

COMPUTER SCIENCE 1: STARTING COMPUTING CSCI 1300



The image part with relationship ID rId3 was not found in the file.

Ioana Fleming / Vipra Gupta
Spring 2018
Lecture 20

Announcements

- Rec 8 due on 3/10
- Hmwk 6 due on 3/11
- **Practicum 2: March 14th, 2018**
 - Loops: while, for
 - Strings
 - Arrays
 - File I/O
 - *sign up on Moodle to take the exam at 6pm*

Agenda

- Today:
 - Object Oriented Programming
 - Classes and objects

Object-Oriented Programming

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

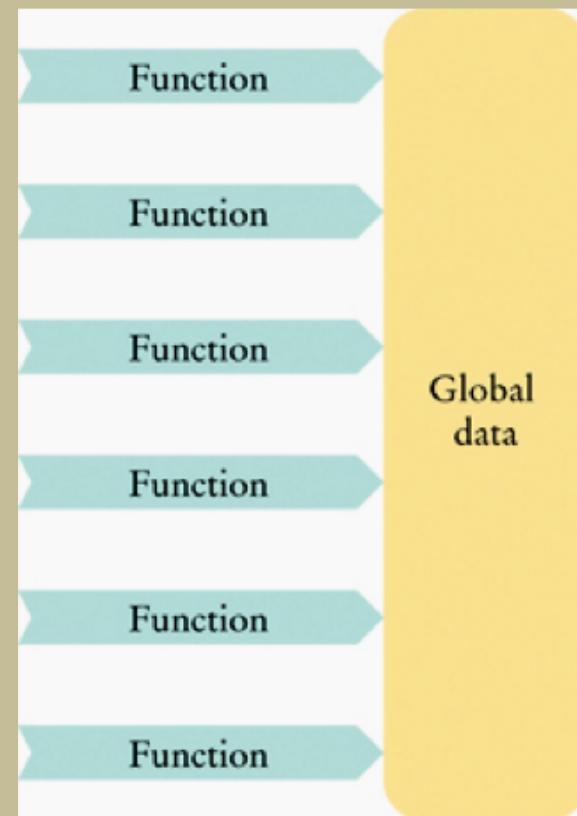
USING GLOBAL VARIABLES

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

Global Variables = variables who have scope everywhere; can
be used in any function

Object-Oriented Programming

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



Example:

Candyland:

- global variables for the current position for each player
- functions for each situation (if licorice, if lollipop, ...) will be able to update the current position without passing and returning values.



Object-Oriented Programming

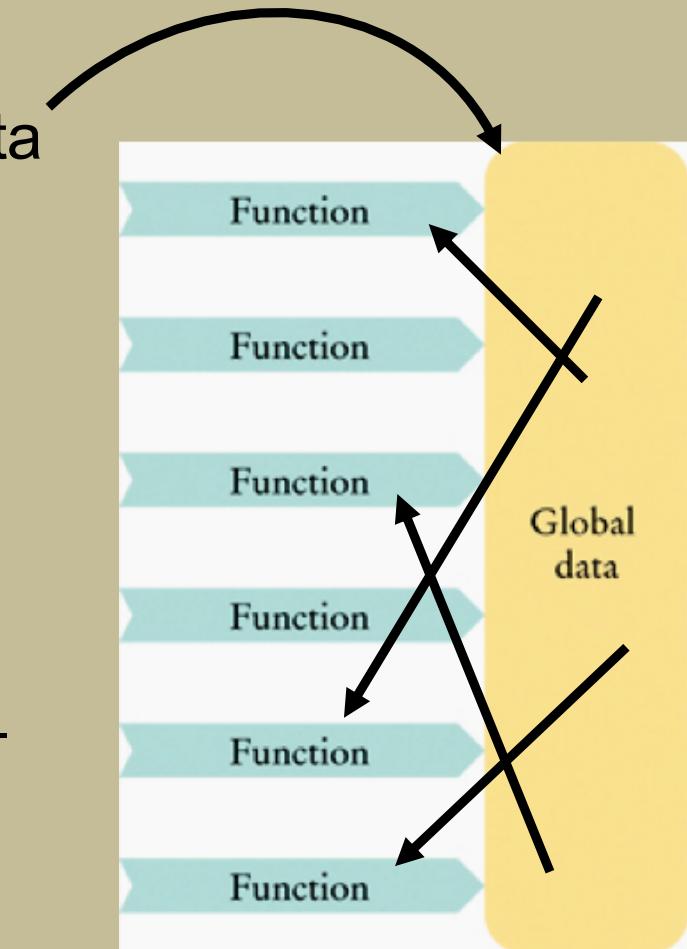
When some part of the global data
needs to be changed:

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

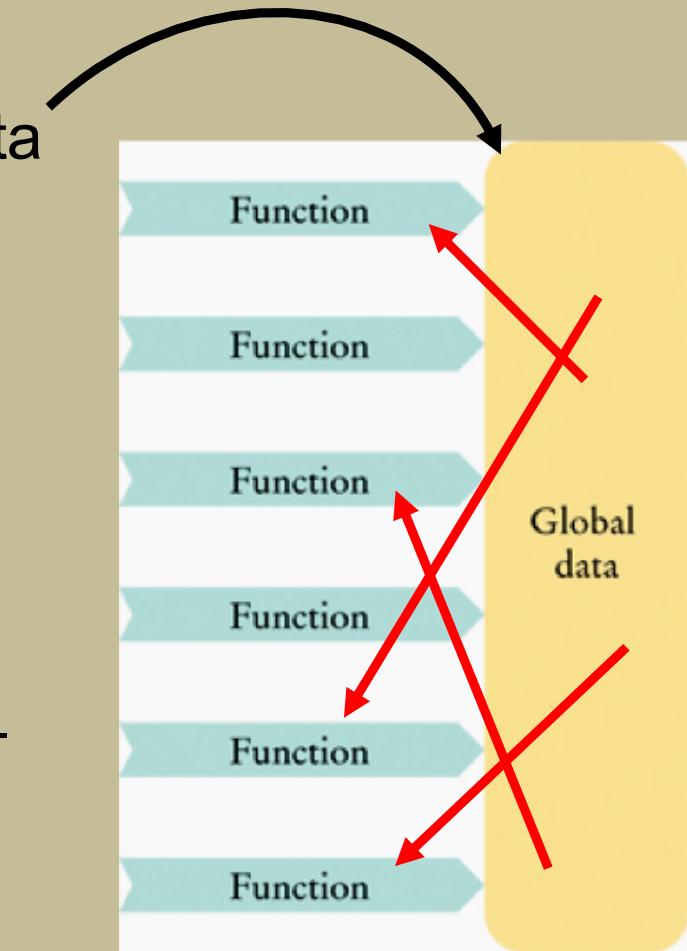
When some part of the global data
needs to be changed:

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

Instead of “variables” and “functions” –
separately.

Objects to the Rescue

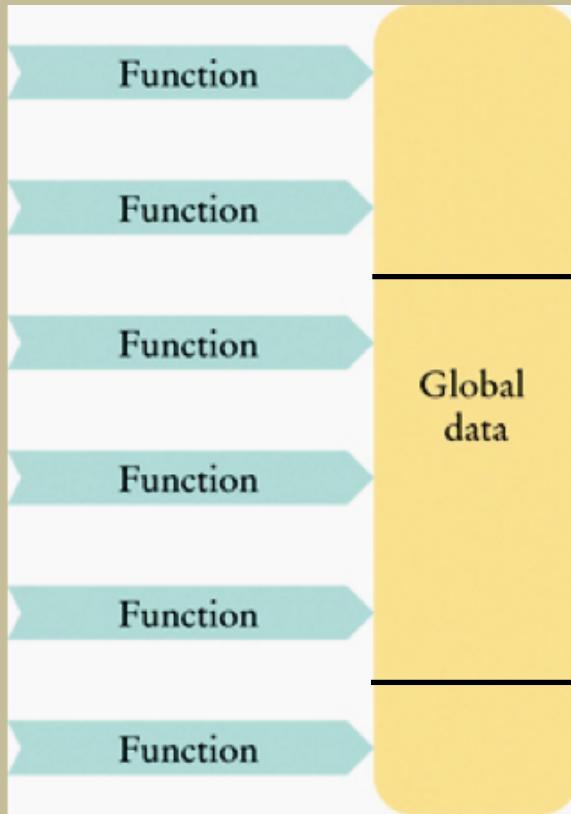
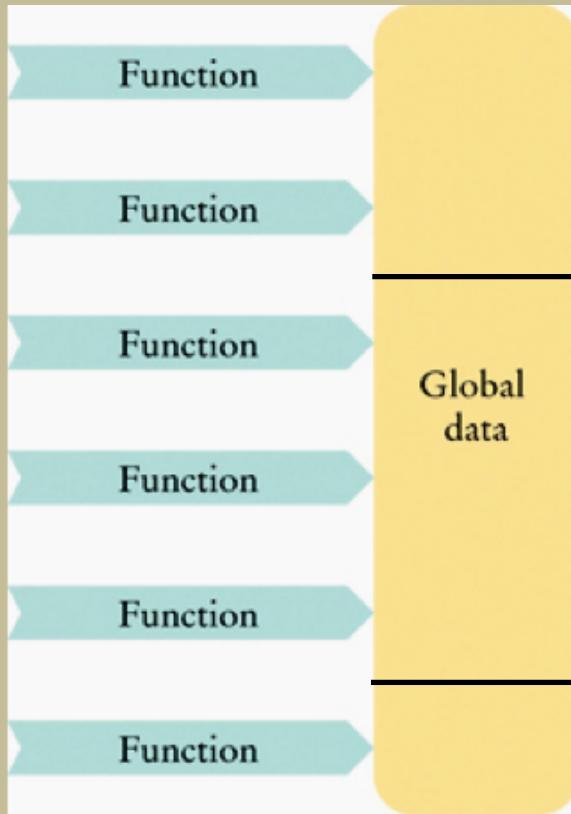


