BLG 102E
Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 14

ISTANBUL TECHNICAL UNIVERSITY

Classes

Objects to the Rescue

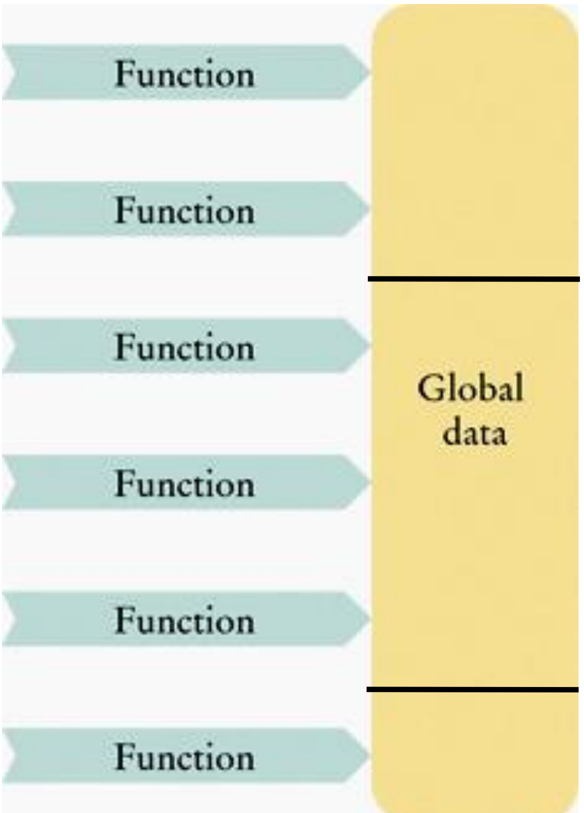
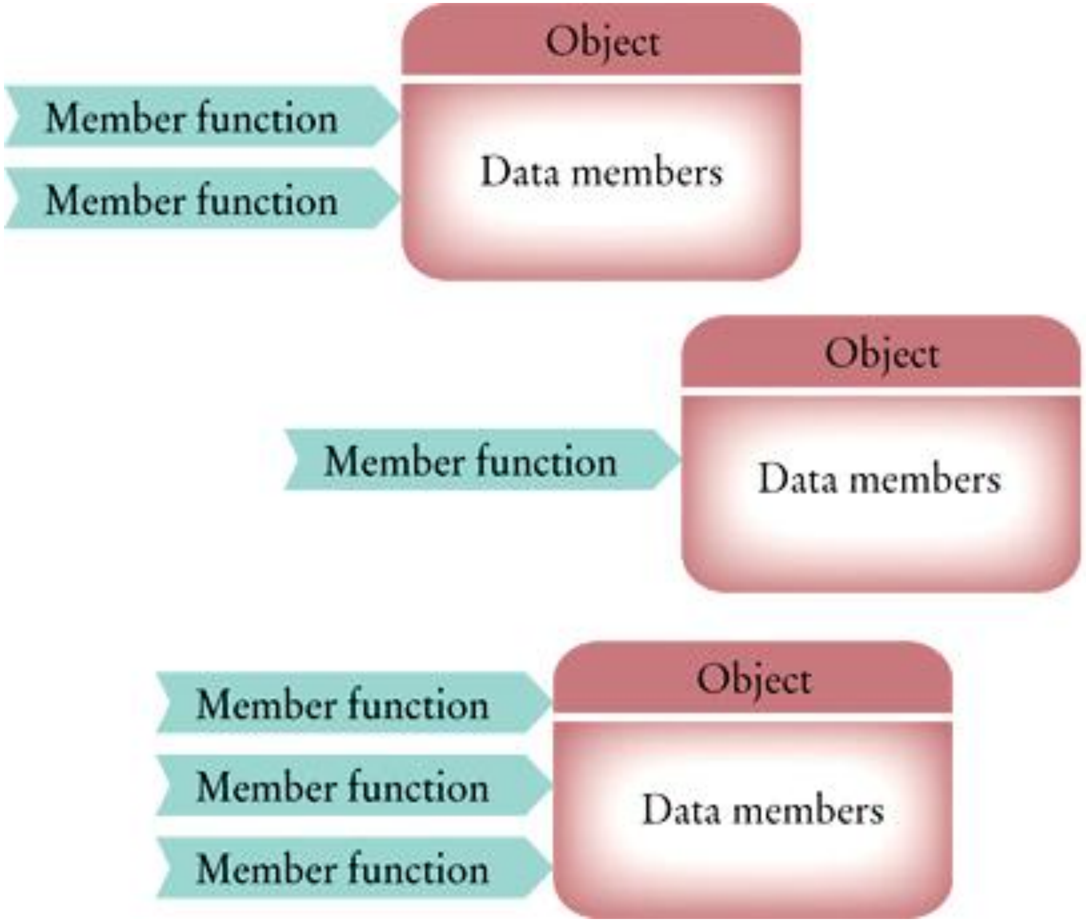


Figure out which functions go with which data.

Objects to the Rescue



From now on, we'll have only objects.

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.*

Classes

Again, to define a class:

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

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.*

Class Definition Syntax

Syntax 9.1 Class Interface

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);

   double get_total() const;
   int get_count() const;

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

Use CamelCase for class names.

Member functions are declared in the class and defined outside.

Public section

Mutator member functions

Accessor member functions

Mark accessors as const.

Private section

Data members should always be private.

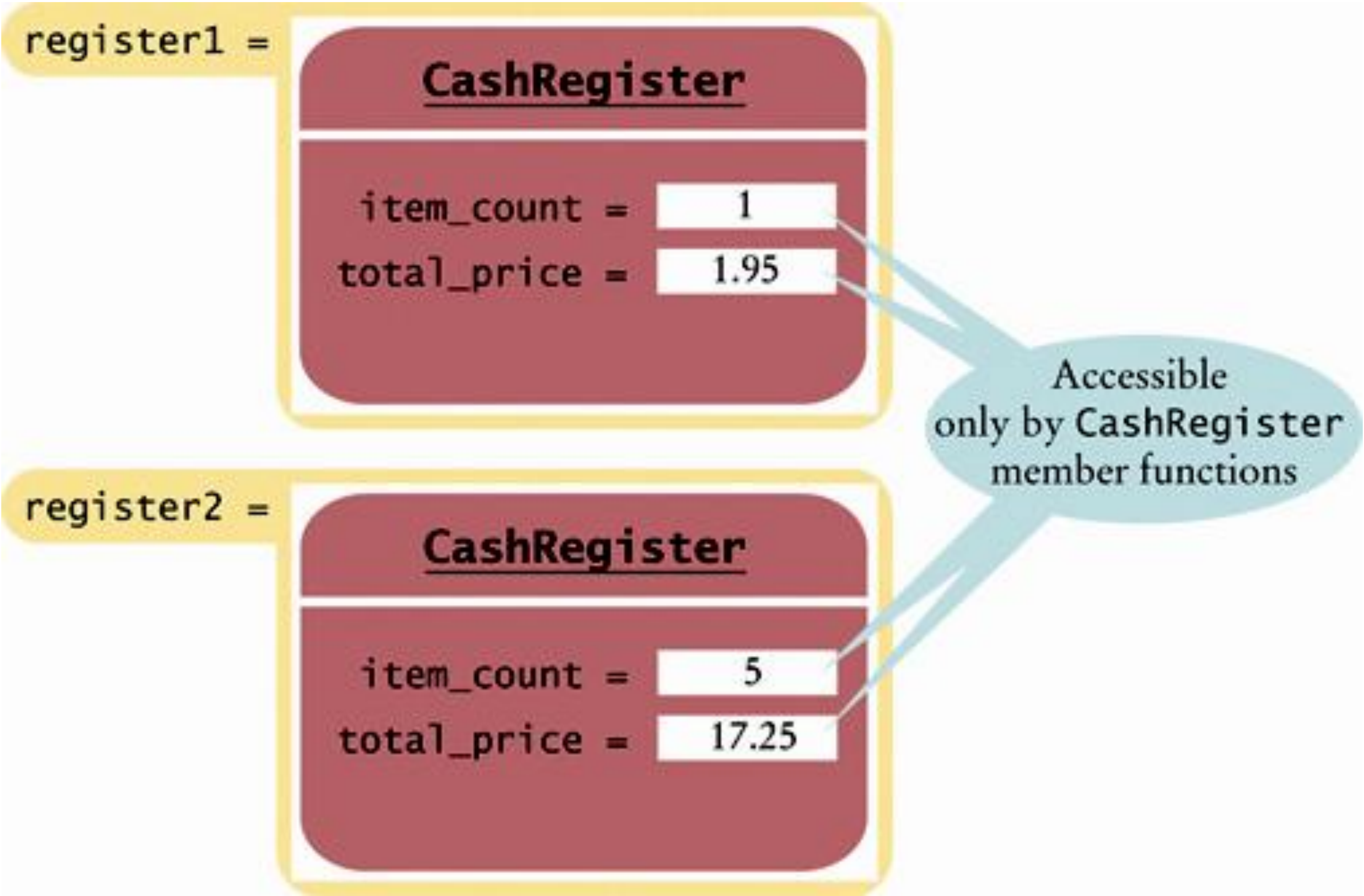
Be sure to include this semicolon.

