CLASSES – A DEEPER LOOK

Chapter 9

TOPICS

Separating Interface from Implementation

Class Scope and Accessing Class Members

Access Functions and Utility Functions

Constructors with Default Arguments

Destructors

TOPICS, CONTINUED

Default Memberwise Assignment and the Default Copy Constructor

Const Objects and const Member Functions

Composition

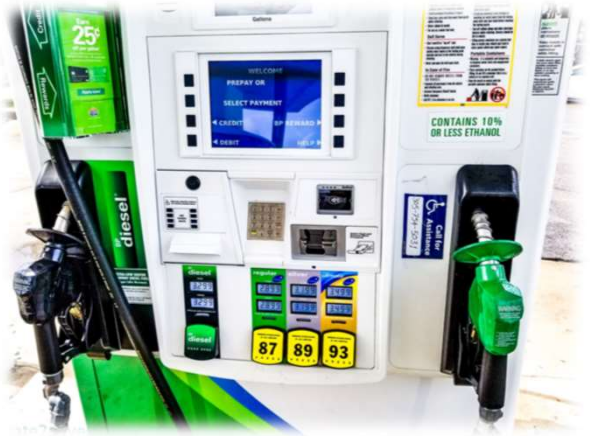
Static Class Members

Separating Interface from Implementation

INTERFACE

The **interface** to a system is the part of the system you interact with

Consider the screen on a gas pump – It provides a connection to three different systems:



- The physical machinery of the pump
- The gas station's sales system
- Your payment system (credit card company, bank, etc.)

To use one, you don't need to know the physics of the machinery, or the name of the sales system

- You are shielded from the implementation details.

Good Software Engineering Uses This Concept Too

It's called the **Principle of Least Privilege**

- Only allow as much access as needed, but no more

So to use a class, the client code only needs to know:

- The member functions that are available to call
- How to call those functions – what data is required as input for them to work
- What result the member functions will send back

Client code DOES NOT need to know HOW those functions are implemented

- Just like you don't need to know how a gas pump works on the inside, or how it communicates with the gas station's sales system
- You just need to know that it does these things

If implementation changes, client code doesn't need to care, and should not have to change

The Interface to a Class

In code, the **interface to a class** describes what **public services** the class provides, and how to request those services

Public services = member functions declared public

So the interface to a class is a list of its public member functions **names**, with their **parameters** and **return types**, in this format:

returnType functionName (parameter list)

This format should look familiar…

The Interface to a Class is a List of Prototypes

returnType functionName (parameter list)

This format is the same format as a **function prototype**.

Remember from chapter 6 that a prototype provides information to the compiler about how a function should be called.

We're going to use prototypes to provide information to the client code that needs to use the functions, without revealing details of the implementations of these functions.

Separating the Interface from the Implementation in C++

To separate interface from implementation, we're going to break up the **class code** (formerly saved as a .h file) into two separate files:

A header file (.h)

A source code file (.cpp)

CLASS HEADER FILE

Is named using the name of the class and the **.h** file extension

Contains the **class definition**, which includes:

1. A list of public and private **data members**

2. Member-function **prototypes**

3. Possibly other things we'll see later

We're going to move member function **definitions** out of the header file.

CLASS SOURCE CODE FILE

Is named using the name of the class and the **.cpp** file extension

Contains only **member function definitions**

EXAMPLE – TIME CLASS

Fig09_01_03: Time.h, Time.cpp, fig09_03.cpp

There are 3 separate files in this example:

Time.h – the Time class header file

Time.cpp – the Time class source code file

Fig09_03.cpp – the driver source code file

- (aka the client code that uses the Time class)

TIME.H CONTAINS THE CLASS DEFINITION

Class definition (lines 11-20)

- Begins with keyword class, the name of the class, and opening curly brace
- Ends with closing curly brace, terminating semicolon

Prototypes (lines 13-15)

- Remember prototypes give the user of the function information, but does not reveal implementation details
- This class has no constructor, but if one was required, it would go here too.

