Chapter Ten: Inheritance, Part II

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;
};
```

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

Implementing Derived Classes

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

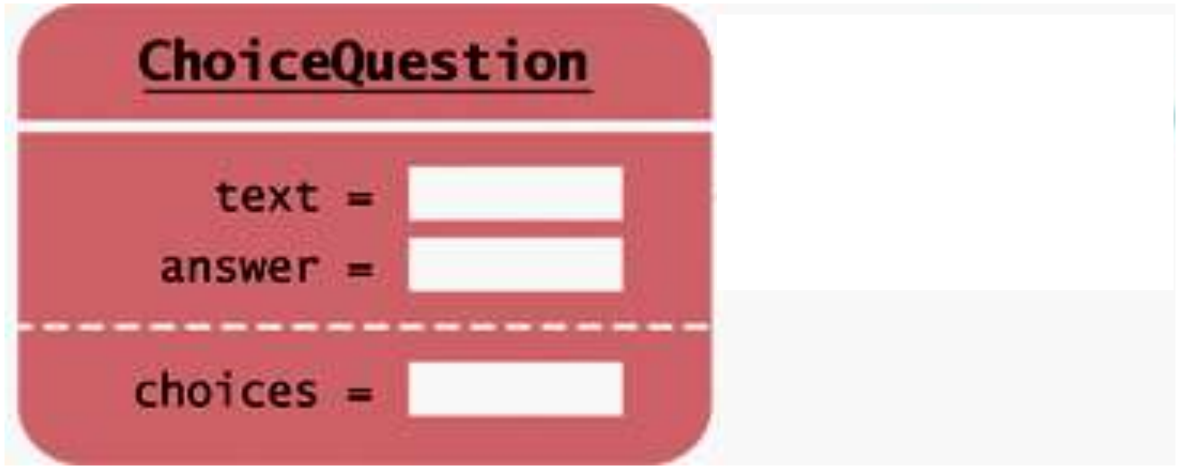
```
class ChoiceQuestion : public Question
{
public:
   // New and changed member
   // functions will go here
private:
   // Additional data members
   // will go here
};
```

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;
};
```

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.

Calling the Base-Class Constructor

If the base-class has a default constructor,
we are in luck: that can be used.

If not…

We have to somehow “call” the base-class constructor from the derived-class’ constructor.

Oh dear!

Calling the Base-Class Constructor

The initialization list to the rescue!

We put the base-class' constructor "call" in the initialization list.

The initialization list code happens before the **{ }** of the constructor so the base-class will be completely set up before any specializations take place.

Overriding Member Functions

The second part is easy – just a data member

```
int ChoiceQuestion::display() const
{
   // Display the question text
      ...
   // Display the answer choices
   for (int i = 0; i < choices.size(); i++)
   {
      cout << i + 1 << ": "
         << choices[i] << endl;
  }
}
```

Overriding Member Functions

The first part seems easy too – call the **`display`** in Question:

```
int ChoiceQuestion::display() const
{
   // Display the question text
   display();


   // Display the answer choices
   ...
```

Overriding Member Functions

Oh no! **`this`** points to me

– and I have a **`display`** member function!

```
int ChoiceQuestion::display() const
{
   // Display the question text
   display();
```

implicit parameter`.display();`

```
   // Display the answer choices
   ...
```

Overriding Member Functions

I'm talking to myself!!!

```
int ChoiceQuestion::display() const
{
    // Display the question text
    display();
```

implicit parameter`.display();`

```
    // Display the answer choices
    ...
}
```

Overriding Member Functions

The solution is to do what you said in the first place:
Call **Question**'s display:

```
int ChoiceQuestion::display() const
{
   // Display the question text
   display();


   // Display the answer choices
   ...
}
```

Overriding Member Functions

The solution is to do what you said in the first place:
Call **Question**'s display:

```
int ChoiceQuestion::display() const
{
   // Display the question text
   Question::display();


   // Display the answer choices
   ...
}
```

Overriding Member Functions

When you override a function, you usually want to *extend* the functionality of the base-class version.

Therefore, you often need to invoke the base-class version before extending it.

Overriding Member Functions

Use the **`BaseClass::function`** notation.

Overriding Member Functions

However, you have no obligation to call the base-class function.

Occasionally, a derived class overrides a base-class function and specifies an entirely different functionality.

A Program With `ChoiceQuestion`

In the following program, the definition of the **`Question`** class, which you have already seen, is placed into **`question.h`**, and the implementation is in **`question.cpp`**.

A Program With `ChoiceQuestion`

ch10/quiz2/test.cpp

```
#include <iostream>
#include <sstream>
#include <vector>
#include "question.h"

class ChoiceQuestion : public Question
{
public:
   /**
      Constructs a choice question with no choices.
   */
   ChoiceQuestion();
```

A Program With `ChoiceQuestion`

ch10/quiz2/test.cpp

```
   /**
    Adds an answer choice to this question.
    @param choice the choice to add
    @param correct true if this is the correct choice,
    false otherwise
   */
   void add_choice(string choice, bool correct);

   void display() const;
private:
   vector<string> choices;
};

ChoiceQuestion::ChoiceQuestion()
{
}
```

A Program With `ChoiceQuestion`

ch10/quiz2/test.cpp

```
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_answer(num_str);
   }
}
```

A Program With `ChoiceQuestion`

ch10/quiz2/test.cpp

```
void ChoiceQuestion::display() const
{
   // Display the question text
   Question::display();

   // Display the answer choices
   for (int i = 0; i < choices.size(); i++)
   {
      cout << i + 1 << ": " << choices[i] << endl;
   }
}
```

A Program With `ChoiceQuestion`

ch10/quiz2/test.cpp

```
int main()
{
   string response;
   cout << boolalpha;

   // Ask a basic question
   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;
```

A Program With `ChoiceQuestion`

ch10/quiz2/test.cpp

```
   // Ask a choice question

   ChoiceQuestion q2;
   q2.set_text("In which country was the inventor of C++ born?");
   q2.add_choice("Australia", false);
   q2.add_choice("Denmark", true);
   q2.add_choice("Korea", false);
   q2.add_choice("United States", false);

   q2.display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << q2.check_answer(response) << endl;

   return 0;
}
```

Common Error: Forgetting the Base-Class Name

Don’t forget to use the

`BaseClass::memberFunction`

notation when you want to call a member function in your base class – especially when you are writing:

`DerivedClass::memberFunction`

(**Yes, Mom.**)

Common Error: Forgetting the Base-Class Name

`Manager` derives from **`Employee`**
which has a method named **`get_salary`**.

