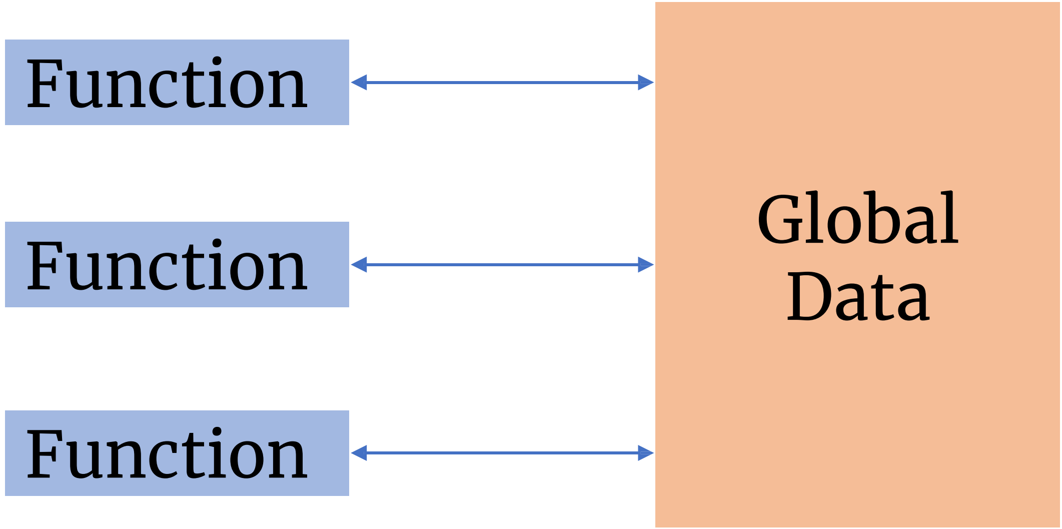
Object-Oriented Programming (OOP) and Classes

Functional Programming

- ***Functional/procedural programming*** is what we have done so far:

  ...a bunch of functions operating on a bunch of data, linked together only by your documentation and planning.



Object-Oriented Programming

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

- **To solve this, computer scientists introduced object-oriented programming (OOP) paradigm.**
    - Tasks are solved by collaborating **objects**
    - An **object** is a set of data + functions that manipulate this data
    - A **class** is a user-defined blueprint or template for an object

Object-Oriented Programming

Object-oriented programming has several advantages over functional programming:

- ★ OOP is faster and easier to execute
- ★ OOP provides a clear structure for the programs
- ★ OOP makes the code easier to maintain, modify and debug
- ★ OOP helps get rid of repetitive code
- ★ OOP makes it possible to create full reusable applications with less code and shorter development time

Object-Oriented Programming

- Everything in C++ is associated with classes and objects, along with its attributes and methods.
  - In real life, **a car is an object.** The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

- Attributes and methods are variables and functions that belong to the class. These are often referred to as "class members".

Classes

- A class is a user defined data type
- Describes a set of objects with the same behavior

Classes

- A class is a template for objects & an object is an instance of a class

Class	**Object**
Fruit	The particular apple you had for b. The rotten orange in the compost
Car	My 2007 Toyota Camry A broken down Ford Escape

Classes

- Most **functions** work on relative data
- **Objects** can be used to work with functions and data together

- A **class** describes a set of objects with the same behavior.
    - e.g. the string class describes the behavior of all strings

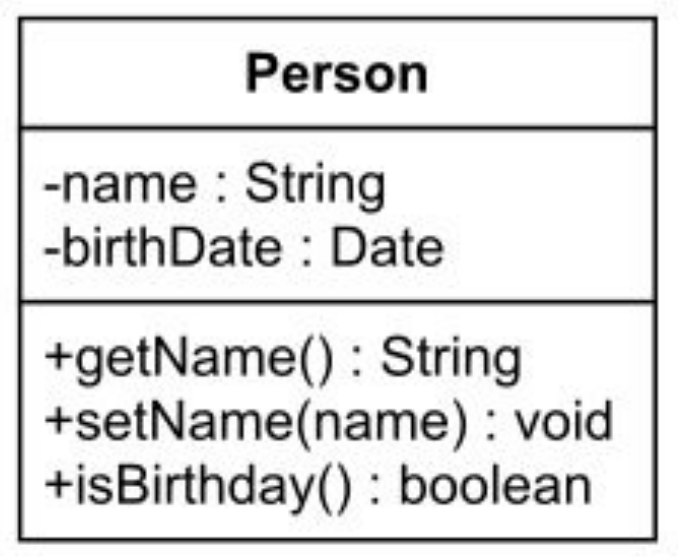
- We use **objects** which store data
    - e.g. *string s*

