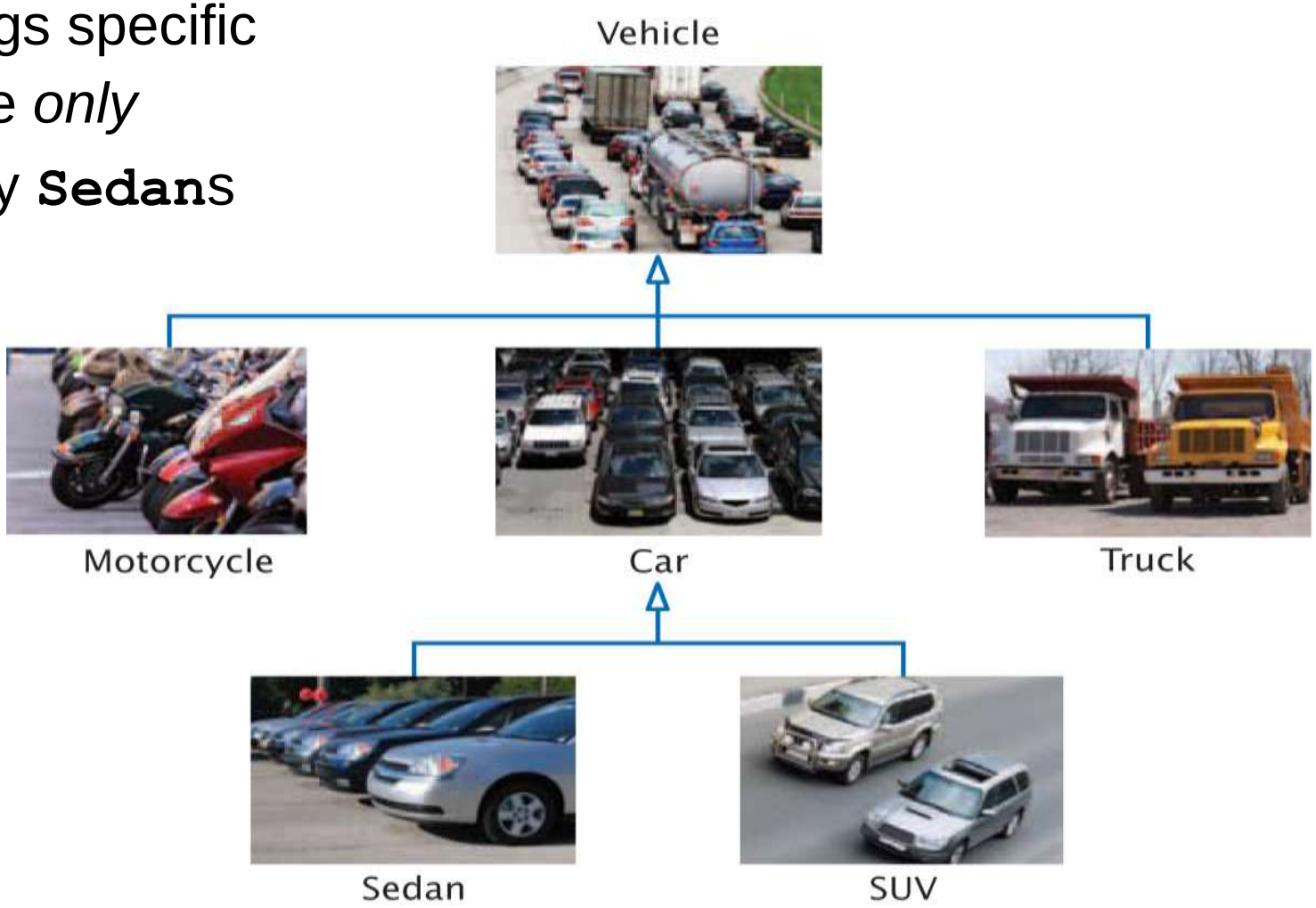
Inheritance: The IS-A Relationship

Everything about being a **`Vehicle`** is inherited by **`Car`**s and **`Truck`**s and **`SUV`**s.



Inheritance: The IS-A Relationship

Those things specific to **`Car`**s are *only* inherited by **`Sedan`**s and **`SUV`**s.



Inheritance Hierarchies

I inherited my ROYALNESS from my parents who inherited it from their parents.

Inheritance Hierarchies

I'm a special version of my base class.

I'm *very* special.

The Substitution Principle

Suppose we have an algorithm that manipulates a **`Vehicle`** object.

Since a car IS-A vehicle, we can supply a **`Car`** object to such an algorithm, and it will work correctly.

The Substitution Principle

The *substitution principle* states
that you can always use a derived-class object
when a base-class object is expected.

The Substitution Principle

If it was good enough for Mama,
it's good enough for me.

The Substitution Principle

Did you know you have already
been working with class hierarchies?

(No! Really?)

The Substitution Principle

Remember your friends **`cin`** and **`cout`**?

(Yes.)

Their types are in an inheritance chain.

(Chains? Like prisoners?)

The Substitution Principle

No, silly.

That's just another phrase for *inheritance hierarchy*.

The Substitution Principle

Look:

```
void process_input(istream& in);
```

You can call this function with
an **`ifstream`** object or with an **`istream`** object.

Why?