Careful:

```
double Manager::get_salary() const
{
   double base_salary = get_salary();
      // Error—should be Employee::get_salary()
   return base_salary + bonus;
}
```

The Slicing Problem

The Slicing Problem

In the `main` function of that last program, there was some repetitive code to display each question and check the responses.

It would be nicer if all questions were collected in an array.

You could then loop to present them to the user:

The Slicing Problem

```
const int QUIZZES = 2;
Question quiz[QUIZZES];
quiz[0].set_text("Who was the inventor of C++?");
quiz[0].set_answer("Bjarne Stroustrup");

ChoiceQuestion cq;
cq.set_text("In which country was the inventor of C++ born?");
cq.add_choice("Australia", false);
...
quiz[1] = cq;

for (int i = 0; i < QUIZZES; i++)
{
   quiz[i].display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << quiz[i].check_answer(response) << endl;
}
```

This works

because **ChoiceQuestion** is a special case of **Question**

The Slicing Problem

However, is it really working?

Here’s a run of the program:

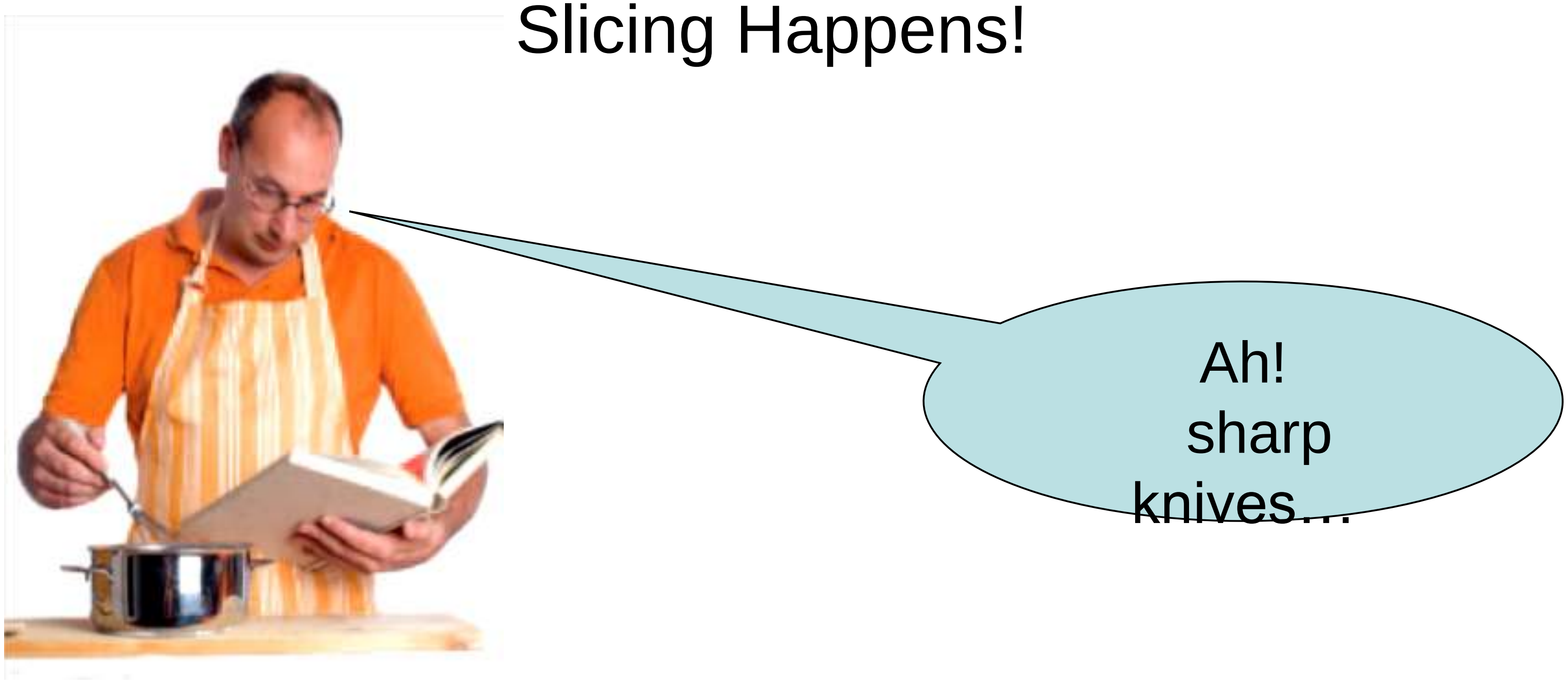
```
Who was the inventor of C++?
Your answer: Bjarne Stroustrup
true
In which country was the inventor of C++ born?
Your answer:
```

Where are the choices?

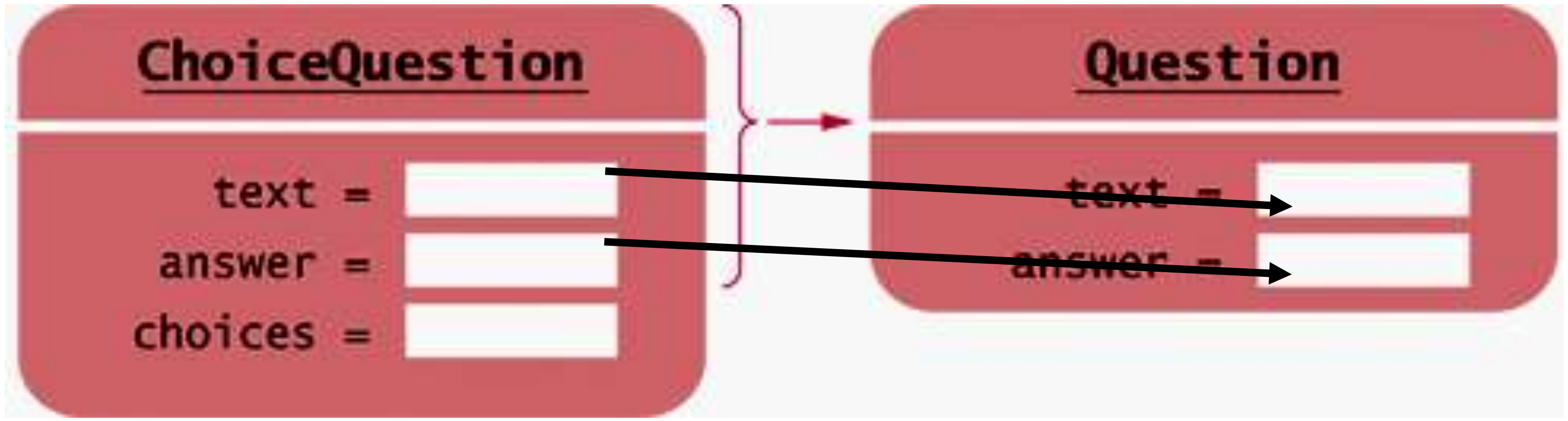
The Slicing Problem

When you assign a derived class to
a base class memory location,

Slicing Happens!



