Chapter Nine: Classes, Part I

C++ for Everyone by Cay Horstmann

Slides by Evan Gallagher & Nikolay Kirov

Chapter Goals

- To understand the concept of encapsulation
- To master the separation of interface and implementation
- To be able to implement your own classes

Object-Oriented Programming

I thought you considered me more
than just a collection of parts.

I’m more than just *function*al.

Am I just an ***object*** to you?

Object-Oriented Programming

I have an onboard computer

– now will you love me for what I am?

Object-Oriented Programming

Did you know that you already are

an

Object Oriented Programmer?

(No way!)

Object-Oriented Programming

Does **`string`** sound familiar?

(Yes…)

Object-Oriented Programming

Does **`string`** sound familiar?

How about **`cin`** and **`cout`**?

(Yes, but...)

An

Object Oriented Programmer

uses objects.

(Wow, I didn't realize…)

But…

a <u>REAL</u>

Object Oriented Programmer

designs and *creates* objects

and then uses them.

Yes, you are mostly

A Programmer Who Writes Functions

To Solve Sub-problems

And that is very good!

As programs get larger,
it becomes increasingly difficult
to maintain a large collection of functions.

It often becomes necessary to use
the *dreaded and deadly* practice of

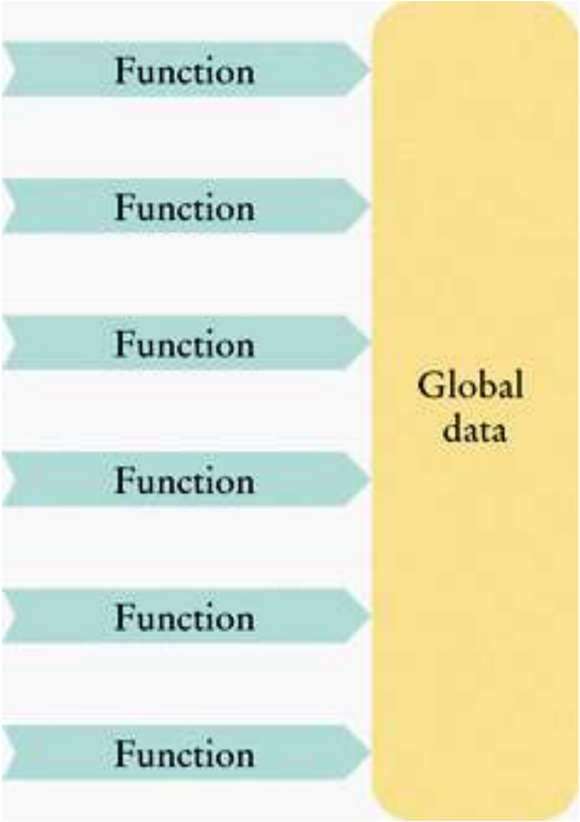
USING GLOBAL VARAIBLES

(**Don't do it, son!**)

Object-Oriented Programming

Global variables are
those defined outside of all
functions – so all functions
have access to them.

But…



Object-Oriented Programming

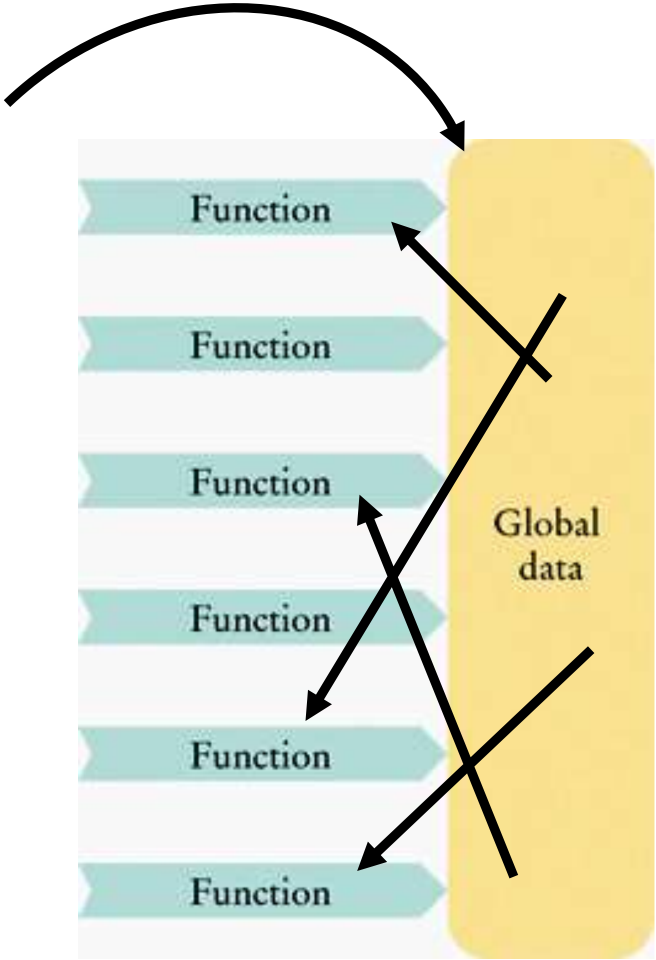
When some part of the global data
needs to be changed:

to improve performance
or to add new capabilities,

a large number of functions
may be affected

– you will have to *rewrite* them –

and *hope* everything still works!



Object-Oriented Programming

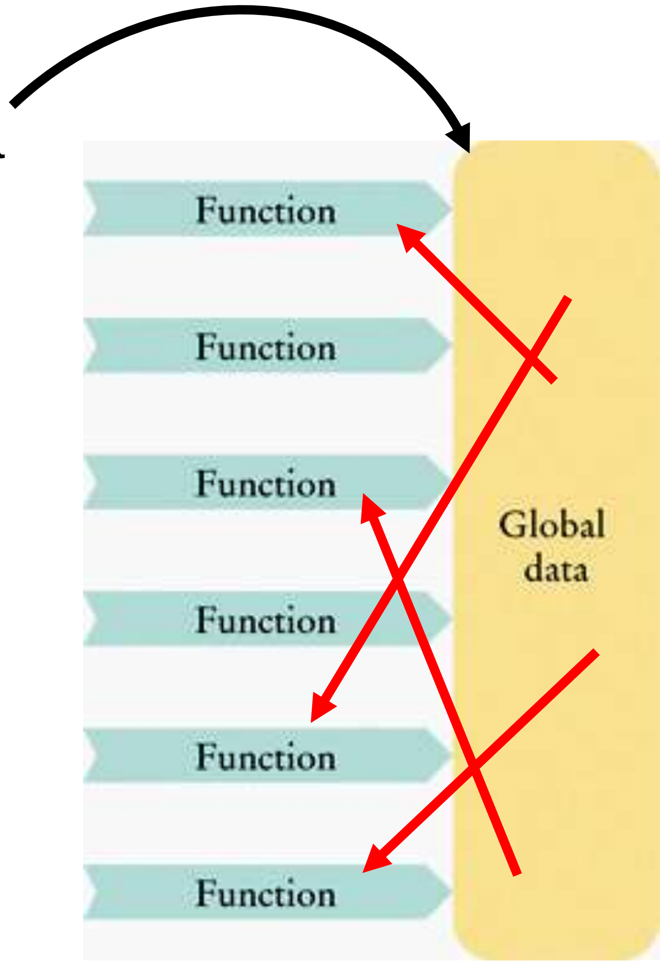
When some part of the global data needs to be changed:

to improve performance
or to add new capabilities,

a large number of functions may be affected

– you will have to *rewrite* them –

and *hope* everything still works!



Ouch!

Objects to the Rescue

Computer scientists noticed that most often functions were working on related data so they invented:

Objects

where **they keep the data and the functions that work with them together.**

No more global variables – *Hurray!*

Objects to the Rescue

objects

Object Oriented Programming

(OOP)

(Not to be confused with oops!, the exclamation.)

Objects to the Rescue

Some new terminology.