Private data members (lines 17-19)

Header Files are Included in Source Code Files

Remember that header files are included in source code files, using the #include directive

Including a header file causes the code in the header file to be copied into the file where the #include appears

This can be a problem if the code that's copied in includes other header files, and those header files are already included in the first source code file.

There can't be two copies of the same prototypes and data member declarations in the same file – compilation error

INCLUDE GUARD

To prevent a header file from being copied in twice, we add some extra lines to it called an **include guard**

- In Time.h, note lines 7, 8, and 22

The include guard prevents the code between **#ifndef** and **#endif** from being #included if the name TIME_H has already been defined.

#ifndef *nameOfClass*_H
#define *nameOfClass*_H
…
#endif

Note naming convention: uppercase with period replaced by underscore

TIME.CPP CONTAINS THE MEMBER FUNCTION DEFINITIONS

setTime, printUniversal and toUniversalString

Note the format of the function names:

ClassName*::*FunctionName

Scope resolution operator ::

- Tells the compiler which class these functions belong to

Without the class name and :: before the name, the compiler would treat them as global functions, like main, not as members of the class

- Would not have access to the class' private data members – causing a compilation error

TIME CLASS – SETTIME FUNCTION

setTime Function (lines 12-23)

If arguments passed into this function are not valid values for setting the time, we need to alert the calling code to the error

throw statement

- Lines 20-21
- Creates an exception object of the type invalid_argument
- Lots of different kinds of exceptions available in C++, or can build custom ones

The text within the parentheses can be a custom error message

- It's passed to the constructor of the exception class

TIME CLASS – TOUNIVERSALSTRING FUNCTION

toUniversalString (lines 26-31)

Uses Time's int data members to create a string representation of the time, using string stream processing with the **ostringstream** class

OSTRINGSTREAM CLASS FOR STRING STREAM PROCESSING

Remember that the standard output stream is **cout**

ostringstream class provides the same functionality as cout, but writes output to string objects in memory, instead of stdout

- Must include the **<sstream>** header

Then ostringstream's **str** member function returns the formatted string

setfill

- A sticky stream manipulator to specify a **fill character** if integer is narrower than width of field

Integers are right-aligned, so fill characters go on left

Left-aligned values – fill characters go on right

CLASS SOURCE CODE MUST INCLUDE THE HEADER FILE

The source code file for the Time class (Time.cpp) must include the header file for the class (Time.h) just like driver programs do

#include on line 7 of Time.cpp:

#include “Time.h”

SOME SOFTWARE ENGINEERING OBSERVATIONS

Separating a class into header and implementation files does not affect how client code uses the class

- Only how the program is compiled and linked

Every change to the header requires recompilation of every source-code file that includes it

- May be a huge task in large systems

So only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header

USING THE TIME CLASS

Fig09_03.cpp

Note call to setTime with invalid values (lines 23-28)

- Enclosed in a try/catch block to demonstrate exception handling

Class Scope and Accessing Class Members

CLASS SCOPE

Class members = data members and member functions of a class

Belong to that class' scope

Within the scope (so, within the class) – refer to class members by their simple name

Outside scope (so, outside of the class) – need a **handle**:

- Object name, or
- Reference to an object, or
- Pointer to an object

Function Variables have Block Scope

Defined within a function

Also called **local variables**

Can't be accessed outside their scope

So, can't be accessed outside of the function in which they are declared

Accessing Class Members w/ Member Selection Operators

Member selection operator

- Used to access a class member from outside of the class
- Which one to use depends on the type of handle you have to the class

Two types:

Dot member selection operator

- Used with object names and with references to objects

Arrow member selection operator

- Used with pointers to objects

MEMBER SELECTION OPERATOR SYNTAX

Dot operator:

ObjectName.memberName

ObjectReference.memberName

Arrow operator syntax:

ObjectPointer–>memberName

Member Selection Operator Examples

Account account;	**account** is the name of an Account object
Account& accountRef{account};	**accountRef** refers to an Account object
Account* accountPtr{&account};	**accountPtr** points to an Account object
account.setBalance(123.45);	call setBalance via the Account object's name
accountRef.setBalance(123.45);	call setBalance via a reference to Account object
accountPtr->setBalance(123.45);	call setBalance via a pointer to Account object