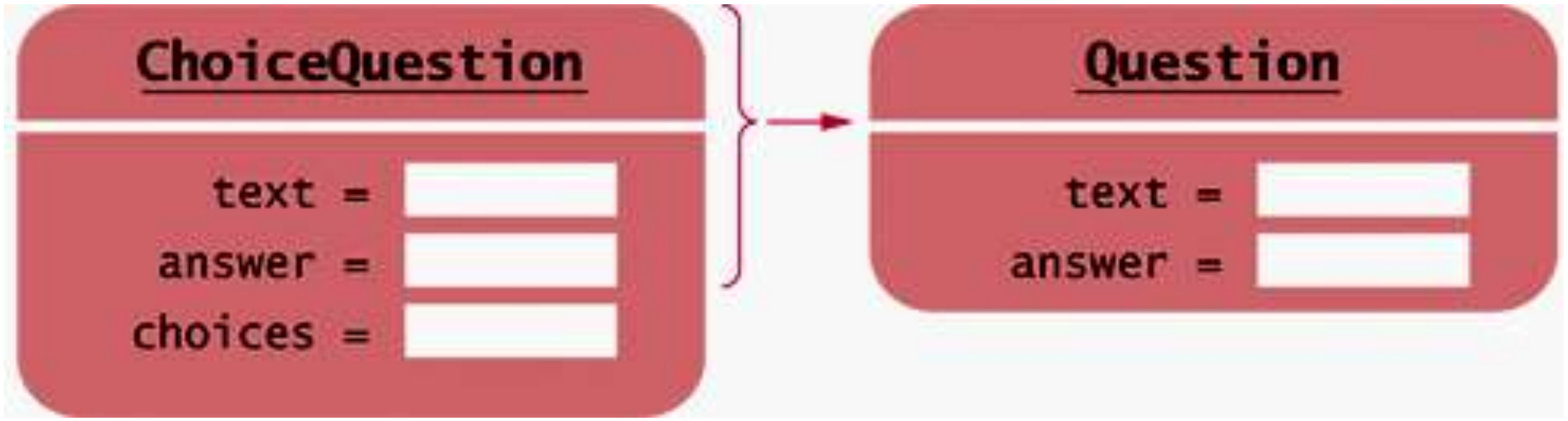
The Slicing Problem



The assignment copies **`text`**

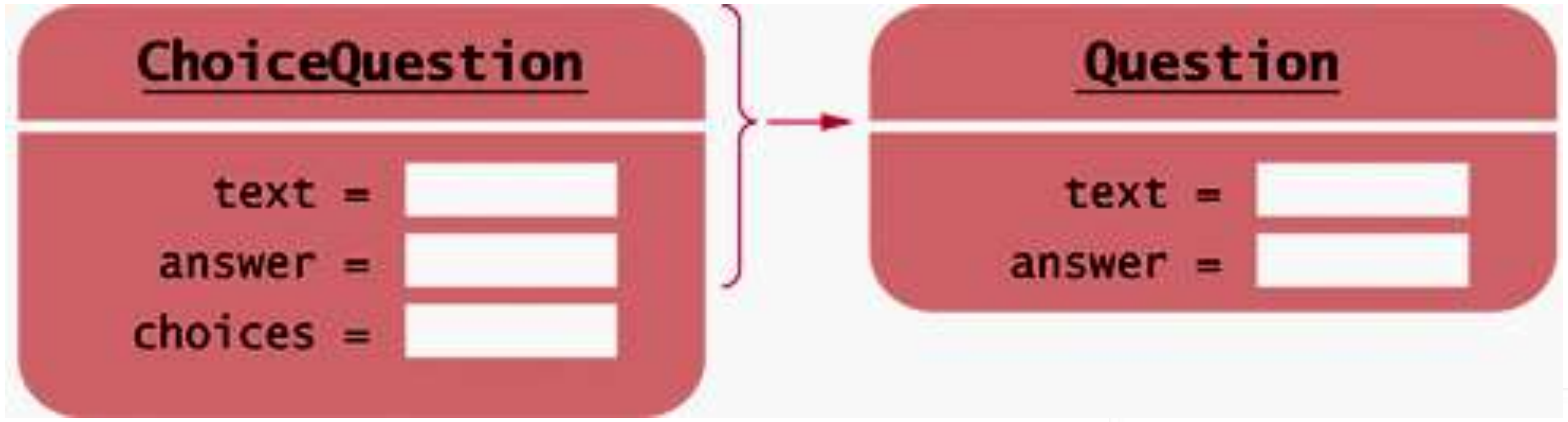
and **`answer`** just fine.