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.

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 public, 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.

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

Not here

Only here

Implicit Parameters

Wait a minute.

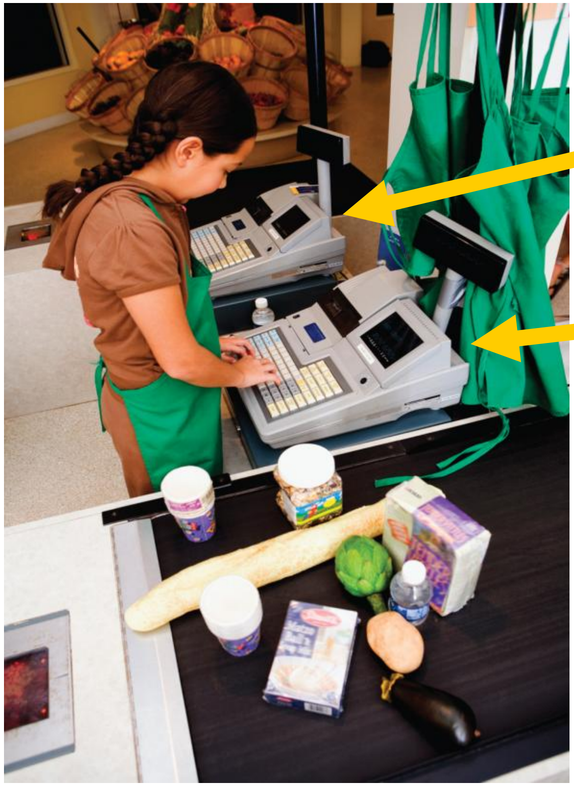
We are changing data members …

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.

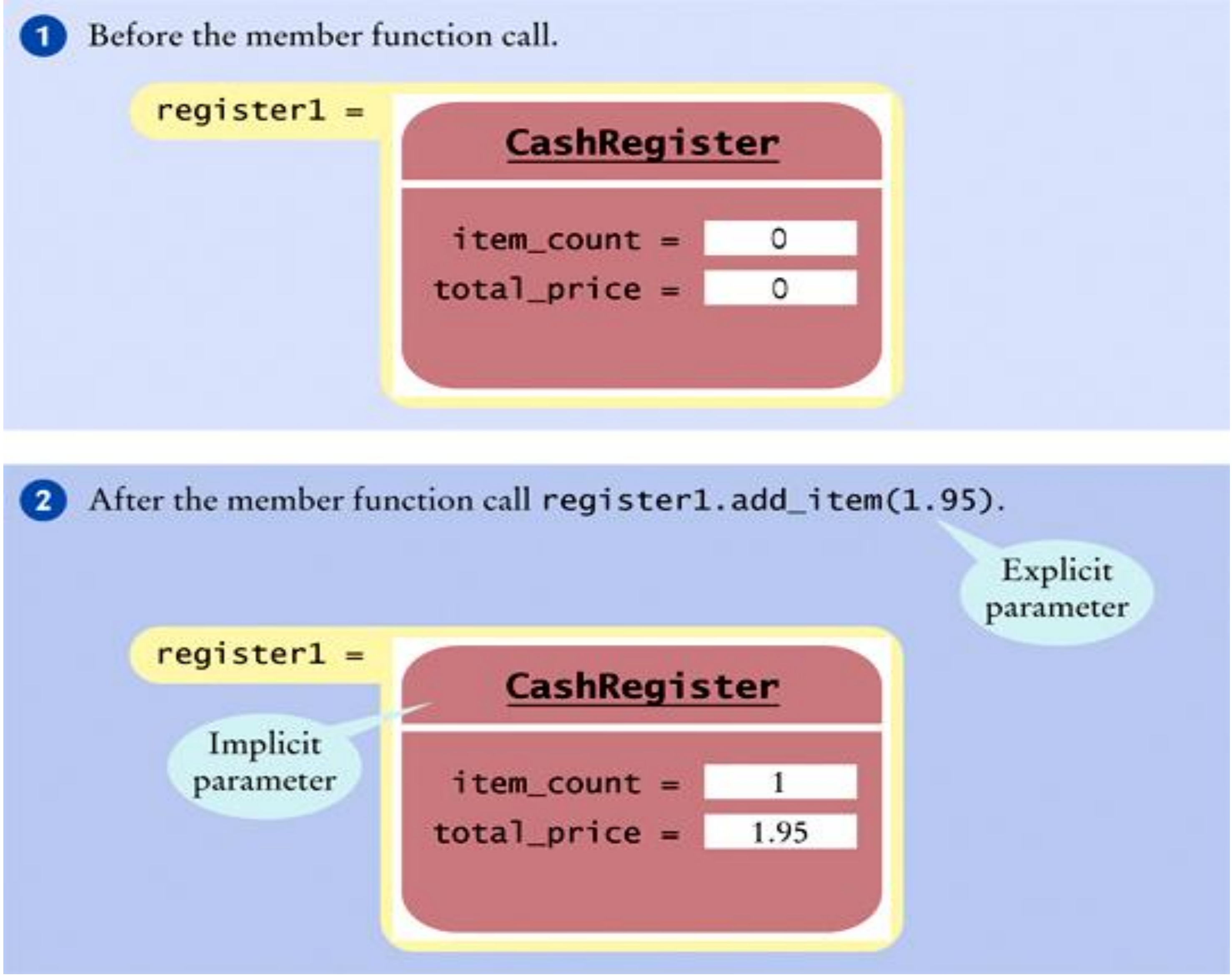
Implicit Parameters

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

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;
}

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

Data members of the implicit parameter

Data member of the implicit parameter

Use const for accessor functions.

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

```
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

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

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

Constructors

```
House house1;
House house2;
House house3;
...
```

A friendly construction worker reading a class definition

Constructors

A constructor *is a member function that* initializes *the data members of an object.*

Constructors

The constructor is automatically called whenever an object is created.

```
CashRegister register1;
```

By supplying a constructor,

you can ensure that all data members are properly set before *any member functions act on an object.*

Constructors

By supplying a constructor,

you can ensure that all data members are properly set before *any member functions act on an object.*

What would be the value of a data member that was not (no way!) properly set?

GARBAGE

Constructors

To understand the importance of constructors, consider the following statements:

```
CashRegister register1;
register1.add_item(1.95);
int count = get_count(); // May not be 1
```

*Notice that the programmer forgot to call **`clear`** before adding items.*

(Smells like "garbage" to me!)

Constructors

Constructors are written to guarantee that an object is always fully and correctly initialized when it is defined.

Constructors

You declare constructors in the class definition:

```
class CashRegister
{
public:
    CashRegister(); // A constructor
...
};
```

Constructors

The name of a constructor is identical to the name of its class:

```
class CashRegister
{
public:
   CashRegister(); // A constructor
...
};
```

Constructors

There must be **no** return type, *not even* ***`void`***.

```
class CashRegister
{
public:
    CashRegister(); // A constructor
...
};
```

Constructors

And, of course, you must define the constructor.

```
CashRegister::CashRegister()
{
    item_count = 0;
    total_price = 0;
}
```

Constructors

To connect the definition with the class, you must use the same `::` *notation*

```
CashRegister::CashRegister()
{
    item_count = 0;
    total_price = 0;
}
```

Constructors

You should choose initial values

for the data members so the object is correct.

```
CashRegister::CashRegister()
{
    item_count = 0;
    total_price = 0;
}
```

Constructors

And still no return type.

```
CashRegister::CashRegister()
{
    item_count = 0;
    total_price = 0;
}
```

Constructors

A constructor with no parameters is called a default constructor*.*

```
CashRegister::CashRegister()
{
    item_count = 0;
    total_price = 0;
}
```

Constructors

Default constructors are called when you define an object and do not specify any parameters for the construction.

