10.1 Objects: Introduction

A large program thought of as thousands of variables and functions is hard to understand. A higher level approach is needed to organize a program in a more understandable way.

In the physical world, we are surrounded by basic items made from wood, metal, plastic, etc. But to keep the world understandable, we think at a higher level, in terms of *objects* like an oven. The oven allows us to perform a few specific operations, like put an item in the oven, or set the temperature.

Thinking in terms of objects can be powerful when designing programs. Suppose a program should record time and distance for various runners, such as a runner ran 1.5 miles in 500 seconds, and should compute speed. A programmer might think of an "object" called RunnerInfo. The RunnerInfo object supports operations like setting distance, setting time, and computing speed. In a program, an ***object*** consists of some internal data items plus operations that can be performed on that data.

PARTICIPATION ACTIVITY 10.1.1: Grouping variables and functions into objects keeps programs understandable.

Animation captions:

1. A program with many variables and functions can be hard to understand
2. Grouping related items into objects keeps programs understandable

Creating a program as a collection of objects can lead to a more understandable, manageable, and properly-executing program.

PARTICIPATION ACTIVITY 10.1.2: Objects.

Some of the variables and functions for a used-car inventory program are to be grouped into an object type named CarOnLot. Select True if the item should become part of the CarOnLot object type, and False otherwise.

1) int carStickerPrice;

- ○ True
- ○ False

2) double todaysTemperature;

- True
- False

3) int daysOnLot;

- True
- False

4) int origPurchasePrice;

- True
- False

5) int numSalespeople;

- True
- False

6) IncrementCarDaysOnLot()

- True
- False

7) DecreaseStickerPrice()

- True
- False

8) DetermineTopSalesperson()

- True
- False

10.2 Classes: Introduction

Using a class

The ***class*** construct defines a new type that can group data and functions to form an object. The below code defines and uses a class named RunnerInfo. This section first focuses on how to *use a class*, with relevant code highlighted below. Later, the section focuses on defining a class.

Figure 10.2.1: Simple class example: RunnerInfo.

```
#include <iostream>
using namespace std;

class RunnerInfo {
   public:                                  // The class' public functions
      void   SetTime(int timeRunSecs);      // Time run in seconds
      void   SetDist(double distRunMiles);  // Distance run in miles
      double GetSpeedMph() const;           // Speed in miles/hour
   private:  // The class' private internal data members
      int    timeRun;
      double distRun;
};

// "RunnerInfo::" means SetTime is a RunnerInfo member function
void RunnerInfo::SetTime(int timeRunSecs) {
   timeRun = timeRunSecs;  // timeRun refers to data member
   return;
}

void RunnerInfo::SetDist(double distRunMiles) {
   distRun = distRunMiles;
   return;
}

double RunnerInfo::GetSpeedMph() const {
   return distRun / (timeRun / 3600.0); // miles / (secs / (hrs / 3600 secs))
}

int main() {
   RunnerInfo runner1; // User-created object of class type RunnerInfo
   RunnerInfo runner2; // A second object

   runner1.SetTime(360);
   runner1.SetDist(1.2);

   runner2.SetTime(200);
   runner2.SetDist(0.5);

   cout << "Runner1's speed in MPH: " << runner1.GetSpeedMph() << endl;
   cout << "Runner2's speed in MPH: " << runner2.GetSpeedMph() << endl;

   return 0;
}
```

```
Runner1's speed in MPH: 12
Runner2's speed in MPH: 9
```


To *use* a class, the programmer declares a variable of a class type, like runner1 in main(). A variable of a class type is known as an ***object***. An object involves data and functions, whereas a typical variable is just data. Like an engineer building multiple bridges from a single blueprint, a programmer can create multiple objects from a single class definition. Above, the programmer also created a second object, runner2, of the same class type.

The class user then called class member functions on the object, such as SetTime(). A ***member function*** is a function that is part of (a "member of") a class. The member functions are (typically) listed after the keyword ***public*** in the class definition. A member-function call uses the "." operator, known as the ***member access operator***.

Construct 10.2.1: Calling a class member function for an object.

```
objectName.MemberFct();
```

A call to an object's member function operates on that object. The following animation provides an example for a PhotoFrame class whose public functions allow setting height and width and calculating area.

PARTICIPATION ACTIVITY 10.2.1: Each object has functions that operate on that object.

Animation captions:

1. The variable declaration creates object frame1, on which several functions can operate.
2. The second variable declaration creates a second distinct object.
3. frame1's function calls operate on the frame1 object.
4. frame2's function calls operate on the frame2 object.
5. frame1's function call operates on frame1's object.

The class user need only use a class' public member functions, called the class ***interface***, and need not directly access internal variables of the class. Such use is akin to a cook using an oven's interface of knobs and buttons, and not reaching inside the oven to adjust the flame.

Don't do this

PARTICIPATION ACTIVITY 10.2.2: Using a class.

The following questions consider *using* class RunnerInfo, which was defined above.

1) Type a variable declaration that creates an object named runnerJoe of class type RunnerInfo.

Check Show answer

2) Type a statement that creates object firstRunner, followed by a statement that creates object secondRunner, both of class type RunnerInfo.

Check Show answer

3) Object runner1 is of type RunnerInfo. Type a statement that sets runner1's time to 100.

Check Show answer

4) If runner1's time was set to 1600, and runner1's distance to 2.0, what do you expect runner1.GetSpeedMph() will return? Type answer as #.#

Check Show answer

Defining a class

To *define* a class, a programmer starts by naming the class, declaring private member variables, and declaring public member functions, as in the initial highlighted text for the RunnerInfo class above. A class' member variables are known as ***data members***. The programmer declares data member variables after the keyword ***private***, making clear that a class user cannot directly access data members. Above, class RunnerInfo has data members timeRun and distRun. Note that the compiler does *not* allocate memory for those variables when the class is defined, but rather when an object is created of that class type.

Construct 10.2.2: Defining a class by naming the class and declaring its members.

```
class ClassName {
  public:
      // Public member functions
   private:
      // Private data members
};
```

Next, the programmer defines the details of each member function, sometimes called the class' ***implementation***. The programmer uses the ***scope resolution operator*** :: as in ClassName::FunctionName(), to inform the compiler that a function is a member function of the indicated class. An example from above is shown again below.

Figure 10.2.2: Defining a member function SetTime for class RunnerInfo.

```
// "RunnerInfo::" means SetTime is a RunnerInfo member function
void RunnerInfo::SetTime(int timeRunSecs) {
  timeRun = timeRunSecs;  // timeRun, a private data member, is accessible to member functions
  return;
}
```

A class' private data members such as timeRun and distRun above, are automatically accessible within member functions. For example, member function RunnerInfo::SetTime assigns its parameter timeRunSecs to private data member timeRun.