- We use member functions that act upon the data
    - e.g. str.length(), str.find(...);

Class: Examples

http://www.cs.utsa.edu/~cs3443/uml/uml.html

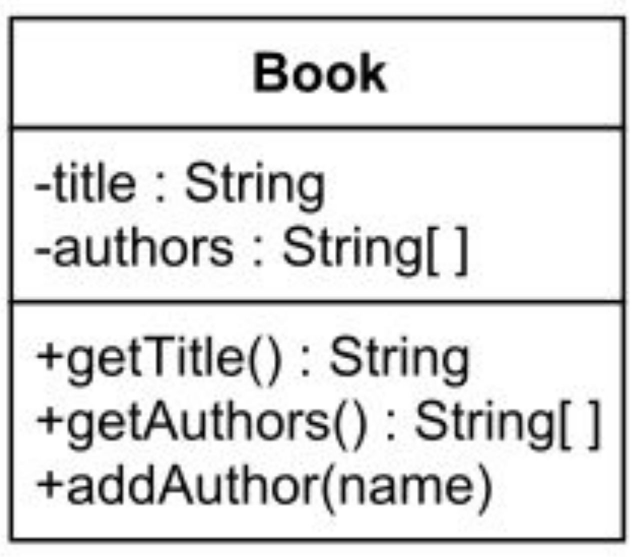
the "-" symbol denotes PRIVATE members and the "+" symbol denotes PUBLIC members



Person

-name : String
-birthDate : Date

+getName() : String
+setName(name) : void
+isBirthday() : boolean

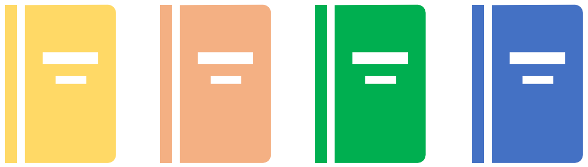


Book

-title : String
-authors : String[ ]

+getTitle() : String
+getAuthors() : String[ ]
+addAuthor(name)

...but more on that later



Implementing a simple class

```
class NameOfClass {

    public:
        // public interface

    private:
        // the data members, private interface

};
```

ONLY difference between structure and class are: access specifier defaults to **private** for class and **public** for struct.

Public vs Private

- **Public** and **Private** are referred to as access specifiers.
  - Sets the accessibility of the class members

- **Public** members (data, functions) can be accessed by anyone (from anywhere).
- **Private** data members can *only* be accessed by the member functions of its own class.
  - They allow a programmer to hide the implementation of a class from a class user.

Classes: A simple example

```
class X {
    public:
        int a;

        int func(int v) {
            return a + v;
        }
};
```

```
int main()
{
        X var;
        var.a = 7;
        int x = var.func(9);

        return 0;
}
```

Implementing a simple class

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

Member function (prototypes)

data member (should *always** be private)