The Slicing Problem



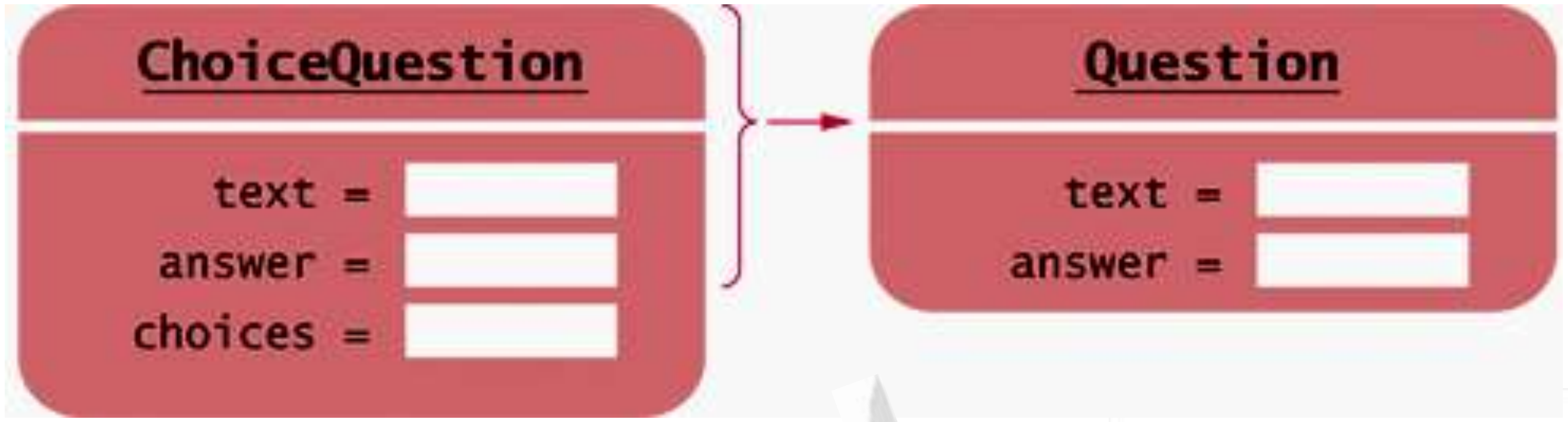
But where does **`choices`** go?

The Slicing Problem

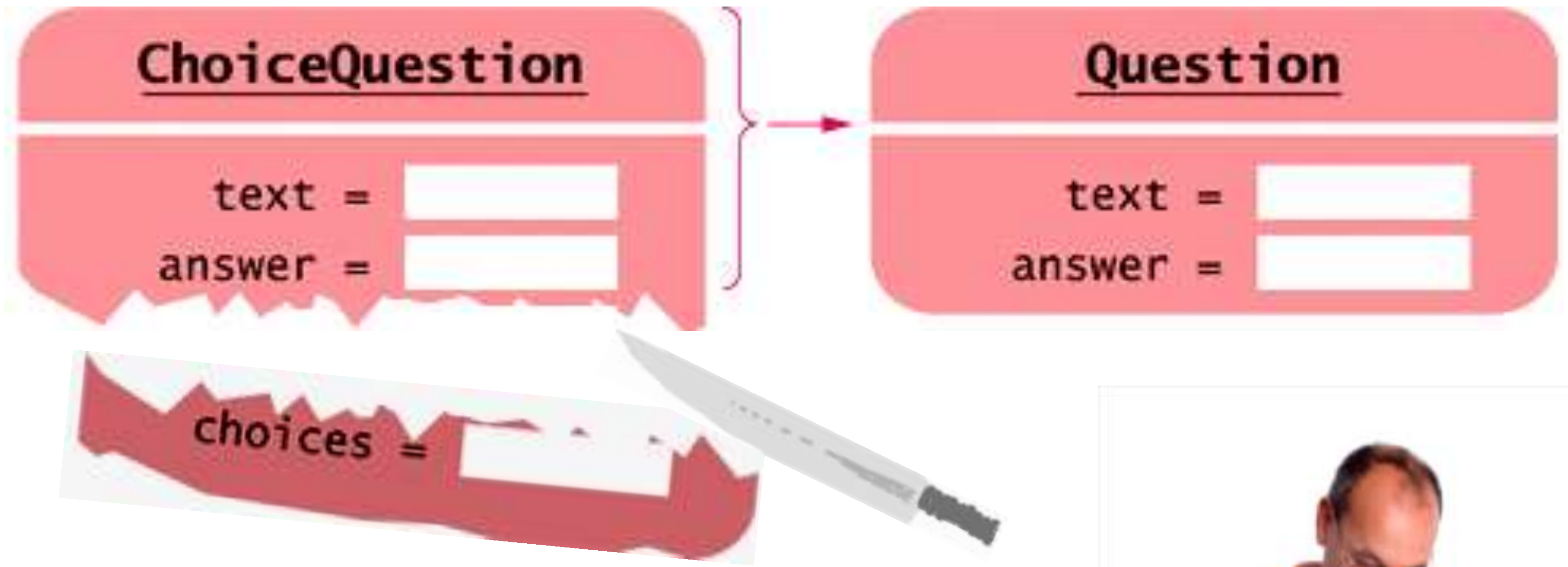


But where does **`choices`** go?