Earlier, RunnerInfo's GetSpeed() member function declaration and definition were followed by the word const. The keyword ***const*** after a member function declaration and definition indicates that the function does not change the value of any data member. The word informs a class user, and also enables the compiler to generate an error if the function attempts a change.

PARTICIPATION ACTIVITY 10.2.3: Defining a class.

Help define class Employee. Follow the class definition examples from above.

1) Type the first line of a class definition for a class named Employee, ending with {.

Check Show answer

2) Type the line that comes after the above class name line and before the public member function declarations.

Check Show answer

3) Declare a member function named SetSalary, having parameter int salaryAmount, and returning void.

Check Show answer

4) Declare a member function named GetSalary, having no parameter, and returning int. The function will not change the value of any data member.

Check Show answer

5) Type the line that comes after the public member functions and before private data members.

Check Show answer

6) Declare a data member named salary of type int.

Check Show answer

7) Type the first line of a function definition for the member function SetSalary described above, ending with {.

Check Show answer

8) Type the statement within member function SetSalary that assigns the value of parameter salaryAmount to data member salary.

Check Show answer

9) Suppose a user declared: Employee clerk1;. Using info from earlier questions, type a statement that sets clerk1's salary to 20000.

Check Show answer

10) Given: class Employee {...}. Is Employee an object? Type yes or no.

Check Show answer

11) Given: Employee clerk1;. Is clerk1 an object? Type yes or no.

Check Show answer

A common error is for a member function definition to not exactly match the member function's declaration in the class definition, resulting in a syntax error. For example, the definition `double RunnerInfo::GetSpeedMph() { ... }`, which is missing the word const, yields the following error message from g++:

```
definition of 'GetSpeedMph' does not match any declaration in 'RunnerInfo'
```

PARTICIPATION ACTIVITY 10.2.4: Class example: Right triangle.

Complete the program involving a class for a right triangle.

Load default template... Run

```
1
2 #include <iostream>
3 #include <cmath>
4 using namespace std;
5
6 class RightTriangle {
7    public:
```

```
 8       void SetSide1(double side1Val);
 9       // FIXME: Declare SetSide2()
10       double GetHypotenuse() const;
11
12    private:
13       double side1;
14       // FIXME: Declare side2
15 };
16
17 void RightTriangle::SetSide1(double side1Val) {
18    side1 = side1Val;
```


This section taught the common use of classes. The class construct is actually more general. For example, data members could be made public. However, good practice is to make ALL data members of a class private, using member functions to access that data.

C++ has a related construct called a struct, which comes from C. In C, a struct can only group data, not functions. In C++, a ***struct*** is nearly identical to a class. The difference is that if a member appears before a label of private or public, then by default, in a struct the member is public while in a class the member is private. Some programmers suggest a good practice is to use struct for simple data grouping with that data being public, while using class for objects (having data and functions, with data private). However, some C++ programmers argue that good practice is to only use class, abandoning struct entirely. Finally, for classes, good practice explicitly uses the public: and private: keywords for clarity, rather than relying on defaults.

Inline member functions

Above, each member function was defined outside the class definition; only the function declaration appeared inside. Alternatively, a member function can be defined inside the class definition, known as an ***inline member function***. Programmers sometimes use inline member functions for very short functions, to keep code compact and readable. Longer inline member functions would clutter the class definition so are usually defined outside.

Figure 10.2.3: Inline member function. SetTime() is defined inside the class (inline), while SetDist() is defined outside.

```
class RunnerInfo {
   public:
      void   SetTime(int timeRunSecs) {
         timeRun = timeRunSecs;
      }
      void   SetDist(double distRunMiles);
      double GetSpeedMph() const;
   private:
      int    timeRun;
      double distRun;
};

void RunnerInfo::SetDist(double distRunMiles) {
   distRun = distRunMiles;

   return;
}
```


Note: Normally, items like variables must be declared before being used, but this rule does not apply within a class definition. Ex: Above, SetTime() accesses timeRun, even though timeRun is declared a few lines after. This rule exception allows a class to have the desired form of a public region at the top and a private region at the bottom: A public inline member function can thus access a private data member even though that private data member is declared after the function.

PARTICIPATION ACTIVITY 10.2.5: Inline member functions.

Consider the following class definition. The code may have some errors.

```
class PickupTruck {
   public:
      void SetLength(double fullLength);
      void SetWidth (double fullWidth) {
         widthInches = fullWidth;
      }
   private:
      double lengthInches;
      double widthInches;
};

void PickupTruck::SetLength(double fullLength) {
   lengthInches = fullLength;
}
```

1) Inside the class definition, SetLength() is declared but not defined.

   - ○ True
   - ○ False

2) Inside the class definition, SetWidth() is declared but not defined.

   - ○ True
   - ○

False

3) SetWidth() is an inline member function.

- True
- False

4) SetWidth() uses widthInches, which is an error because widthInches is declared after that use.

- True
- False

5) If the programmer defines SetWidth() inline as above, then the programmer should probably define SetLength() as inline too.

- True
- False

Exploring further:

- Classes from cplusplus.com
- Classes from msdn.microsoft.com

CHALLENGE ACTIVITY 10.2.1: Classes.

Start

Type the program's output.


```
He is Joe
```

```
#include <iostream>
using namespace std;

class Person {
   public:
      void SetFirstName(string firstNameToSet);
      string GetFirstName() const;
   private:
      string firstName;
};

void Person::SetFirstName(string firstNameToSet) {
   firstName = firstNameToSet;
}

string Person::GetFirstName() const {
   return firstName;
}

int main() {
   string userName;
   Person person1;

   userName = "Joe";

   person1.SetFirstName(userName);
   cout << "He is " << person1.GetFirstName();

   return 0;
}
```


1 2 3 4

Check Next

CHALLENGE ACTIVITY 10.2.2: Basic class use.

Print person1's kids, apply the IncNumKids() function, and print again, outputting text as below. End each line with newline.
Sample output for below program:

```
Kids: 3
New baby, kids now: 4
```



```
1 #include <iostream>
2 using namespace std;
3
4 class PersonInfo {
5    public:
6       void   SetNumKids(int personsKids);
7       void   IncNumKids();
8       int    GetNumKids() const;
```

```
 9    private:
10       int    numKids;
11 };
12
13 void PersonInfo::SetNumKids(int personsKids) {
14    numKids = personsKids;
15    return;
16 }
17
18 void PersonInfo::IncNumKids() {
```


Run

View your last submission

CHALLENGE ACTIVITY 10.2.3: Basic class definition.