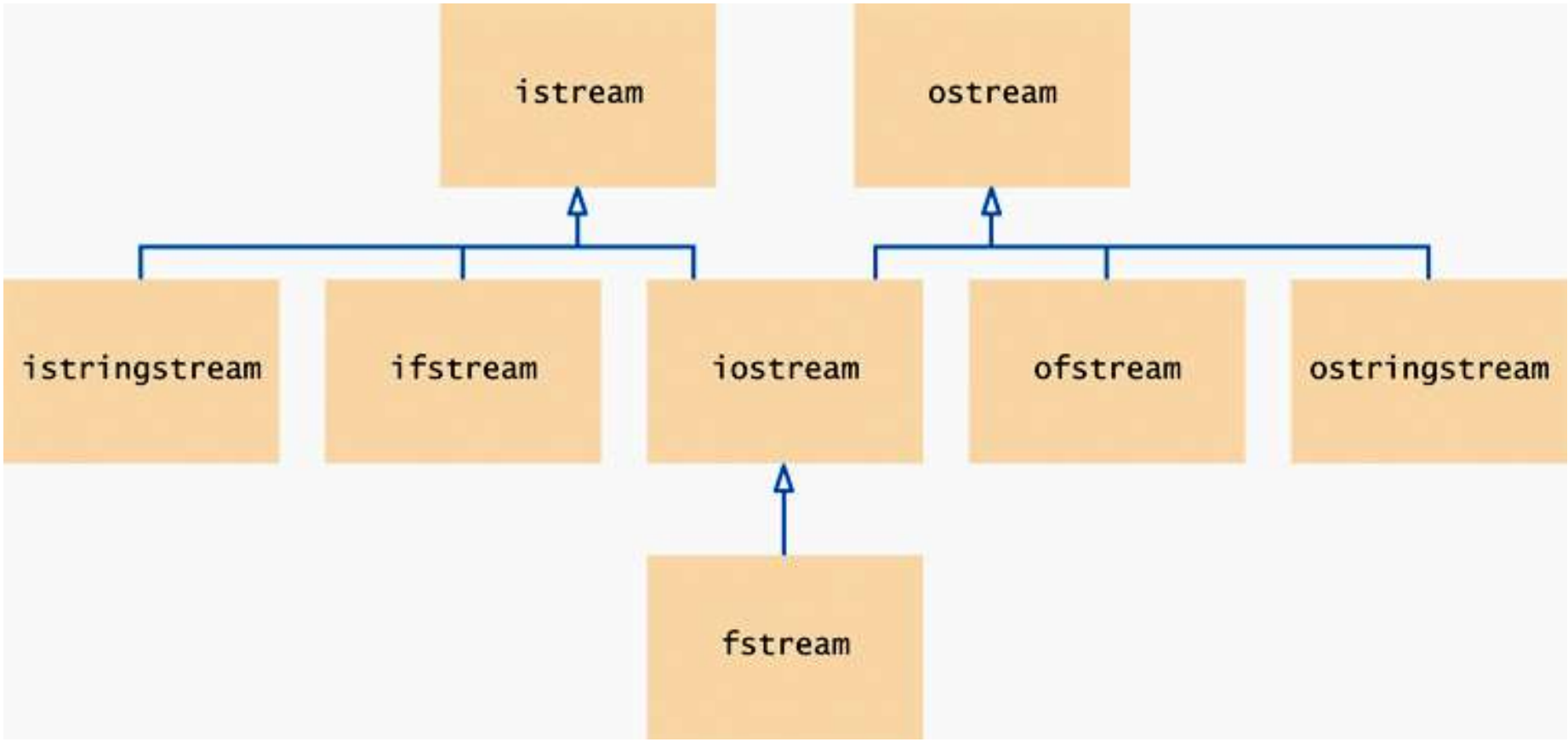
The Substitution Principle

Because **`istream`** is more *general* than **`ifstream`**.

```
void process_input(istream& in);
```

This works by inheritance:

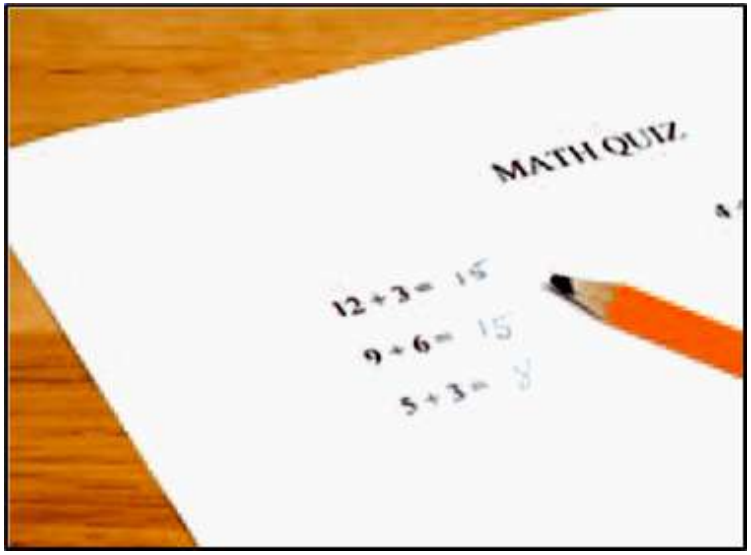
The C++ Stream Hierarchy



`istream` is the base class of **`ifstream`**.

`ifstream` inherits from **`istream`**.