Don’t forget the semicolon

Example17A

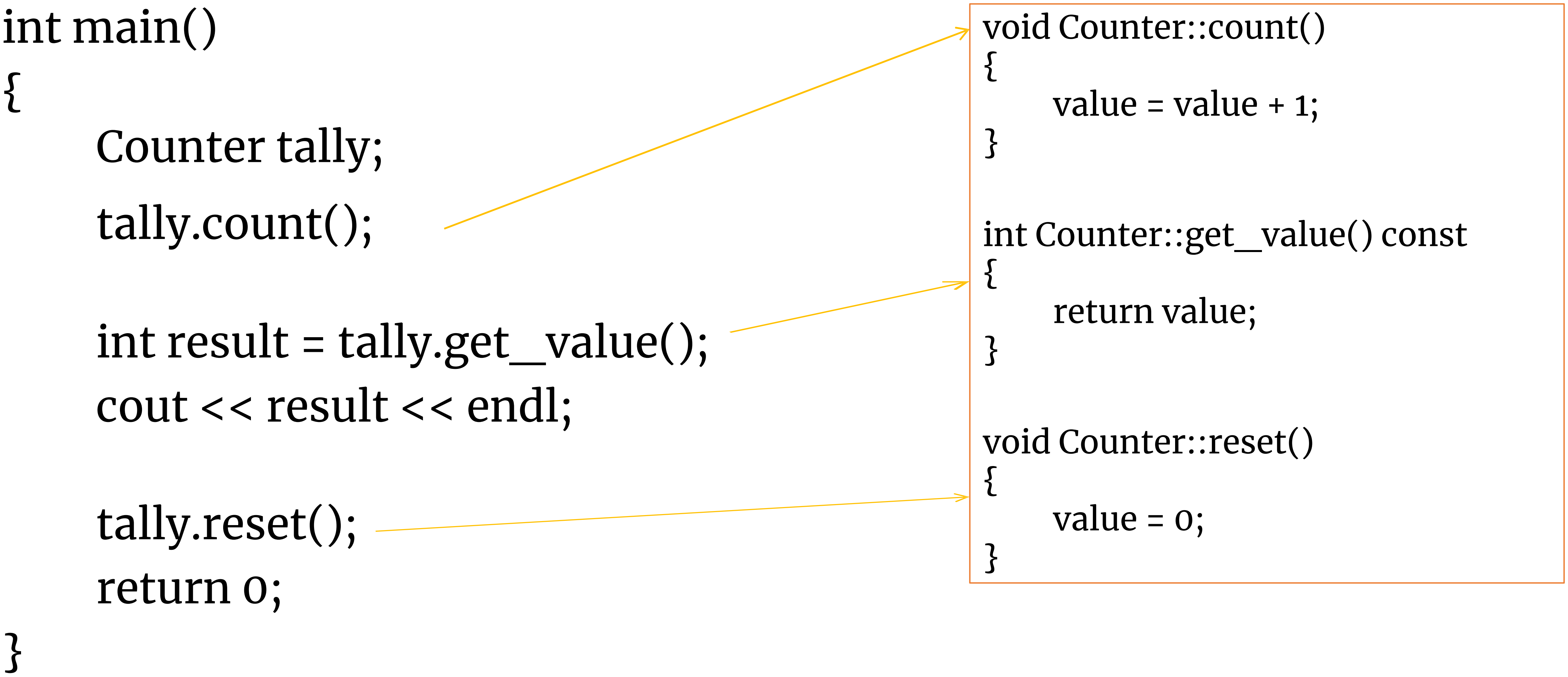
Implementing a simple class

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

Use the Counter:: prefix to indicate that we're defining the count function of the Counter class.

```
void Counter::count()
{
        value = value + 1;
}
```

Implementing a simple class



```
int main()
{
    Counter tally;
    tally.count();

    int result = tally.get_value();
    cout << result << endl;

    tally.reset();
    return 0;
}
```

```
void Counter::count()
{
    value = value + 1;
}

int Counter::get_value() const
{
    return value;
}

void Counter::reset()
{
    value = 0;
}
```

Aside: const keyword after function

```
void Counter::count()
{
	value = value + 1;
}

int Counter::get_value() const
{
	return value;
}

void Counter::reset()
{
	value = 0;
}
```

const member **function is** a member **function** that is guaranteed to **not modify the object** or call any non-const member functions (as they may modify the object).

Implementing a simple class

```
int main()
{
    Counter tally;
    tally.count();

    int result = tally.get_value();
    cout << result << endl;

    tally.reset();
    return 0;
}
```

The first line in the main() creates an object (i.e. instance of the Counter class). We can use the object we created to call member functions contained within the Counter class. Use dot notation.

When the individual objects are created, they inherit all the variables and functions from the class.

Encapsulation Motivation

- What will happen if I try the following code?

```
int main() {
        ...
        cout << tally.value << endl;
}
```

Error: use count() instead.

