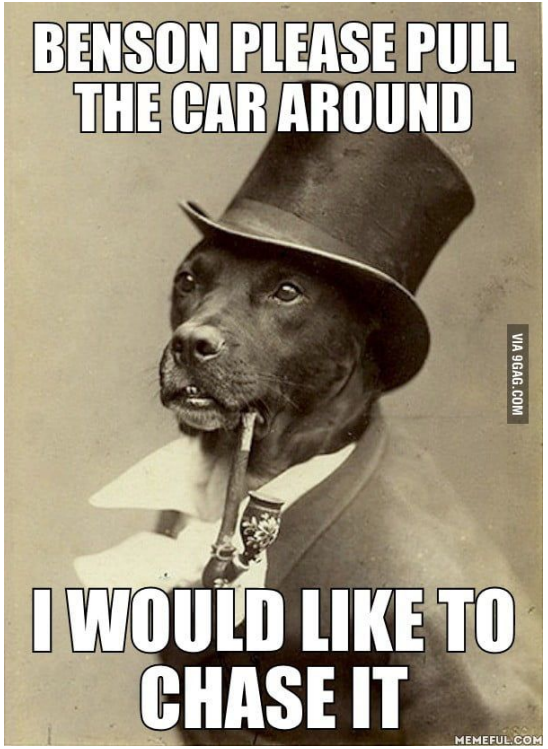
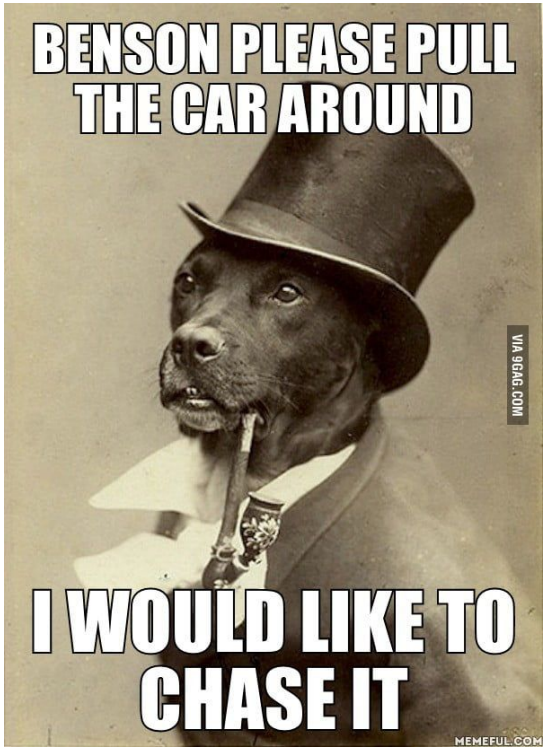
Lecture 22: Classes: ***Let's do this.***



Announcements and reminders

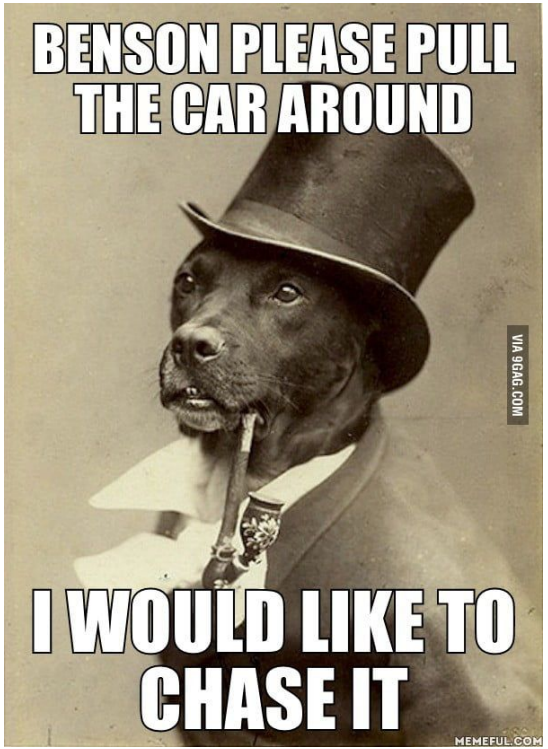
- HW7 posted, due Wednesday March 13, by 11 pm
- It's a lot of work. Friday, we'll talk specifics about ***how*** we can map what we're doing in here to the homework! And more Jedis.
- Mid-term TA FCQs -- let us know how things are going in recitation. It helps us help you! And that's just cute.



