© Cameron Strathdee/iStockphoto.

Chapter Nine: Classes

Chapter Goals

- To understand the concept of encapsulation
- To master the separation of interface and implementation
- To be able to implement your own classes
- To understand how constructors and member functions act on objects
- To discover appropriate classes for solving programming problems
- To distribute a program over multiple source files

Topic 1

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

Object-Oriented Programming

- You have learned to structure programs into functions.
  - This is an excellent practice, but not good enough.
  - As programs get larger, it becomes increasingly difficult to maintain all the functions and separate datasets.

- To solve this problem, computer scientists invented **object-oriented programming**
  - tasks are solved by collaborating objects.
  - An object is a set of data plus functions that manipulate the data
  - A "`class`" is a blueprint or template for an object with data members and member functions.

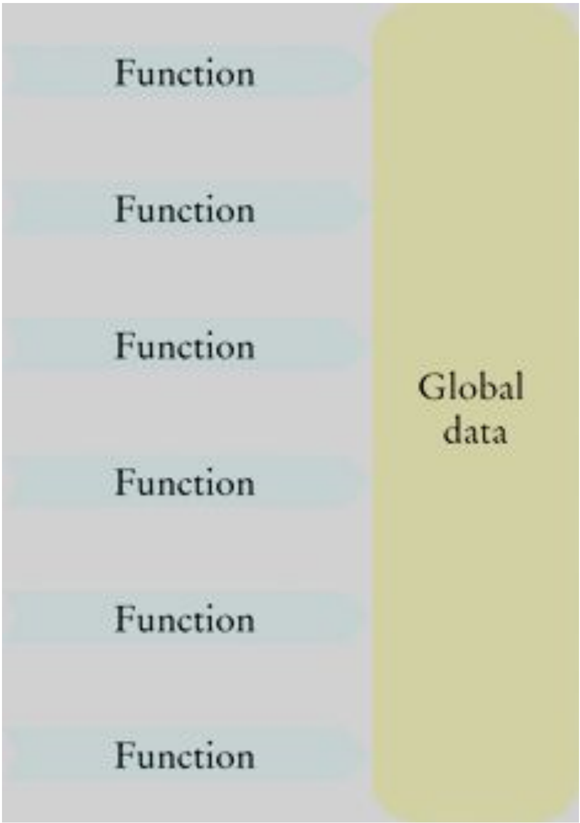
- Did you know that you already are an Object Oriented Programmer?
  - `string,cin,cout,streams` are all classes or objects

The Problem with Functional Programming

Functional programming is what you have done (mostly) so far, with a bunch of functions operating on a bunch of data, linked together only by your documentation and planning.

When some part of the data needs to be changed:

to improve performance
or to add new capabilities,
a large number of functions
will have to be modified, *and there is no mechanism to ensure correctness*



Objects to the Rescue

Computer scientists noticed that functions work on related data so they invented:

Objects

where data and the functions that work with them are bundled together.

The C++ language syntax rules guarantee that changes to the class (object) data structure will be matched by changes in the built-in functions.

And these changes are "under the hood", hidden from users of your code. This hiding is known as "encapsulation".

Object Terminology

Some new terminology.

The data stored in an object are called:

data members

The functions that work on data members are:

member functions

The list of member functions is the *public interface* of the class.

Encapsulation and the Interface

When you use **`string`** or **`stream`** objects, you did not know their data members.

Encapsulation means that they are hidden from you.

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.

Classes

A `class` describes a set of objects with the same behavior.

You would create the **`Car`** `class` to represent cars as objects.

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

Topic 2

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 a Simple Class

- Let's make a class that models a *tally counter*
  - mechanical device that is used to count
    - for example, to find out how many people board a bus
- When the operator pushes a button, the counter value advances by one.
  - We model this operation with a count function.
- A counter has a display to show the current value
  - we use a `get_value` function instead.
- A counter has another button to reset the count to 0
  - We use a `reset` function to model it.