Define the missing function. licenseNum is created as: (100000 * customID) + licenseYear.
Sample output:

```
Dog license: 77702014
```

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class DogLicense{
 5    public:
 6       void   SetYear(int yearRegistered);
 7       void   CreateLicenseNum(int customID);
 8       int    GetLicenseNum() const;
 9    private:
10       int    licenseYear;
11       int    licenseNum;
12 };
13
14 void DogLicense::SetYear(int yearRegistered) {
15    licenseYear = yearRegistered;
16    return;
17 }
18
19 // FIXME: Write CreateLicenseNum()
```


Run

View your last submission

10.3 Classes example (with vectors)

One area where the power of classes becomes clear is when used together with vectors. The following illustrates.

Figure 10.3.1: Class example. Seat reservation system.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

/*** Seat class definition ***/
class Seat {
public:
   void Reserve(string resfirstName, string resLastName, int
resAmountPaid);
   void MakeEmpty();
   bool IsEmpty() const;
   void Print() const;

private:
   string firstName;
   string lastName;
   int    amountPaid;
};

void Seat::Reserve(string resfirstName, string resLastName, int
resAmountPaid) {
   firstName  = resfirstName;
   lastName   = resLastName;
   amountPaid = resAmountPaid;

   return;
}

void Seat::MakeEmpty() {
   firstName  = "empty";
   lastName   = "empty";
   amountPaid = 0;

   return;
}

bool Seat::IsEmpty() const {
   return(firstName == "empty");
}

void Seat::Print() const {
   cout << firstName << " " << lastName  << ", ";
   cout << "Paid: "  << amountPaid << endl;

   return;
}
/*** End definitions for Seat class ***/


/*** Functions for vector of Seat objects ***/
void SeatsMakeEmpty(vector<Seat>& seats) {
   unsigned int i = 0;
```

```
Enter command (p/r/q): p
0: empty empty, Paid: 0
1: empty empty, Paid: 0
2: empty empty, Paid: 0
3: empty empty, Paid: 0
4: empty empty, Paid: 0

Enter command (p/r/q): r
Enter seat number
(0..4): 2
Enter first name: Mike
Enter last name: Jones
Enter amount paid: 300
Completed.

Enter command (p/r/q): r
Enter seat number
(0..4): 2
Seat already occupied.

Enter command (p/r/q): r
Enter seat number
(0..4): 3
Enter first name: Alicia
Enter last name:
Gonzales
Enter amount paid: 275
Completed.

Enter command (p/r/q): p
0: empty empty, Paid: 0
1: empty empty, Paid: 0
2: Mike Jones, Paid: 300
3: Alicia Gonzales,
Paid: 275
4: empty empty, Paid: 0

Enter command (p/r/q): x
Invalid command.

Enter command (p/r/q): r
Enter seat number
(0..4): 6
Seat number too large.

Enter command (p/r/q): q
Quitting.
```

```
   for (i = 0; i < seats.size(); ++i) {
      seats.at(i).MakeEmpty();
   }

   return;
}

void SeatsPrint(vector<Seat> seats) {
   unsigned int i = 0;

   for (i = 0; i < seats.size(); ++i) {
      cout << i << ": ";
      seats.at(i).Print();
   }

   return;
}

void SeatsCreateReservation(vector<Seat>& seats) {
   string firstName, lastName;
   int amountPaid = 0;
   unsigned int seatNum = 0;
   Seat seat;

   cout << "Enter seat number (0..";
   cout << seats.size() - 1 << "): ";
   cin  >> seatNum;

   if (seatNum > (seats.size() - 1)) {
      cout << "Seat number too large." << endl;
   }
   else if ( !(seats.at(seatNum).IsEmpty()) ) {
      cout << "Seat already occupied." << endl;
   }
   else {
      cout << "Enter first name: ";
      cin >> firstName;
      cout << "Enter last name: ";
      cin >> lastName;
      cout << "Enter amount paid: ";
      cin >> amountPaid;

      seat.Reserve(firstName, lastName, amountPaid);
      seats.at(seatNum) = seat;

      cout << "Completed." << endl;
   }

   return;
}
/*** End functions for vector of Seat objs ***/

int main() {
   char   userKey = '-';
   vector<Seat> seats(5);

   SeatsMakeEmpty(seats);

   while (userKey != 'q') {
      cout << endl << "Enter command (p/r/q): ";
      cin >> userKey;

      if (userKey == 'p') { // Print seats
         SeatsPrint(seats);
      }
      else if (userKey == 'r') { // Reserve seat
         SeatsCreateReservation(seats);
      }
      else if (userKey == 'q') { // Quit
         cout << "Quitting." << endl;
```


©zyBooks 04/05/18 21:46 261830
Julian Chan
WEBERCS2250ValleSpring2018

```
        }
        else {
           cout << "Invalid command." << endl;
        }
     }

     return 0;
}
```


The Seat class provides public member functions for managing each seat, such as reserving a seat, checking if a seat is empty, and printing a seat. main() creates a vector of Seat objects, and then calls functions that operate on a vector of Seat objects.

The reader may notice that the program could be further improved by creating a new class named Seats. The class Seats would have a vector of Seat objects as a private data member, and have member functions MakeEmpty, Print, and CreateReservation. Classes containing other classes are common.

PARTICIPATION ACTIVITY 10.3.1: Seat reservation example using a class and vector.

The following refers to the above seat reservation example.

1) The class definition at the top of the code allocates memory for one object of type Seat.
   - True
   - False

2) The Seat class has 7 member functions.
   - True
   - False

3) The & is optional in: `void SeatsCreateReservation(vector<Seat>& seats) { ... }`
   - True
   - False

4) `vector<Seat> seats(5)` allocates 5 Seat objects.
   - True
   - False

5) `vector<Seat> seats(5)` allocates 5 * 4 or 20 member functions.

- True
- False

10.4 Mutators, accessors, and private helpers

A class' public functions are commonly classified as either mutators or accessors. A ***mutator*** function may modify ("mutate") the class' data members. An ***accessor*** function accesses data members but does not modify them.

The following example illustrates for a video game class that maintains two scores A and B for two players.