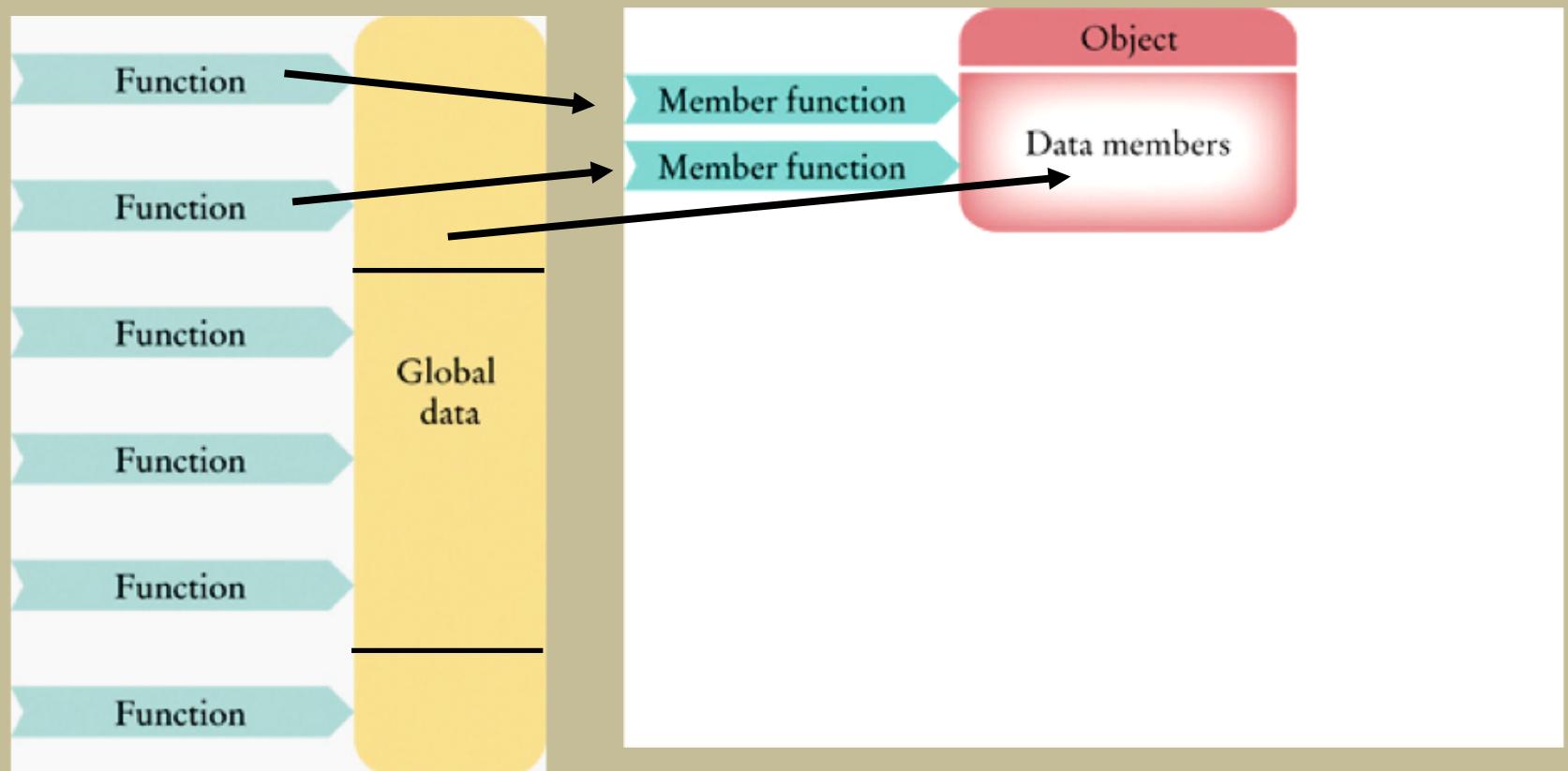
Figure out which functions go with which data.