- Observe that value is a private variable.
- Private members can only be accessed by member functions of the same class.

Object-Oriented Programming: Encapsulation

- Recall every class has a **public interface** (a collection of member functions through which the objects of the class can be manipulated) and **private members** (that can only be manipulated via the public interface).

- **Encapsulation**: the act of providing a public interface with which you can access your private data.
  - Enables changes in the implementation without affecting users of a class, or misuse of a class
  - It is considered good practice to declare your class attributes as private (as often as you can).
  - Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts

Structure vs Class

In C++, a structure is almost the same as a class.

The biggest difference: by default, a **Structure** is not secure and will not hide its implementation details from the end-user while a **class** is secure and will hide its programming and designing details.

Default Structure

Field	Value
name	Jane Doe
email	jane@email.com
birthyear	2003
address	Fort Collins, CO

`student1.birthyear = 2004;` ✓

Default Class

Field	Value
name	Jane Doe
email	jane@email.com
birthyear	2003
address	Fort Collins, CO

`student1.birthyear = 2004;` ✗

Methods

- The member functions of a class are called methods.

- There are two kinds of methods:
  - Mutators: can *change* data members
  - Accessors: only access data members

AKA, **getters** (get the value of private attributes) and **setters** (set the value of private attributes)!

Accessor vs Mutator Functions

- Recall, member functions which do not modify data have the word const as the last word of the prototype.

```
int Counter::get_value() const
```

- These are called "**accessor**" functions or GETTERS

- "Mutator" functions or SETTERS modify the data members of the object.

Example17B: Cash Register

Cash Register

- Let’s think about what we expect a cash register to do:

- 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.

Cash Register

- So, let’s start by defining the class:

```
class CashRegister {
        public:
                // public interface methods will go here
        private:
                // data members will go here
}
```

Cash Register

- And now add the methods that we need:

```
class CashRegister {
        public:
                void clear();
                void add_item(double price);
                double get_total() const;
                int get_count() const;

        private:
                  // data members will go here
}
```

These are just prototypes, we still need to define them.

Cash Register

- And now add the methods that we need:

```
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/
Getters

Cash Register

- And now add the methods that we need:

```
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/
Setters

Cash Register

- Now, let’s list out the private data members required:

```
class CashRegister {
        public:
                // public interface methods

        private:
                int item_count;
                double total_price;
}
```

Cash Register

- We can call the member functions by first creating an object 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.