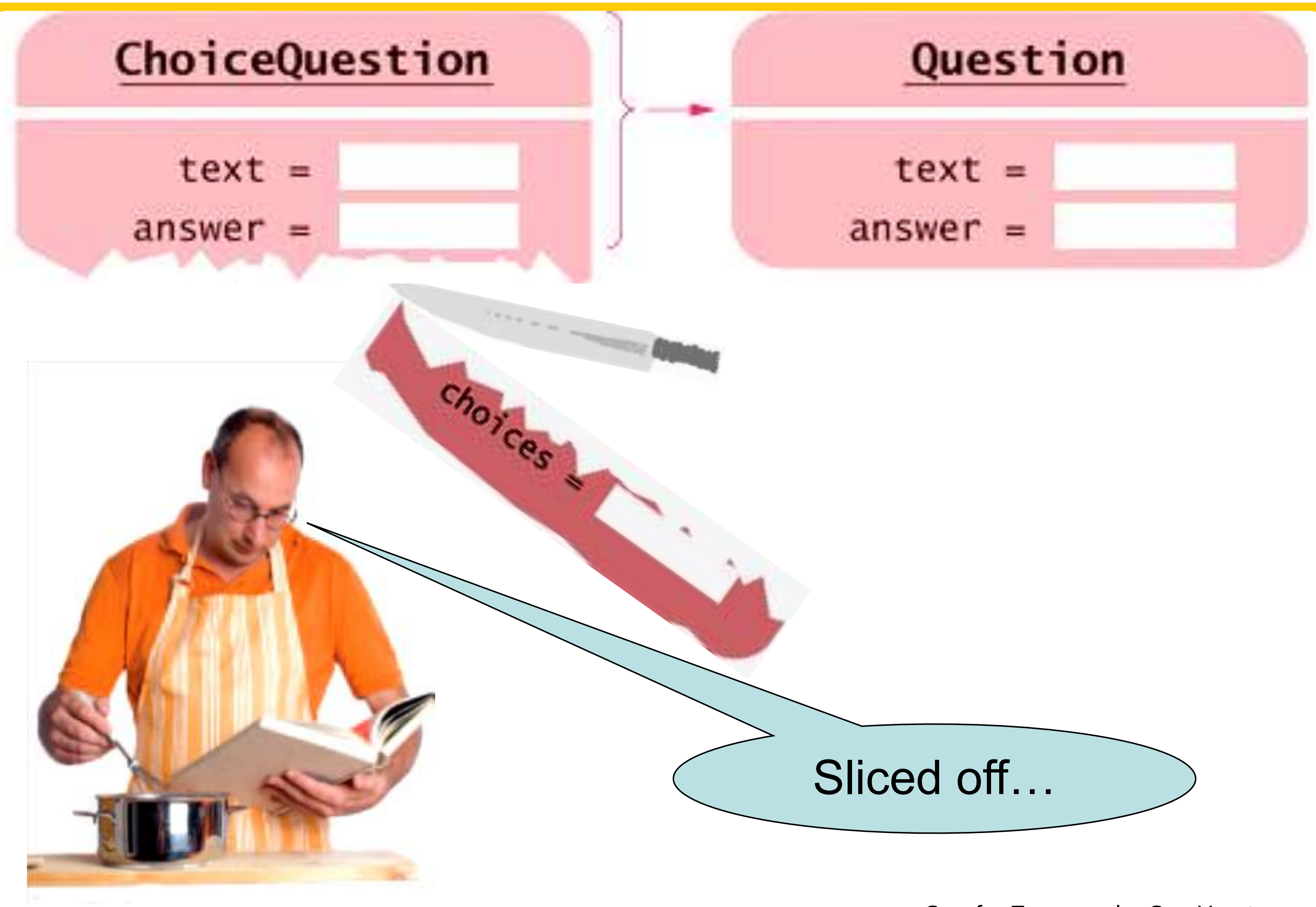
The Slicing Problem



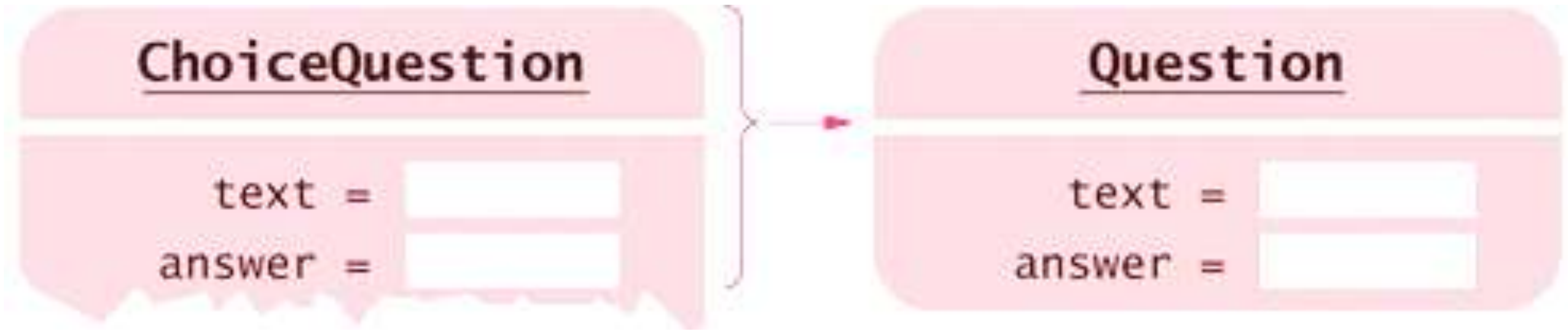
The Slicing Problem



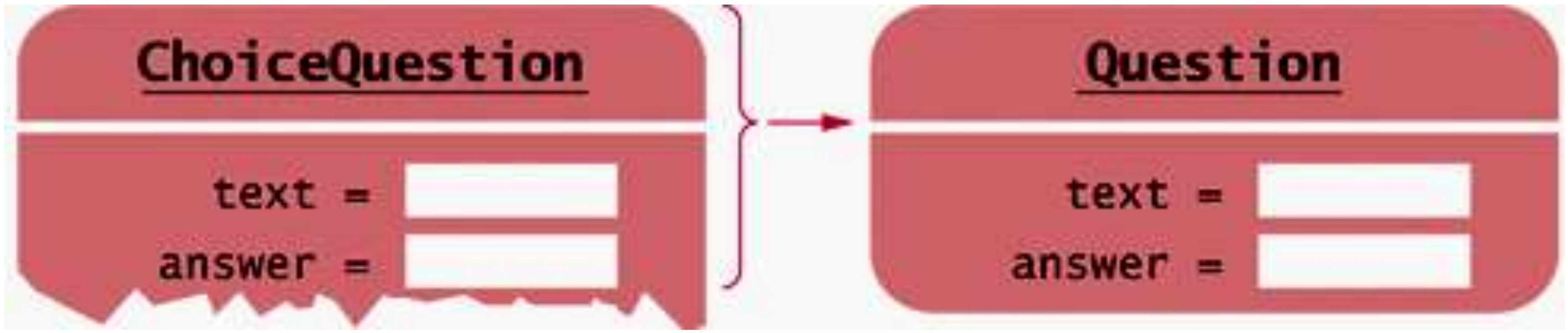
The Slicing Problem



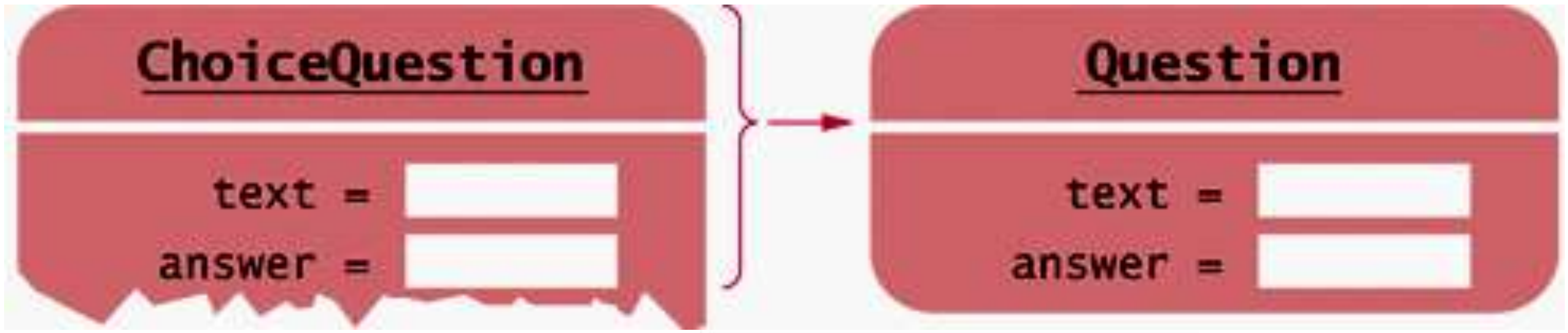
The Slicing Problem



The Slicing Problem

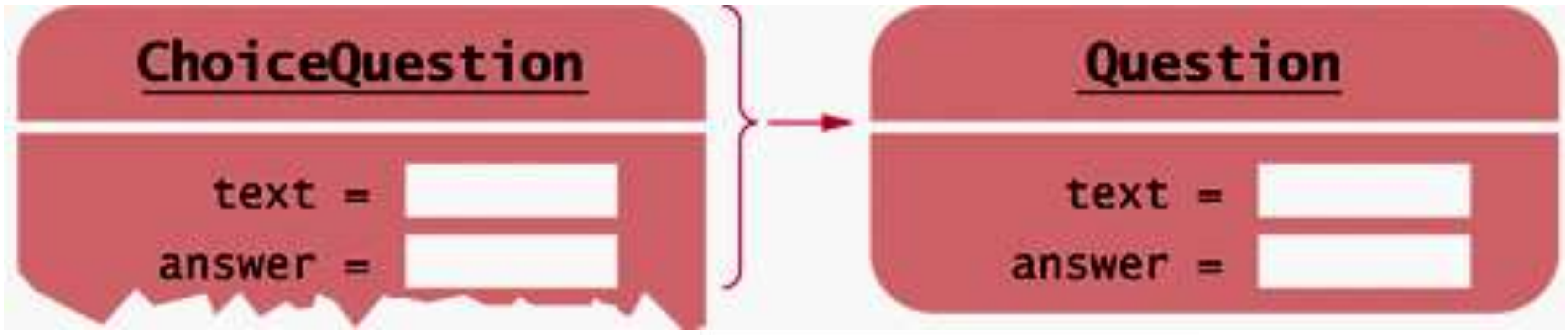


The Slicing Problem