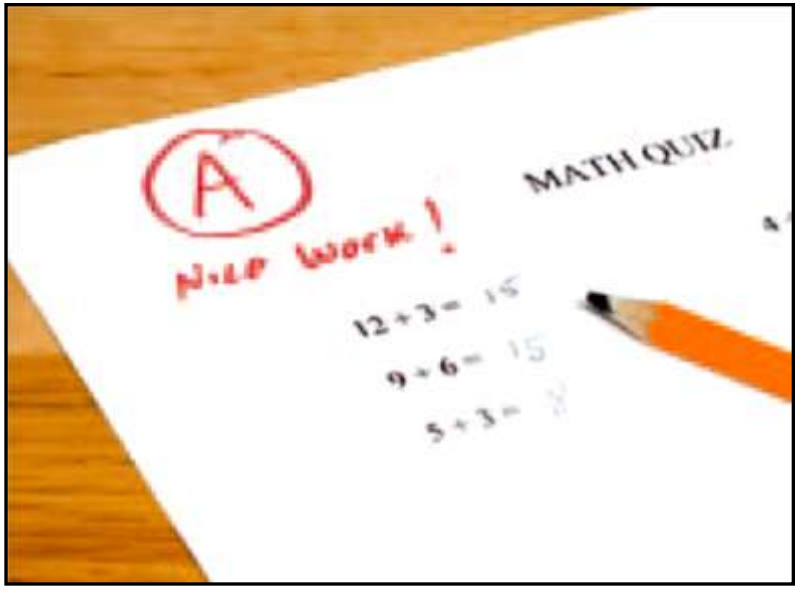
Quiz Time



OK.

QUIZ TIME!

Quiz Time



OK.

QUIZ TIME!

Everyone likes taking quizzes.

So let's take one.

(Oh no!)

Quiz Time

You *don't* like taking quizzes?

(Not really …)

OK

Let's *create* a Quiz Question hierarchy.

(Whew!)

Design Phase of Question Program

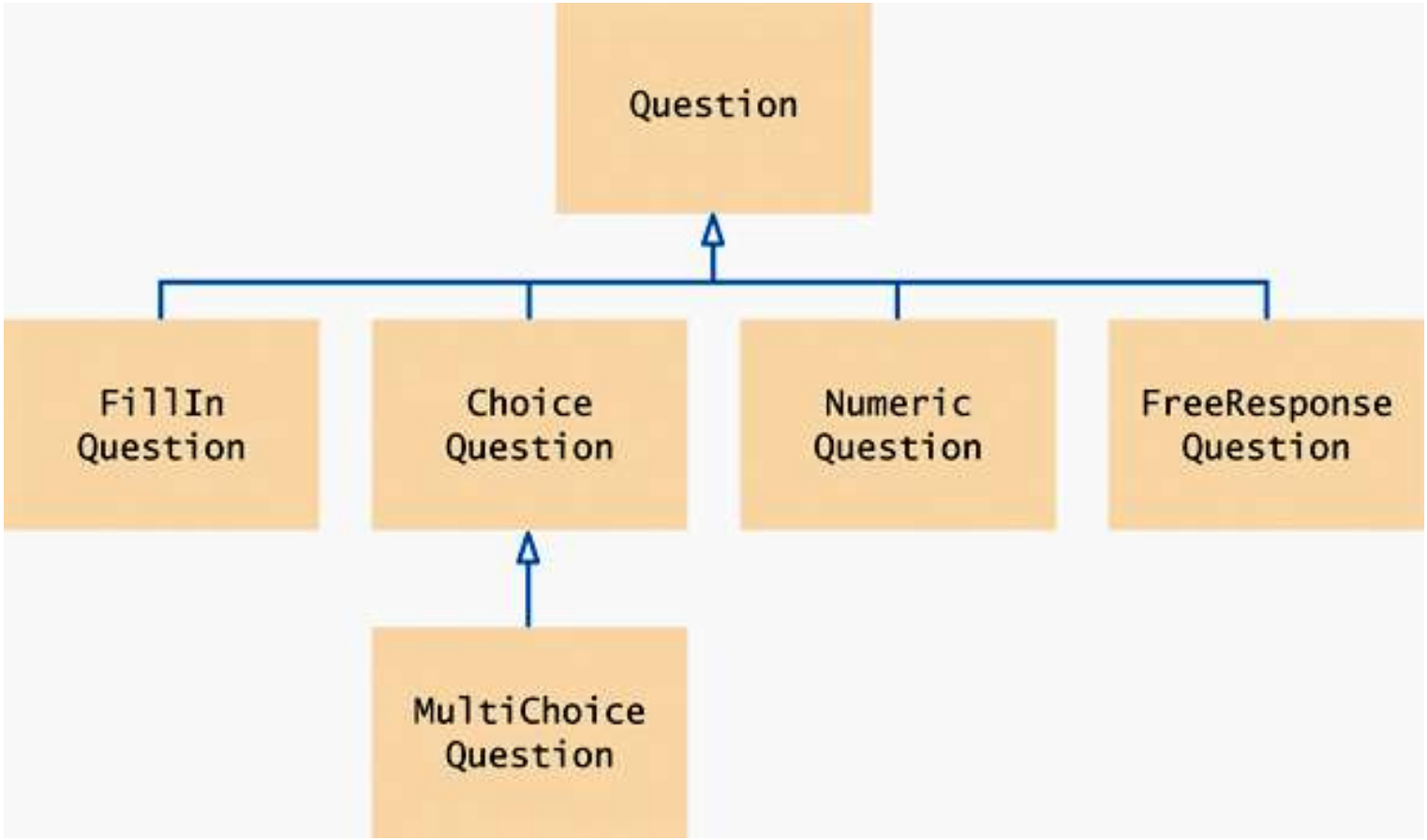
We will try to make this as general as we can because often quizzes consist of different kinds of questions:

- Fill-in-the-blank
- Choice (single or multiple)
- Numeric
  (we'll allow approximate answers to be OK)
- Free response

(We like multiple guess questions.)

Design Phase of Question Program

Here is the UML diagram that resulted from our analysis:



The Base Class: `Question`

At the root of this hierarchy is the **`Question`** type.