Access Functions and Utility Functions

FUNCTIONS WITHIN A CLASS

Functions within a class may be public or private

Functions within a class may be used for different things

We've seen getters and setters before

We can categorize class functions according to their purpose

We're going to see three types:

- Access functions
- Predicate functions
- Utility functions

Access Functions

Access functions read or display class data

Do not modify class data

Getters are an example

Access functions are declared **const**, so that it's clear that no change to the object should occur

PREDICATE FUNCTIONS

Type of access function that tests truth or false

Returns a **bool**

- Data type that is either true or false

Example: **isEmpty()** function of the vector class

Returns true if the vector object does not contain any objects

- "It is true that the vector is empty"

Returns false if the vector object contains any objects

- "It is false that the vector is empty"

UTILITY FUNCTIONS

Support operations of a class' other member functions

- Aka helper function

Declared private

Not intended for use by client code that uses the class

Intended for use only within the class, by the other functions

Often contains code that might be in common among other functions, and would have to be duplicated in more than one place

CONSTRUCTORS WITH DEFAULT ARGUMENTS

CONSTRUCTORS WITH DEFAULT ARGUMENTS – EXAMPLE

Fig09_05_07: Time.h, Time.cpp, fig09_07.cpp

Time class uses a **default constructor** with **default arguments**

- Time.h (line 13) and Time.cpp (lines 11-13)

Note that a constructor that defaults ALL its arguments can be invoked with no arguments

Declared **explicit**

- More in chapter 10

Can be at most **one default constructor** per class

Using a Constructor with Default Arguments

Time class constructor has 3 integer parameters

- Used to set the hour, minute, and second data members of a Time object

When working with time, a 0 is a valid value

- If the hour is 0, it refers to midnight

We can make it easier for client code to create a Time class by providing default values

Note: default constructor argument values can be anything, not just 0

- 0 just happens to work for this example

Syntax of a Constructor with Default Arguments

Example from Time.h:

explicit Time(int = 0, int = 0, int = 0);

Parts of this statement:

explicit – required keyword

Time – name of the class

int = 0 – the **data type** of each parameter, along with a **default value**

Invoking a Constructor with Default Arguments

The client code that creates a Time object can create it in four ways, depending on the number of arguments passed to the constructor

fig09_07.cpp – lines 15-18

Constructor call in driver	No. of arguments passed to constructor	What happens in Time constructor
Time t1;	None	Hour, minute, and second set to the default value 0
Time t2{2};	1	Hour set to 2; minute and second set to the default value 0
Time t3{21, 34};	2	Hour set to 21; minute set to 34; and second set to the default value 0
Time t4{12, 25, 42};	All 3	Hour set to 12; minute set to 25; and second set to 42

TYPES OF CONSTRUCTORS

On the previous slide, we demonstrated calling a constructor four different ways

Remember that a constructor has the same name as the class

So using a default constructor is similar to overloading a function

- Same name, different parameters

But we can also write more than one constructor

OVERLOADED CONSTRUCTORS

Separate constructors with **different parameter lists**

Must provide a prototype for each version of the constructor

Must provide a separate constructor definition for each overloaded version

Overloaded constructors can be used with, or instead of, default constructor arguments

Delegating Constructors

When using overloaded constructors, we may have separate constructors that perform similar but maybe not exactly the same tasks

It's helpful to call the other constructor to do that work, instead of repeating it

When using overloaded constructors, we can invoke a constructor from within another constructor in the same class

Delegating constructor – a constructor that calls another constructor within a class

DESTRUCTORS

Destructors Perform Termination Cleanup

Constructors create objects by reserving memory space

When the object is no longer needed, it is good practice to reclaim that memory space

The memory space will eventually be reclaimed by the operating system when the program ends, but if an object is no longer needed before that, we should clean things up

Destructor

- Performs termination housekeeping before object's memory is reclaimed

DESTRUCTOR SYNTAX

Prototype: ***~nameOfClass***

Definition: ***nameOfClass*::~*nameOfClass*() {**
```
        //statements to be performed
        //when object no longer needed
    }
```