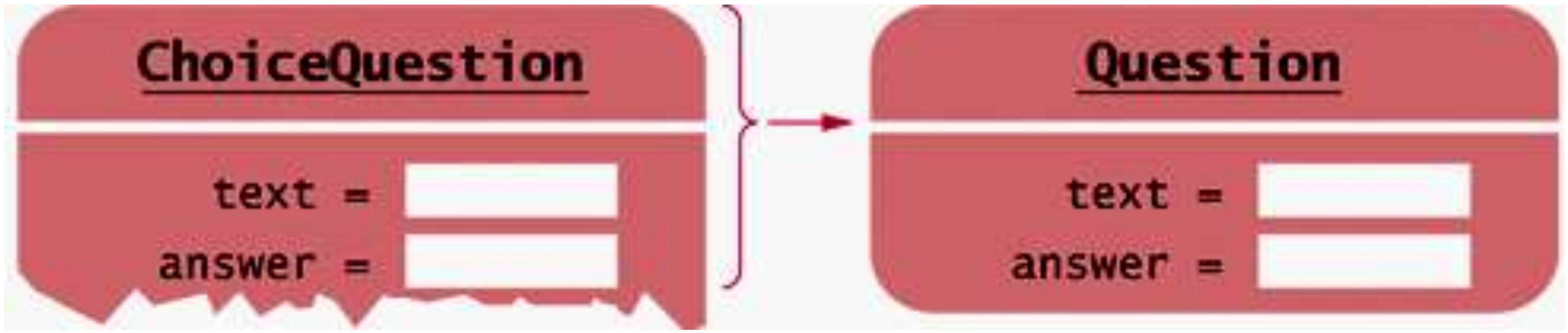


SLICED OFF!

The Slicing Problem



The Slicing Problem



The assignment copies *only* the base portion.

Pointers to Base and Derived Classes

To access objects from *different* classes
in a class hierarchy, use pointers

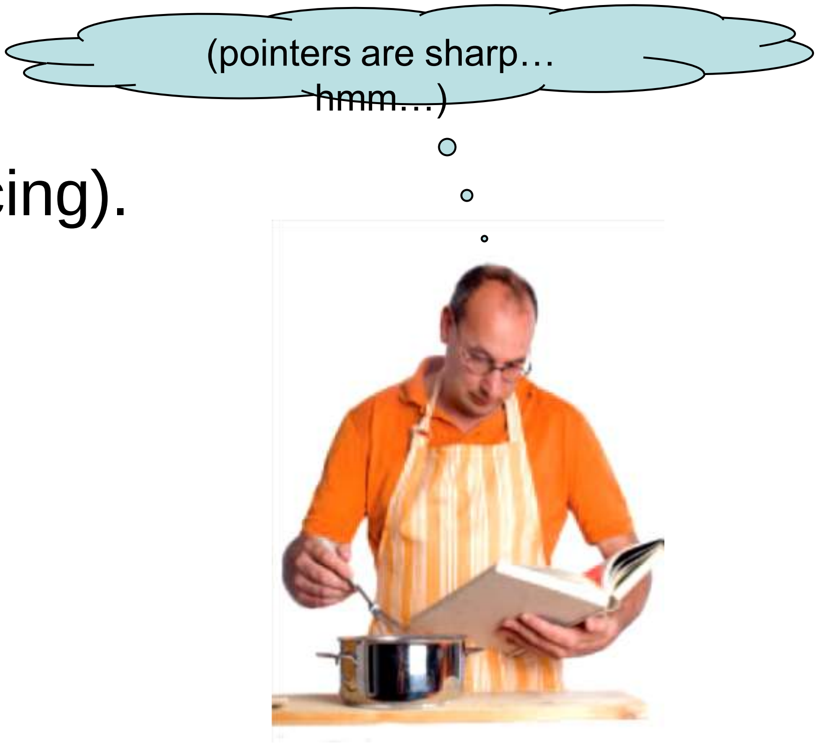
(to avoid slicing).