The Base Class: `Question`

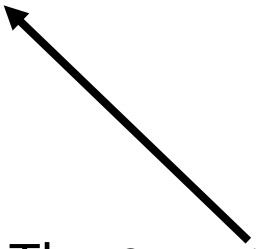
We want a object of Question type to work like this:

First, the programmer sets the question text
and the correct answer in the Question object.

The Base Class: `Question`

Then, when it's time to run the testing program with a user taking the test, the programmer asks the **`Question`** to display the text of the question:

```
Who was the inventor of C++?
```



The **`Question`** object displays this

The Base Class: `Question`

That programmer then gets the use's response and passes it to the **`Question`** object for evaluation:

```
Who was the inventor of C++?
Your answer: Bjarne Stroustrup
true
```

The **`Question`** object displays this

The Base Class: `Question` – Member Variables

To work as shown, a **`Question`** object would contain:

- The question's text
  - **`string text;`**
- The correct answer
  - **`string answer;`**

The Base Class: `Question` – Constructor

- `string text;`
- `string answer;`

What initial values should these have?

What could possibly be a reasonable initial value?

And – we'll write mutators to allow setting them later.

So a default **`Question`** constructor that does nothing is fine.

The **`string`** class constructor gives us empty **`string`**s.

The Base Class: `Question` – Accessors

A **`Question`** object should be able to:

- Display its text
  - **`void display() const;`**
- Check whether a given response is a correct answer
  - **`bool check_answer(string response) const;`**

The Base Class: `Question` – Mutators

And have these mutators:

- Set the question's text
  - `void set_text(string question_text);`
- Set the correct answer
  - `void set_answer(string correct_response);`

The Base Class: `Question` –

```
class Question
{
public:
   Question();
   void set_text(string question_text);
   void set_answer(string correct_response);
   bool check_answer(string response) const;
   void display() const;
private:
   string text;
   string answer;
};
```

`Question` Class Test Program

Here's a complete program to test our Question class.

ch10/quiz1/test.cpp

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Question
{
public:
   /**
      Constructs a question with empty text and answer.
   */
   Question();
```

`Question` Class Test Program

ch10/quiz1/test.cpp

```
/**
   @param question_text the text of this question
*/
void set_text(string question_text);

/**
   @param correct_response the answer to this question
*/
void set_answer(string correct_response);

/**
   @param response the response to check
   @return true if the response was correct,false
   otherwise
*/
bool check_answer(string response) const;
```

Question Class Test Program

ch10/quiz1/test.cpp

```
   /**
      Displays this question.
   */
   void display() const;

private:
   string text;
   string answer;
};

Question::Question()
{
}

void Question::set_text(string question_text)
{
   text = question_text;
}
```

`Question` Class Test Program

ch10/quiz1/test.cpp

```
void Question::set_answer(string correct_response)
{
   answer = correct_response;
}

bool Question::check_answer(string response) const
{
   return response == answer;
}

void Question::display() const
{
   cout << text << endl;
}
```

`Question` Class Test Program

ch10/quiz1/test.cpp

```
int main()
{
   string response;

   // Show Boolean values as true, false
   cout << boolalpha;

   Question q1;
   q1.set_text("Who was the inventor of C++?");
   q1.set_answer("Bjarne Stroustrup");

   q1.display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << q1.check_answer(response) << endl;

   return 0;
}
```

boolalpha

Did you notice this in the code?

```
// Show Boolean values as true, false
cout << boolalpha;
```

The **boolalpha** manipulator causes Boolean values to be displayed as the output strings:

"true" for **true**

and **"false"** for **false**

– otherwise, numbers would be displayed.

Implementing Derived Classes

Now for those different kinds of questions.

All of the different kinds *IS-A* Question

so we code by starting with the base class (**`Question`**)
and then we write code for what makes them
special versions of more general **`Question`** type.

Implementing Derived Classes

I know about being *special*.

Implementing Derived Classes

Through inheritance, each of these types will have the data members and member functions set up in class **`Question`**.

– plus “specialness-es”

(We don’t rewrite the member functions)

(code reuse in action – all right!)

Implementing Derived Classes

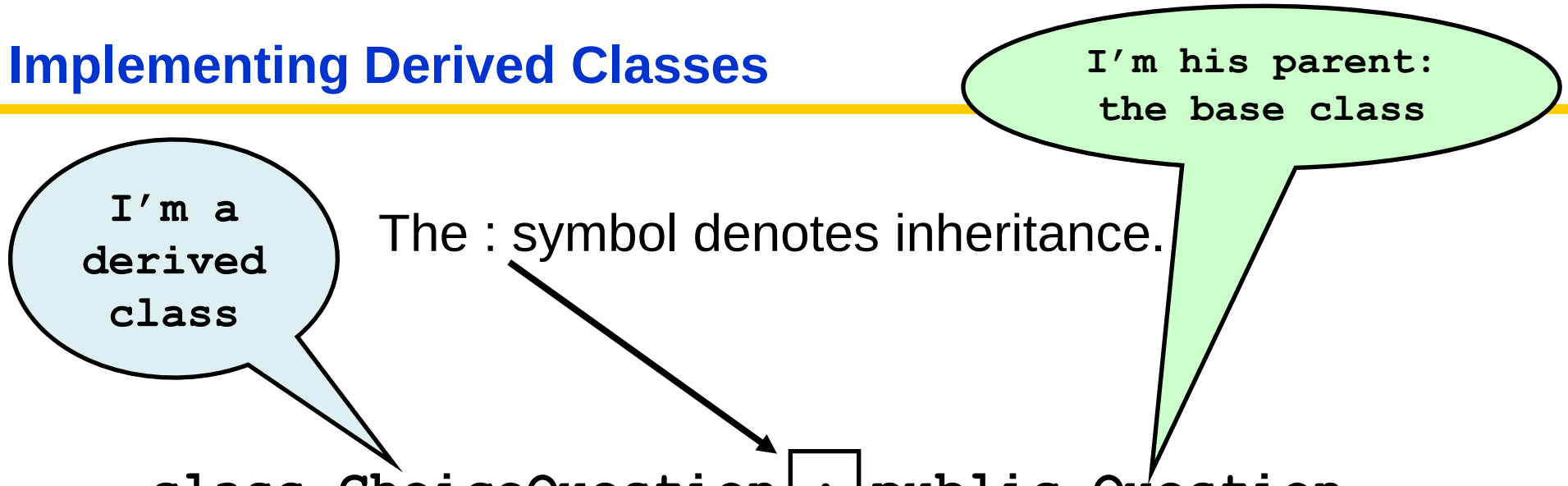
That's *me!*

Implementing Derived Classes

We will start with the “choice question” kind of question:

```
class ChoiceQuestion : public Question
{
public:
   // New and changed member
   // functions will go here
private:
   // Additional data members
   // will go here
};
```

Implementing Derived Classes




```
class ChoiceQuestion : public Question
{
public:
   // New and changed member
   // functions will go here
private:
   // Additional data members
   // will go here
};
```

Implementing Derived Classes

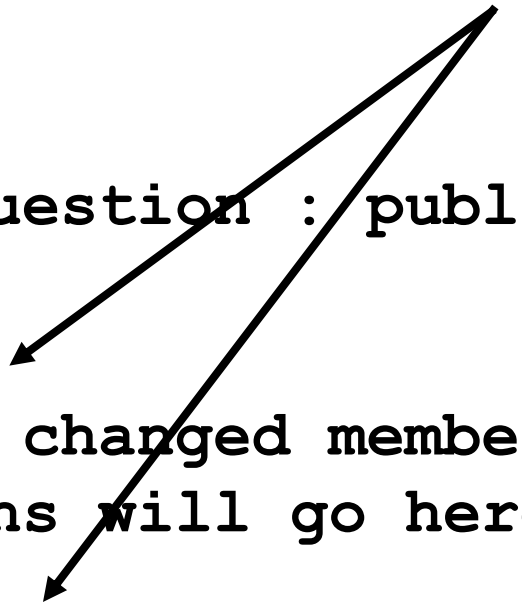
The keyword **`public`** makes sure this is an IS-A relationship.

```
class ChoiceQuestion : public Question
{
public:
   // New and char      mber
   // 
   // will go here
};
```

ChoiceQuestion IS-A **Question**

Implementing Derived Classes

We are telling the compiler to start with the **`Question`** class and add these things to it.



```
class ChoiceQuestion : public Question
{
public:
   // New and changed member
   // functions will go here
private:
   // Additional data members
   // will go here
};
```

Implementing Derived Classes – Analysis of the Problem

So what are these new things?

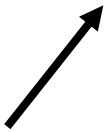
How does a **`ChoiceQuestion`** differ from its base class?

It's use in the interaction with a user will be different:

Implementing Derived Classes – Analysis of the Problem

After a programmer has set the question text and the several multiple choice answers the **`ChoiceQuestion`** object is asked to display:

```
In which country was the inventor of C++ born?
1: Australia
2: Denmark
3: Korea
4: United States
```



The Question object displays all this

Implementing Derived Classes – Analysis of the Problem

The programmer then gets the user's input,

and sends it to the **`ChoiceQuestion`** object to see if it is correct.

```
In which country was the inventor of C++ born?
1: Australia
2: Denmark
3: Korea
4: United States
Your answer: 2
true
```

The Question object displays this

Implementing Derived Classes – Coding

The code will have to make **`ChoiceQuestion`** be a specialized form of a **`Question`** object.

Implementing Derived Classes

`ChoiceQuestion` must have:

- Storage for the various choices for the answer
  - **`Question`** has the text and correct answer, not these
- A member function for adding another choice
- A display function
  - The designer of the **`Question`** class could not have known how to display this sort of multiple choice question. It only has the question itself, not the choices.
  - In the **`ChoiceQuestion`** class you will have to *rewrite* the display function **`display`**.
    - This is called *overriding* a member function.

Implementing Derived Classes – Coding

After specifying the class you are inheriting from, you only write the differences:

```
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

Implementing Derived Classes – Coding

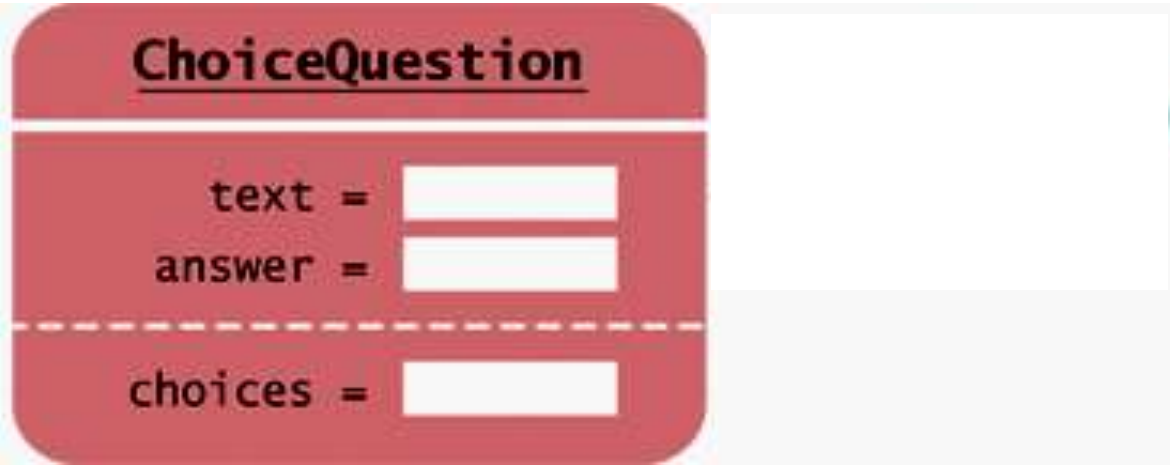
Where is the **`set_text`** member function?