The data stored in an object are called:

data members

The functions that work on data members are:

member functions

No more variables and functions –
separately.

Objects to the Rescue

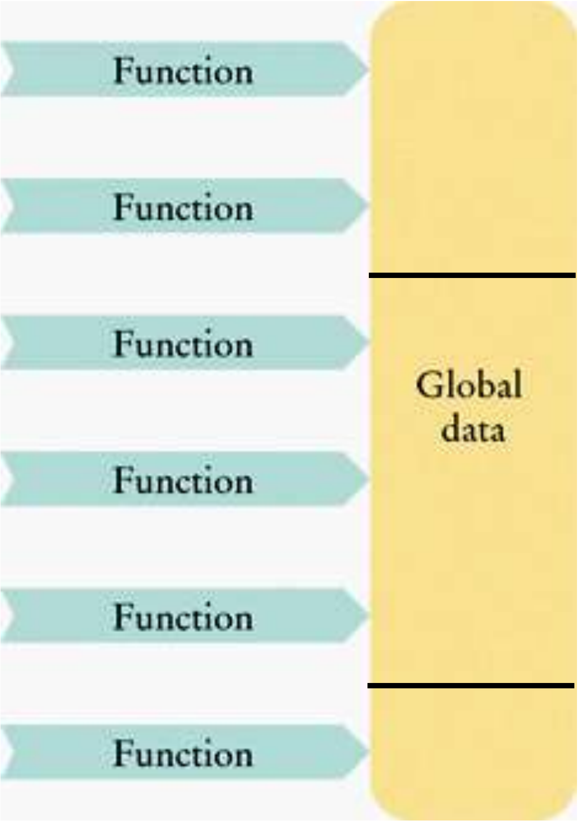
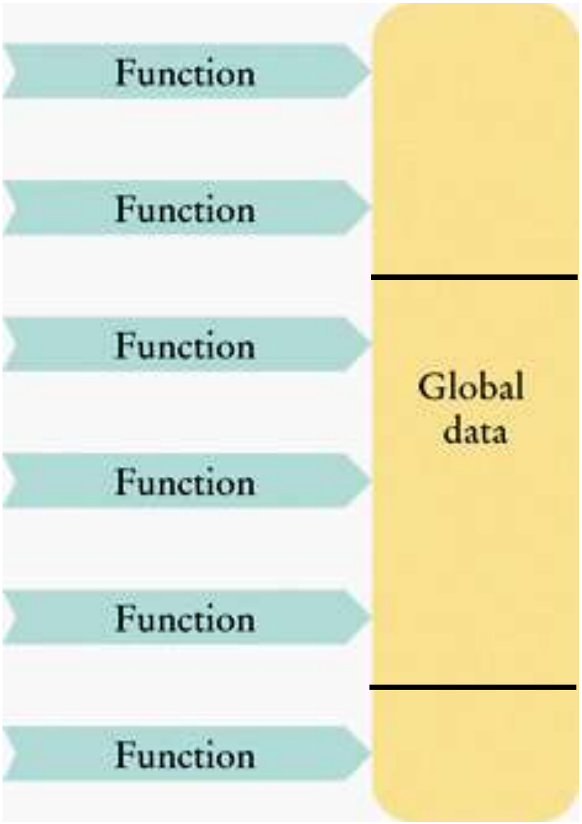


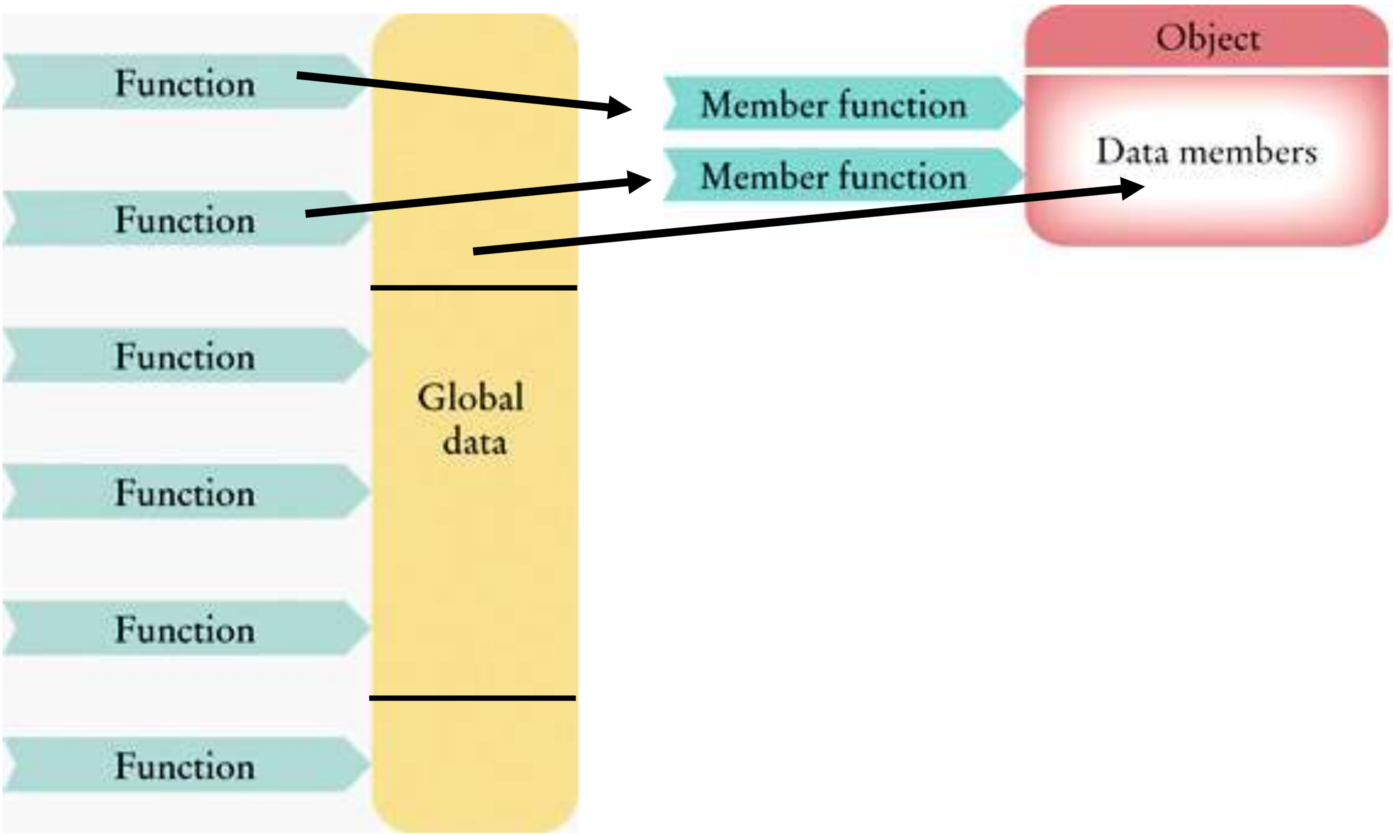
Figure out which functions go with which data.