Cash Register

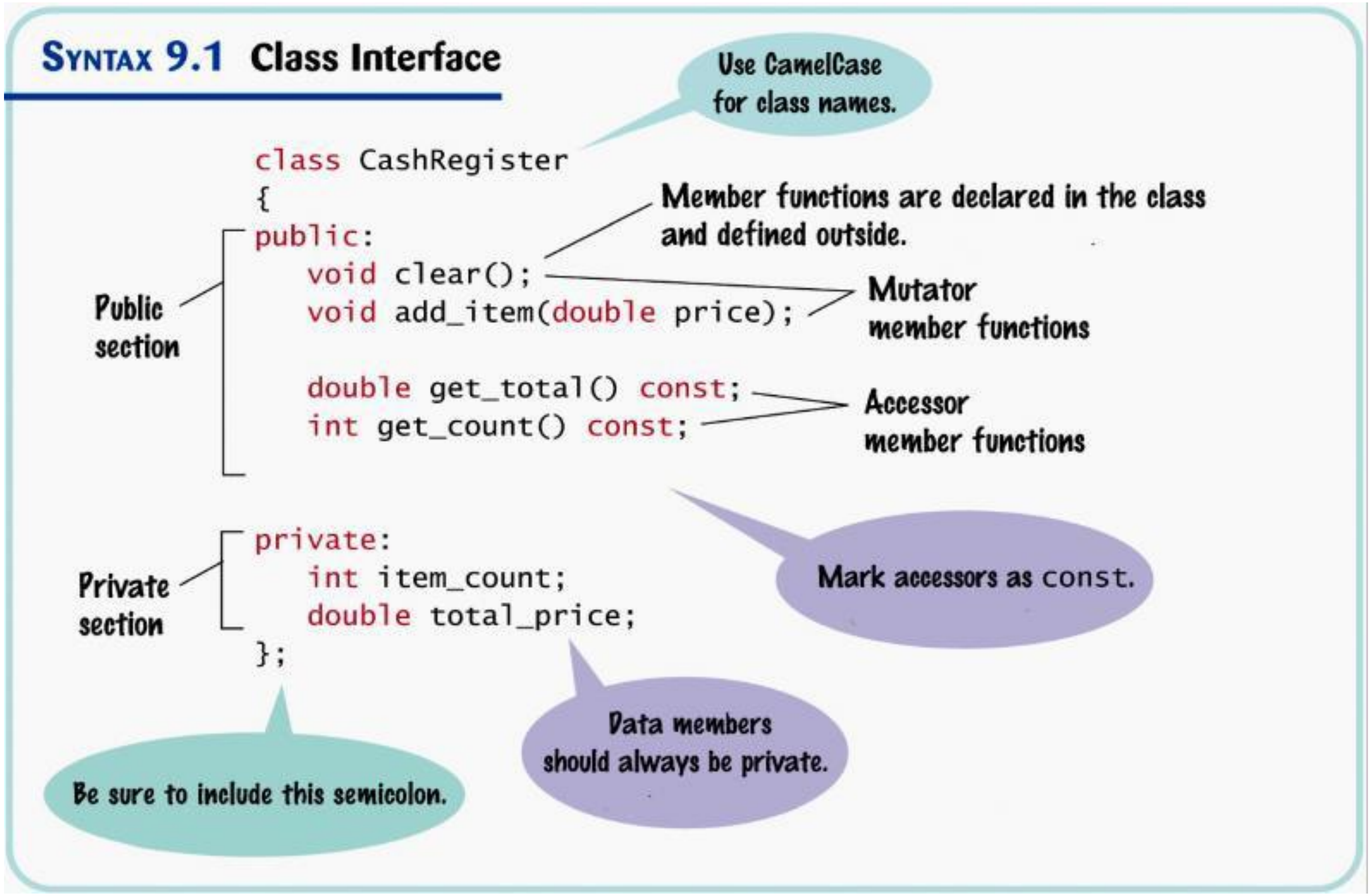
- This statement will print the current total:

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

- Accessor functions/getters return values.
- CANNOT say

```
cout << register1.total_price << endl;
```

Cash Register



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;
};
```

Use CamelCase for class names.

Member functions are declared in the class and defined outside.

Mutator member functions

Accessor member functions

Public section

Private section

Mark accessors as const.

Data members should always be private.

Be sure to include this semicolon.

Cash Register

```
CashRegister register1;
```

Data Member	Value
item_count	1
total_price	1.95

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

```
CashRegister register2;
```

Data Member	Value
item_count	3
total_price	8.45

```
register2.add_item(2.95);
register2.add_item(4.00);
register2.add_item(1.50);
```

Implicit Parameter

- When a member function is called – maybe in 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.

Cash Register: Implementation

- Now, let’s implement this!

Wrapping Up

- ★ Problem Set 4 due Sunday July 4 at 11:59
  - ○ File I/O
  - ○ There will be interview grading
- ★ Quiz 4 this week is our last quiz!
  - ○ Functions, Vectors, File I/O
  - ○ Released tomorrow, due Friday 11:59 PM
- ★ Recitation tomorrow
  - ○ File I/O
  - ○ Classes and objects basics
- ★ Mid-term FCQs for TAs and GPTIs have been sent out
  - ○ Please take a minute to fill them out!
  - ○ The survey will remain open until July 2nd

References

This lecture was inspired and adapted from the following:

★ **Sanskar Katiyar**'s version of CSCI 1300 (Summer 2020)
★ **Asa Ashraf**'s version of CSCI 1300 (Spring 2020)
★ **Taylor Dohmen**'s version of CSCI 1300 (Summer 2019)
★ **Varsha Koushik**'s version of CSCI 1300 (Summer 2019)
★ **Cay Horstmann**'s Brief C++: Late Objects (3e)
★ **David Malan**'s CS50 at Harvard