Last time on *Intro Computing…*

We saw…

- What is this **object-oriented programming** you speak of?
- What an ***object*** and a ***class*** is!
- What makes up a class and what is **encapsulation**!
  - Member functions (**public interface**)
  - Data members (private data - encapsulated)
- Different types of member functions
  - Getters (accessors) and setters (mutators)

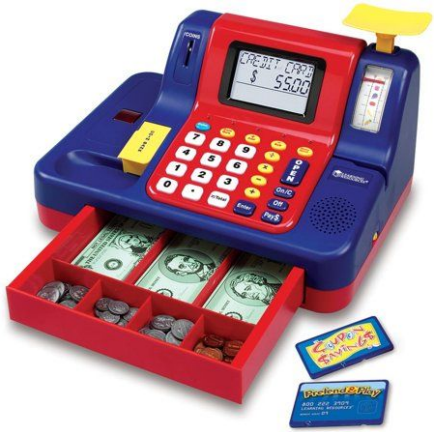


Now:

… how do we get classier, and implement these things?

Last time on *Intro Computing…*

```
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;
};
```



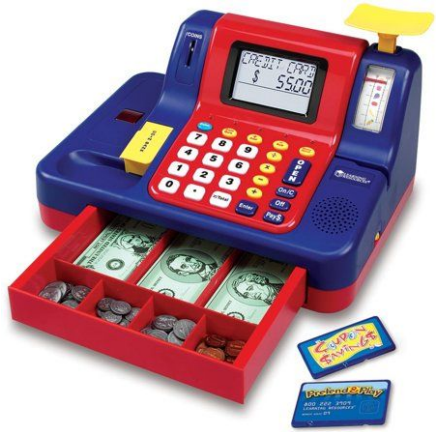
Now that we have the interface, we need to ***actually define*** what all this junk is!

→ start by **implementing the member functions**

Implementing the member functions

Start with the `add_item()` member function:

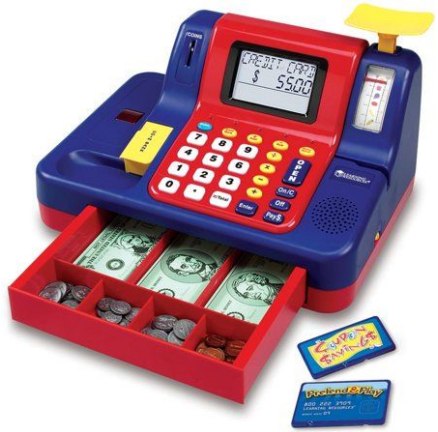
```
void add_item( double price ) {


}
```



Implementing the member functions

Start with the add_item() member function:

```
void add_item( double price ) {
    item_count++;           // added an item, so increment item counter
    total_price = total_price + price;  // added item price too
}
```

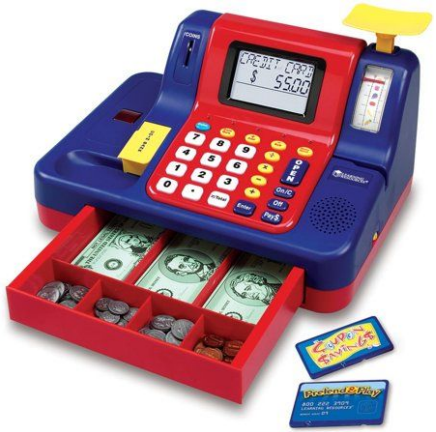


One more thing to add: as written, there is **no connection** to the CashRegister class!