Code for the Tally Counter Class: Interface

- To define the structure of a fully fledged class, we use syntax very similar to what we used with data-only classes.

```
class Counter
{
public:
   void reset();
   void count();
   int get_value() const;
private:
   int value;
};
```

- In the **`public:`** area are the function prototype statements.
  - These are the "interface" of the class that can be used in **`main`**
- In the **`private:`** area are the data members
- By convention, we name our classes starting with a Capital letter

  `CamelCase`

Code for the Tally Counter Class: Functions

- We define the member functions immediately after the interface
  - They must be denoted as member functions by prefixing the function name with the class name followed by 2 colons:

```
void Counter::count()
{
    value++;
}
void Counter::reset()
{
    value = 0;
}
int Counter::get_value() const
{
    return value;
}
```

- The `get_value()` member function is required so that users can know the count
  - Users are NOT PERMITTED to access the `private: value` variable
    - Only member functions can access private data

Code for the Tally Counter Class: `main`

```
int main() //define and use 2 Counter objects to test class
{
   Counter tally;
   tally.reset();
   tally.count();
   tally.count();
   int result = tally.get_value();
   cout << "Value of tally: " << result << endl;
   tally.count();
   tally.count();
   result = tally.get_value();
   cout << "Value of tally: " << result << endl;

   Counter concert_counter;
   concert_counter.reset();
   concert_counter.count();
   concert_counter.count();
   concert_counter.count();
   result = concert_counter.get_value();
   cout << "Value of concert_counter: " << result << endl;
   return 0;
}
```

Class Debrief

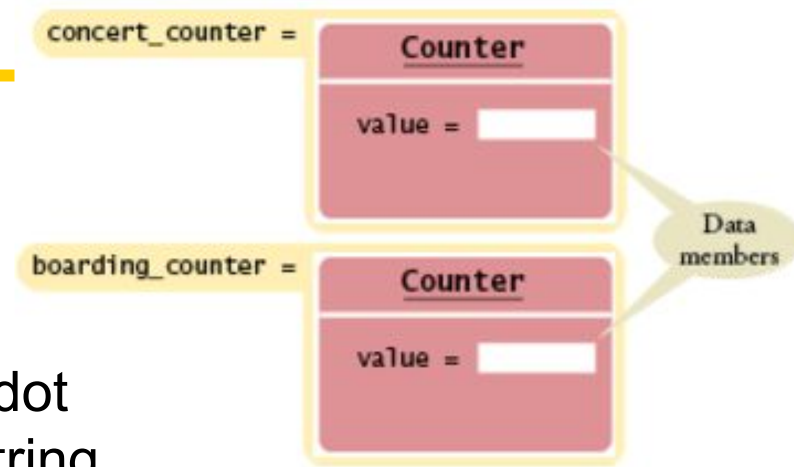


Figure 2 Data Members

- Each object has its own private data members
  - As shown in Figure 2
- Member functions are called with the dot notation, just like they were with the string classes

```
tally.reset();
concert_counter.reset();
concert_counter.count();
```

- Member functions which do not modify data have the word const as the last word of their prototype
  - **`int Counter::get_value() const`**
  - These are called "accessor" functions

Practice It: A Bug Class

Fill in the code below for a `class Bug`, to model a bug climbing a pole.

- Each time the `up()` member function is called, the bug climbs 10 cm.
  - Whenever it reaches the top of the pole (at 100 cm), it slides back to the bottom.
- Also implement a member function `reset()` that starts the Bug at the bottom
- and a member function `int get_position` that returns the current position
- See the textbook, Ch. 9 Section 2 Self-check 4, for the `main()` code to test the class.

Practice It: A Bug Class

```
#include <iostream>
using namespace std;
class Bug
{
public:
   ...
private:
   int position = 0;
};

int Bug::get_position() const
{   ...}

void Bug::reset()
{   ...}

void Bug::up() // bug climbs 10 cm, and @ 100,
{   ...}       // resets back to position 0
```