Where is the **`string text;`** data member?

Right there

```
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

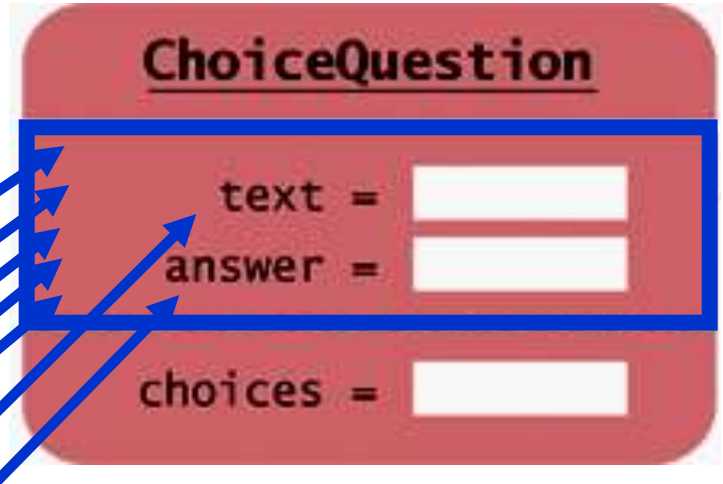
Derived Classes




```
class Question
{
public:
   Question();
   void set_text(string question_text);
   void set_answer(string correct_response);
   bool check_answer(string response) const;
   void display() const;
private:
   string text;
   string answer;
};
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

`ChoiceQuestion` is *one* type, made of two subtypes.

Derived Classes

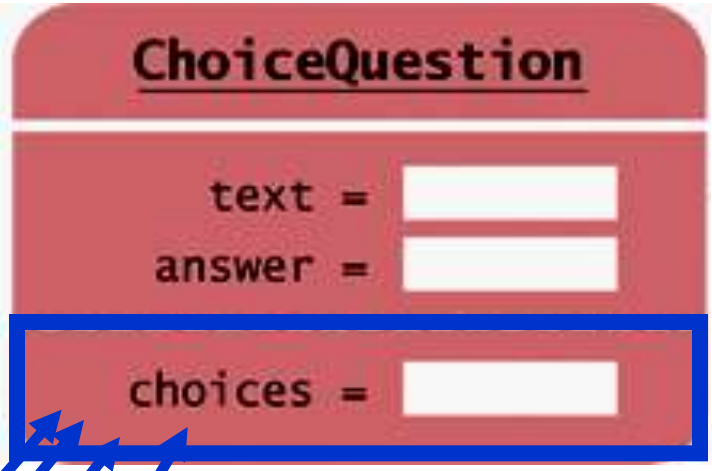


members from **class Question**

```
class Question
{
public:
   Question();
   void set_text(string question_text);
   void set_answer(string correct_response);
   bool check_answer(string response) const;
   void display() const;
private:
   string text;
   string answer;
};
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

It is based on the **Question** type so it has those parts.

Derived Classes

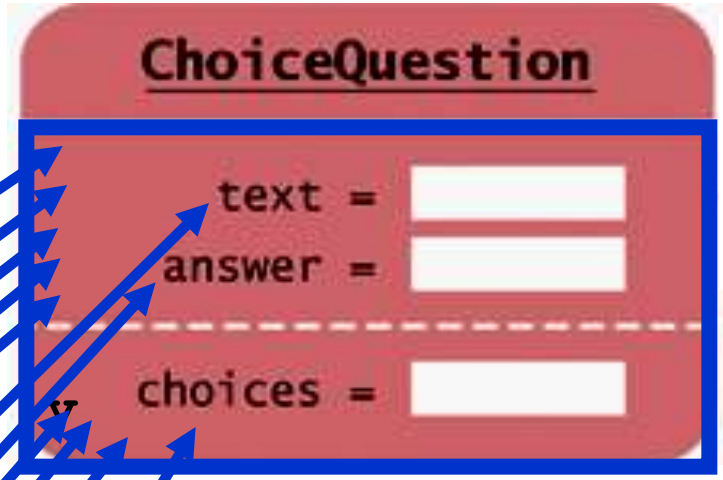


specializations

```
class Question
{
public:
   Question();
   void set_text(string question_text);
   void set_answer(string correct_response);
   bool check_answer(string response) const;
   void display() const;
private:
   string text;
   string answer;
};
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

And, added to those parts from the Question type, it has its own specializations (its *specialness-es*).

