Topic 3

1. Object oriented programming
2. Implementing a simple class
3. <u>Specifying the public interface</u>
4. Designing the data representation
5. Member functions
6. Constructors
7. Problem solving: tracing objects
8. Problem solving: discovering classes
9. Separate compilation
10. Pointers to objects
11. Problem solving: patterns for object data

Specifying the Public Interface of a Class

We will design a cash register class, starting with the public interface. The interface consists of all member functions that a user of the class may need.

By observing a real cashier working, we realize we need 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.

Class Definition Syntax

To define a class you write:

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

`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
};
```

It is legal to declare the private members before the public section, but most programmers place the public section first.

It is also legal to have private functions and public data members, but these rarely are appropriate.

Member Functions: Accessors and Mutators

There are two kinds of member functions:

- Mutator: modifies the data members of the object. For example,

```
void clear();
```

- Accessor: does not modify data members. For example,

```
double get_total() const;
```

Figure 3 The Interface of the CashRegister Class

Accessors

This statement will print the current total:

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

Common Error: (Shown in small font, enlarge to see)

Can you find the error?

```
class MysteryClass
{
public:
    ...
private:
    ...
}  // ERROR: Forgot semicolon

int main()
{


    ...
}
```

Topic 4

1. Object oriented programming
2. Implementing a simple class
3. Specifying the public interface
4. Designing the data representation
5. Member functions
6. Constructors
7. Problem solving: tracing objects
8. Problem solving: discovering classes
9. Separate compilation
10. Pointers to objects
11. Problem solving: patterns for object data

Data Representation

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

Each **`CashRegister`** object has member functions `get_count` and `get_total`, so it must store the item count of the sale that is rung up.

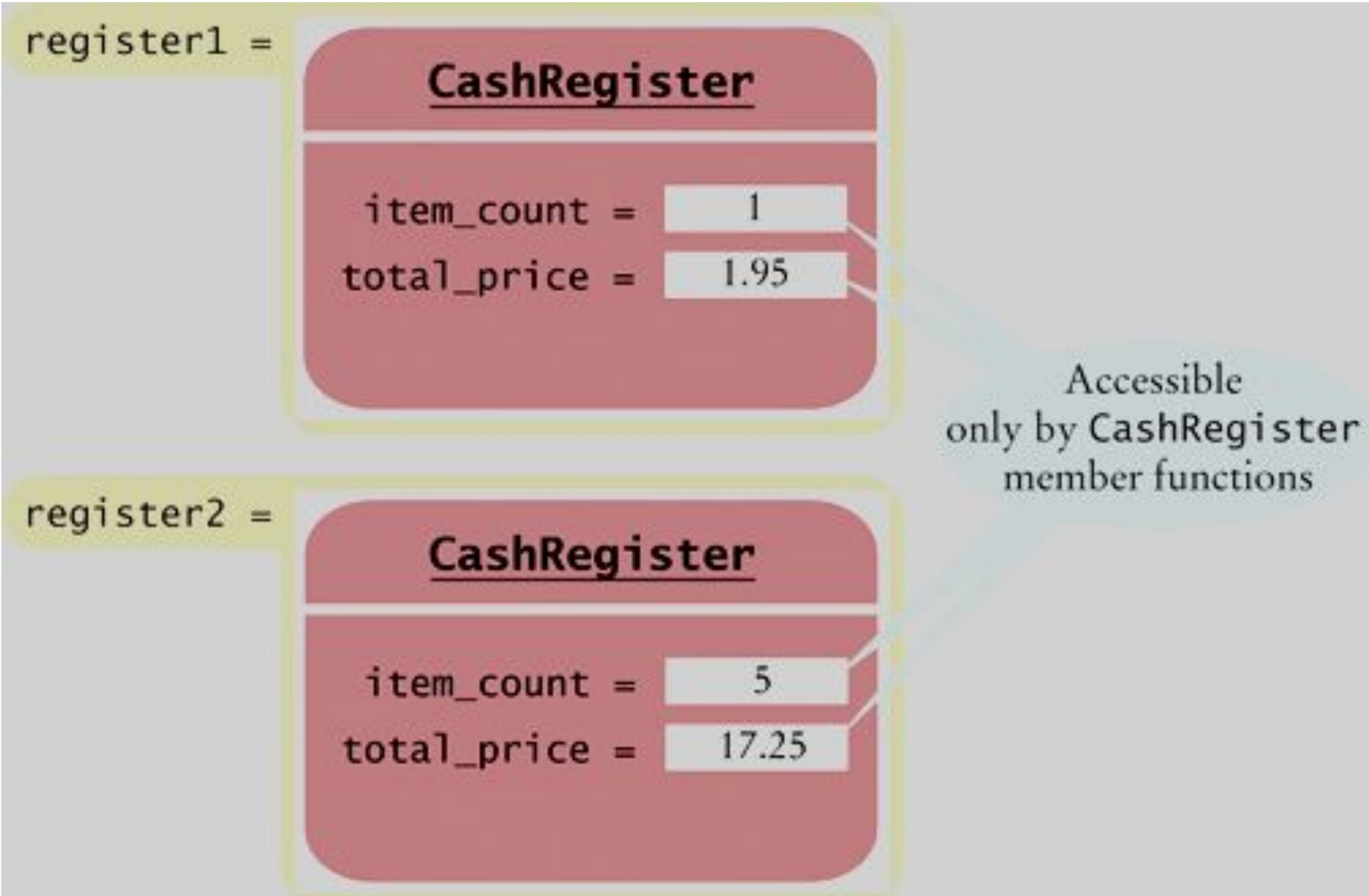
It must either store all entered prices (as a vector or array) and compute the total in the function call, or it must store the total.

Since the latter is simpler and adequate, we'll just store the total.

The Complete Cash Register Interface, with Data