Implementing the member functions

Start with the `add_item()` member function:

```
void CashRegister::add_item( double price ) {
    item_count++;           // added an item, so increment item counter
    total_price = total_price + price;  // added item price too
}
```



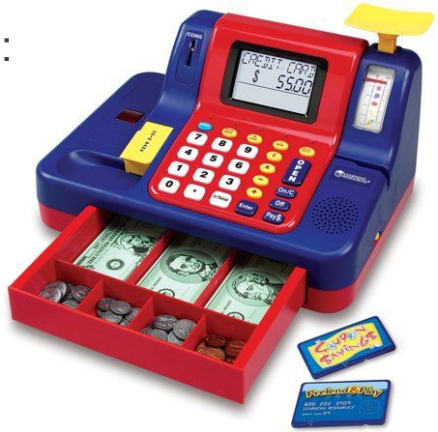
One more thing to add: as written, there is **no connection** to the CashRegister class!

→ so we specify for our member functions:

`CashRegister::[`*member function name*`]`

Implementing the member functions

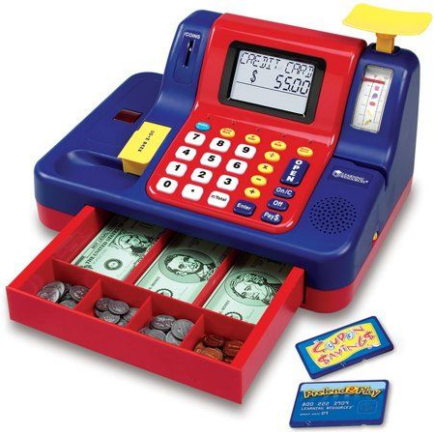
We do ***not*** need the `CashRegister::` declaration when defining the class:



```
class CashRegister {
public:
    ...
    void add_item( double price );
    ...
private:
    ...
};

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

Implicit parameters

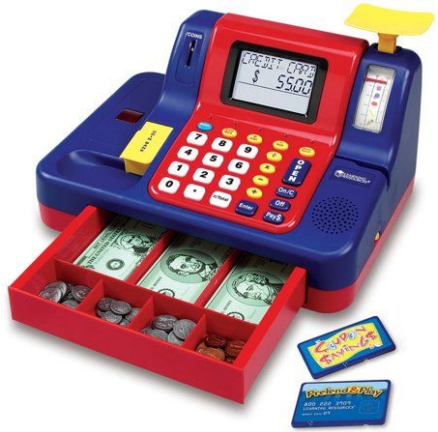
When we call `add_item( 1.95 )`, how does it know ***which*** `item_count` to increment, or which `total_price` to increase?



```
CashRegister register1, register2;
... [stuff happens] ...

register1.add_item( 1.95 );
```

Implicit parameters

When we call `add_item( 1.95 )`, how does it know ***which*** `item_count` to increment, or which `total_price` to increase?

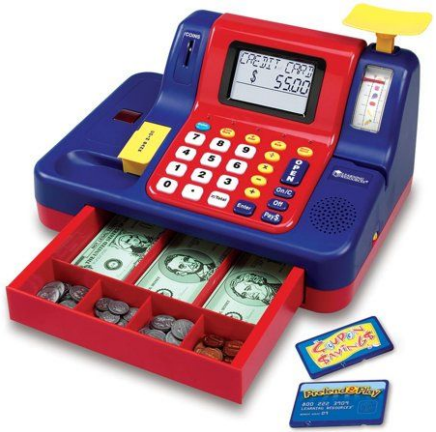


```
CashRegister register1, register2;
... [stuff happens] ...