Derived Classes



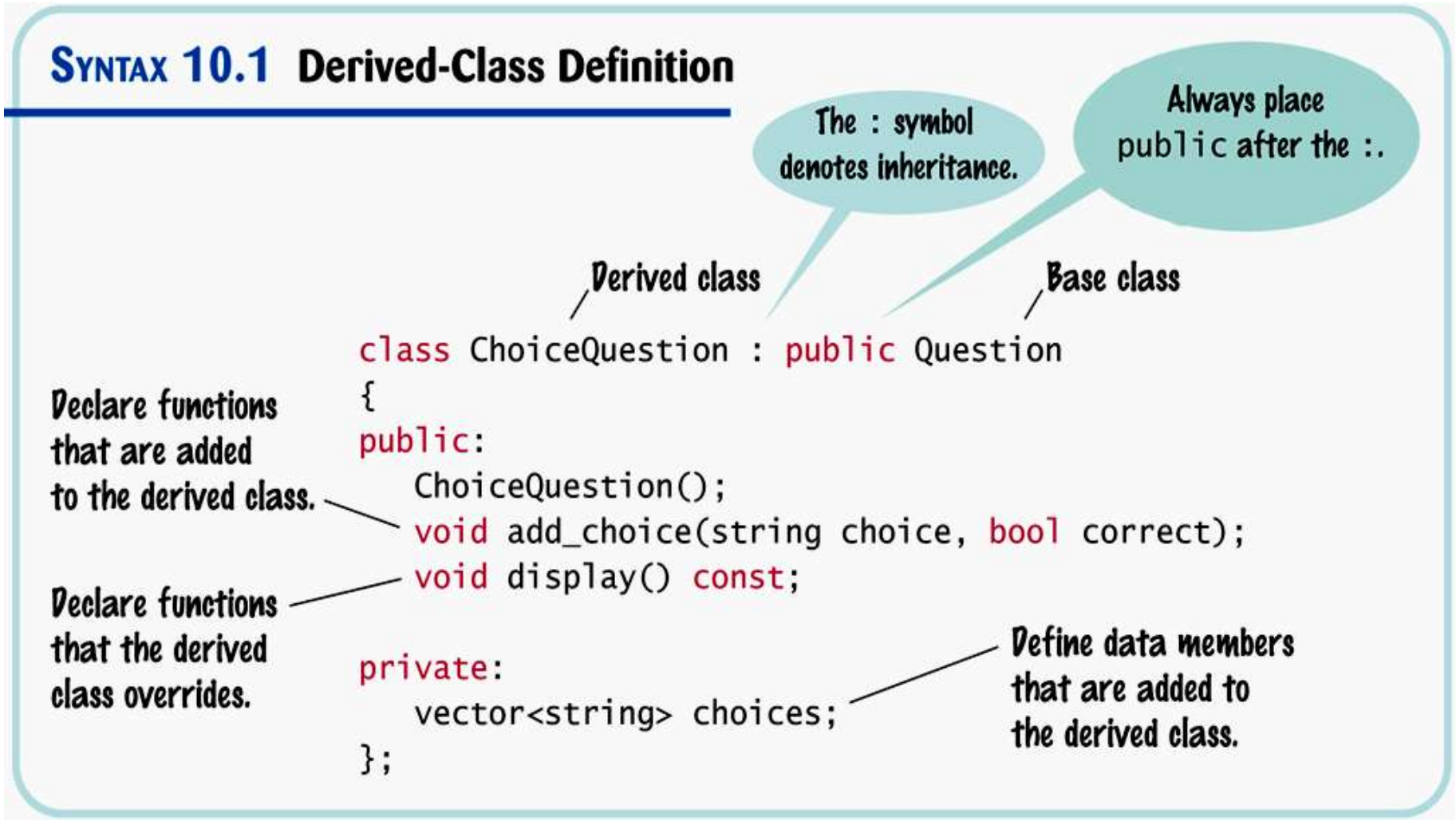

```
class Question
{
public:
   Question();
   void set_text(string question_text);
   void set_answer(string correct_response);
   bool check_answer(string response) const;
   void display() const;
private:
   string text;
   string answer;
};
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

to make *one* type:

`ChoiceQuestion`

Derived Classes – Syntax

Syntax 10.1 Derived-Class Definition



Implementing Derived Classes – Coding

The derived class inherits all data members and all functions that it does not override.

Implementing Derived Classes – Coding

But…

private means private.

Implementing Derived Classes – Coding

Consider:

```
Choice_question choice_question;
choice_question.set_answer("2");
```

(OK, a public method in a derived part.)

Implementing Derived Classes – Coding

How about:

```
Choice_question choice_question;
choice_question.answer = "2";
```

(Well, it *did* inherit that data member …)

Implementing Derived Classes – Coding

No, private means private!

```
Choice_question choice_question;
choice_question.answer = "2";
// will not compile - private
```

(OK – private means private even if inherited.)

Implementing Derived Classes – Coding

This means that when you are writing the **`ChoiceQuestion`** member functions, you cannot directly access any private data members in **`Question`**.

(Oh, dear. What to do?)

Implementing Derived Classes – Coding

A good design would be
for this function to take a choice
and somehow an indication that this
choice is the correct answer.

(bool! bool! bool!)

Implementing Derived Classes – Coding

Very good:

```
void ChoiceQuestion::add_choice(string choice, bool correct)
{
   choices.push_back(choice);
   if (correct)
   {
      // Convert choices.size() to string
      ostringstream stream;
      stream << choices.size();
      string num_str = stream.str();

      // Set num_str as the answer
      ...
   }
}
```

...but

Implementing Derived Classes – Coding

Oh dear

```
void ChoiceQuestion::add_choice(string choice, bool correct)
{
   choices.push_back(choice);
   if (correct)
   {
      // Convert choices.size() to string
      ostringstream stream;
      stream << choices.size();
      string num_str = stream.str();

      // Set num_str as the answer
      ...
   }
}
```

answer is private!

Implementing Derived Classes – Coding

Happily, the designer of **`Question`** provided accessors!

```
void ChoiceQuestion::add_choice(string choice, bool correct)
{
   choices.push_back(choice);
   if (correct)
   {
      // Convert choices.size() to string
      ostringstream stream;
      stream << choices.size();
      string num_str = stream.str();

      // Set num_str as the answer
      set_answer(num_str);
   }
}
```

Common Error

Here is the class definition for **`ChoiceQuestion`** again.

It's wrong – we made a *small* mistake.

Can you find it?