Objects to the Rescue



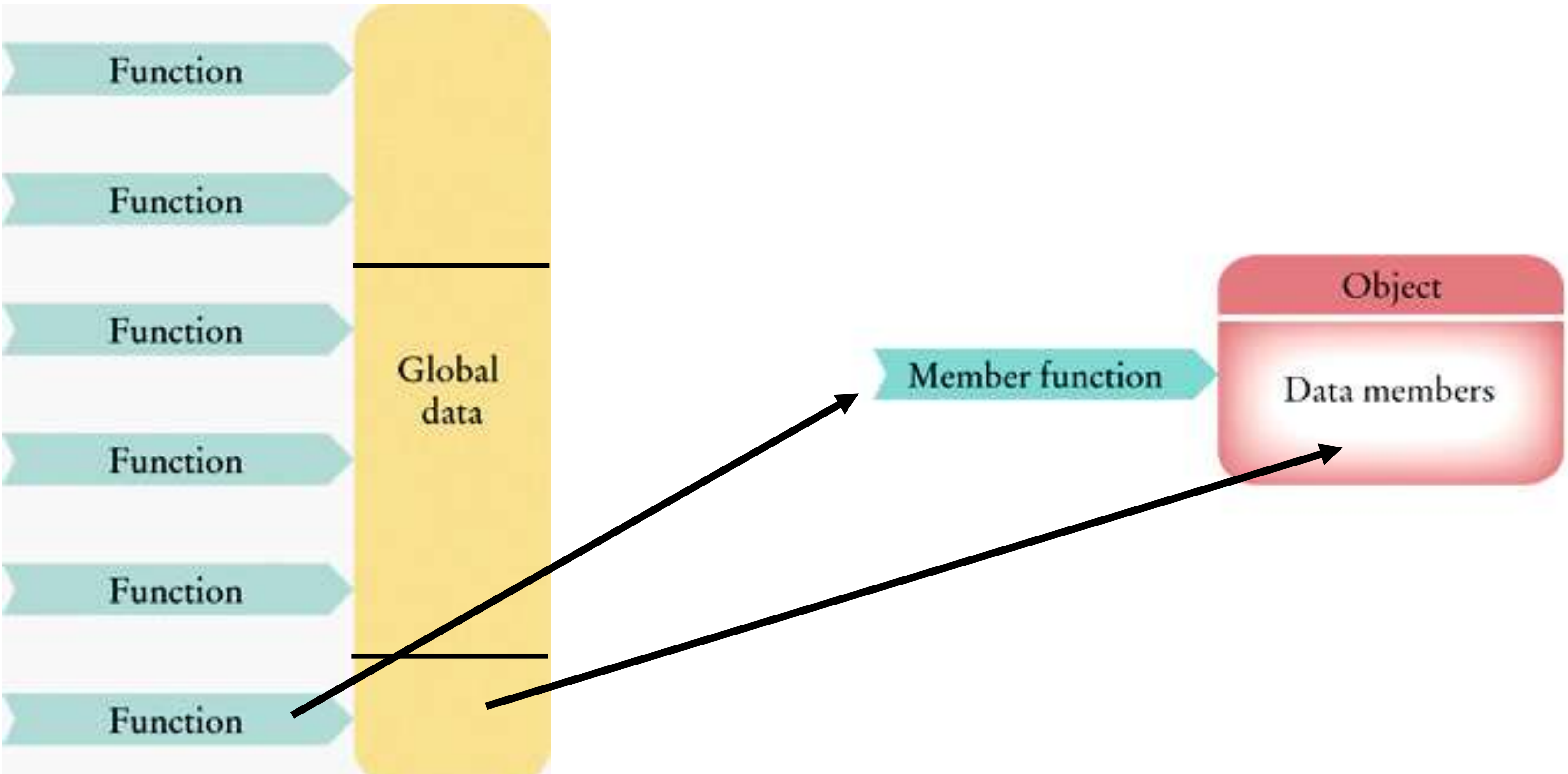
Create an object for each set of data.

Objects to the Rescue



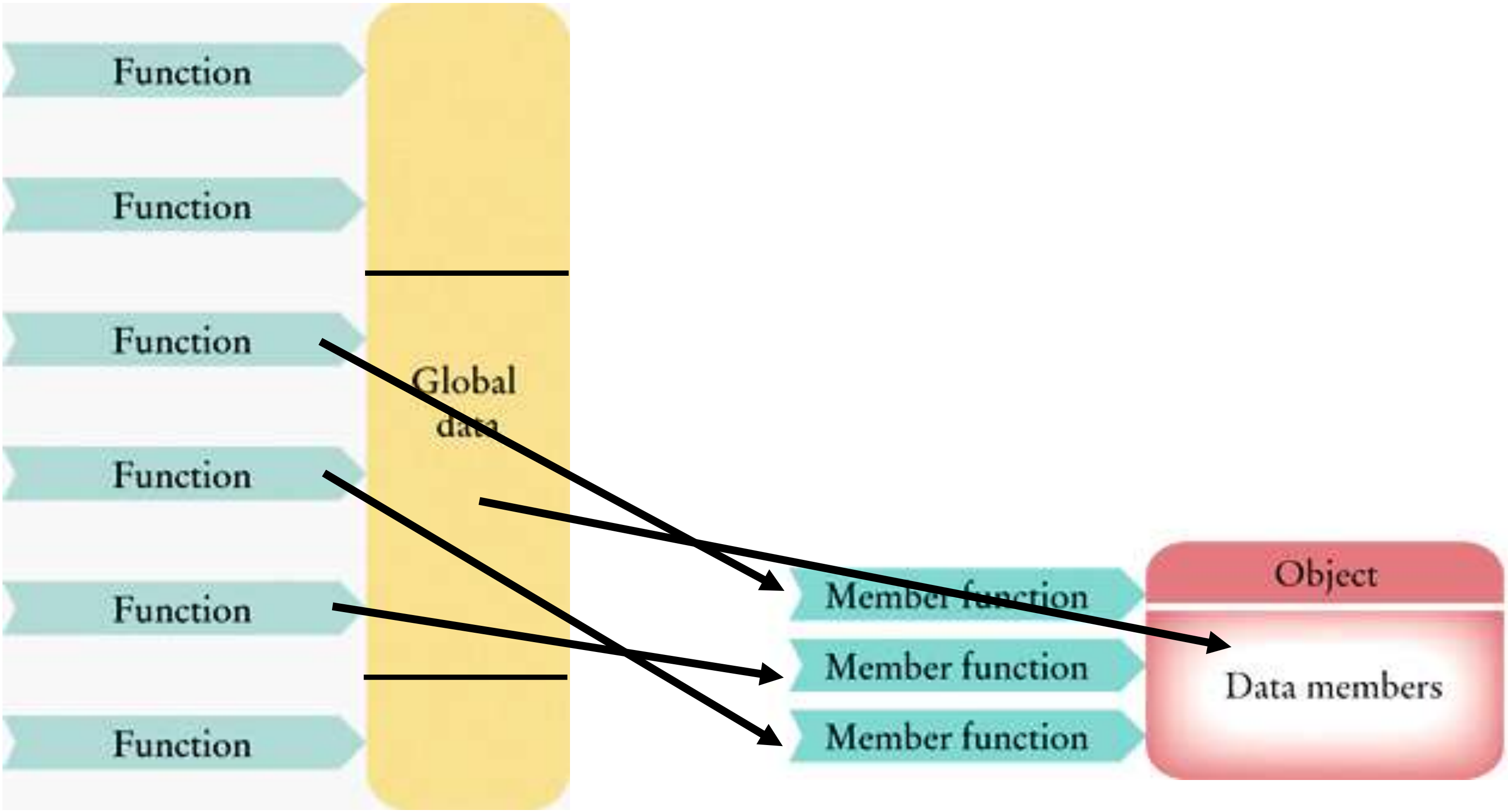
Create another object for another set.

Objects to the Rescue



And now, a third object.