Objects to the Rescue



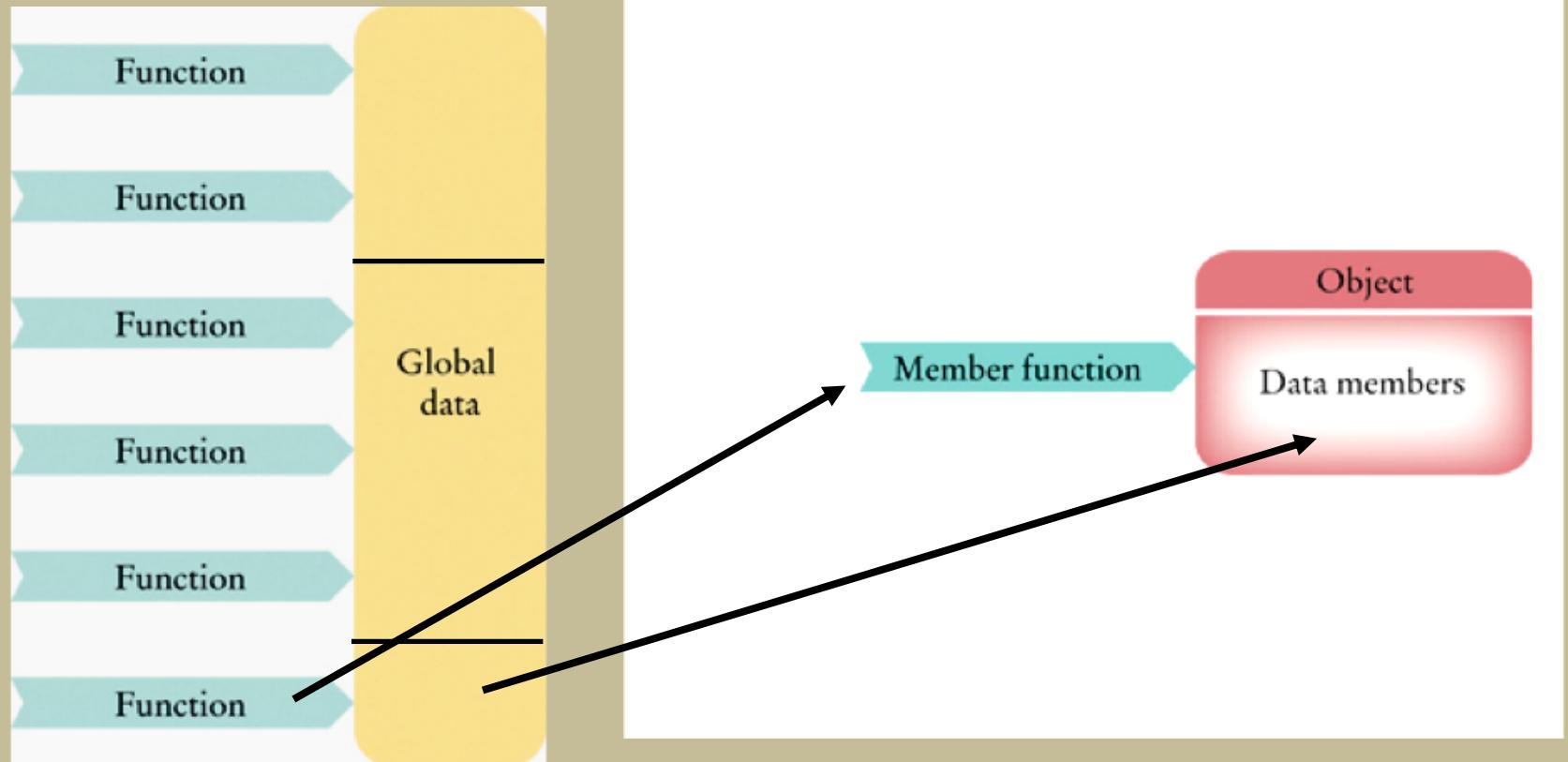
Create an object for each set of data.