```
CashRegister register1;
```

Notice that you do NOT use an empty set of parentheses.

Constructors

register1.item_count *and* ***register1.total_price*** *are set to zero as they should be.*

```
CashRegister register1;
```

Constructors

Constructors can have parameters,
and constructors can be overloaded:

```
class BankAccount
{
public:
    // Sets balance to 0
    BankAccount();
    // Sets balance to initial_balance
    BankAccount(double initial_balance);
    // Member functions omitted
private:
    double balance;
};
```

Constructors

When you construct an object, the compiler chooses the constructor that matches the parameters that you supply:

```
BankAccount joes_account;
   // Uses default constructor
BankAccount lisas_account(499.95);
   // Uses BankAccount(double) constructor
```

Constructors

It is good design to think *about what values you should put in numeric and pointer data members.*

They will be garbage if you don't set them in the constructor.

Constructors

Data members of classes that have constructors will not be garbage.

For example, the ***`string`*** *class has a default constructor that sets* ***`string`****s to the* empty string *("").*

Constructors

THINK: is the default **`string`** OK?

```
...
private:
    string name;
    double hourlyRate;
};
```

THINK, then set.

Common Error: Trying to Use the Constructor to Reset

You cannot use a constructor to “reset” a variable.
It seems like a good idea but you can’t:

```
CashRegister register1;
...
register1.CashRegister(); // Error
```

Constructors – The System Default Constructor

If you write no constructors at all,
the compiler automatically generates
a system default constructor
that initializes all data members of
class type with their default constructors

(which is just garbage for numeric and pointer data members).

Initialization Lists

When you construct an object whose data members are themselves objects, those objects are constructed by their class's default constructor.

However, if a data member belongs to a class without *a default constructor, you need to invoke the data member's constructor explicitly.*

Initialization Lists

A class to represent an order might not have a default constructor:

```
class Item:
public:
    Item(string item_descript, double item_price);
    // No other constructors
    ...
};
```

Initialization Lists

A class to represent an order would most likely have an Item type data member:

```
class Order
{
public:
    Order(string customer_name,
          string item_descript,
          double item_price);
...
private:
    Item article;
    string customer;
};
```

Initialization Lists

The Order constructor must call the Item constructor.

This is done in the **initializer list***.*

The initializer list goes before the opening brace of the constructor by putting the name of the data member followed by their construction arguments:

```
Order::Order(string customer_name,
             string item_description,
             double item_price)
      : article(item_description, item_price)
   ...
```

Initialization Lists

Any other data members can also be initialized in the initializer list by putting their initial values in parentheses after their name, just like the class type data members.
These must be separated by commas:

```
Order::Order(string customer_name,
             string item_description,
             double item_price)
      : article(item_description, item_price),
        customer(customer_name)
{
}
```

Notice there’s nothing to do in the body of the constructor now.

Tracing Objects

Recall how you hand traced code
to help you understand functions.

Adapting tracing for objects
will help you understand objects.

Grab some index cards
(blank ones).

Tracing Objects

You know that the **`public:`** *section is for others.*
That's where you'll write methods for their use.

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);
   double get_total() const;
   int get_count() const;
private:
   int item_count;
   double total_price;
};

...

CashRegister reg1;
```

That will be the front of the card.

Tracing Objects

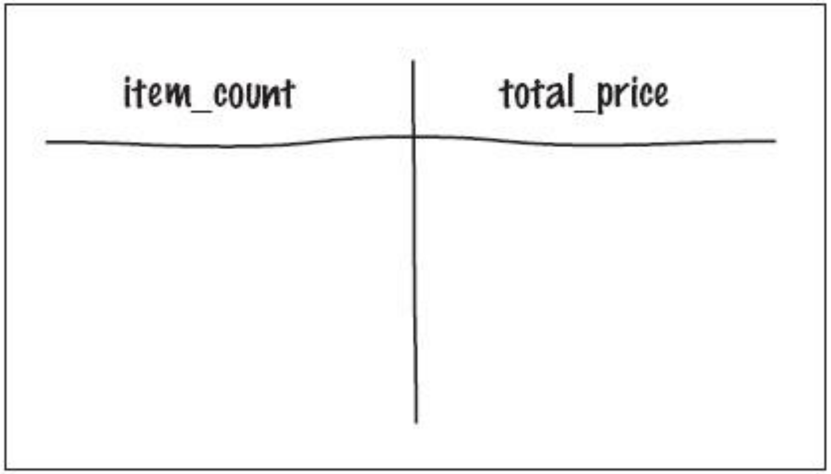
You know that the **`private:`** section is for your data – they are not allowed to mess with it except through the public methods you provide.

```
class CashRegister
{
public:
    void clear();
    void add_item(double price);
    double get_total() const;
    int get_count() const;
private:
    int item_count;
    double total_price;
};

...

CashRegister reg1;
```

That will be the back of the card.



back

Tracing Objects

You'll need a card for every variable.

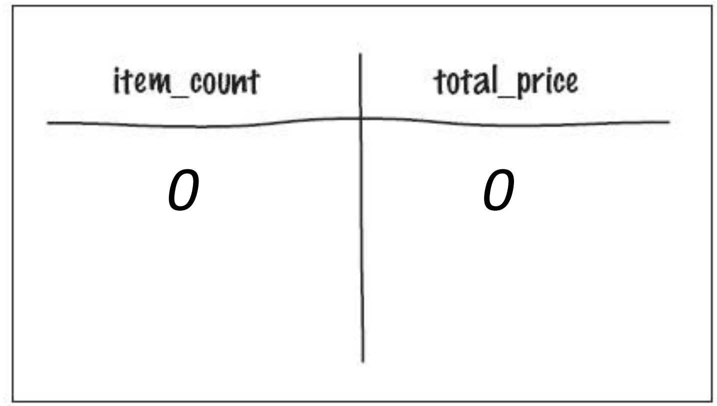
You might want to make several now.

Tracing Objects

```
CashRegister reg1;
```

When an object is constructed, add the variable's name to the front of a card and fill in the initial values.

front



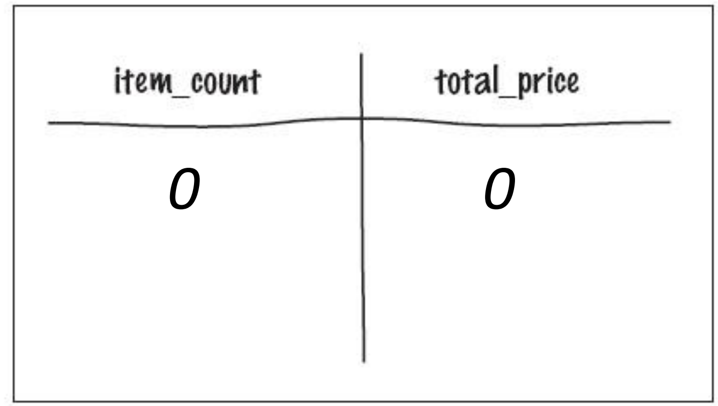
back

Tracing Objects

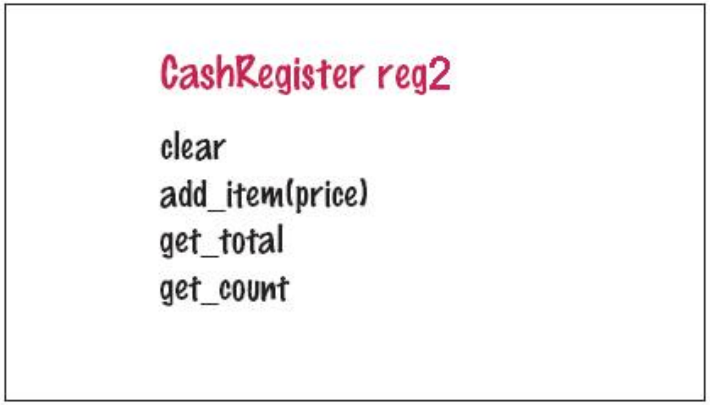
```
CashRegister reg1;
CashRegister reg2;
```

You would do this for every variable.