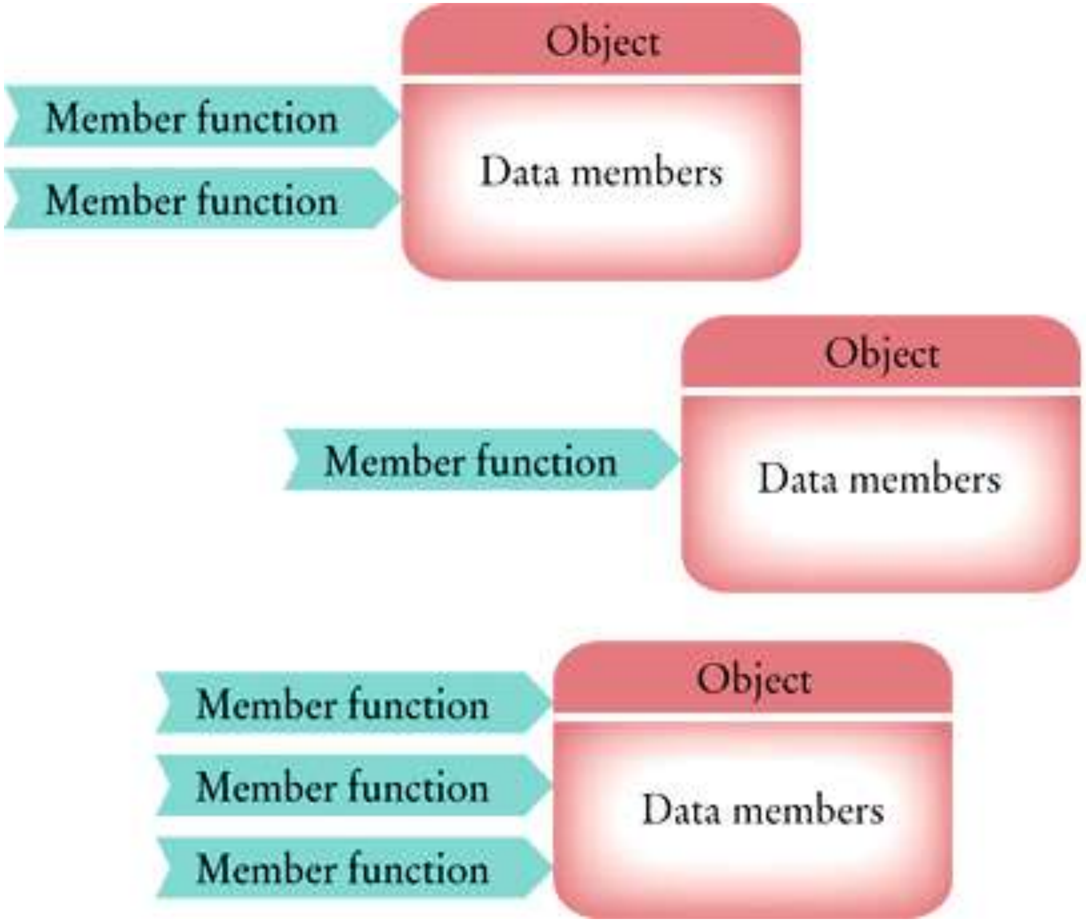
Objects to the Rescue



Objects to the Rescue

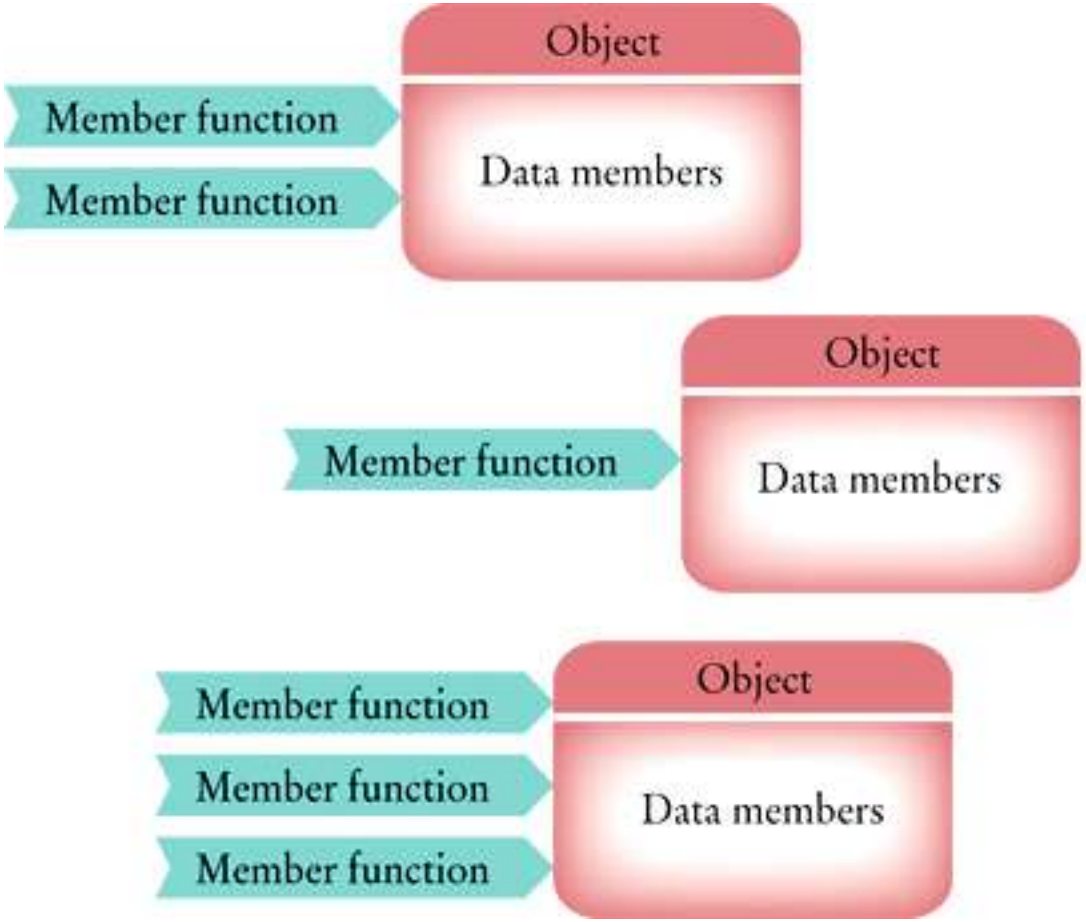
Get rid of those global variables.

Objects to the Rescue



From now on, we'll have only objects.

Objects to the Rescue



Ah.

Encapsulation

The data members are

encapsulated

They are **hidden** from other parts of the program and **accessible** only through their own member functions.

Encapsulation

Now when we want to change the way
that an object is implemented,
only a small number of
functions need to be changed,

and they are the ones *in* the object.

Encapsulation

Because most real-world programs need
to be updated often during their lifetime,
this is an important **advantage**
of object-oriented programming.

Program evolution becomes much more manageable.

(Ah …)

Encapsulation

When you use `string` or `stream` objects,

you did not know their data members.

Encapsulation means that they are hidden from you.

(That's good – you might have messed them up.)

Encapsulation and the Interface

But you were allowed to call member functions such as **`substr`**,

and you could use operators such as

`[]` (subscript for **`vector`**s)

or

`>>` (input stream for objects of **`ifstream`** class)

(which are actually functions).

You were given an *interface* to the object.

Encapsulation and the Interface

All those member functions and operators *are* the interface to the object.

Encapsulation and the Interface

I wonder how the engine really works,
and the speedometer,
and the gas gauge,
and that little thingy over there…

Encapsulation and the Interface

And I better stop thinking about all this
or I won’t be able to drive!

Object-Oriented Programming

So you like my interface.

Don't get me started…

Classes

In C++, a programmer doesn’t implement a single object.

Instead, the programmer implements a *class*.

Classes

A class describes a set of objects with the same behavior.

You would create the **`Car`** class to represent cars as objects.

Object-Oriented Programming

*An **object** ?!*

Defining Classes

To define a class,

you must specify the *behavior*

by providing implementations for the *member functions*,

and by defining the *data members* for the objects …

Defining Classes

Oops! (the exclamation),

I’m a little early – sorry.

I, *camel*, will be *back* later.

Classes

Again, to define a class:

- Implement the member functions to specify the behavior.
- Define the data members to hold the object's data.

Designing the Class

We will design a cash register object.

Designing the Class

By observing a real cashier working,
we realize our cash register design needs
member functions to do the following:

- Clear the cash register to start a new sale.
- Add the price of an item.
- Get the total amount owed and the count of items purchased.

These activities will be our *public interface*.

The public interface is specified by *declarations* in the class definition.

