# Topic 10

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

Pointers to Objects
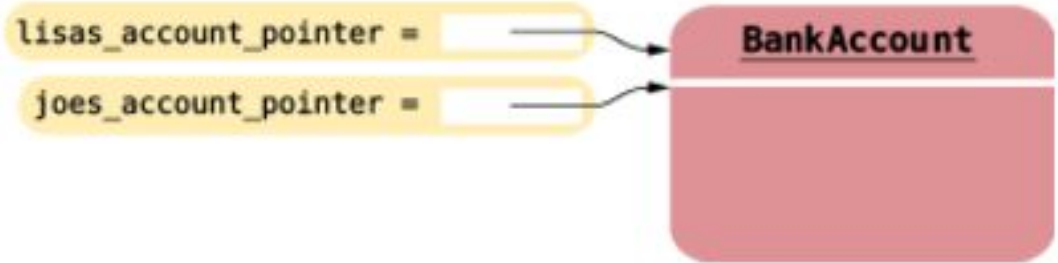
Pointers to objects permit us to dynamically allocate them and to share objects.

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

The `new` operator returns a pointer to the allocated object.

Now we can copy the pointer, without copying the object:

```
BankAccount* joes_account_pointer =
lisas_account_pointer;
```

Remember to call the `delete` operator on the pointer before exiting the program, to reclaim the dynamic memory:

```
delete joes_account_pointer;
```

Pointers and the -> Operator

Given:

```
CashRegister* register_pointer = new CashRegister;
```

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

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

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
```

The preferred syntax is the arrow operator:

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

The `this` Pointer

Each member function of every class has a built-in parameter, called ***this***, a pointer to the *implicit parameter*. (The object)

If you call

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

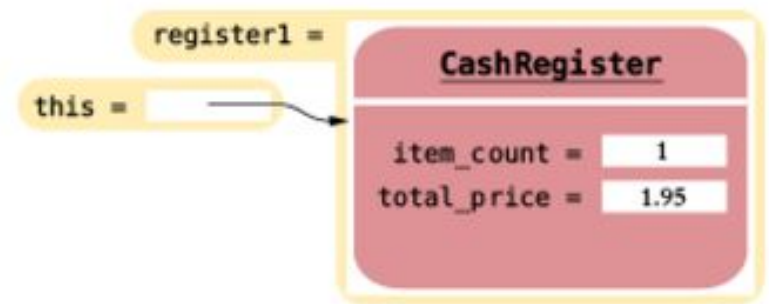then the **this** pointer has type **CashRegister*** and points to the **register1** object.

register1 =

this =

CashRegister

item_count = 1

total_price = 1.95

Figure 11 The this Pointer

The `this` Pointer: Example

You can use the `this` pointer inside the definition of a member function. For example,

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

However, you don't really need the `this` pointer in this case, as you saw in previous versions of the function.

The `this` Pointer: Motivation

The real reason to use the `this` pointer is to clarify duplicate variable names, as either the data members or other explicit parameters to a function. For example, we might add a constructor to the class that carries over the previous day's **total_price**:

```
CashRegister::CashRegister(double total_price)
{
    item_count =0;
    this->total_price = total_price;
}
```

However, we recommend just giving the explicit parameter a different name (such as **initial_total_price**), to avoid the confusion that the above code might create.

Practice It: Object Pointers

Given:

```
int n; string* p = nullptr;
```
Write the statements to implement the tasks.

Task	Statement	Explanation
Dynamically allocates a string object and save the address in the pointer variable p.	p = new string;	The new operator allocates a new object from the free store and returns its address.
Deallocates the object that was allocated in the previous task.	delete p;	The delete operator deallocates the memory block with the given address.
Dynamically allocate a string object with contents "Hi"and save the address in the pointer variable p.	p = new string("Hi");	You need to call a constructor to initialize the string object.
Invoke the length member function on the object that was allocated in the previous step and save the result in the integer variable n.	n = p->length();	Use the -> operator to call a member function on a pointer to an object.

Practice It: Object Pointers and `this`

Write the constructor implementation of a Point class:

```cpp
class Point
{
public:
    Point(int x, int y);
    // Member functions omitted
private:
    int x;
    int y;
};


Point::Point(int x, int y)
{

    _____

    _____

}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Keeping a Total

- Many classes need to track a quantity that can go up or down as functions are called. Keep a data member that represents the current total.

```
double total_price;
```

- Examples of what must be done to the total:

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

Counting Events

For classes that need to count events, some examples:
```
int item_count;   //Keep a counter
```

Increment the counter in those member functions that correspond to the events that you want to count.
```
void CashRegister::add_item(double price)
{
    total_price = total_price + price;
    item_count++;
}
```
You may need to clear the counter:
```
void CashRegister::clear()
{
    total_price = 0;
    item_count = 0;
}
```
There may or may not be a member function that reports the count to the class user.