Figure 10.4.1: Mutator, accessor, and private helper methods.

```
#include <iostream>
using namespace std;

class GameInfo {
   public:
      void SetPlayer1PlayA(int playScore); // Mutator
      void SetPlayer1PlayB(int playScore); // Mutator
      void SetPlayer2PlayA(int playScore); // Mutator
      void SetPlayer2PlayB(int playScore); // Mutator

      int  GetPlayer1PlayA() const;        // Accessor
      int  GetPlayer1PlayB() const;        // Accessor
      int  GetPlayer2PlayA() const;        // Accessor
      int  GetPlayer2PlayB() const;        // Accessor

      int  GetPlayer1HighScore() const;    // Accessor
      int  GetPlayer2HighScore() const;    // Accessor

   private:
      int player1PlayA;
      int player1PlayB;
      int player2PlayA;
      int player2PlayB;
      int MaxOfPair(int num1, int num2) const; // Private helper fct
};

void GameInfo::SetPlayer1PlayA(int playScore) {
   player1PlayA = playScore;
}

void GameInfo::SetPlayer1PlayB(int playScore) {
   player1PlayB = playScore;
}

void GameInfo::SetPlayer2PlayA(int playScore) {
```

```
   player2PlayA = playScore;
}

void GameInfo::SetPlayer2PlayB(int playScore) {
   player2PlayB = playScore;
}

int  GameInfo::GetPlayer1PlayA() const {
   return player1PlayA;
}

int  GameInfo::GetPlayer1PlayB() const {
   return player1PlayB;
}

int  GameInfo::GetPlayer2PlayA() const {
   return player2PlayA;
}

int  GameInfo::GetPlayer2PlayB() const {
   return player2PlayB;
}

int GameInfo::GetPlayer1HighScore() const {
   return MaxOfPair(player1PlayA, player1PlayB);
}

int GameInfo::GetPlayer2HighScore() const {
   return MaxOfPair(player2PlayA, player2PlayB);
}

int GameInfo::MaxOfPair(int num1, int num2) const {
   if (num1 > num2) {
      return num1;
   }
   else {
      return num2;
   }
}

int main() {
   GameInfo funGame;

   funGame.SetPlayer1PlayA(88);
   funGame.SetPlayer1PlayB(97);
   funGame.SetPlayer2PlayA(74);
   funGame.SetPlayer2PlayB(40);

   cout << "Player1 playA: " << funGame.GetPlayer1PlayA() << endl;

   cout << "Player1 max: " << funGame.GetPlayer1HighScore() << endl;
   cout << "Player2 max: " << funGame.GetPlayer2HighScore() << endl;

   return 0;
}
```

```
Player1 playA: 88
Player1 max: 97
Player2 max: 74
```


Commonly, a data member has a pair of associated functions: a mutator for setting its value, and an accessor for getting its value, as above. Those functions are also known as a ***setter*** and ***getter*** functions, respectively, and typically have names that start with set or get.

Additional mutators and accessors may exist that aren't directly associated with one data member; the above has two additional accessors for getting a player's high score. These additional mutators and accessors often have names that start with words other than set or get, like compute, find, print, etc.

Accessor functions usually are defined as const, to enforce that they do not change data members. The keyword ***const*** after a member function's declaration and definition causes the compiler to report an error if the function modifies a data member. If a const member function calls another member function, that function must also be const.

PARTICIPATION ACTIVITY 10.4.1: Mutators and accessors.

1) A mutator should not change a class' private data.
   - True
   - False

2) An accessor should not change a class' private data.
   - True
   - False

3) A private data item sometimes has a pair of associated set and get functions.
   - True
   - False

4) Accessor functions must be defined as const.
   - True
   - False

5) If a const accessor function calls a non-const member function, a data member might get changed.
   - True
   - False

A programmer commonly creates private functions to help public functions carry out their tasks, known as ***private helper functions***. Above, private function maxOfPair() helps public functions getPlayer1HighScore() and getPlayer2HighScore(), thus avoiding redundant code.

PARTICIPATION ACTIVITY 10.4.2: Private helper functions.

1) A class' private helper function can be

called from main().

- True
- False

2) A private helper function typically helps public functions carry out their tasks.

- True
- False

3) A private helper function may not call another private helper function.

- True
- False

4) A public member function may not call another public member function.

- True
- False

CHALLENGE ACTIVITY 10.4.1: Mutators, accessors, and private helpers.

Start

Type the program's output.

```
Puppy
```

```
#include <iostream>
using namespace std;

class Dog {
   public:
      void SetAge(int monthsToSet);
      string GetStage() const;
   private:
      int months;
};

void Dog::SetAge(int monthsToSet) {
   months = monthsToSet;
}

string Dog::GetStage() const {
   string stage;
   if (months < 10) {
      stage = "Puppy";
   }
   else if (months < 25) {
      stage = "Adolescence";
   }
   else if (months < 70) {
      stage = "Adulthood";
   }
   else {
      stage = "Senior";
   }

   return stage;
}

int main() {
   Dog buddy;

   buddy.SetAge(1);

   cout << buddy.GetStage();
   return 0;
}
```


1 2 3 4

Check Next

10.5 Initialization and constructors

A good practice is to initialize all variables when declared. This section deals with initializing the data members of a class when a variable of the class type is declared.

Data member initialization (C++11)

Since C++11, a programmer can initialize data members in the class definition. Any variable declared of that class type will initially have those values.

Figure 10.5.1: A class definition with initialized data members.

```
#include <iostream>
#include <string>
using namespace std;

/*** ShopItem class definition ***/
class ShopItem {
public:
   void SetNameAndPrice(string itemName, int itemPrice);
   void Print() const;

private:
   string name = "NoName"; // Ex: "Bag of salad"
   int    price = -1;      // Price in cents. Ex: 199
};

void ShopItem::SetNameAndPrice(string itemName, int itemPrice) {
   name  = itemName;
   price = itemPrice;
}

void ShopItem::Print() const {
   cout << "Name: "  << name  << ", ";
   cout << "Price: " << price << endl;
}
/*** End definitions for ShopItem class ***/


int main() {
   ShopItem item1;  // Auto-calls default constructor

   item1.Print();

   item1.SetNameAndPrice("Soap", 385);
   item1.Print();

   return 0;
}
```

```
Name: NoName, Price: -1
Name: Soap, Price: 385
```


PARTICIPATION ACTIVITY 10.5.1: Initialization.

Consider the example above.

1) When item1 is initially declared, what is the value of item1's price?

Check Show answer

2) After the call to SetNameAndPrice(),

what is the value of item1's price?

Check Show answer

Constructors

C++ provides a special class member function, known as a ***constructor***, that is called *automatically* when a variable of that class type is defined, and which can be used to initialize data members. The following illustrates.

Figure 10.5.2: Adding a constructor member function to the ShopItem class.

```
Name: NoName, Price: -1
Name: Soap, Price: 385
```

```
#include <iostream>
#include <string>
using namespace std;

/*** ShopItem class definition ***/
class ShopItem {
public:
   void SetNameAndPrice(string itemName, int itemPrice);
   void Print() const;
   ShopItem();  // Default constructor

private:
   string name;  // Ex: "Bag of salad"
   int    price; // Price in cents. Ex: 199
};

ShopItem::ShopItem() {  // Default constructor
   name  = "NoName";    // Default name
   price = -1;          // Default price

   return;
}

void ShopItem::SetNameAndPrice(string itemName, int itemPrice) {
   name  = itemName;
   price = itemPrice;

   return;
}

void ShopItem::Print() const {
   cout << "Name: "  << name  << ", ";
   cout << "Price: " << price << endl;

   return;
}

/*** End definitions for ShopItem class ***/

int main() {
   ShopItem item1;  // Auto-calls default constructor

   item1.Print();

   item1.SetNameAndPrice("Soap", 385);
   item1.Print();

   return 0;
}
```


The constructor has the same name as the class. The constructor function has no return type, not even void.