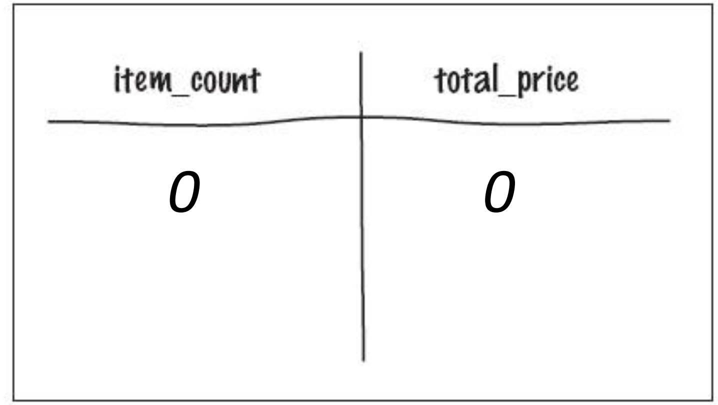
front



back



front

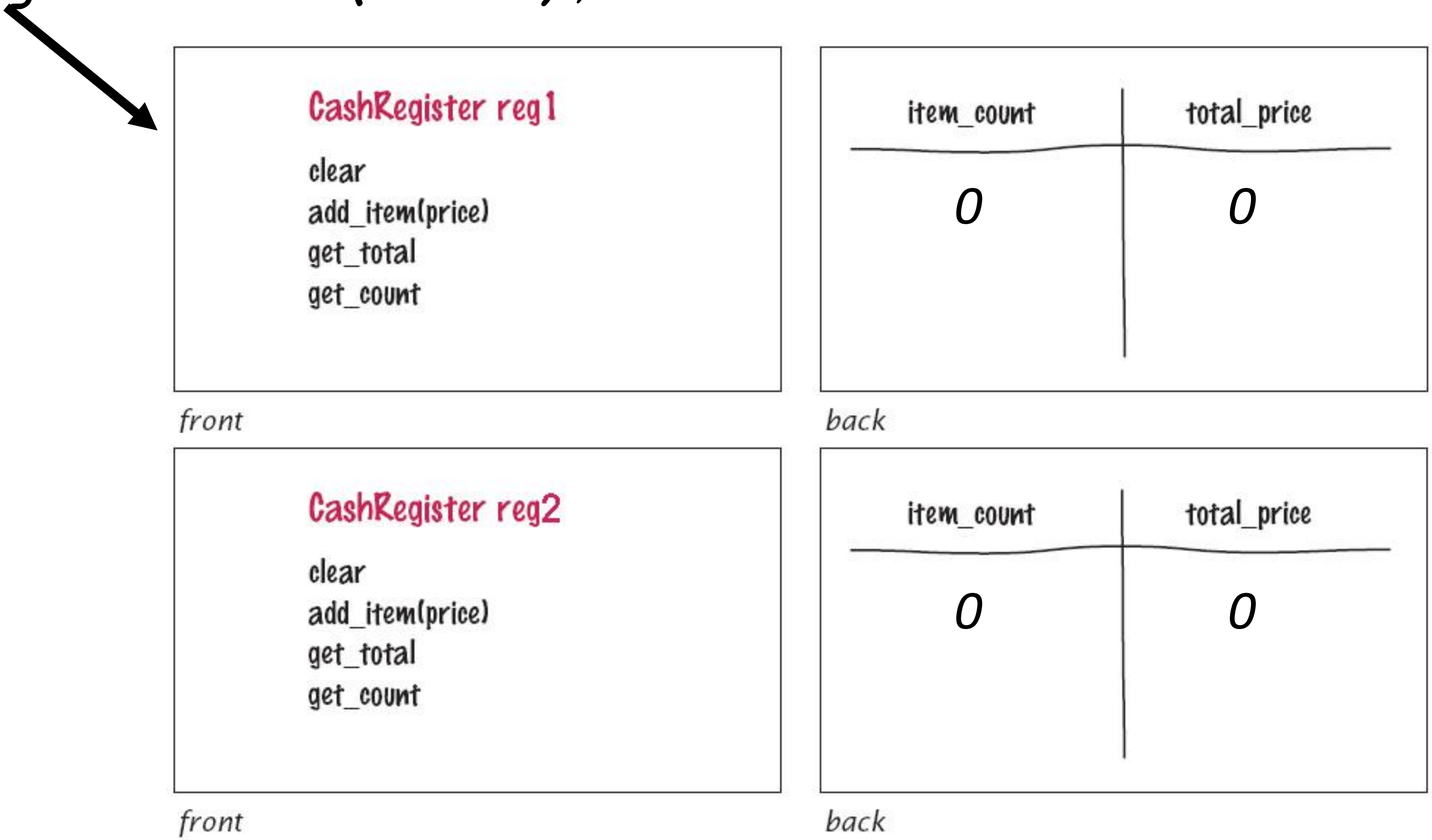


back

Tracing Objects

```
CashRegister reg1;
CashRegister reg2;
reg1.addItem(19.95);
```

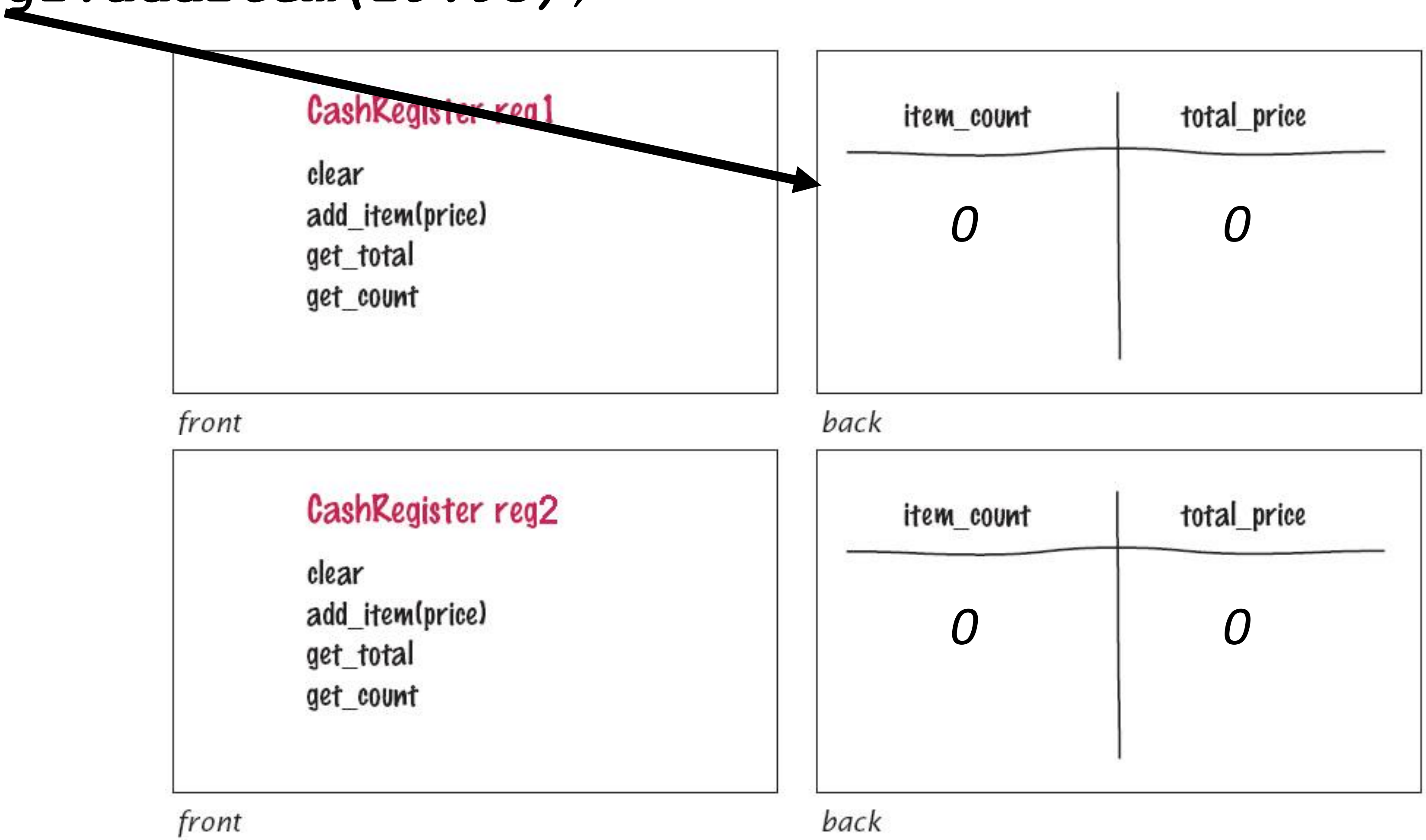
When a method is invoked, grab the right card...