Collecting Values

Some objects collect numbers, strings, or other objects. For example, each multiple-choice question has a number of choices. A cash register may need to store all prices of the current sale. Use an array to store these, and a member function to add to the list:

```cpp
class Question
{
   . . .
private:
   string choices[5];

   . . .
};
void Question::set_choice(int pos, string choice)
{
   choices[pos] = choice;
}
```

Managing Properties of an Object

A property is a value of an object that an object user can set and retrieve. For example, a Student object may have a name and an ID.

An object property needs a getter and setter member functions. Provide a data member to store the property's value:

```cpp
class Student
{
public:
    Student(string name_, int id_);
    string get_name() const;
    int get_id() const;
    void set_name(string new_name);
private:
    string name;
    int id;
};

void Student::set_name(string new_name) // Includes Error checking
{
    if (new_name.length() > 0) { name = new_name; }
}
```

Some properties should not change after initialization in the constructor, and thus need no setter function. For example, a student's ID.

Modeling Objects with Distinct States(1)

Some objects' behavior depends on what happened in the past. For example, a Fish object may look for food when it is hungry and ignore food after it has eaten.

Such an object needs a state variable to remember whether it has recently eaten.

Supply a data member that models the state, together with some constants for the state values:

```cpp
class Fish
{
public:
    const int NOT_HUNGRY = 0;
    const int SOMEWHAT_HUNGRY = 1;
    const int VERY_HUNGRY = 2;
    void eat();
    void move();

private:
    int hungry = NOT_HUNGRY;
};
```

Determine which member functions change the state. If a fish has just eaten food, it won't be hungry. But as the fish moves, it will get hungrier.

```cpp
void Fish::eat()
{
    hungry = NOT_HUNGRY;
    . . .
}
void Fish::move()
{
    if (hungry == VERY_HUNGRY)
    {
        cout << "Looking for food" << endl;
    } else . . .
    if (hungry < VERY_HUNGRY) { hungry++; }
}
```

Finally, determine where the state affects behavior. A fish that is very hungry will want to look for food first.

Describing the Position of an Object

- Some objects move around during their lifetime, and they remember their current position. For example,
    - A train drives along a track and keeps track of the distance from the terminus.
    - A bug living on a grid crawls from one grid location to the next, in one of 4 directions.
- If the object moves along a line, represent the position as a distance.

    ```
    double distance_from_terminus;
    ```
- If the object moves in a grid, remember its current location and direction in the grid:

    ```
    int row, column;
    int direction; // 0 = North, 1 = East, 2 = South, 3 = West
    ```
- When you model a physical object such as a cannonball, you need to track both the position and the velocity, possibly in 2 or 3 dimensions.

    ```
    double z_position, z_velocity;
    ```
- There will be member functions that update the position. In the simplest case, you may be told by how much the object moves:

    ```
    void Train::move(double distance_moved)
    {
        distance_from_terminus = distance_from_terminus +
                                                distance_moved;
    }
    ```

Understand the concepts of objects and classes.

•A class describes a set of objects with the same behavior. An object is a collection of related data plus member functions that manipulate the data.

•Every class has a public interface: a collection of member functions through which the objects of the class can be manipulated.

•Encapsulation is the act of providing a public interface and private data and function internals, hiding implementation details.

•Encapsulation enables changes in the implementation without affecting users of a class.

Understand data members and member functions of a simple class.

- The member functions of a class define the behavior of its objects.

- An object's data members represent the state of the object.

- Each object of a class has its own set of data members.

- A member function can access the data members of the object on which it acts.

- A private data member can only be accessed by the member functions of its own class.

Formulate the public interface of a class in C++.

•You can use member function declarations and function comments to specify the public interface of a class.

•A mutator member function changes the object on which it operates.

•An accessor member function does not change the object on which it operates. Use const with accessors.

Choose data members to represent the state of an object.

•An object holds data members that are accessed by member functions.

•Private data members can only be accessed by member functions of the same class, and we usually make all data private.

Implement member functions of a class.

•Use the `ClassName::` prefix when defining member functions.

•The implicit parameter `this` is a pointer to the object on which a member function is applied.

•Explicit parameters of a member function are listed in the function definition.

•When calling another member function on the same object, do not use the dot notation.

```cpp
void CashRegister::add_items(int qnt, double prc)
{
    for (int i = 1; i <= qnt; i++)
    {
        add_item(prc); //calling another member function
    }
}
```

Design and implement constructors.

•A constructor is called automatically when an object is created.

•The name of a constructor is the same as the class name.

•A default constructor has no arguments.

•A class can have multiple constructors. ("overloaded")

•The compiler picks the constructor that matches the arguments.

•Be sure to initialize all number and pointer data members in a constructor.

```cpp
class BankAccount
{
public:
    // "Default" constructor: Sets balance=0
    BankAccount();
    // Sets balance to initial_balance
    BankAccount(double initial_balance);
    // . . . Member functions omitted
private:
    double balance;
};
```

Use the technique of object tracing for visualizing object behavior.

•Write the member functions on the front of a card, and the data member values on the back.

•Update the values of the data members when a mutator member function is called.