register1.add_item( 1.95 );
```

`register1` → pass as an **implicit parameter** into the `add_item()` function

Implicit parameters

When we call add_item(1.95), how does it know ***which*** item_count to increment, or which total_price to increase?

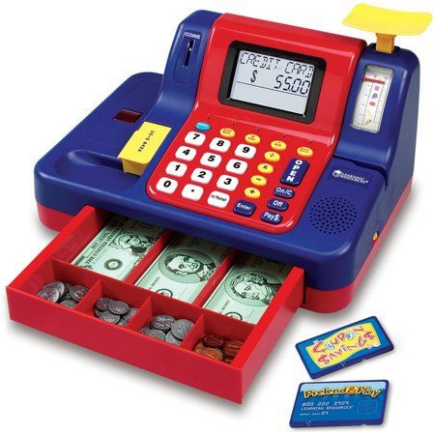


```
CashRegister register1, register2;
... [stuff happens] ...

register1.add_item( 1.95 );
```

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

Implicit parameters

When we call add_item(1.95), how does it know ***which*** item_count to increment, or which total_price to increase?



```
CashRegister register1, register2;
... [stuff happens] ...

register1.add_item( 1.95 );



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

Implicit parameters

When we call add_item(1.95), how does it know ***which*** item_count to increment, or which total_price to increase?

registerl.add_item(1.95) knows to add 1 item with price $1.95 to ***registerl*** the same way strl.length() knows to take the length of ***strl***

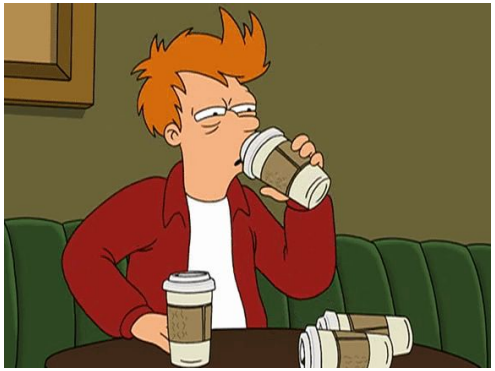
```
CashRegister registerl, register2;
... [stuff happens] ...

registerl.add_item( 1.95 );
```

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

We got member functions on member functions up in here

… Do the kids still say that?

Alright, nevermind.



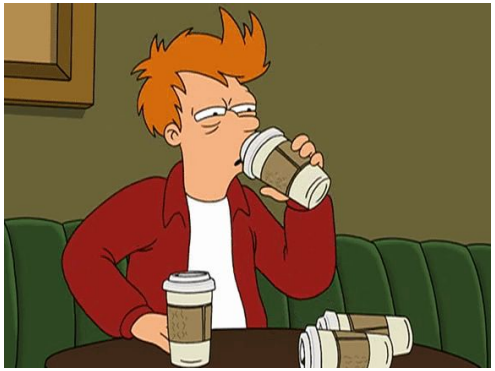
The situation: S'pose we have CSCI 1300 homework due soon, so we buy a dozen nice coffees to go.

… And maybe something for our friends too.

We got member functions on member functions up in here

… Do the kids still say that?

Alright, nevermind.



> **The situation:** S'pose we have CSCI 1300 homework due soon, so we buy a dozen nice coffees to go.
> … And maybe something for our friends too.

We ***could*** do…

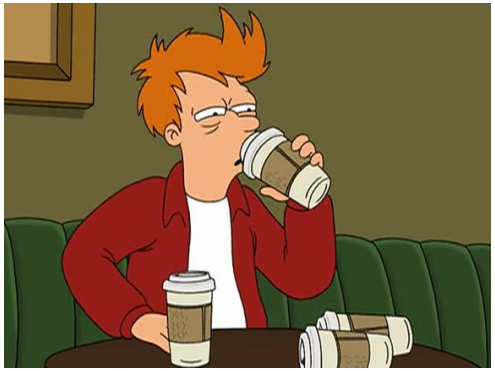
```
register1.add_item( 1.95 );
register1.add_item( 1.95 );
register1.add_item( 1.95 );
... (12 times) ...
```

… but any time you're sitting there hitting Ctrl+C; Ctrl+V over and over again, you start to feel: ***surely there is a better way!***

We got member functions on member functions up in here

We can define a ***new*** member function to add multiples of the same item:

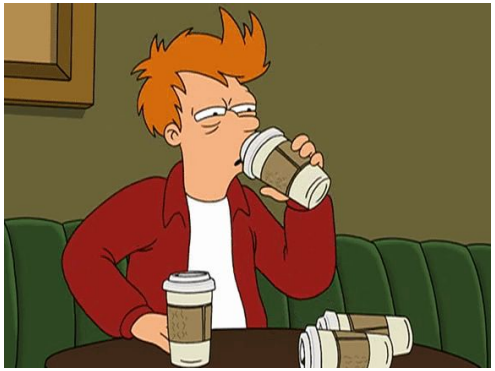
```
void CashRegister::add_items( int quant, double price ) {



}
```



We got member functions on member functions up in here

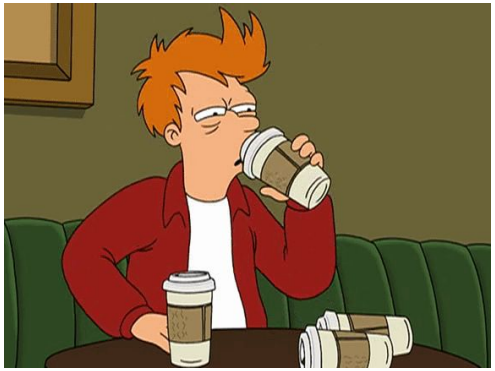
We can define a ***new*** member function to add multiples of the same item:

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



We got member functions on member functions up in here

We can define a ***new*** member function to add multiples of the same item:



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

Again, the `CashRegister` object is an **implicit parameter argument**

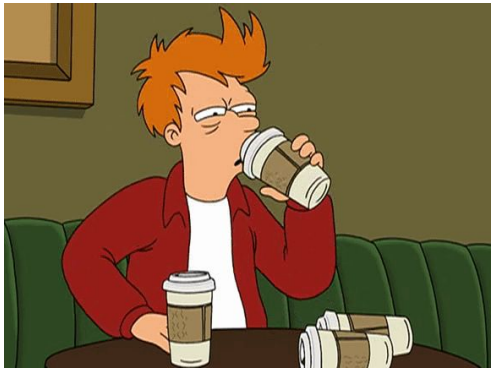
→ knows to `add_item()` to the object that `add_items()` is called on

```
register1.add_items(12, 1.95);
```
← knows to modify data members of `register1`, so we do not need to include `register1.add_item()`

What about the getter (accessor) member functions?

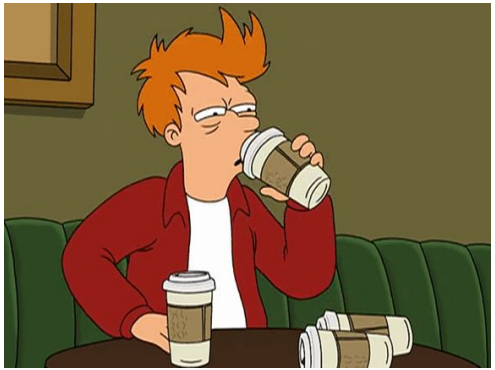
Let's write up the `get_count()` getter function:

```
get_count( )
{

}
```



What about the getter (accessor) member functions?

Let's write up the `get_count()` getter function:



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

Cool.

Often, we want to be able to display our objects' data members so we can debug, or just check the progress of a program.

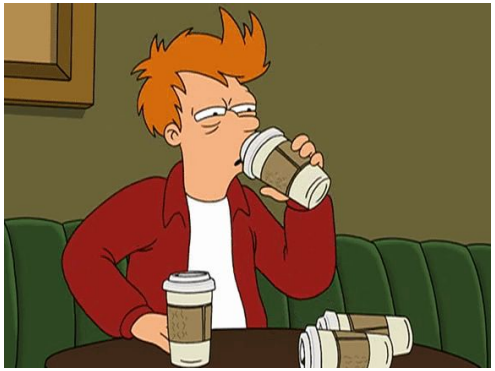
→ let's write a new member function to do that!

What about the getter (accessor) member functions?