The member function definition begins with `ShopItem::ShopItem() {...}`, which may look strange but is logical: ShopItem:: indicates this will be a ShopItem member function, and ShopItem() is the function name.

The constructor is called automatically for a variable declaration like: ShopItem item1. Calling item1.Print() immediately after the variable declaration prints the values assigned by the constructor, namely "NoName" and "-1".

A constructor that can be called without any arguments is called a ***default constructor***, like the constructor above. If a class does not have a programmer-defined constructor, then the compiler *implicitly* defines a default constructor having no statements.

PARTICIPATION ACTIVITY 10.5.2: Remove the constructor.

Run the program. Then remove the default constructor declaration and definition, run, and note the different initial values.

Load default template... Run

```
#include <iostream>
#include <string>
using namespace std;

/*** ShopItem class definition ***/
class ShopItem {
public:
   void SetNameAndPrice(string itemName, int itemPrice);
   void Print() const;
   ShopItem();  // Default constructor

private:
   string name;  // Ex: "Bag of salad"
   int    price; // Price in cents. Ex: 199
};

ShopItem::ShopItem() {  // Default constructor
   name  = "NoName";    // Default name
   price = -1;          // Default price
```

PARTICIPATION ACTIVITY 10.5.3: Default constructors.

Assume a class named Seat.

1) A default constructor declaration in class Seat { ... } is:

```
class Seat {
   ...
   void Seat();
}
```


- ○ True
- ○ False

2) A default constructor definition has this form:

```
Seat::Seat() {
   ...
}
```

- True
- False

3) Omitting a default constructor is essentially the same as defining a constructor with no statements.

- True
- False

4) The following calls the default constructor once:

```
Seat mySeat;
```

- True
- False

5) The following calls the default constructor once:

```
Seat seat1;
Seat seat2;
```

- True
- False

6) The following calls the default constructor 5 times:

```
vector<Seat> seats(5);
```

- True
- False

Note: Since C++11, data members can be initialized in the class definition as in `int price = -1;`, which is usually preferred over using a constructor. However, sometimes initializations are more complicated, in which case a constructor is needed.

Exploring further:

- Constructors from msdn.microsoft.com

CHALLENGE ACTIVITY 10.5.1: Basic constructor definition.

Define a constructor as indicated. Sample output for below program:

```
Year: 0, VIN: -1
Year: 2009, VIN: 444555666
```

```
#include <iostream>
using namespace std;

class CarRecord {
   public:
      void   SetYearMade(int originalYear);
      void   SetVehicleIdNum(int vehIdNum);
      void   Print() const;
      CarRecord();
   private:
      int    yearMade;
      int    vehicleIdNum;
};

// FIXME: Write constructor, initialize year to 0, vehicle ID num to -1.

/* Your solution goes here  */

void CarRecord::SetYearMade(int originalYear) {
```


Run

10.6 Constructor overloading

Programmers often want to provide different initialization values when creating a new object. A class creator can ***overload*** a constructor by defining multiple constructors differing in parameter types. The following illustrates.

Figure 10.6.1: Overloading a constructor of a Seat class.

```
class Seat {
   public:
      ...
   Seat(); // Default constructor
   Seat(string resfirstName, string resLastName, int resAmountPaid); // Second constructor
      ...
};

Seat::Seat() { // Default constructor
   firstName  = "none";
   lastName   = "none";
   amountPaid = -1;
}

Seat::Seat(string resfirstName, string resLastName, int resAmountPaid) { // Second constructor
   firstName  = resfirstName;
   lastName   = resLastName;
   amountPaid = resAmountPaid;
}
```


Table 10.6.1: Overloaded constructor example.

`Seat mySeat;`	Calls the default constructor to initialize the variable.
`Seat mySeat("Nobody", "Here", 0);`	Calls the second constructor.
`Seat mySeat("Nobody", "Here");`	Yields a compiler error because no constructor has just two string parameters.
`vector<Seat>  mySeats(5);`	Calls the default constructor for each vector element.

In the last case involving a vector, there is no simple way for the programmer to cause the second constructor to be called. Several similar cases exist where the default constructor will automatically be called. However, *if the programmer creates any constructor, then no implicit default constructor is created*. So if the programmer creates any constructor, the programmer should also include a default constructor that can be called without any arguments, to handle such cases (else a compiler error is generated).

Programmers commonly create a default constructor having parameters with default values. That constructor is a default constructor because it can be called without arguments, while also supporting initialization by the user. The following code illustrates.

Figure 10.6.2: Default constructor using default parameter values.

```
class Seat {
   public:
      ...
   Seat(string resfirstName = "none", string resLastName = "none", int resAmountPaid = -1); //
Default constructor
      ...
};

Seat::Seat(string resfirstName, string resLastName, int resAmountPaid) { // Default constructor
   firstName  = resfirstName;
   lastName   = resLastName;
   amountPaid = resAmountPaid;
}
```


That constructor will be called for all of the earlier four example variable declarations.

The initialization values could alternatively appear in the parameter list of the function definition rather than of the function declaration, but some programmers state that putting the initialization values in the declaration makes the initialization values more noticeable to a class user.

PARTICIPATION ACTIVITY 10.6.1: Constructor overloading.

Given:

```
ShopItem::ShopItem() {  // Default constructor
   name  = "NoName";    // Default name
   price = -1;          // Default price
   return;
}

ShopItem::ShopItem(string itemName, int itemPrice) {
   name  = itemName;
   price = itemPrice;
   return;
}
```

1) Which constructor is called for:

```
ShopItem item1;
```

- First
- Second
- Error

2) Which constructor is called for:

```
ShopItem item2(250, "Pasta");
```


- First
- Second
- Error

3) Which constructor is called for:

```
ShopItem item3("Shampoo",
```

```
199);
```

- First
- Second
- Error

4) Which constructor is called for:

```
ShopItem item2("Nuts");
```

- First
- Second
- Error

5) Could the second constructor be rewritten as follows without yielding an error?

```
ShopItem(string itemName =
"none", int itemPrice = 0);
```

- Yes
- No

6) Could the two constructors be replaced by this one constructor?

```
ShopItem(string itemName =
"none", int itemPrice = 0);
```

- Yes
- No

CHALLENGE ACTIVITY 10.6.1: Constructor overloading.