Tracing Objects

```
CashRegister reg1;
CashRegister reg2;
reg1.addItem(19.95);
```

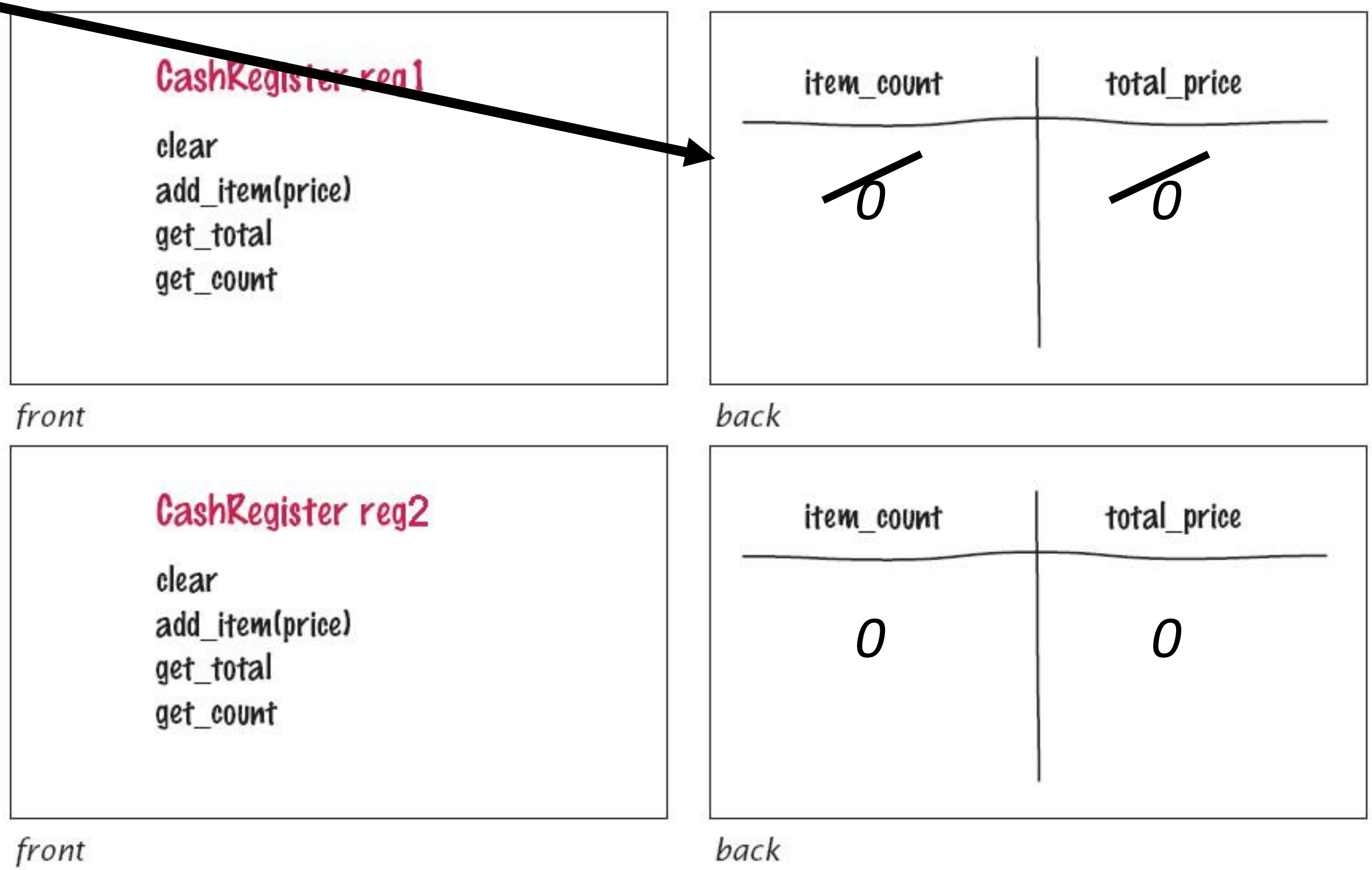
…flip it over…



Tracing Objects

```
CashRegister reg1;
CashRegister reg2;
reg1.addItem(19.95);
```

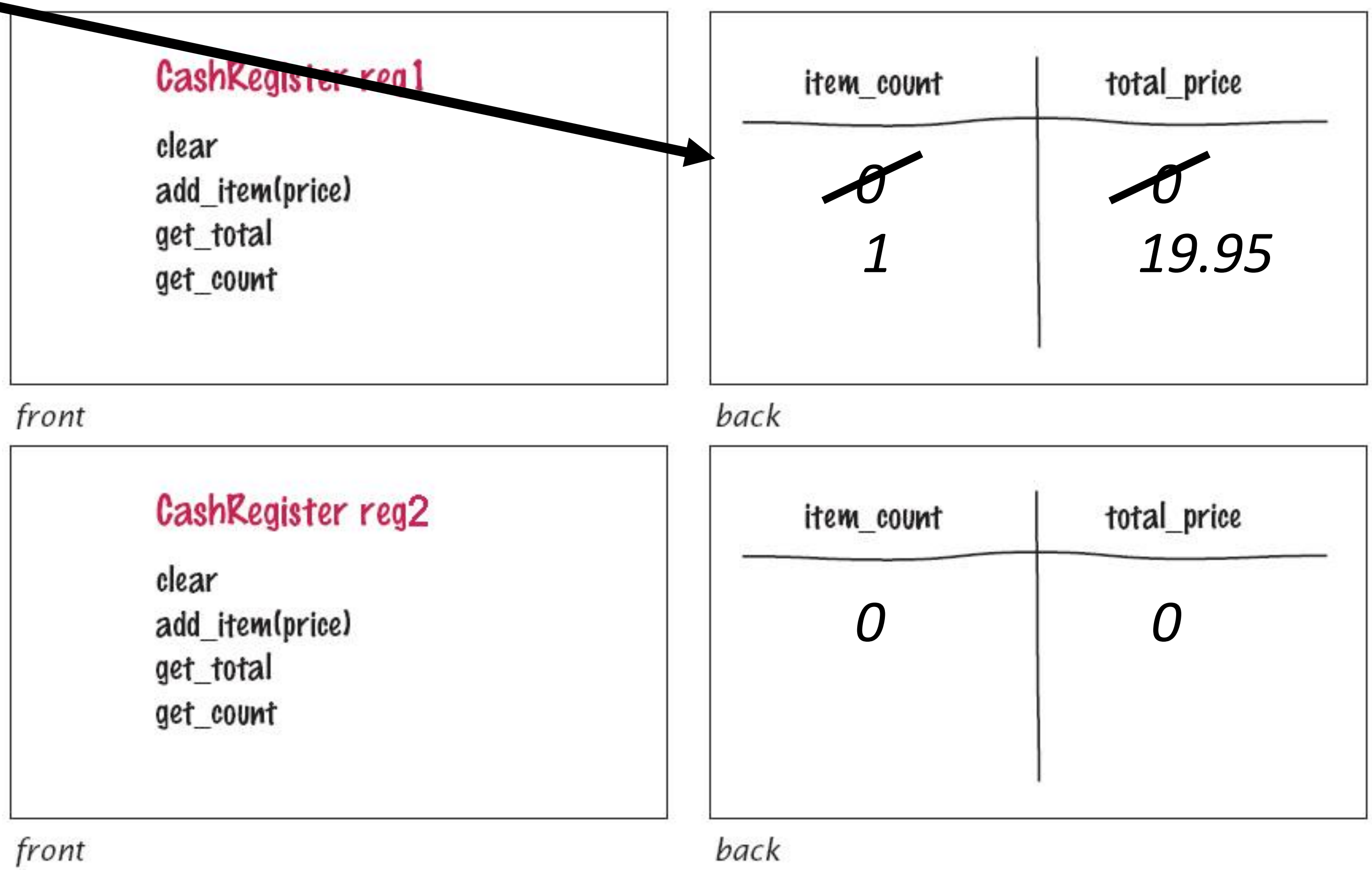
…cross out the old values...



Tracing Objects

```
CashRegister reg1;
CashRegister reg2;
reg1.addItem(19.95);
```

…then write the new values below.



Tracing Objects

These cards can help you in development when you need to add more functionality:

Suppose you are asked to get the sales tax.

Tracing Objects

You would add that to the front of the cards.
Grab any card – they will all have to be redone.

Add the newly requested method.

Then flip it over and start thinking.

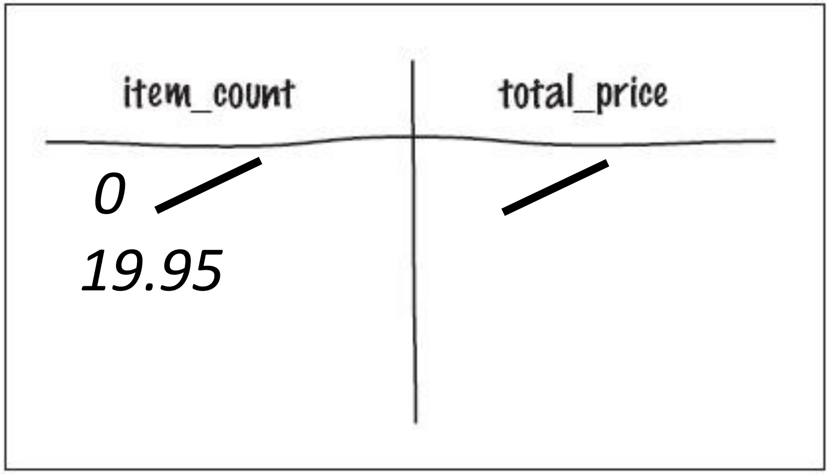
Tracing Objects

You would add that to the front of the cards.
Grab any card – they will all have to be redone.

Add the newly requested method.

Then flip it over and start thinking.