Often, we want to be able to display our objects' data members so we can debug, or just check the progress of a program.

→ let's write a new function to do that!

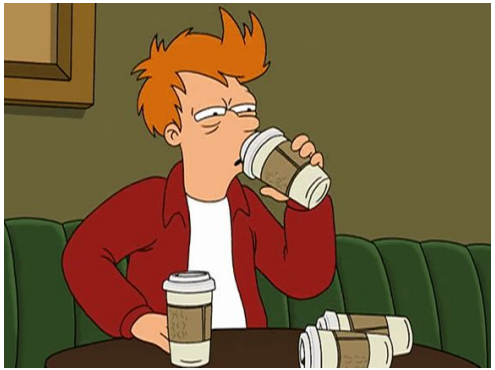
→ **will take a `CashRegister` variable as input parameter argument**



```
display( CashRegister reg ) {


}
```

What about the getter (accessor) member functions?

Often, we want to be able to display our objects' data members so we can debug, or just check the progress of a program.



→ let's write a new function to do that!

→ **will take a `CashRegister` variable as input parameter argument**

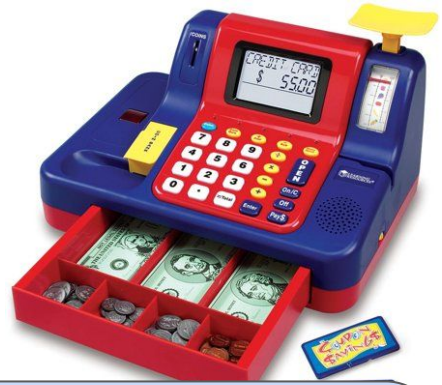
```
void display( CashRegister reg ) {
    cout << reg.get_count();
    cout << "  $" << fixed << setprecision(2) << reg.get_total() << endl;
}
```

Note: display() is ***not*** part of the CashRegister class!

→ So we need to include the dot notation `reg.get_count()` (for example) in order to use the member functions (just doing `get_count()` won't work)

Implementing the *other* member functions

```
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;
};
```

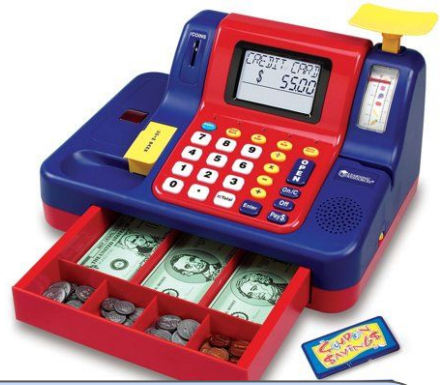


Cash registers in the wild typically start out an order with **item_count = 0** and **total_price = 0**, right?

→ Can we set up **CashRegister** objects to do this by default?

Implementing the *other* member functions

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



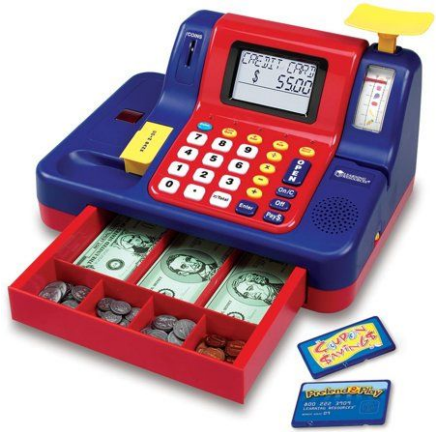
Cash registers in the wild typically start out an order with **item_count = 0** and **total_price = 0**, right?

→ Can we set up **CashRegister** objects to do this by default?

→ Totally! Special member function called a **constructor**

Implementing the *other* member functions

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



Now let's finish off defining **and testing** these member functions!

→ ***registerTest1.cpp***

What just happened?!

We just saw…

- How to define different types of member functions
  - Getters (accessors) and setters (mutators)
- How to access and mutate (get and set) data members from inside and outside of the class
  - That dot notation -- when and how to use it!