The data members are defined there also.

Classes

To define a class you write:

```
class NameOfClass
{
public:
   // the public interface
private:
   // the data members
};
```

Classes

Any part of the program should be able to call the member functions – so they are in the *public section*.

```
class NameOfClass
{
public:
    // the public interface
private:
    // the data members
};
```

Data members are defined in the *private section* of the class. **Only member functions of the class can access them.** They are hidden from the rest of the program.

Classes

Here is the C++ syntax for the **`CashRegister`** class definition:

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Classes

The public interface has the four activities that we decided this object should support.

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Classes

Notice that these are declarations.

They will be defined later.

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Defining Classes

Hi.

I'm back.

I'm here for *style* purposes.

Defining Classes

Remember, earlier, when I said, “I, *camel*, will be *back* later.”

That was to help you with the style for class names:

CAMEL BACK – well, actually it’s CAMEL CASE.

Defining Classes

I personally think CAMEL BACK is
more *stylish* than CAMEL CASE.

And I consider *my* style to be immaculate.

Just look… at *me* of course:

Defining Classes

Look at my head and my humps.
(*Very* cute!)

That's how your class names should look:

Each "word" should start with an uppercase letter.
(*Very* good style!)

Defining Classes

What should you choose for the name of the class to represent me?

Defining Classes

```
class TwoHumpCamel
{
```

```
};
```

Defining Classes

I'll be going now…

Defining Classes

but don’t forget: class names should be…

Defining Classes

…CAMEL CASE

Methods

There are two kinds of member functions:

- *Mutators*
- *Accessors*

Mutators

A mutator is a function that
modifies
the data members of the object.

Mutators

`CashRegister` has two mutators: `clear`

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Mutators

`CashRegister` has two mutators: **`clear`** and **`add_item`**.

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

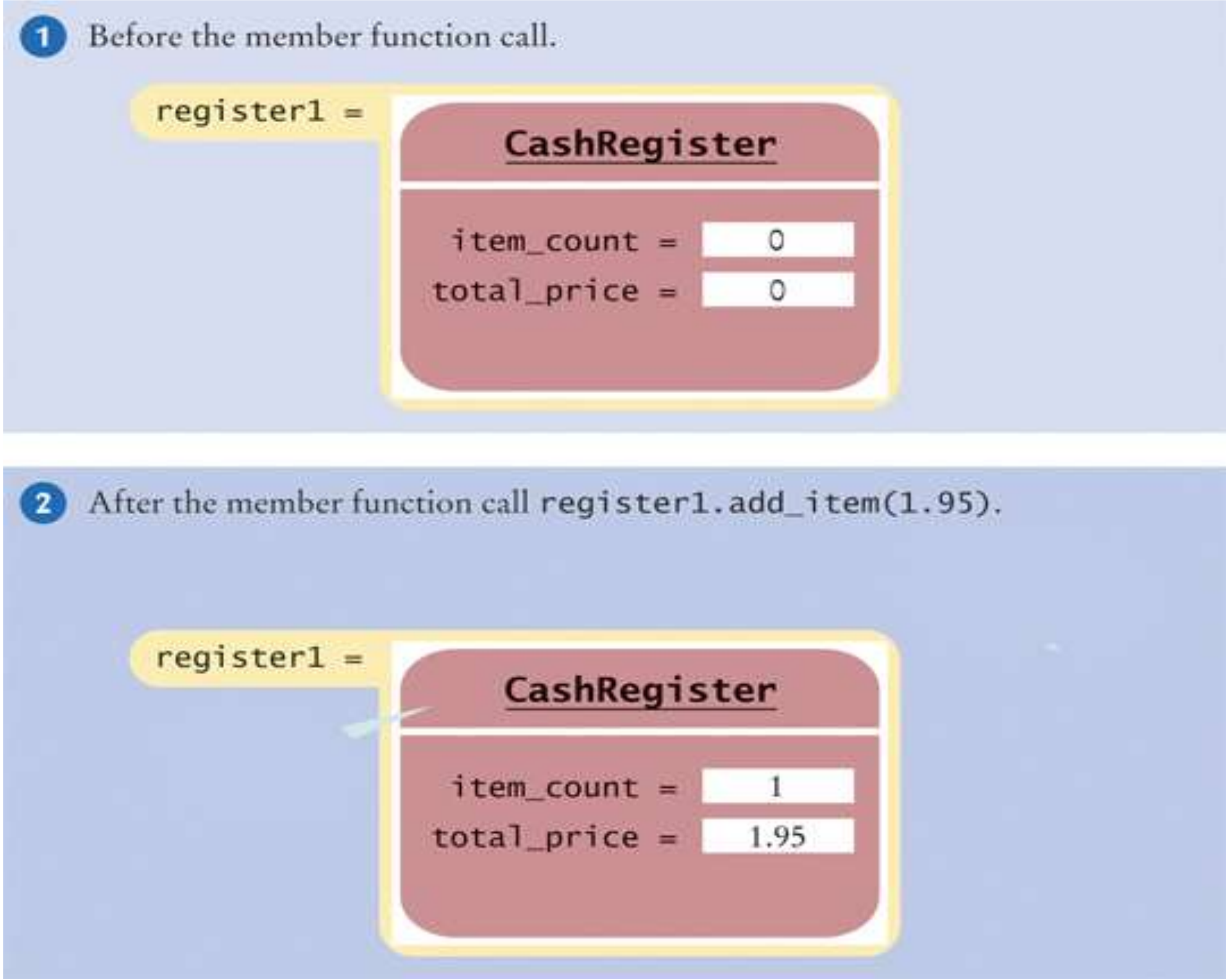
Mutators

You call the member functions by first creating a variable of type **`CashRegister`** and then using the dot notation:

```
CashRegister register1;
...
register1.clear();
...
register1.add_item(1.95);
```

Because these are mutators, the data stored in the class will be changed.

Mutators



1 Before the member function call.
register1 =
CashRegister
item_count = 0
total_price = 0
2 After the member function call register1.add_item(1.95).
register1 =
CashRegister
item_count = 1
total_price = 1.95

Accessors

An accessor is a function that
queries
a data member of the object.

It returns the value of a data member to its caller.

Accessors

`CashRegister` has two accessors: **`get_total`**

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Accessors

`CashRegister` has two accessors: **`get_total`** and **`get_count`**.

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   // data members will go here
};
```

Accessors

Because accessors should never change the data in an object, you should make them `const`.

```
class CashRegister
{
public:
    void clear();
    void add_item(double price);
    double get_total() const;
    int get_count() const;
private:
    // data members will go here
};
```

not here

Accessors

This statement will print the current total:

```
cout << register1.get_total() << endl;
```

Mutators and Accessors: The Interface

The interface for our class:

Common Error:

Can you find the error?

```
class MysteryClass
{
public:
    ...
private:
    ...
}
int main()
{


    ...
}
```

Common Error: Missing Semicolon

Don't forget that semicolon!