0
1



back

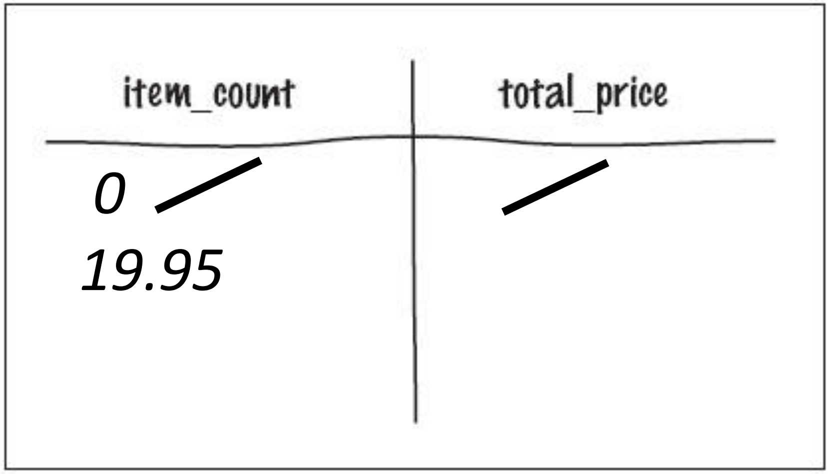
Tracing Objects

I have to calculate the sales tax.

Do I have enough information here on the back of this card?

I can only use these and any values passed in through parameters and global variables.

0
1



back

Tracing Objects

Tax rate?

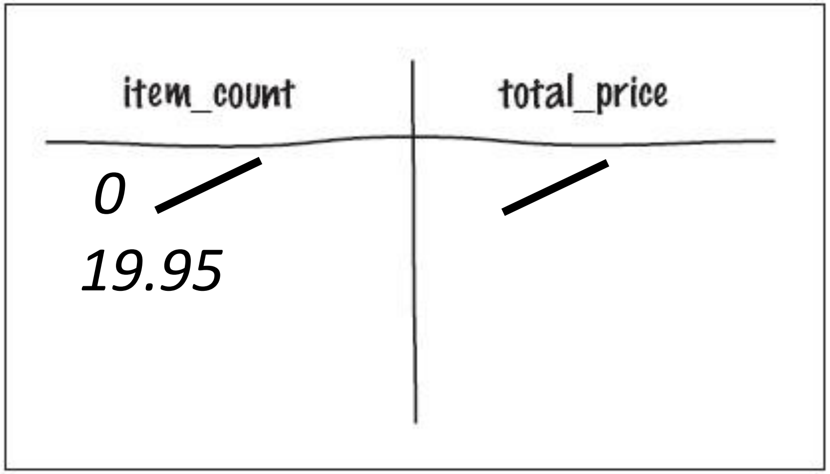
Need a new data member ***`tax_rate`*** *for this which would be set in the constructor to a global constant.*

Are all items taxable?

Need to add another parameter for taxable-or-not to ***`add_item`*** *which would appropriately update...*

…what???

Need a new data member: ***`taxable_total`****.*



0 0

1 19.95

back

Tracing Objects

Add these things and do some tracing.

```
CashRegister reg2(TAX_RATE);
reg2.addItem(3.95, false);
reg2.addItem(19.95, true);
```

item_count	total_price	taxable_total	tax_rate
~~0~~	~~0~~	~~0~~	7.5
~~1~~	~~3.95~~		
2	23.90	19.95	

Discovering Classes

One simple approach for discovering classes
and member functions is to look for
the nouns and verbs
in the problem description.

Discovering Classes

Often times,

- *nouns correspond to classes, and*
- *verbs correspond to member functions.*

Discovering Classes

Many classes are abstractions of real-life entities.

- ***`BankAccount`***
- ***`CashRegister`***

Discovering Classes

Generally, concepts from the problem domain,
be it science, business, or a game, make good classes.

The name for such a class should be
a noun that describes the concept.

Other frequently used classes represent
system services such as files or menus.

Not Discovering Classes

What might not *be a good class?*

If you can't tell from the class name what an object of the class is supposed to do, then you are probably not on the right track.

Not Discovering Classes

For example, you might be asked to write a program that prints paychecks.

You start by trying to design a ***`class PaycheckProgram`***.

Not Discovering Classes

```
class PaycheckProgram
```

?

What would an object of this class do?

Not Discovering Classes

```
class PaycheckProgram
```

? ?

An object of this class would have to

do everything!

Not Discovering Classes

`class PaycheckProgram`

? ? ?

That doesn't simplify anything.

A better class would be:

Discovering Classes

`class Paycheck`

! ! ! ! !

Not Discovering Classes

Another common mistake, made particularly by those who are used to writing programs that consist of functions, is to turn an action *into a* class*.*

Not Discovering Classes

For example, if you are to compute a paycheck, you might consider writing a

`class ComputePaycheck`*.*

Not Discovering Classes

`class ComputePaycheck`

*But can you visualize a “**`ComputePaycheck`**” object?*

A thing *that* is *a* ***`computePaycheck`****?*

Not Discovering Classes

```
class ComputePaycheck
```

The fact that "computepaycheck" is **not a noun** *tips you off that you are on the wrong track.*

On the other hand, a "paycheck" class makes intuitive sense.

Discovering Classes

You can visualize *a paycheck object.*

You can then think about useful member functions of the ***`Paycheck`*** *class, such as* ***`compute_taxes`****, that help you solve the problem.*

“Has-a” relationship

When you analyze a problem description,
you often find that you have multiple classes.

It is then helpful to consider how these classes are related.

One of the fundamental relationships between classes
*is the “*aggregation*” relationship*

(which is informally known as the “has-a” relationship).

“Has-a” relationship

The aggregation relationship states that objects of one class contain objects of another class.

“Has-a” relationship

Consider a quiz that is made up of questions.

Since each quiz has one or more questions,
we say that the class ***`Quiz`*** aggregates *the class* ***`Question`***

Discovering Classes

In summary, when you analyze a problem description, you will want to carry out these tasks:

- *Find the concepts that you need to implement as classes. Often, these will be nouns in the problem description.*
- *Find the responsibilities of the classes. Often, these will be verbs in the problem description.*
- *Find relationships between the classes that you have discovered.*

Separate Compilation

When you write and compile small programs,
you can place all your code into a single source file.

When your programs get larger or you
work in a team, that situation changes.

You will want to split your code into separate source files.

Separate Compilation

There are two reasons why this split becomes necessary.

First, it takes time to compile a file, and it seems silly to wait for the compiler to keep translating code that doesn't change.

If your code is distributed over several source files, then only those files that you changed need to be recompiled.

Separate Compilation

The second reason becomes apparent when you work with other programmers in a team.

It would be very difficult for multiple programmers to edit a single source file simultaneously.

Therefore, the program code is broken up so that each programmer is solely responsible for a separate set of files.

Separate Compilation

If your program is composed of multiple files, some of these files will define data types or functions that are needed in other files.

There must be a path of communication between the files.

In C and C++, that communication happens through the inclusion of header files.

Yes, ***`#include`****.*

Separate Compilation

The code will be in two kinds of files:

header files (.h files)
*(which will be **`#include`**-ed)*

source files (.c or .cpp files)
*(which should never be **`#include`**-ed)*

Separate Compilation

A header file contains (.h extension)

- *the interface:*
  - *Definitions of classes.*
  - *Definitions of constants.*
  - *Declarations of nonmember functions.*

Separate Compilation

A source file contains (.c or .cpp extension)

- *the implementation:*
  - *Definitions of member functions.*
  - *Definitions of nonmember functions.*

Separate Compilation

There will also be either:

a “tester” program

or

the real problem solution

This is where **`main`** *goes.*

Separate Compilation

For the **`CashRegister`** *class,*
you create a pair of files:

`cashregister.h`

the interface – the class definition

`cashregister.cpp`

the implementation – all the member function definitions

Separate Compilation: The Cash Register Program

This is the header file, ***`cashregister.h`***

Notice the first two lines.

There is an ending ***`#endif`*** *at the end of the file.*

This makes sure the header is only included once.