Call: **delete *nameOfObject*;**

~ is the bitwise complement operator (tilde on a keyboard)

May not specify arguments or return value

CALLING A DESTRUCTOR

Called implicitly when object is destroyed

Exists implicitly even if one is not explicitly defined

Called explicitly with the **delete** operator

Default Memberwise Assignment and the Default Copy Constructor

Default Memberwise Assignment

Also called **copy assignment**

Uses assignment operator to assign data members of one object to those of another object of the same class

Enables copying objects using one statement, rather than using get and set functions to copy individual values

Default Memberwise Assignment Example

Fig09_14_16: fig09_16.cpp, Date.h, Date.cpp

Date class has a default constructor (Date.h, line 12)

fig09_16.cpp creates two Date objects (lines 9 and 10)

- One created with 3 arguments
- The other uses the defaults

Then it copies the object date1 to the object date2, using default memberwise assignment (line 15)

DEFAULT COPY CONSTRUCTOR

When passing objects by value to functions, and when returning objects from functions, each data member is also copied

This is handled through the **default copy constructor**

Provided by compiler to allow passing of objects by value to functions and returning objects from functions

CONST OBJECTS AND CONST MEMBER FUNCTIONS

CONST OBJECTS

Can create const objects, just like any const variable

Declaring objects as const can improve performance

Rules for Const Objects and const Member Functions

Cannot call a member function of a const object unless that function is also const

Get functions should always be declared const

Cannot declare constructors as const

- "const"ness enforced from end of constructor to beginning of destructor

COMPOSITION

Composition: Objects as Members of Classes

Data members of a class can be of any type

- Fundamental types, like int
- Types from a standard library, like string
- Programmer-defined types, like other classes

Composition

- When a class contains a data member that is an object of another class, it is called composition

Also called a “has-a” relationship

COMPOSITION EXAMPLE

Fig09_18_22

- Date.cpp, Date.h, Employee.cpp, Employee.h, fig09_22.cpp

The Employee class contains objects of the Date class to represent a birth date and a hire date

The relationship between Employee and Date is composition, because we can say an Employee “has a” birth date or hire date

We don’t say that a Date “has an” Employee, so the opposite relationship is not composition

SOME NOTES ON THIS EXAMPLE – EMPLOYEE.CPP INITIALIZER LIST

Note member initializer list in constructor (lines 13-16)

Begins with : (colon), and each initializer on a line by itself

For clarity, list the member initializers in the order that the class's data members are declared.

SOME NOTES ON THIS EXAMPLE – DATE CLASS DEFAULT COPY CONSTRUCTOR

Date class has a **default copy constructor**

Note that no constructor is provided that receives a Date object

But the Employee constructor's member-initializer list initializes birthDate and hireDate objects by passing Date objects to their constructors

This is ok because the compiler implicitly defines a default copy constructor, and each data member of the argument passed in is copied appropriately

STATIC CLASS MEMBERS

STATIC CLASS MEMBERS ARE SHARED

Usually each object of a class has its own copy of all class data members

Data members declared **static** are **SHARED** by all objects of the class

Functions that access static data members must also be declared static

Demonstrating Static Class Members

Fig09_28_30 – fig09_30.cpp, Employee.cpp, Employee.h

This example has a static data member (count) and a static function (getCount())

STATIC DATA MEMBERS

Declared in the header

Defined and initialized in the class source code file

count

- Data member will be shared by all objects of the Employee class – each object will not get its own copy
- Declared in the header file using keyword static (Employee.h, line 23)
- Initialized to 0 in global scope in the class source code file (Employee.cpp, line 8)

Note that static keyword used only in the header file, where the data member is declared

Static Member Functions

Also declared in the header (via a prototype)

Defined in the class source code file

getCount

- Returns the value of a static data member, so it too must be declared static
- Prototype in the header file uses keyword static (Employee.h, line 17)
- Defined in the class source code file (Employee.cpp, line 12)