Objects to the Rescue



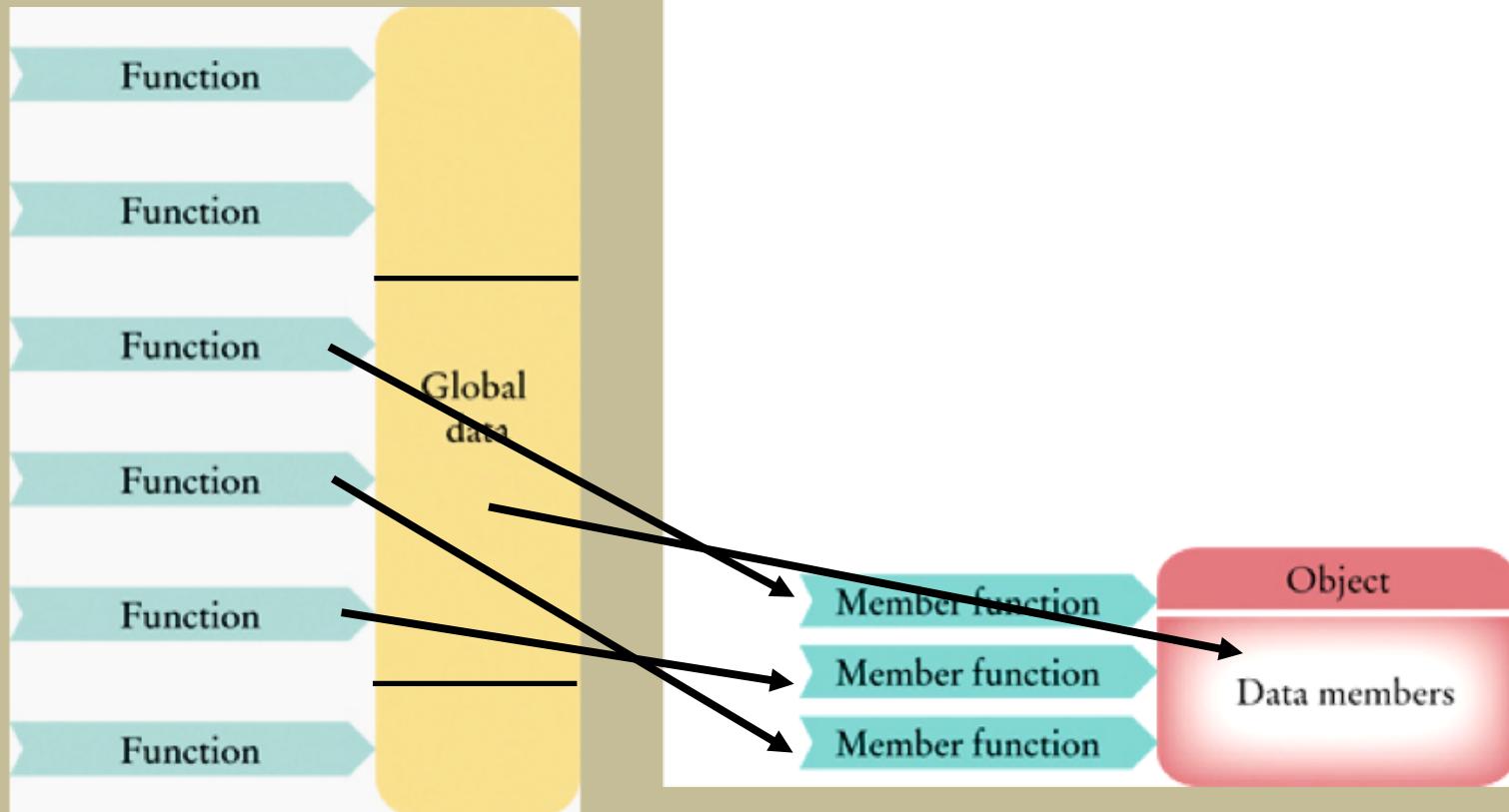
Create another object for another set.