Write a second constructor as indicated. Sample output:

```
User1: Minutes: 0, Messages: 0
User2: Minutes: 1000, Messages: 5000
```



```
1 #include <iostream>
2 using namespace std;
3
4 class PhonePlan{
5    public:
6       PhonePlan();
```

```
7       PhonePlan(int numMinutes, int numMessages);
8       void   Print() const;
9    private:
10      int    freeMinutes;
11      int    freeMessages;
12 };
13
14 PhonePlan::PhonePlan() {     // Default constructor
15    freeMinutes  = 0;
16    freeMessages = 0;
17    return;
18 }
```


Run

10.7 Constructor initializer lists

A ***constructor initializer list*** is an alternative approach for initializing data members in a constructor, coming after a colon and consisting of a comma-separated list of variableName(initValue) items.

Figure 10.7.1: Member initialization: (left) Using statements in the constructor, (right) Using a constructor initializer list.

```
#include <iostream>
using namespace std;

class SampleClass {
   public:
      SampleClass();
      void Print() const;

   private:
      int field1;
      int field2;
};

SampleClass::SampleClass() {
   field1 = 100;
   field2 = 200;
}

void SampleClass::Print() const {
   cout << "Field1: " << field1 <<
endl;
   cout << "Field2: " << field2 <<
endl;
}

int main() {
   SampleClass myClass;
   myClass.Print();
   return 0;
}
```

```
Field1: 100
Field2: 200
```

```
#include <iostream>
using namespace std;

class SampleClass {
   public:
      SampleClass();
      void Print() const;

   private:
      int field1;
      int field2;
};

SampleClass::SampleClass() : field1(100),
field2(200) {
}

void SampleClass::Print() const {
   cout << "Field1: " << field1 << endl;
   cout << "Field2: " << field2 << endl;
}

int main() {
   SampleClass myClass;
   myClass.Print();
   return 0;
}
```

```
Field1: 100
Field2: 200
```


PARTICIPATION ACTIVITY 10.7.1: Member initialization.

1) Convert this constructor to use a constructor initializer list.

```
MyClass::MyClass() {
   x = -1;
   y = 0;
}
```

```
MyClass::MyClass()
{
}
```

Check Show answer

The approach is important when a data member is a class type that must be explicitly constructed. Otherwise, that data member is by default constructed. Ex: If you have studied vectors, consider a data member consisting of a vector of size 2.

Figure 10.7.2: Member initialization in a constructor.

```
#include <iostream>
#include <vector>
using namespace std;

class SampleClass {
   public:
      SampleClass();
      void Print() const;

   private:
      vector<int> itemList;
};

SampleClass::SampleClass() {
   // itemList gets default constructed,
size 0
   itemList.resize(2);
}

void SampleClass::Print() const {
   cout << "Item1: " << itemList.at(0) <<
endl;
   cout << "Item2: " << itemList.at(1) <<
endl;
}

int main() {
   SampleClass myClass;
   myClass.Print();
   return 0;
}
```

```
Item1: 0
Item2: 0
```

```
#include <iostream>
#include <vector>
using namespace std;

class SampleClass {
   public:
      SampleClass();
      void Print() const;

   private:
      vector<int> itemList;
};

SampleClass::SampleClass() : itemList(2) {
   // itemList gets constructed with size 2
}

void SampleClass::Print() const {
   cout << "Item1: " << itemList.at(0) <<
endl;
   cout << "Item2: " << itemList.at(1) <<
endl;
}

int main() {
   SampleClass myClass;
   myClass.Print();
   return 0;
}
```

```
Item1: 0
Item2: 0
```


On the left, the constructor initially creates a vector of size 0, then resizes to size 2. On the right, itemList(2) is provided in the SampleClass constructor initialization list, causing the vector constructor to be called with size 2. Using the initialization list avoids the inefficiency of constructing and then modifying an item.

Note: Since C++11, the data member could have been initialized in the class definition: `vector<int> itemList(2);`. However, initialization lists are still useful for other cases.

PARTICIPATION ACTIVITY 10.7.2: Constructor initializer list.

Consider the example above.

1) On the left, itemList is first constructed with size 0, then resized to size 2.

   - True
   -

False

2) On the right, itemList is first constructed with size 0, then resized to size 2.

- True
- False

Exploring further:

- Classes from cplusplus.com, see "Member initialization in constructors" section.
- Constructors from msdn.microsoft.com, see "Member lists".

10.8 Unit testing (classes)

Like a chef who tastes the food before allowing it to be served to diners, a programmer should test a class before allowing it to be used in a program. Testing a class can be done by creating a special program, sometimes known as a ***testbench***, whose job is to thoroughly test the class. The process of creating and running a program that tests a specific item (or "unit"), such as a function or a class, is known as ***unit testing***.

PARTICIPATION ACTIVITY 10.8.1: Unit testing of a class.

Animation captions:

1. Typical program may not thoroughly use all class items.
2. Tester class' job is to thoroughly test all class items
3. After testing, class is ready for use. Tester class kept for later tests.

Figure 10.8.1: Unit testing of a class.

Class to test:

```
#include <iostream>
using namespace std;

// Note: This class intentionally has
errors

class StatsInfo {
public:
   void SetNum1(int numVal);
   void SetNum2(int numVal);
   int  GetNum1() const;
   int  GetNum2() const;
   int  GetAverage() const;
   void PrintNums()  const;

private:
   int num1;
   int num2;
};

void StatsInfo::SetNum1(int numVal) {
   num1 = numVal;
}

void StatsInfo::SetNum2(int numVal) {
   num2 = numVal;
}

int StatsInfo::GetNum1() const {
   return num1;
}

int StatsInfo::GetNum2() const {
   return num1;
}

int StatsInfo::GetAverage() const {
   return num1 + num2 / 2;
}
```

Testbench:

```
int main() {
   StatsInfo testData;

   // Typical testbench tests more thoroughly

   cout << "Beginning tests." << endl;

   // Check set/get num1
   testData.SetNum1(100);
   if (testData.GetNum1() != 100) {
      cout << "   FAILED set/get num1" << endl;
   }

   // Check set/get num2
   testData.SetNum2(50);
   if (testData.GetNum2() != 50) {
      cout << "   FAILED set/get num2" << endl;
   }

   // Check GetAverage()
   testData.SetNum1(10);
   testData.SetNum2(20);
   if (testData.GetAverage() != 15) {
      cout << "   FAILED GetAverage for 10, 20"
<< endl;
   }

   testData.SetNum1(-10);
   testData.SetNum2(0);
   if (testData.GetAverage() != -5) {
      cout << "   FAILED GetAverage for -10, 0"
<< endl;
   }

   cout << "Tests complete." << endl;

   return 0;
}
```

```
Beginning tests.
   FAILED set/get num2
   FAILED GetAverage for 10, 20
   FAILED GetAverage for -10, 0
Tests complete.
```

The testbench program creates an object of the class, then invokes public functions to ensure they work as expected. Above, the test for num2's set/get functions failed. Likewise, the GetAverage function failed. You can examine those functions and try to find the bugs.