```
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;
};
```

Example of Two CashRegister Objects with Data Members



Encapsulation Motivation

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

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

The encapsulation mechanism guarantees:

1. 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.

2. If we need to change or improve implementation details later, these should not affect users of the public class interface.

Topic 5

1. Object oriented programming
2. Implementing a simple class
3. Specifying the public interface
4. Designing the data representation
5. Member functions
6. Constructors
7. Problem solving: tracing objects
8. Problem solving: discovering classes
9. Separate compilation
10. Pointers to objects
11. Problem solving: patterns for object data

Implementing the Member Functions

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

Implementing the member functions.

NOT a Member Function

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

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 unless we prefix the function name in the header with

`CashRegister::`

Member Functions

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

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

/* NOTE that we do NOT declare the item_count or
total_price variables in the member functions –
they only get declared in the Class interface
definition */
```

Implicit Parameters

In the member function call (in `main`):

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

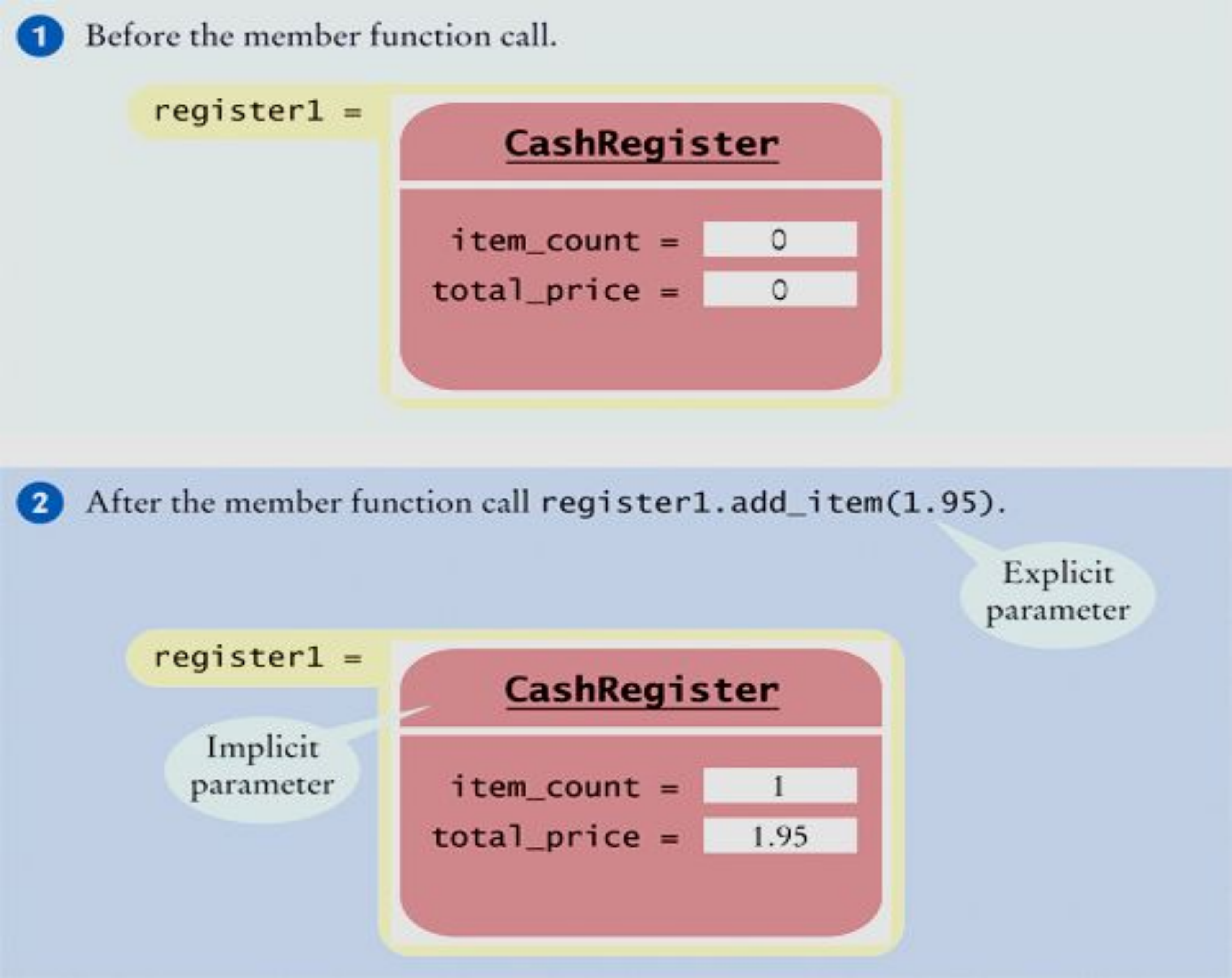
The variable **`register1`** is an *implicit parameter to the member function*. But you don't include it in your code:

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

Whenever a member function accesses a variable in the Class's data, the compiler automatically includes the implicit parameter and a dot (shown fictitiously in ***italics*** below):

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

Implicit Parameters vs. Explicit



Calling a Member Function from a Member Function

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

Let's add a member function to add multiple copies of the same item to the total. This new function calls the single-unit function via a loop:

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

Calling a Member Function from Another: no Dot

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

And, of course, the object remains an implicit parameter for both functions.

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

The Cash Register Program, Part 1

```
// sec05/cashregister.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:
   /**
      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, Part 2

```
    /**
       @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 , Part 3

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

The Cash Register Program, Part 4 (NOT a member function)

```
/**
   Displays the item count and total
   price of a cash register.
   @param reg the cash register to display
 NOT a member function of the class !!
 So the CashRegister must be passed as an
   explicit parameter - is not implicit.
*/
void display(CashRegister reg)
{
   cout << reg.get_count() << " $"
      << fixed << setprecision(2)
      << reg.get_total() << endl;
}
```

The Cash Register Program, `main()` and the output

```
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;
}
```

Program Run Output:

```
Item 1: $1.95
Item 2: $2.90
Item 3: $5.40
```

Practice It: The CashRegister

- Trace through the function calls of `main()`, filling in this diagram of the values of register1's data members:

```
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;
}
```

total_price	item_count

Programming Tip: `const` Correctness (1)

You should declare all accessor functions with the const reserved word.

For example, suppose you write:

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

When you compile your code, no error is reported.

Programming Tip: `const` Correctness (2)

But suppose that another programmer uses your `CashRegister` class in a function:

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

For efficiency, that programmer declared the `registers[]` parameter as `const`.

But the call `registers[i].display()` will not compile. Because `CashRegister::display` is not tagged as `const`, the compiler suspects that the call may modify `registers[i]`.