Objects to the Rescue



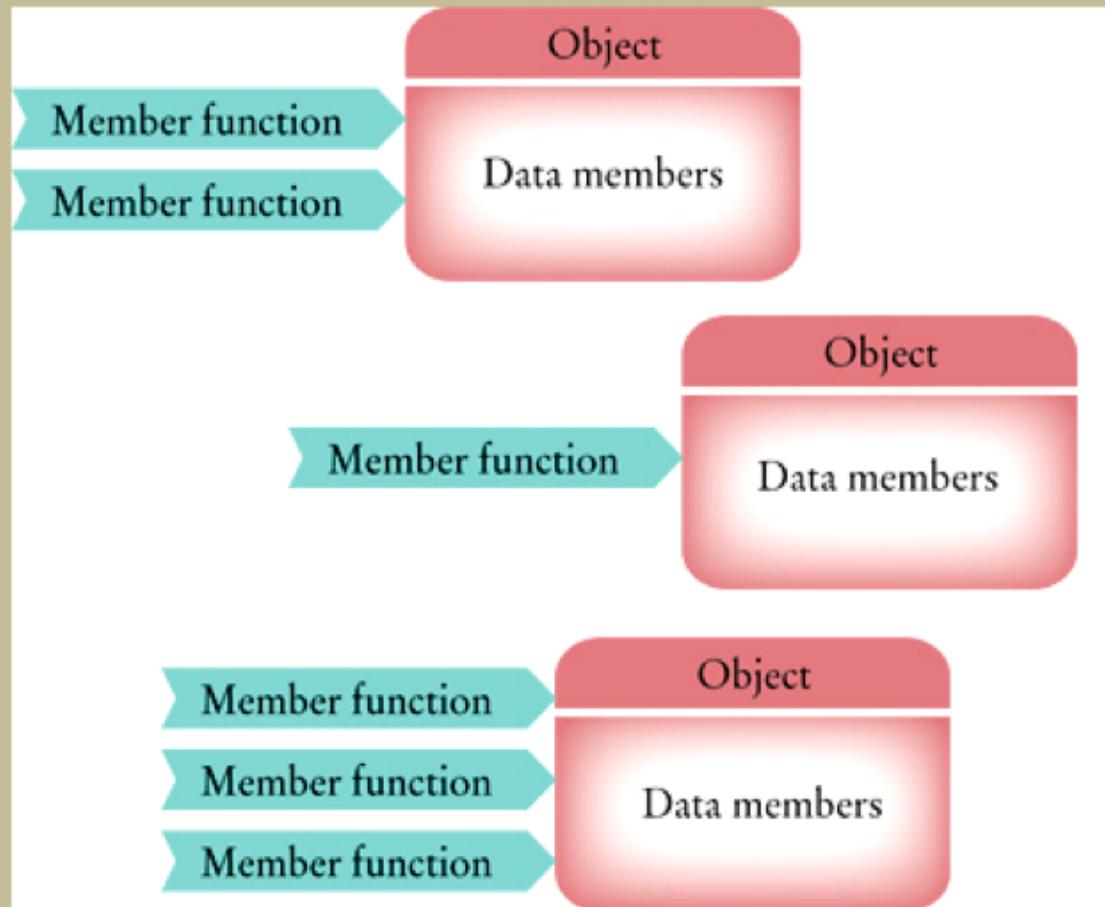
And now, a third object. You get the idea

Objects to the Rescue



Get rid of those global variables.

Objects to the Rescue



From now on, we'll have only objects.

Objects examples

Point object:

- what would the data members be? (for a 2D point)
- for a 3D point?

Car object:

- what would the data members be?
 - make
 - model
 - color
 - mileage
 - year
 - electric (yes/no)

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

- 1. Benefits developer:** *When working on a program that is being developed over a long period of time, it is common for implementation details to change, usually to make objects more efficient or more capable. Encapsulation is crucial to enabling these changes. When the implementation is hidden, the developer is free to make improvements. Because the implementation is hidden, these improvements do not affect the programmers who use the objects.*
- 2. Benefits programmers who use the objects:** *They can put your classes to work without having to know their implementations, just as you are able to make use of the **string** and **stream** classes without knowing their internal details.*

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 `[]` or `>>`
(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...

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.

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 ...

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

Let's 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.



Classes

These activities will be our *public interface*.

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

Remember function declaration (prototype)?

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

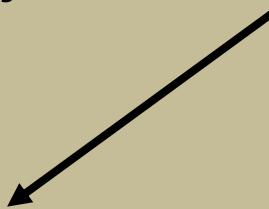
Here is the C++ syntax for the **CashRegister** class definition:

```
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 three 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