```
class MysteryClass
{
public:
    ...
private:
    ...
}        // Forgot semicolon
int main()
{
    // Many compilers report
    // that error here in main!
    ...
}
```

Encapsulation

Let's continue with the design of **`CashRegister`**.

Each **`CashRegister`** object must store the total price and item count of the sale that is currently rung up.

We have to choose an appropriate data representation.

Encapsulation

This is pretty easy:

Encapsulation

`item_count` for the count

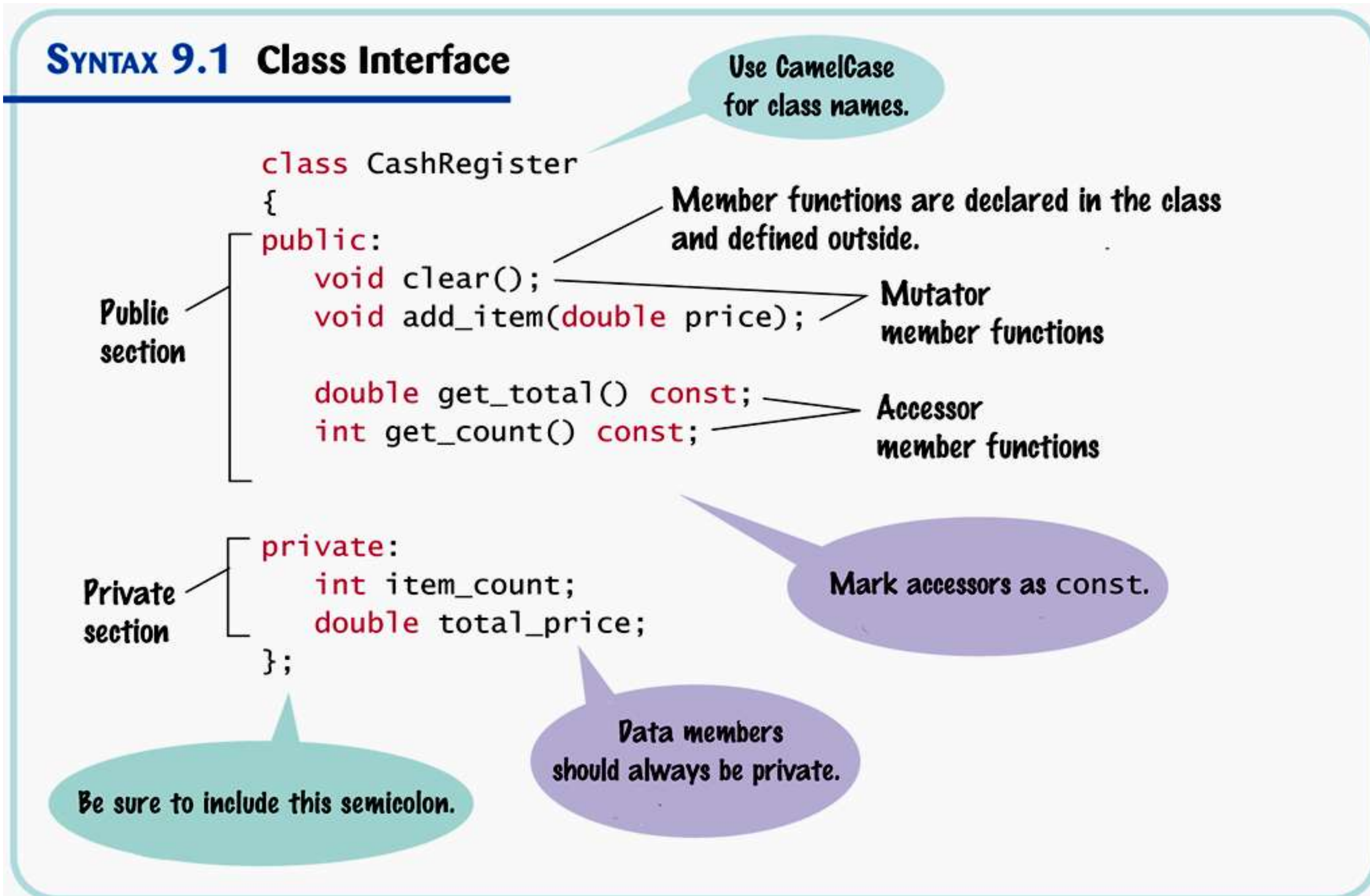
```
class CashRegister
{
public:
   // interface
private:
   int item_count;
   double total_price;
};
```

Encapsulation

`total_price` for the total

```
class CashRegister
{
public:
   // interface
private:
   int item_count;
   double total_price;
};
```

Class Definition Syntax

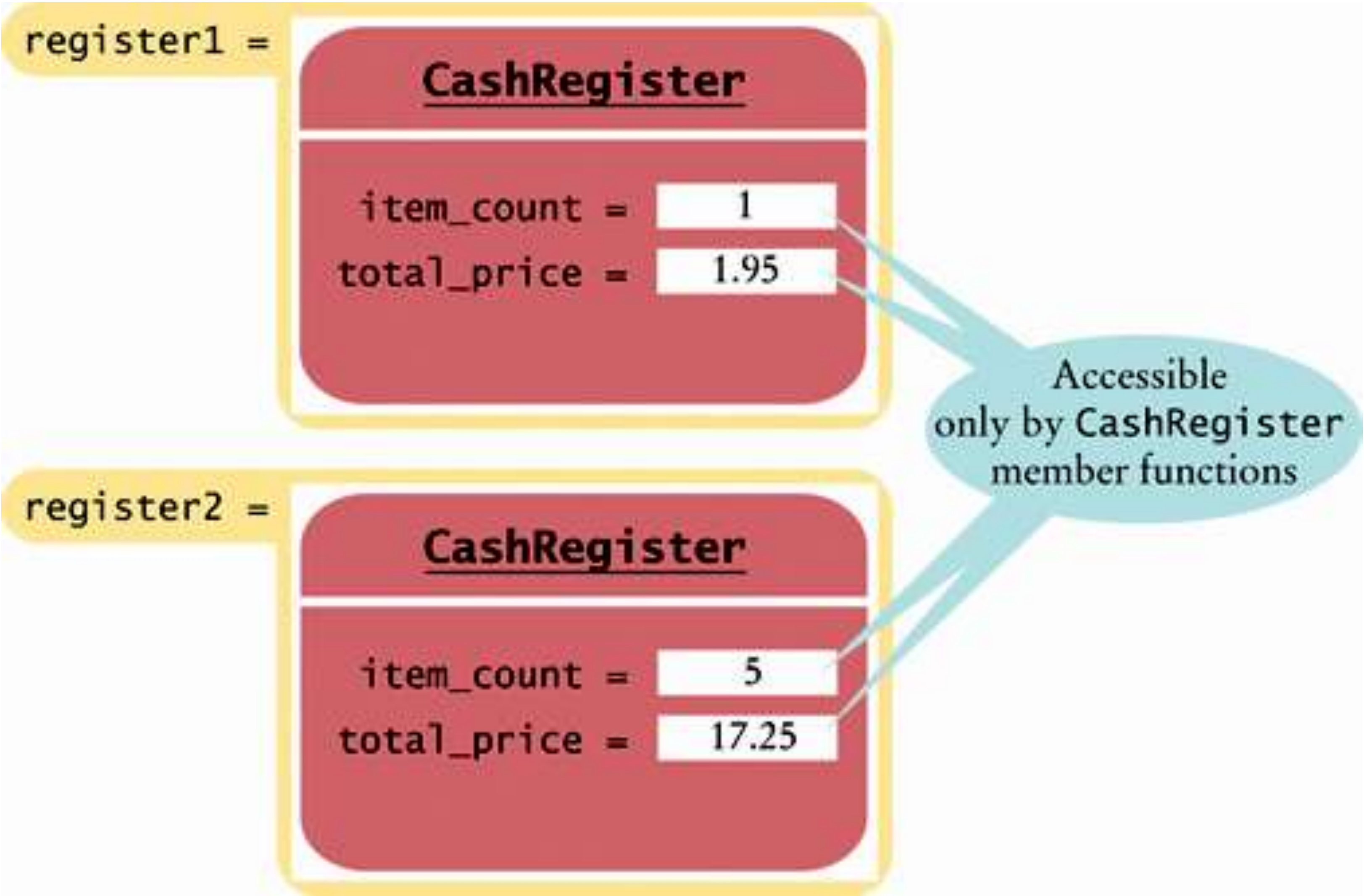


Encapsulation

Every **`CashRegister`** object has
a separate copy of these data members.