I'm putting the
knife away… …forever

Pointers to Base and Derived Classes

To access objects from *different* classes
in a class hierarchy, use pointers

(to avoid slicing).



Pointers to Base and Derived Classes

Pointers to the various objects all have the same size:
the size of a memory address.

(convenient)

Pointers to Base and Derived Classes

Pointers to base classes can hold pointers to *ANY* object publicly derived from it

– as far down the inheritance chain as you want to go.

(very general)

Pointers to Base and Derived Classes

The opposite will not work:

Assigning a base pointer to a derived pointer location will generate a compiler error.

(pointer well taken)

(make that: point well taken)

Pointers to base and Derived Classes

To manage a set of *ANY* **`Questions`**

including any IS-A **`Question`**s:

- Fill-in-the-blank
- Numeric
- Free response
- Choice (single)

Pointers to base and Derived Classes

To manage a set of *ANY* Questions

and *Questions inheriting from* IS-A Questions

- Fill-in-the-blank
- Numeric
- Free response
- Choice (single)
  - Choice (multiple – inherits from Choice (single)

Pointers to base and Derived Classes

… and `Questions` that are just

a gleam in a programmer's eye

- Fill-in-the-blank
- Numeric

  Free response
- Choice (single)
  - Choice (multiple – inherits from Choice (single)
- Essay
  - Essay written longhand
    - in pomegranate juice

Pointers to base and Derived Classes

To manage all of these, use a **`vector<Question*>`**
of an array of **`Question*`** and store
only pointers to the different kinds of **`Question`**s

- Fill-in-the-blank
- Numeric
  Free response
- Choice (single)
  - Choice (multiple – inherits from Choice (single)
- Essay
  - Essay written longhand
    - in pomegranate juice

Code for Pointers to base and Derived Classes

Notice the use of **`new`** and **`->`** :

```
Question* quiz[2];

quiz[0] = new Question;
quiz[0]->set_text("Who was the inventor of C++?");
quiz[0]->set_answer("Bjarne Stroustrup");

ChoiceQuestion* cq_pointer = new ChoiceQuestion;
cq_pointer->set_text("In which country… …C++ born?");
cq_pointer->add_choice("Australia", false);
...
quiz[1] = cq_pointer;
```

Code for Pointers to base and Derived Classes

And the assignment of a derived-class pointer to a base:

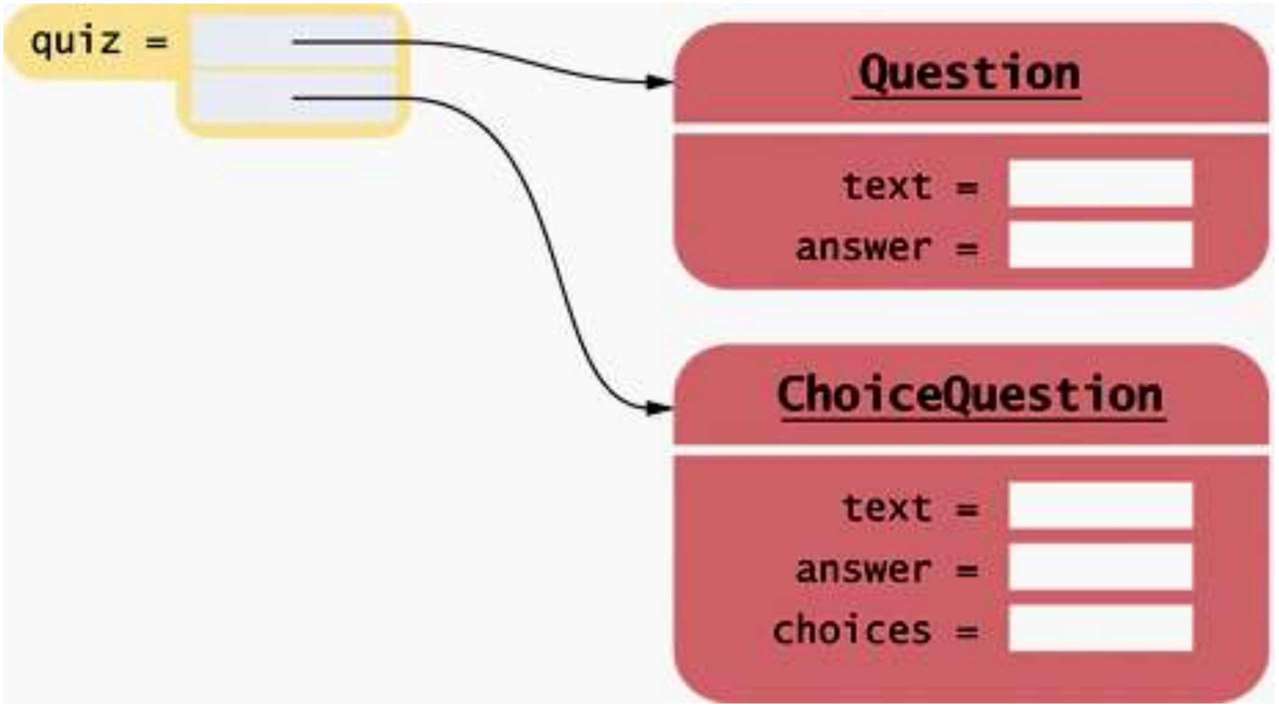
```
Question* quiz[2];


quiz[0] = new Question;
quiz[0] -> set_text("Who was the inventor of C++?");
quiz[0] -> set_answer("Bjarne Stroustrup");

ChoiceQuestion* cq_pointer = new ChoiceQuestion;
cq_pointer -> set_text("In which country… …C++ born?");
cq_pointer -> add_choice("Australia", false);
...
quiz[1] = cq_pointer;
```

Vector of Pointers to Base to Manage Base and Derived

```
Question* quiz[2];
quiz[0] = new Question;
quiz[1] = new ChoiceQuestion;
```



Using `vector<Base*>`

The code to present all questions – any kind of **`Question`** – is:

```
for (int i = 0; i < QUIZZES; i++)
{
   quiz[i]->display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << quiz[i]->check_answer(response) << endl;
}
```

Virtual Functions

(take a deep breath)

When you call the **`display`** member function on a **`Question*`** pointer that currently contains a pointer to a **`ChoiceQuestion`**, you want the **`choices`** to be displayed,

right?