Always write these. Use the name of the as shown.

ch09/cashregister.h

```
#ifndef CASHREGISTER_H
#define CASHREGISTER_H

/**
   A simulated cash register that tracks
   the item count and the total amount due.
*/
class CashRegister
```

Separate Compilation: The Cash Register Program

ch09/cashregister.h

```
/**
   A simulated cash register that tracks
   the item count and the total amount due.
*/
class CashRegister
{
public:
   /**
      Constructs a cash register with
      cleared item count and total.
   */
   CashRegister();

   /**
      Clears the item count and the total.
   */
   void clear();
```

Separate Compilation: The Cash Register Program

ch09/cashregister.h

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

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

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

Separate Compilation: The Cash Register Program

ch09/cashregister.h

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

#endif
```

You include this header file whenever the definition of the **`CashRegister`** *class is required.*

Since this file is not a standard header file, you must enclose its name in quotes, not **`<. . .>`**, *when you include it, like this:*

```
#include "cashregister.h"
```

And now the implementation (.cpp) file:

Separate Compilation: The Cash Register Program

*Notice that the implementation file **`#include`**s its own header file.*

ch09/cashgregister.cpp

```
#include "cashregister.h"

CashRegister::CashRegister()
{
   clear();
}

void CashRegister::clear()
{
   item_count = 0;
   total_price = 0;
}
```

Separate Compilation: The Cash Register Program

ch09/cashgregister.cpp

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

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

Separate Compilation

Notice that the member function comments are in the header file, not the .cpp file.

They are part of the interface, not the implementation.

Separate Compilation

There's no ***`main`****!*

HELP!

Now, someone who wants to use your class will write their own ***`main`*** *and* ***`#include`*** *your header.*

Like this:

Separate Compilation: The Cash Register Program

ch09/registertest2.cpp

```
#include <iostream>
#include <iomanip>
#include "cashregister.h"
using namespace std;

/**
   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;
}
```

Separate Compilation: The Cash Register Program

ch09/registertest2.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;
}
```

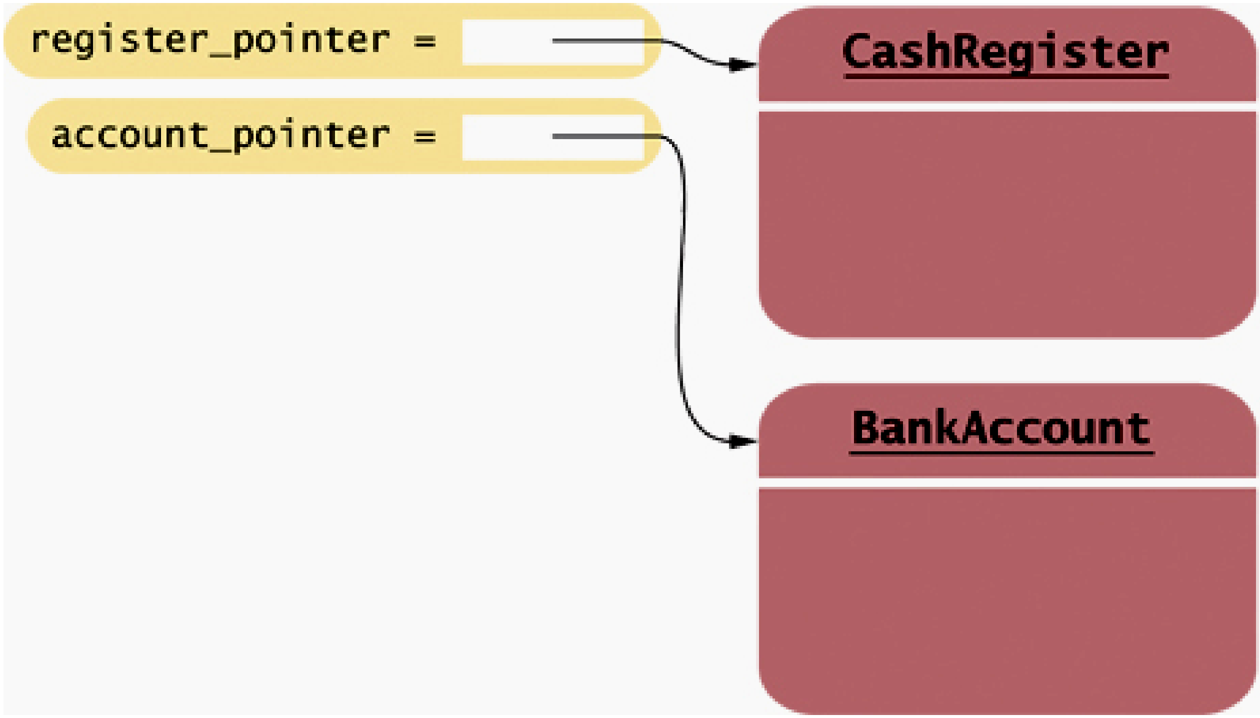
Pointers to Objects

Dynamically Allocating Objects

How about dynamic objects?

Fine:

Pointers to Objects




```
CashRegister* register_pointer = new CashRegister;
BankAccount* account_pointer = new BankAccount(1000);
```

Accessing: The -> Operator

Because **`register_pointer`** *is a pointer to a* **`CashRegister`** *object, the value* **`*register_pointer`** *denotes the* **`CashRegister`** *object itself.*

To invoke a member function on that object, you might call

```
(*register_pointer).add_item(1.95);
```

Accessing: The –> Operator

The parentheses are necessary because in C++ the dot operator takes precedence over the ***** *operator.*

The expression without the parentheses would be a syntax error:

```
*register_pointer.add_item(1.95);
     // Error – you can't apply . to a pointer
```

Pointers to Objects

Because calling a member function through a pointer is very common, the designers of C++ supply an operator to abbreviate the “follow pointer and access member” operation.

That operator is written **->** *and usually pronounced as “arrow”.*

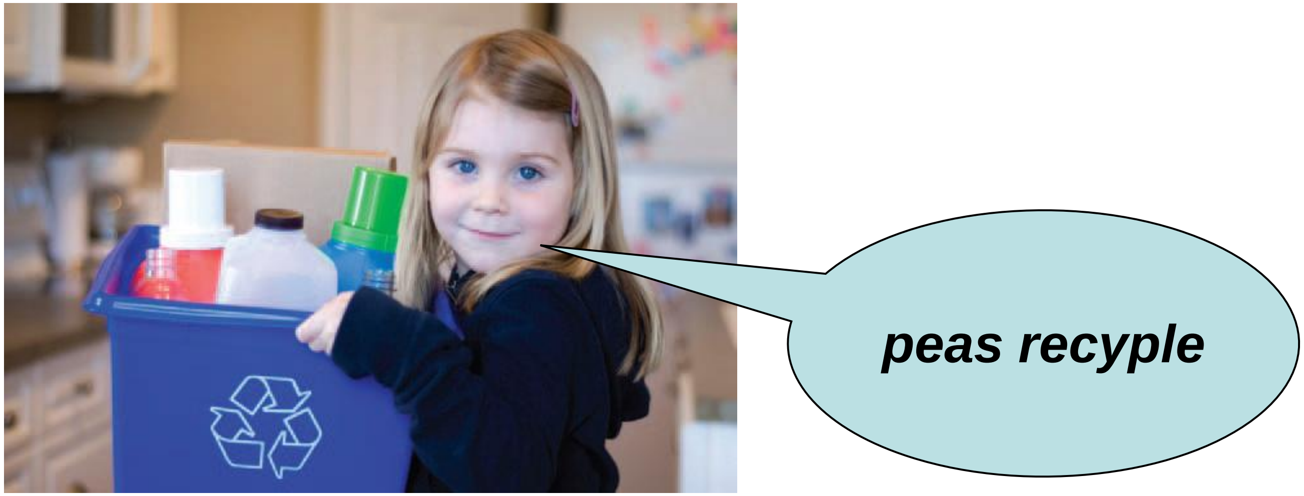
Here is how you use the “arrow” operator:

```
register_pointer->add_item(1.95);
```

Destructors and Resource Management

When a programmer uses new to obtain a dynamic array, she is requesting a system resource.

And as all good recyclers know…



…resources are limited and should be returned.

Destructors and Resource Management

THE HEAP**:**
No Entry Without
Permission Of OS!

```
class String
{
    ...
private:
    char* char_array;
}
String::String(const char initial_chars[])
{
    char_array = new char[strlen(initial_chars) + 1];
    strcpy(char_array, initial_chars);
}
```

The characters of a **String** are stored on the heap, a system resource.

Destructors and Resource Management

Constructors don’t really construct (they initialize).

There is another method that doesn’t really do what it’s name implies:

the destructor*.*

(Not in any way associated with professional wrestling.)

Destructors and Resource Management

A destructor, like a constructor, is written without a return type and its name is the tilde *character followed by the name of the class:*

```
~String
```

Destructors and Resource Management

A class can have only one destructor

and

it cannot have any parameters.

```
String::~String()
{ ...
```

Destructors and Resource Management

Destructors don't really destruct:

```
String::~String()
{ ...
```

Destructors and Resource Management

Destructors don't really destruct:

they are used to recycle resources.

```
String::~String()
{ ...
```

Destructors and Resource Management

Destructors don't really destruct:

they are used to recycle resources.

```
String::~String()
{
    delete[] char_array;
}
```

Destructors and Resource Management

THE HEAP:
No Entry Without
Permission Of OS!