```
CashRegister register1;
CashRegister register2;
```

Encapsulation



Encapsulation

Because the data members are private, this won’t compile:

```
int main()
{
    ...
    cout << register1.item_count;
            // Error—use get_count() instead
    ...
}
```

Encapsulation

A good design principle:

Never have any public data members.

Son, consider that an addition to the RULES!

I know you can make data members public,

but don't.

Just don't do it!

Encapsulation and Methods as Guarantees

One benefit of the encapsulation mechanism is

we can make guarantees.

Encapsulation and Methods as Guarantees

We can write the mutator for **`item_count`** so that **`item_count`** cannot be set to a negative value.

If **`item_count`** were pubic, it could be directly set to a negative value by some misguided (or worse, devious) programmer.

Encapsulation and Methods as Guarantees

There is a second benefit of encapsulation that is particularly important in larger programs:

Things Change.

Encapsulation and Methods as Guarantees

Well, that’s not really a benefit.

Things change means:

Implementation details often need to change over time …

Encapsulation and Methods as Guarantees

You want to be able to make your classes more efficient or more capable, without affecting the programmers that use your classes.

The benefit of encapsulation is:

As long as those programmers do not depend on the implementation *details*, you are free to change them at any time.

The Interface

The interface should not change even
if the details of how they are implemented change.

The Interface

A driver switching to an electric car
does not need to relearn how to drive.

Object-Oriented Programming

How dare you compare my interface with that, that…

I'm shocked!

Implementing the Member Functions

Now we have what the interface does,
and what the data members are,
so what is the next step?

Implementing the member functions.

Implementing the Member Functions

The details of the **`add_item`** member function:

```
void add_item(double price)
{
   item_count++;
   total_price = total_price + price;
}
```

Implementing the Member Functions

Unfortunately this is NOT the **`add_item`** member function.

It is a separate function, just like you used to write.

It has no connection with the **`CashRegister`** class

```
void add_item(double price)
{
   item_count++;
   total_price = total_price + price;
}
```

Implementing the Member Functions

To specify that a function is a *member* function
of your class you must write

`CashRegister::`

in front of the member function's name:

Implementing the Member Functions

To specify that a function is a *member* function
of your class you must write

`CashRegister::`

in front of the member function's name:

Not here

```
void CashRegister::add_item(double price)
{
   item_count++;
   total_price = total_price + price;
}
```

Implementing the Member Functions

Use **`CashRegister::`** *only* when defining the function – not in the class definition.

```
class CashRegister
{
public:
    ...
private:
    ...
};
```

Not here

Only here

```
void CashRegister::add_item(double price)
{
    item_count++;
    total_price = total_price + price;
}
```

Implicit Parameters

Wait a minute.

We are changing data members …

BUT THERE'S NO VARIABLE TO BE FOUND!

Which variable is **`add_item`** working on?

Implicit Parameters

Oh No! We've got two cash registers!

`CashRegister register2;`

`CashRegister register1;`

Which cash register is **`add_item`** working on?

Implicit Parameters

When a member function is called:

```
CashRegister register1;
...
register1.add_item(1.95);
```

The variable to the left of the dot operator is *implicitly* passed to the member function.

In the example, **`register1`** is the *implicit parameter*.

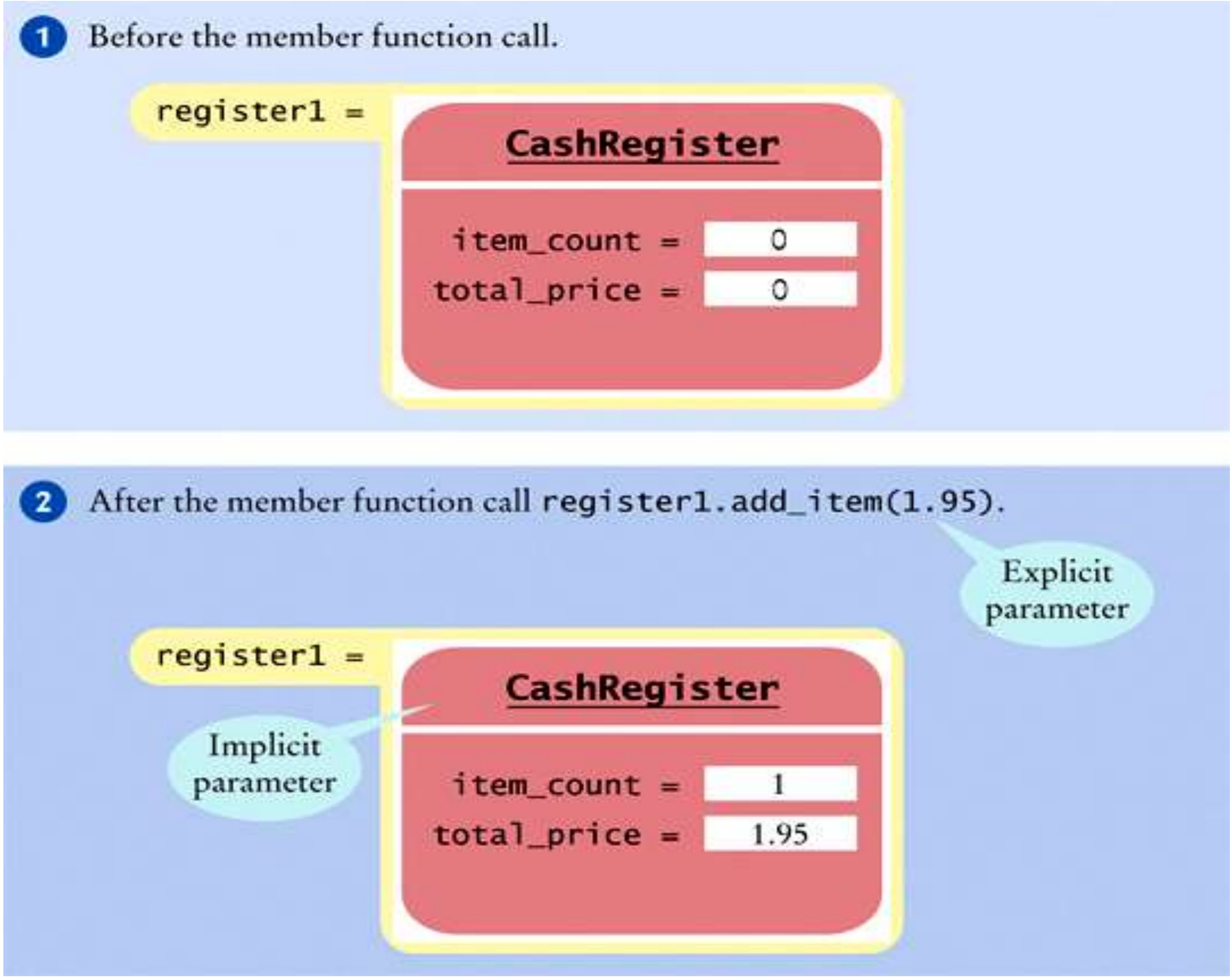
Implicit Parameters

The variable **`register1`** is an *implicit parameter*.