```
class ChoiceQuestion : Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

Common Error: Private Inheritance


```
class ChoiceQuestion : Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

Common Error: Private Inheritance

If you do not specify public inheritance,
you get *private inheritance* and everything is a mess.

```
class ChoiceQuestion : private Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

Common Error: Private Inheritance

If you do not specify **public** inheritance,
you get *private inheritance* and everything is a mess.

(**Be careful, son!**)

Common Error: Replicating Base Class Members

You know all about private access:

A derived class has no access to
the private data members of the base class.

Common Error: Replicating Base Class Members

But when some programmers encounter a compiler error, they (not you, of course) don't stop and THINK.

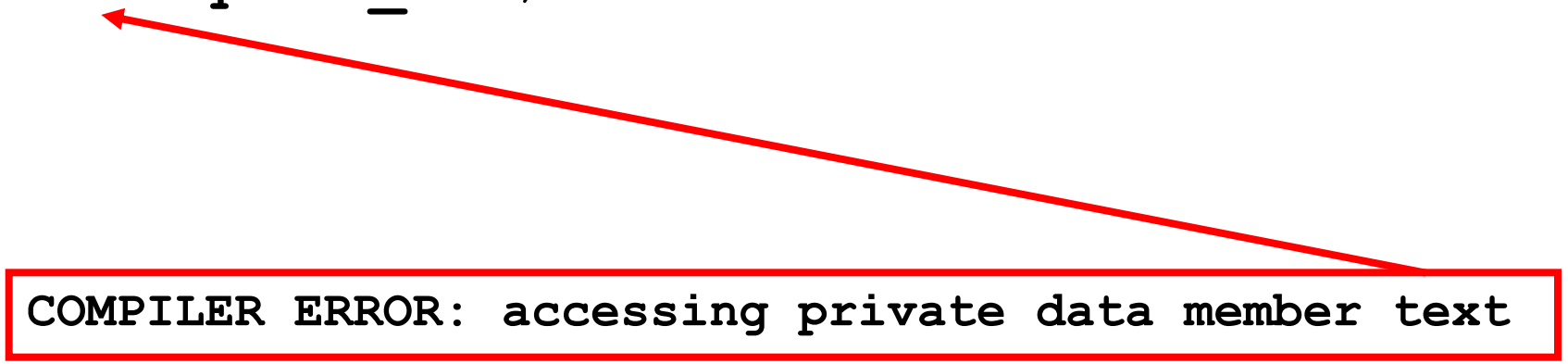
They just start hacking.

```
ChoiceQuestion::ChoiceQuestion(string quest_txt)
{
   text = quest_txt;
}
```

Common Error: Replicating Base Class Members

And an “easy” fix seems to be to add the data member that the compiler is complaining about.

```
ChoiceQuestion::ChoiceQuestion(string quest_txt)
{
   text = quest_txt;
}
```



Common Error: Replicating Base Class Members

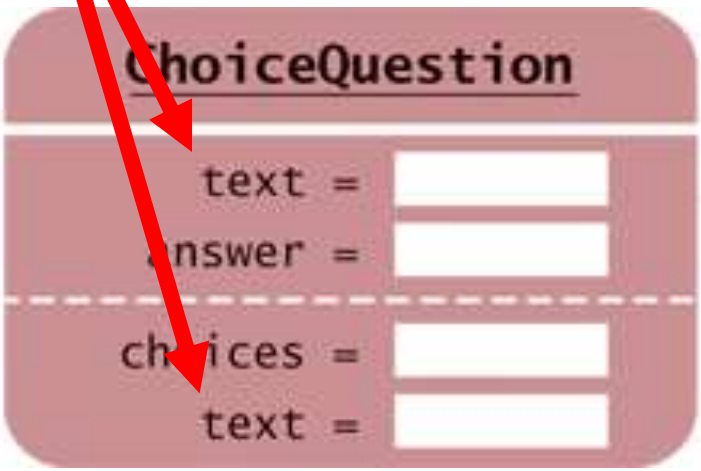
And an “easy” fix seems to be to add the data member that the compiler is complaining about.

```
class ChoiceQuestion : public Question
{
...
private:
   vector<string> choices;
   string text;
}
```

Common Error: Replicating Base Class Members

Oh dear!

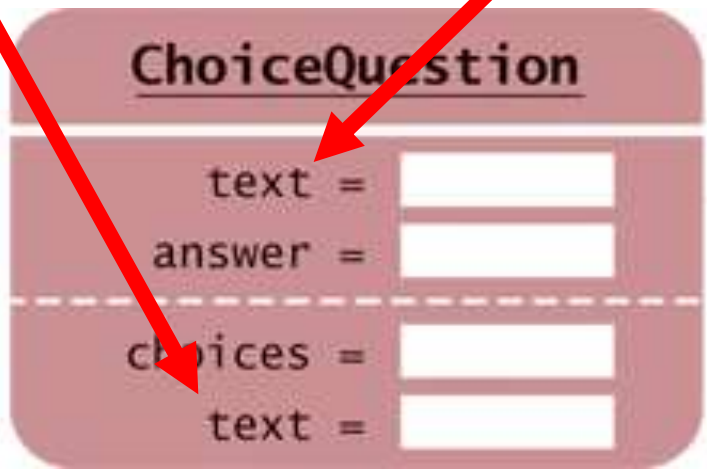
TWO test data members!



Common Error: Replicating Base Class Members

One set by the constructor

and the other displayed!



Oh dear! Oh Dear! OH DEAR!!!

End Chapter Ten: Inheritance, Part I

Slides by Evan Gallagher & Nikolay Kirov