```
class TwoHumpCamel  
{  
}  
;
```

Three black arrows point from the words "TwoHumpCamel" in the code above to the two humps and the head of the camel in the image below.

Class name should follow *camelCase*:

From Wikipedia: writing compound words or phrases such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation

Methods

There are two kinds of member functions:

- *Mutators*
- *Accessors*

Mutators

- A mutator is a function that *modifies* the data members of the object.
 - increment the item count
 - add to the total bill
 - clear all data members

Mutators

CashRegister has two mutators: **clear**

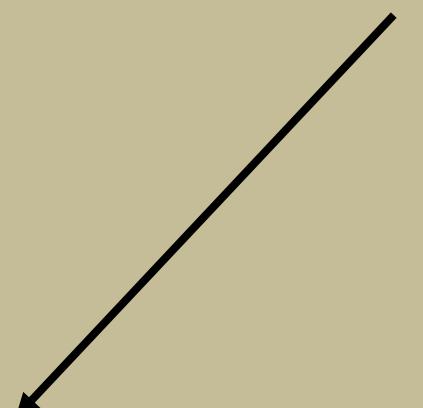
```
class CashRegister
{
public:
    void clear();
    void add_item(double price);
    double get_total() const;
    int get_count() const;
private:
    // data members will go here
};
```



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:

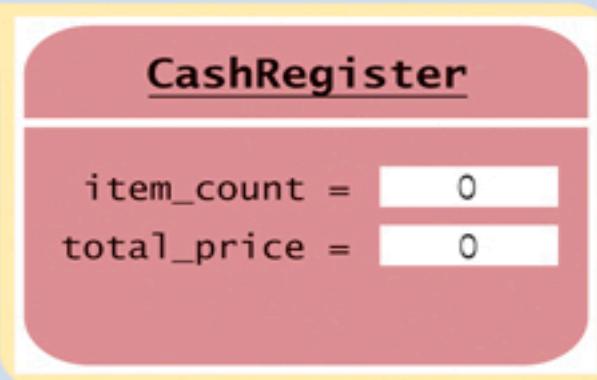
```
CashRegister register1;  
...  
register1.clear();  
...  
register1.add_item(1.95);
```

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

Mutators

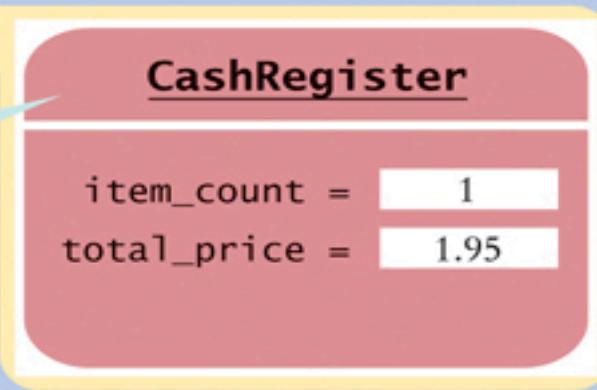
- 1 Before the member function call.

register1 =



- 2 After the member function call `register1.add_item(1.95)`.

register1 =



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.
 - get the total bill
 - get the item count

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;
```

Accessors functions return values. Mutators do not

Mutators and Accessors: The Interface

The interface for our class:



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;
};
```

Public section

Use CamelCase
for class names.

Member functions are declared in the class
and defined outside.

Mutator
member functions

Accessor
member functions

Private section

Mark accessors as const.

Data members
should always be private.

Be sure to include this semicolon.

Common 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

```
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;
};
```

Encapsulation

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

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

Encapsulation

```
register1 =
```

CashRegister

item_count = 1

total_price = 1.95

```
register2 =
```

CashRegister

item_count = 5

total_price = 17.25

Accessible
only by CashRegister
member functions

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.

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 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.

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.



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;
}
```

The diagram illustrates the scope of the `CashRegister::` prefix. An annotation box labeled "Not here" with an arrow points to the `public:` section, indicating that the prefix is not used there. Another annotation box labeled "Only here" with an arrow points to the `add_item` function definition, indicating that the prefix is used only in member function definitions.

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 - maybe from main():

```
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*.

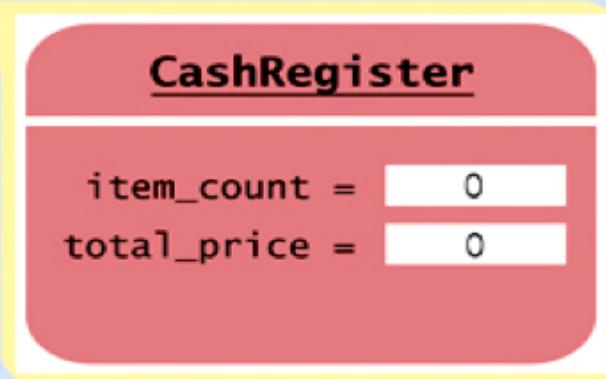
```
register1.add_item(1.95);  
  