CashRegister reg1

clear
add_item(price)
get_total
get_count

item_count | total_price
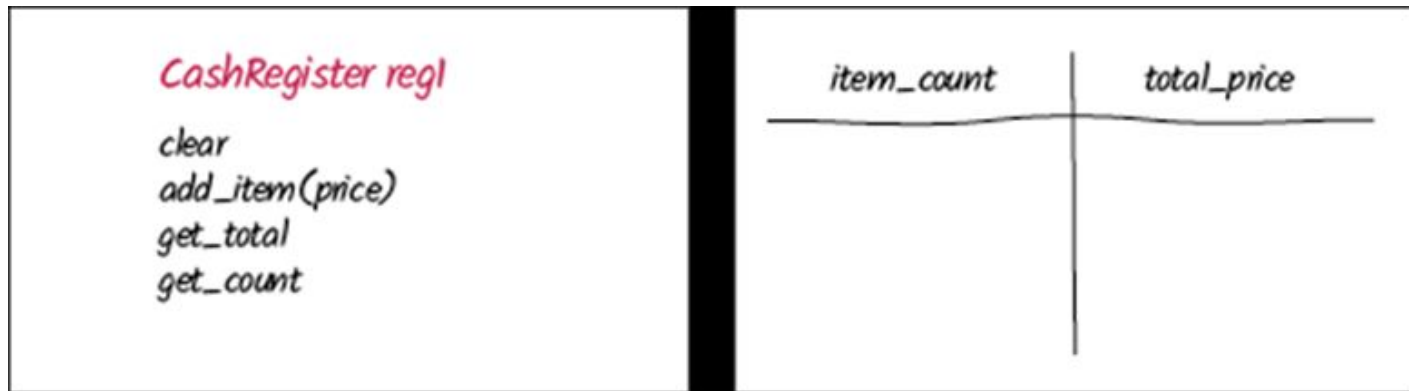
Discover classes that are needed for solving a programming problem.

•To discover classes, look for nouns in the problem description.

•Concepts from the problem domain are good candidates for classes.

•Verbs in the description will inspire member functions required to manipulate the class data.

•A class aggregates another if its objects contain objects of the other class.

Separate the interface and implementation of a class in header and source files.

•The code of complex programs is distributed over multiple files.

•Header files contain the definitions of classes and declarations of nonmember functions.

•Source files contain function implementations.
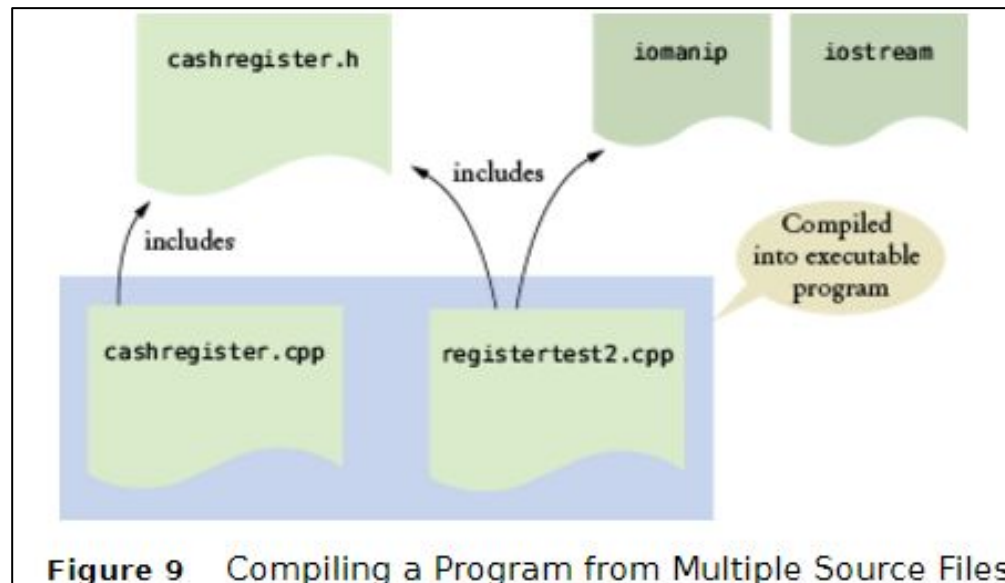

Figure 9 Compiling a Program from Multiple Source Files

Big C++ by Cay Horstmann

Use pointers to objects and manage dynamically allocated objects.

•Use the `new` operator to obtain an object that is located on the free store.

•The `new` operator returns a pointer to the allocated object.

•When an object allocated on the free store is no longer needed, use the `delete` operator to reclaim its memory.

•Use the `->` operator to invoke a member function through a pointer, or to access a private data member from the "`this`" pointer

•In a member function, the `this` pointer points to the implicit parameter.

Use patterns to design the data representation of a class.

•An data member for the total is updated in member functions that increase or decrease the total amount.

•A counter that counts events is incremented in member functions that correspond to the events.

•An object can collect other objects in an array.

•An object property can be accessed with a getter member function and changed with a setter member function.

•If your object can have one of several states that affect the behavior, supply a data member for the current state.

•To model a moving object, you need to store and update its position.