The memory for the characters in a string are properly recycled..

```
void fun()
{
    String name("Harry");
    ...
}
```

Heap memory is allocated by the constructor

Do you see a method being invoked?

Right there!

Destructors and Resource Management

THE HEAP:
No Entry Without
Permission Of OS!

Destructors are automatically invoked when an object of that type is no longer needed.

The memory for the characters in a string are properly recycled..

```
void fun()
{
    String name("Harry");
    ...
}
```

`String::~String()` *is invoked right there.*

Destructors and Resource Management

THE HEAP**:**
No Entry Without
Permission Of OS!

Unfortunately, it's more complicated when assignment comes along:

```
void no_fun()
{
    String name1("Harry");
    String name2("Sally");
    name1 = name2;
    ...
}
```

What happened to the memory for "Harry"?

Now what?!

Heap memory is allocated by both the constructors

Destructors and Resource Management

This is not a topic covered in these slides.

It involves:

the destructor

and

another kind of constructor - the copy constructor

and

rewriting how the assignment operation works.

These three topics together are called The Big Three.

(Again, not in any way associated with professional wrestling.)

English?

And now for another English sort of thing:

The **`this`** *pointer.*

(Yes, it's correct English)

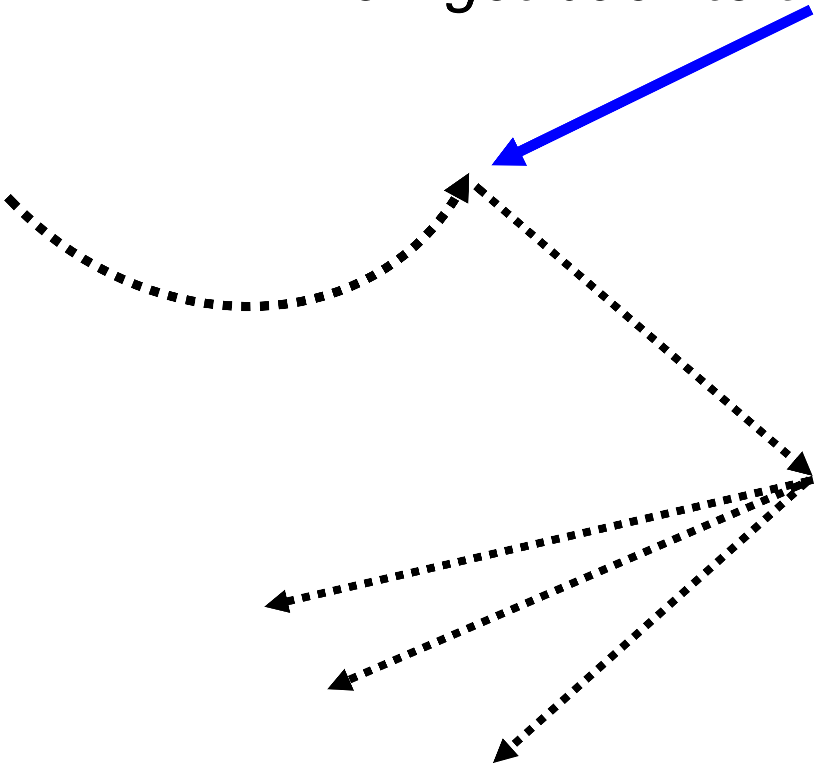
(if you are talking about C++.)

Remember, way back there,

when we said:

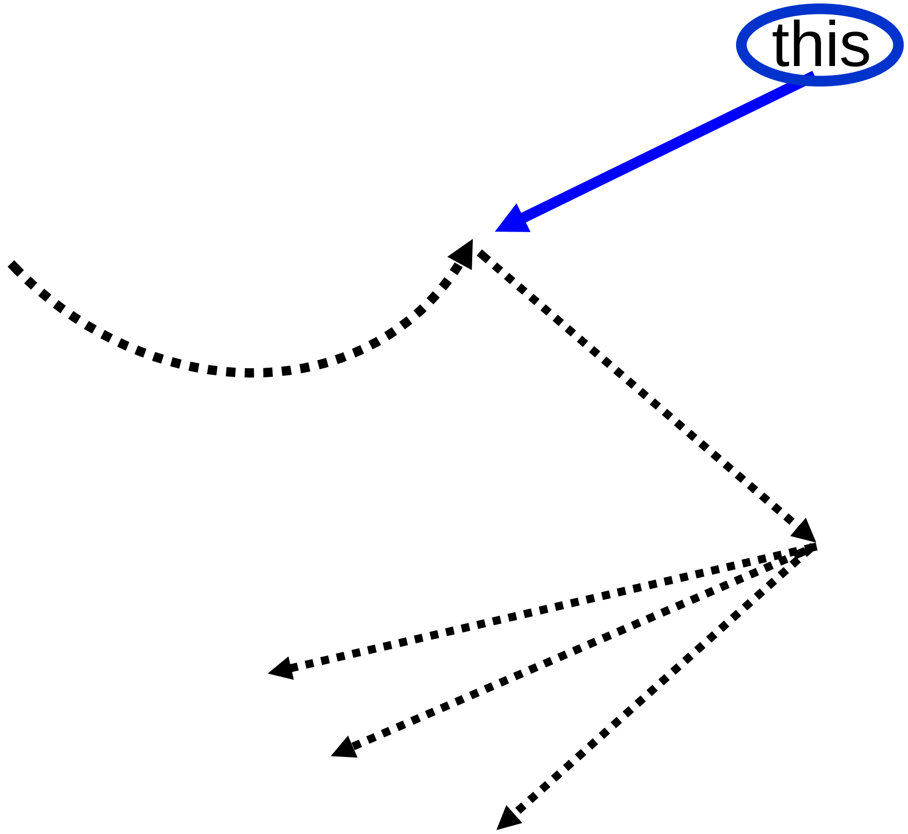
Implicit Parameters

“We’ll get back to this, *later …”*



Well, now is later!

Implicit Parameters



That's the ***this*** pointer.

Implicit Parameters

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

`this` **`= register1`** *(assigned by the system)*

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

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

this.

The `this` Pointer

Each member function has a special parameter variable,
called **`this`**,
which is a pointer to the implicit parameter.

The `this` Pointer

For example, consider the member function

```
CashRegister::add_item
```

The `this` Pointer

If you call

... register1.add_item(1.95) ...

*then the **`this`** pointer has type **`CashRegister*`** and points to the **`register1`** object.*

The `this` Pointer

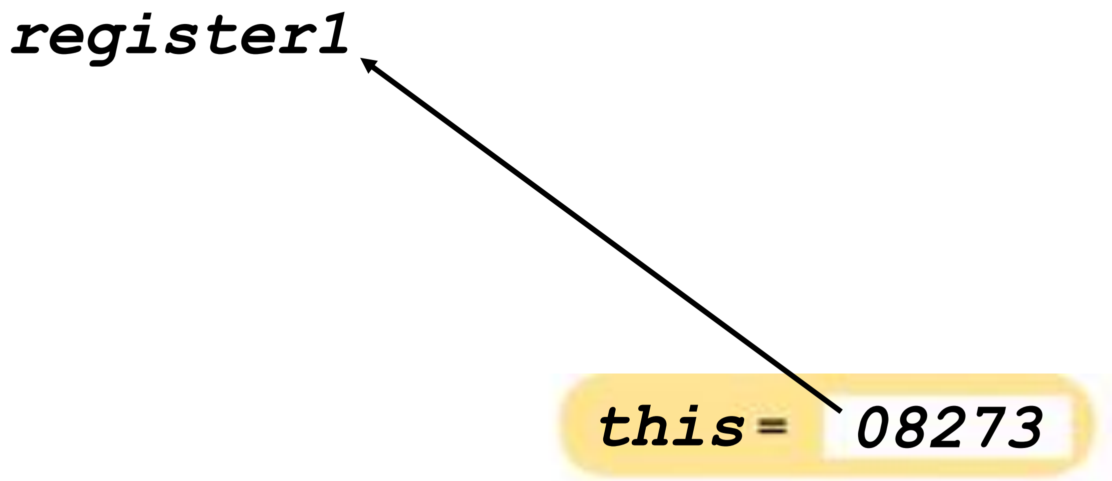
(I don’t see it.)

No, but you can use it:

The `this` Pointer

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

The *this* Pointer



The ***this*** *pointer is made to point to the implicit variable.*

(The system did that assignment behind your back.)

(Thank you!)

The `this` Pointer

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

The `this` Pointer

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

this *points at the that* **implicit parameter***.*

The `this` Pointer

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

The ***`this`*** *pointer is not necessary here, but some programmers like to use the* ***`this`*** *pointer to make it*

very, very *clear*

that ***`item_count`*** *and* ***`total_price`*** *are data members—not (plain old) variables or parameters.*