void CashRegister::add_item(double price)  
{  
    implicit parameter.item_count++;  
    implicit parameter.total_price =  
        implicit parameter.total_price + price;  
}
```

The diagram illustrates the flow of the implicit parameter `register1`. A dotted arrow originates from the `register1` variable in the first line of code and points to the `add_item` function definition. Another dotted arrow originates from the `price` parameter in the function definition and points to its corresponding assignment statement in the body. A solid arrow originates from the `register1` variable in the first line of code and points directly to the `total_price` variable in the assignment statement.

Implicit Parameters

- 1 Before the member function call.

register1 =

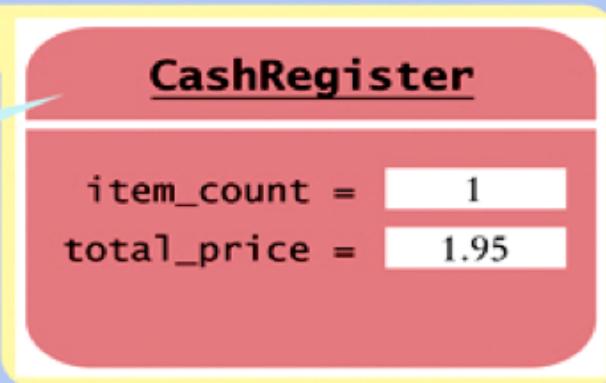


- 2 After the member function call `register1.add_item(1.95)`.

register1 =

Implicit parameter

Explicit parameter



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

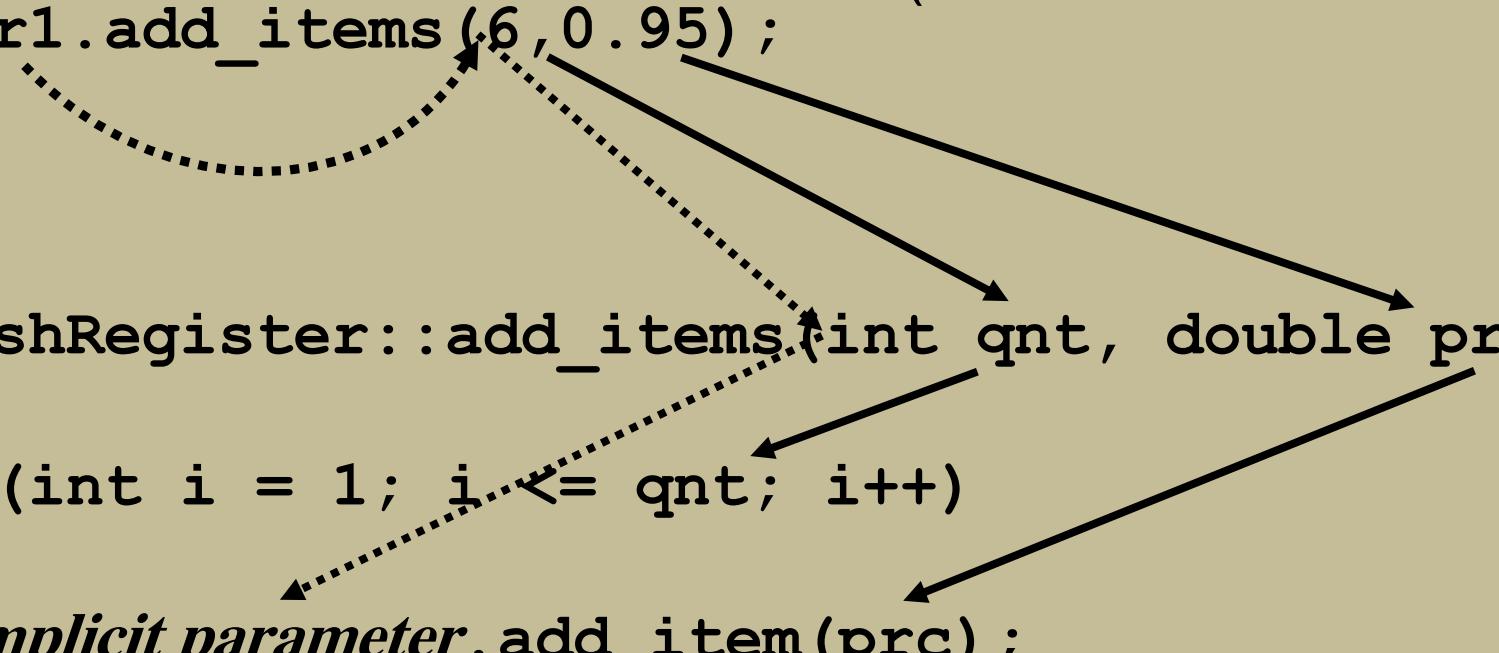
So how does this work?

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.

```
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

Explicit parameter

Use `const` for accessor functions.

The Cash Register Program

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

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

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

```
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;
}
```

registertest1.cpp

The Cash Register Program

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;
}
```

registertest1.cpp