(*Whew* … yes.)

Virtual Functions

But that’s not what happens.

(Oh dear)

Virtual Functions

For reasons of efficiency, by default, the call

```
quiz[i]->display();
```

always calls **`Question::display`**
because the type of **`quiz[i]`** is **`Question*`**.

(Do we give up hope now and go home?)

Virtual Functions

In C++, you must alert the compiler
that the function call needs to *not* be the default,

that the function should be the one *in* the thing pointed to.

(How?)

You use the **`virtual`** reserved word for this purpose.

Virtual Functions

Notice where **`virtual`** is written.

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

The **`virtual`** reserved word must be used in the base class.

Virtual Functions

All functions with the same name and parameter types in derived classes are then automatically virtual.

Virtual Functions

Although not needed, it is considered good practice to write the **`virtual`** reserved word in the derived-class functions.

```
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   virtual void display() const;
private:
...
};
```

Virtual Functions

You do not write **`virtual`** in the function definition:

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

Virtual Functions

With virtual member functions,

```
quiz[i]->display();
```

what you call is what you get!

(Got it!)

Polymorphism

The quiz vector collects a mixture of all kinds of `Question`s.

Such a collection is called *polymorphic*
(literally, “of multiple shapes”).

Polymorphism

Objects in a polymorphic collection have some commonality but are not necessarily of the same type.

Inheritance is used to express this commonality.

`virtual` functions enable variations in behavior.

Polymorphism

Virtual functions give programs a great deal of flexibility.

The question presentation loop describes
only the *general* mechanism:

“Display the question, get a response, and check it”.

Polymorphism

Each object knows
on its own
how to carry out *its* specific version
of these general tasks:

“Display the question”
(my way)
and
“Check a response”
(my way).

Polymorphism

Using virtual functions makes programs easily extensible.

Polymorphism

Suppose we want to have a new
kind of question for calculations,
where we are willing to accept an approximate answer.

All we need to do is to define a new class **`NumericQuestion`**,
with its own **`check_answer`** function.

Polymorphism

Then we can populate any quiz vector with a mixture of plain questions, choice questions,

and these new numeric questions.

They will fit in just fine because:

they IS-A Questions.

(That’s so nice.)

Polymorphism

The code that presents the questions need not be changed at all!

The calls to the `virtual` functions automatically select the correct member functions of the newly defined classes.

The Complete Program with Polymorphism

ch10/quiz3/question.h

```
#ifndef QUESTION_H
#define QUESTION_H

#include <string>
using namespace std;

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

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

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

The Complete Program with Polymorphism

ch10/quiz3/question.h

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

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

private:
   string text;
   string answer;
};

#endif
```

The Complete Program with Polymorphism

ch10/quiz3/choicequestion.h

```
#ifndef CHOICEQUESTION_H
#define CHOICEQUESTION_H

#include <vector>
#include "question.h"

class ChoiceQuestion : public Question
{
public:
   /**
      Constructs a choice question with no choices.
   */
   ChoiceQuestion();
   /**
      Adds an answer choice to this question.
      @param choice the choice to add
      @param correct true if this is the correct choice,
      false otherwise
   */
   void add_choice(string choice, bool correct);
```

The Complete Program with Polymorphism

ch10/quiz3/choicequestion.h

```
   virtual void display() const;

private:
   vector<string> choices;
};
#endif
```

The Complete Program with Polymorphism

ch10/quiz3/test.cpp

```
#include <iostream>
#include "question.h"
#include "choicequestion.h"
int main()
{
   string response;
   cout << boolalpha;

   // Make a quiz with two questions
   const int QUIZZES = 2;
   Question* quiz[QUIZZES];

   quiz[0] = new Question;
   quiz[0]->set_text("Who was the inventor of C++?");
   quiz[0]->set_answer("Bjarne Stroustrup");
```

The Complete Program with Polymorphism

ch10/quiz3/test.cpp

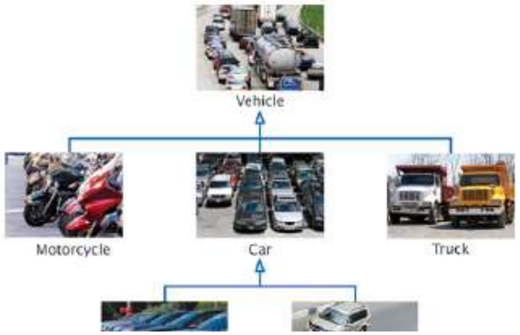
```
   ChoiceQuestion* cq_pointer = new ChoiceQuestion;
   cq_pointer->set_text("In which country… …C++ born?");
   cq_pointer->add_choice("Australia", false);
   cq_pointer->add_choice("Denmark", true);
   cq_pointer->add_choice("Korea", false);
   cq_pointer->add_choice("United States", false);
   quiz[1] = cq_pointer;

   // Check answers for all questions
   for (int i = 0; i < QUIZZES; i++)
   {
      quiz[i]->display();
      cout << "Your answer: ";
      getline(cin, response);
      cout << quiz[i]->check_answer(response) << endl;
   }
   return 0;
}
```

CHAPTER SUMMARY

Explain the notions of inheritance, base class, and derived class.

- A derived class inherits data and behavior from a base class.
- You can always use a derived-class object in place of a base-class object.



Implement derived classes in C++.

- A derived class can override a base-class function by providing a new implementation.
- The derived class inherits all data members and all functions that it does not override.
- Unless specified otherwise, the base-class data members are initialized with the default constructor.
- The constructor of a derived class can supply arguments to a base-class constructor.

CHAPTER SUMMARY

Describe how a derived class can override functions from a base class.

- A derived class can inherit a function from the base class, or it can override it by providing another implementation.
- Use *BaseClass*::*function* notation to explicitly call a base-class function.

Describe virtual functions and polymorphism.

- When converting a derived-class object to a base class, the derived-class data is sliced away.
- A derived-class pointer can be converted to a base-class pointer.
- When a virtual function is called, the version belonging to the actual type of the implicit parameter is invoked.
- Polymorphism (literally, "having multiple shapes") describes objects that share a set of tasks and execute them in different ways.

End Chapter Ten: Inheritance, Part II