A good practice is to create the testbench program to automatically check for correct execution rather than relying on a user reading program output, as done above. The program may print a message for each failed test, but not each passed test, to ensure failures are evident. ***Assert statements*** are commonly used for such checks (not discussed here). Also, good practice is to keep each test independent from the previous case, as much as possible. Note, for example, that the get average test did not rely on values from the earlier set/get tests. Also note that different values were used for each set/get (100 for num1, 50 for num2) so that problems are more readily detected.

A goal of testing is to achieve complete ***code coverage***, meaning all code is executed at least once. Minimally for a class, that means every public function is called at least once. Of course, the programmer of a class knows about a class' implementation and thus will want to also ensure that every private helper function is called, and that every line of code within every function is executed at least once, which may require multiple calls with different input values for a function with branches.

While achieving complete code coverage is a goal, achieving that goal still does not mean the code is correct. Different input values can yield different behavior. Tests should include at least some typical values and some ***border cases***, which for integers may include 0, large numbers, negative numbers, etc. A good testbench includes more test cases than the above example.

PARTICIPATION ACTIVITY 10.8.2: Unit testing of a class.

1) A class should be tested individually (as a "unit") before being used in another program.
   - True
   - False

2) A testbench that calls each function at least once ensures that a class is correct.
   - True
   - False

3) If every line of code was executed at least once (complete code coverage) and all tests passed, the class must be bug free.
   - True
   - False

4) A programmer should test all possible values when testing a class, to ensure correctness.
   - True
   - False

5) A testbench should print a message for each test case that passes and for each that fails.

- True
- False

A testbench may be in a separate file from the class, such as in file MyClassTest.cpp for a class MyClass.

The testbench should be maintained for the lifetime of the class code, and run again (possibly updated first) whenever a change is made to the class. Running an existing testbench whenever code is changed is known as ***regression testing***, due to checking whether the change caused the code to "regress", meaning to fail previously-passed test cases.

Testbenches themselves can be complex programs, with thousands of test cases, each requiring tens of statements, that may themselves contain errors. Many tools and techniques exist to support testing, not discussed here. Due to testbench complexity and importance, many companies employ test engineers whose sole job is to test. Testing commonly occupies a large percentage of program development time, e.g., nearly half of a commercial software project's development effort may go into testing.

Exploring further:

- C++ Unit testing frameworks from accu.org.

CHALLENGE ACTIVITY 10.8.1: Unit testing of a class.

Write a unit test for addInventory(). Call redSweater.addInventory() with parameter sweaterShipment. Print the shown error if the subsequent quantity is incorrect. Sample output for failed unit test given initial quantity is 10 and sweaterShipment is 50:

```
Beginning tests.
   UNIT TEST FAILED: addInventory()
Tests complete.
```

Note: UNIT TEST FAILED is preceded by 3 spaces.


```
1 #include <iostream>
2 using namespace std;
3
4 class InventoryTag {
5 public:
6    InventoryTag();
7    int getQuantityRemaining() const;
8    void addInventory(int numItems);
9
```

```
10 private:
11    int quantityRemaining;
12 };
13
14 InventoryTag::InventoryTag() {
15     quantityRemaining = 0;
16 }
17
18 int InventoryTag::getQuantityRemaining() const {
19    return quantityRemaining;
```

Run

10.9 The 'this' implicit parameter

An object's member function is called using the syntax `object.Function()`. The compiler converts that syntax into a function call with the object implicitly passed as a parameter—actually a pointer to the object (which is similar to a pass by reference parameter). So you can think of `object.Function(...)` getting converted to `Function(object, ...)`. The object is known as an ***implicit parameter*** of the member function.

Within a member function, the implicitly-passed object pointer is accessible via the name ***this***. In particular, a member can be accessed as `this->member`. The -> is the member access operator for a pointer, similar to the "." operator for non-pointers.

Figure 10.9.1: Using 'this' to refer to an object's member.

```
Square's area: 1.44
```

```
#include <iostream>
using namespace std;

class ShapeSquare {
   public:
      void   SetSideLength(double sideLength);
      double GetArea() const;
   private:
      double sideLength;
};

void ShapeSquare::SetSideLength(double sideLength) {
   this->sideLength = sideLength;
   // Data member      Parameter
   return;
}

double ShapeSquare::GetArea() const{
   return sideLength * sideLength; // Both refer to data member
}

int main() {
   ShapeSquare square1;

   square1.SetSideLength(1.2);
   cout << "Square's area: " << square1.GetArea() << endl;

   return 0;
}
```


Using `this->` makes clear that a class member is being accessed. Such use is essential if a data member and parameter have the same identifier, as in the above example, because the parameter name dominates.

PARTICIPATION ACTIVITY 10.9.1: The 'this' implicit parameter.

Given a class Spaceship with private data member numYears and public member function:
`void Spaceship::AddNumYears(int numYears)`

1) In AddNumYears(), which would set data member numYears to 0?
   - numYears = 0;
   - this.numYears = 0;
   - this->numYears = 0;

2) In AddNumYears(), which would assign the parameter numYears to the data member numYears?
   - numYears = this->numYears;
   - this->numYears = numYears;

3) In AddNumYears(), which would add the parameter

numYears to the existing value of data member numYears?

- this->numYears = this->numYears + numYears;
- this->numYears = numYears + numYears;
- Not possible.

4) In main(), given variable declaration: Spaceship ss1, which sets ss1's numYears to 5?

- ss1.numYears = 5;
- ss1->numYears = 5;
- this->numYears = 5;
- None of the above.

The following animation illustrates. When an object's member function is called, the object's memory address is passed to the function via the implicit "this" parameter. An access in that member function to `this->hrs` first goes to the object's address, then to the hrs data member. If that member function instead had the assignment hrs = h, because no other item in the member function is named hrs, then hrs is essentially converted by the compiler to `this->hrs`.

PARTICIPATION ACTIVITY 10.9.2: How a member function works.

Animation captions:

1. travTime is an object of class type ElapsedTime.
2. When travTime's SetTime() member function is called, travTime's memory address is passed t the function via the implicit "this" parameter.
3. The implicitly-passed object pointer is accessible within the member function via the name "this". Ex: this->hours first goes to travTime's address, then to the hours data member.

PARTICIPATION ACTIVITY 10.9.3: The 'this' pointer.