```
register1.add_item(1.95);
```

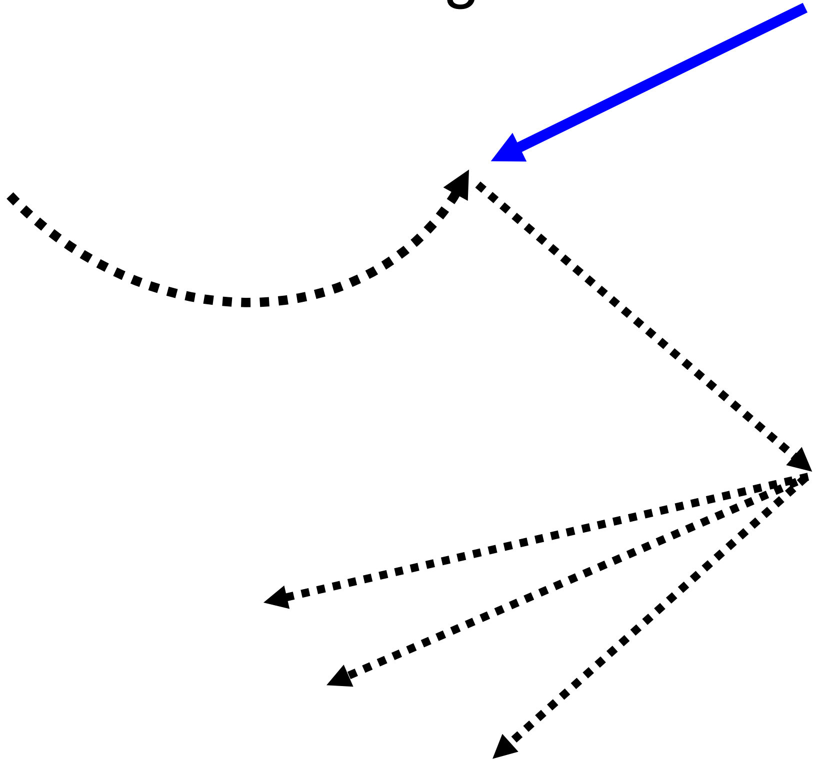
```
void CashRegister::add_item(double price)
{
    implicit parameter.item_count++;
    implicit parameter.total_price =
            implicit parameter.total_price + price;
}
```

Implicit Parameters



Implicit Parameters

We’ll get back to *this*, later …



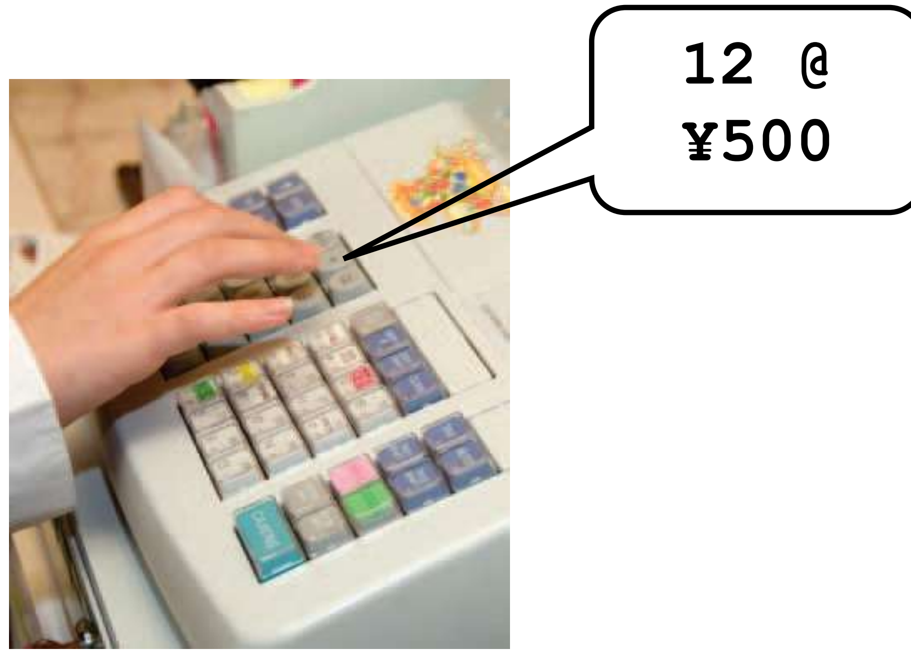
Calling a Member Function from a Member Function

Let’s add a member function that adds
multiple instances of the same item.

Calling a Member Function from a Member Function

Like when we are programming…

and we get a dozen strong, black coffees to go.



Calling a Member Function from a Member Function

We have already written the **`add_item`** member function

and

the same good design principle of

code reuse with functions

is still fresh in our minds, so:

```
void CashRegister::add_items(int qnt, double prc)
{
   for (int i = 1; i <= qnt; i++)
   {
      add_item(prc);
   }
}
```

Calling a Member Function from a Member Function

When one member function calls another member function on the same object, you do ***not*** use the dot notation.

```
void CashRegister::add_items(int qnt, double prc)
{
    for (int i = 1; i <= qnt; i++)
    {
        add_item(prc);
    }
}
```

Calling a Member Function from a Member Function

So how does this work?

Remember our friend: *implicit parameter*!

It's as if it were written to the left of the dot

(which also isn't there)

```
register1.add_items(6,0.95);


void CashRegister::add_items(int qnt, double prc)
{
   for (int i = 1; i <= qnt; i++)
   {
      implicit parameter.add_item(prc);
   }
}
```

Calling a Member Function from a Member Function

Syntax 9.2 Member Function Definition

Use *ClassName*:: before the name of the member function.

Explicit parameter

```
void CashRegister::add_item(double price)
{
   item_count++;
   total_price = total_price + price;
}
```

Data members of the implicit parameter

```
int CashRegister::get_count() const
{
   return item_count;
}
```

Data member of the implicit parameter

Use const for accessor functions.

C++ for Everyone by Cay Horstmann

The Cash Register Program

ch09/registertest1.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;
/**
   A simulated cash register that tracks
   the item count and the total amount due.
*/
class CashRegister
{
public:
```

The Cash Register Program

ch09/registertest1.cpp

```
class CashRegister
{
public:
   /**
      Clears the item count and the total.
   */
   void clear();

   /**
      Adds an item to this cash register.
      @param price the price of this item
   */
   void add_item(double price);
```

The Cash Register Program

ch09/registertest1.cpp

```
    /**
       @return the total amount of the current sale
    */
    double get_total() const;

    /**
       @return the item count of the current sale
    */
    int get_count() const;

private:
    int item_count;
    double total_price;
};
```

The Cash Register Program

ch09/registertest1.cpp

```
void CashRegister::clear()
{
   item_count = 0;
   total_price = 0;
}
void CashRegister::add_item(double price)
{
   item_count++;
   total_price = total_price + price;
}
double CashRegister::get_total() const
{
   return total_price;
}
```

The Cash Register Program

ch09/registertest1.cpp

```
int CashRegister::get_count() const
{
   return item_count;
}

/**
   Displays the item count and total
   price of a cash register.
   @param reg the cash register to display
*/
void display(CashRegister reg)
{
   cout << reg.get_count() << " $"
      << fixed << setprecision(2)
      << reg.get_total() << endl;
}
```

The Cash Register Program

ch09/registertest1.cpp

```
int main()
{
   CashRegister register1;
   register1.clear();
   register1.add_item(1.95);
   display(register1);
   register1.add_item(0.95);
   display(register1);
   register1.add_item(2.50);
   display(register1);
   return 0;
}
```

`const` Correctness

You should declare all accessor functions in C++ with the **`const`** reserved word.

`const` Correctness

But let’s say, just for the sake of checking things out

– you would never do it yourself, of course –

suppose you did not make **`display`** **`const`**:

```
class CashRegister
{
   void display(); // Bad style—no const
   ...
};
```

`const` Correctness

This will compile with no errors.

```
class CashRegister
{
   void display(); // Bad style—no const
   ...
};
```

`const` Correctness

But son, it's not just about you.

`const` Correctness

What happens when some other, well intentioned, good design-thinking programmer uses your class, an array of them actually, in a function.

Very correctly she makes the array **`const`**.

```
void display_all(const CashRegister[] registrs)
{
    for (int i = 0; i < NREGISTERS; i++)
    {
        registrs[i].display();
    }
}
```

`const` Correctness

Son, look what you've done!

`const` Correctness

The compiler (correctly) notices that
`registrs[i].display()`
is calling a NON-CONST **`display`** method
on a CONST **`CashRegister`** object.

```
void display_all(const CashRegister[] registrs)
{
    for (int i = 0; i < NREGISTERS; i++)
    {
        registrs[i].display();
    }
}
```

compiler error

`const` Correctness

Son…

Yes, it's actually her fault for not reading your code closely enough but that is no excuse for your bad behavior.

End Chapter Nine: Classes, Part II

Slides by Evan Gallagher & Nikolay Kirov

C++ for Everyone by Cay Horstmann