1) Complete the code to assign the value of minutes to data member minutes, using `this->` notation.

```
void
ElapsedTime::SetMinutes(int
minutes) {

      [          ]    =
minutes;

      return;

}
```


Check Show answer

2) Complete the code to assign the value of parameter hours to data member hours, using `this->` notation.

```
void ElapsedTime::SetHours(int
hours) {

      [          ];

      return;

}
```

Check Show answer

Exploring further:

- The 'this' pointer from msdn.microsoft.com.

CHALLENGE ACTIVITY 10.9.1: The this implicit parameter.

Define the missing member function. Use "this" to distinguish the local member from the parameter name.

```
1 #include <iostream>
2 using namespace std;
3
```

```
class CablePlan{
   public:
      void  SetNumDays(int numDays);
      int   GetNumDays() const;
   private:
      int   numDays;
};

// FIXME: Define SetNumDays() member function, using "this" implicit parameter.
void CablePlan::SetNumDays(int numDays) {

   /* Your solution goes here  */

   return;
}
```


Run

10.10 Operator overloading

C++ allows a programmer to redefine the functionality of built-in operators like +, -, and *, to operate on programmer-defined objects, a process known as ***operator overloading***. Suppose a class TimeHrMn has data fields hours and minutes. Overloading + would allow the following.

Figure 10.10.1: Operator overloading allows use of operators like + on classes.

```
TimeHrMn time1(3, 22);
TimeHrMn time2(2, 50);
TimeHrMn timeTot;

timeTot = time1 + time2;
timeTot.Print();
```

```
H:5, M:72
```

The + operator was somehow redefined to add TimeHrMn objects' hours and minutes fields separately (3 + 2 is 5, 22 + 50 is 72), leading to simple readable code.

Overloading an operator is straightforward, but with a slightly odd syntax. To overload +, the programmer creates a member function named operator+, as shown below.

Although + requires left and right operands as in `time1 + time2`, the member function only requires the right operand (rhs: right-hand-side) as the parameter, because the left operand is the calling object. In other words, `time1 + time2` is equivalent to the function call `time1.operator+(time2)`, which is valid syntax but almost never used.

Figure 10.10.2: Overloading the + operator.

```
#include <iostream>
using namespace std;

class TimeHrMn {
public:
   TimeHrMn(int timeHours = 0, int timeMinutes = 0);
   void Print() const;
   TimeHrMn operator+(TimeHrMn rhs) ;
private:
   int hours;
   int minutes;
};

// Overload + operator for TimeHrMn
TimeHrMn TimeHrMn::operator+(TimeHrMn rhs) {
   TimeHrMn timeTotal;

   timeTotal.hours   = hours   + rhs.hours;
   timeTotal.minutes = minutes + rhs.minutes;

   return timeTotal;
}

TimeHrMn::TimeHrMn(int timeHours, int timeMinutes) {
   hours   = timeHours;
   minutes = timeMinutes;

   return;
}

void TimeHrMn::Print() const {
   cout << "H:" << hours << ", " << "M:" << minutes << endl;

   return;
}

int main() {
   TimeHrMn time1(3, 22);
   TimeHrMn time2(2, 50);
   TimeHrMn timeTot;

   timeTot = time1 + time2;
   timeTot.Print();

   return 0;
}
```



```
H:5, M:72
```

In the statement `timeTotal.hours = hours + rhs.hours;`, hours refers to the left operand's (time1 above) hours member. If you have studied the *this* implicit parameter, then note that the statement is equivalent to `timeTotal.hours = this->hours + rhs.hours;`.

PARTICIPATION ACTIVITY 10.10.1: Operator overloading basics.

1) Given `TimeHrMin time1(10, 0)` and `TimeHrMin time2(3, 5)`, and

the above overloading of +, what is timeTot.hours after `timeTot = time1 + time2`?

Check Show answer

2) Write the start of a TimeHrMn member function definition that overloads the - operator, naming the parameter rhs.

```
// Overloaded '-' function definition

{
   /* Implementation */
}
```

Check Show answer

3) Which parameter should be removed from this line, that strives to overload the * operator? Type the parameter name only; don't list the type.

```
TimeHrMn TimeHrMn::operator*
(TimeHrMn lhs, TimeHrMn rhs)
{
```

Check Show answer

When an operator like + has been overloaded, the compiler determines which + operation to invoke based on the operand types. In `4 + 9`, the compiler sees two integer operands and thus applies the built-in + operation. In `time1 + time2`, where time1 and time2 are TimeHrMn objects, the compiler sees two TimeHrMn operands and thus invokes the programmer-defined function.

A programmer can define several functions that overload the same operator, as long as each involves different types so that the compiler can determine which to invoke.

Figure 10.10.3: Overloading the + operator multiple times.

```
H:5, M:72
H:13, M:22
99
```

```
#include <iostream>
using namespace std;

class TimeHrMn {
public:
   TimeHrMn(int timeHours = 0, int timeMinutes = 0);
   void Print() const;
   TimeHrMn operator+(TimeHrMn rhs);
   TimeHrMn operator+(int rhsHours);
private:
   int hours;
   int minutes;
};

// Operands: TimeHrMn, TimeHrMn. Call this "A"
TimeHrMn TimeHrMn::operator+(TimeHrMn rhs) {
   TimeHrMn timeTotal;

   timeTotal.hours   = hours   + rhs.hours;
   timeTotal.minutes = minutes + rhs.minutes;

   return timeTotal;
}

// Operands: TimeHrMn, int. Call this "B"
TimeHrMn TimeHrMn::operator+(int rhsHours) {
   TimeHrMn timeTotal;

   timeTotal.hours = hours + rhsHours;
   timeTotal.minutes = minutes; // Stays same

   return timeTotal;
}

TimeHrMn::TimeHrMn(int timeHours, int timeMinutes) {
   hours  = timeHours;
   minutes = timeMinutes;

   return;
}

void TimeHrMn::Print() const {
   cout << "H:" << hours << ", " << "M:" << minutes << endl;

   return;
}

int main() {
   TimeHrMn time1(3, 22);
   TimeHrMn time2(2, 50);
   TimeHrMn timeTot;
   int num = 91;

   timeTot = time1 + time2; // Invokes "A"
   timeTot.Print();

   timeTot = time1 + 10;    // Invokes "B"
   timeTot.Print();

   cout << num + 8 << endl; // Invokes built-in add

   // timeTot = 10 + time1; // ERROR: No (int, TimeHrMn)

   return 0;
}
```


©zyBooks 04/05/18 21:46 261830
Julian Chan
WEBERCS2250ValleSpring2018

The first + involves two TimeHrMn operands, so the compiler invokes the first operator+ function ("A"). The second + involves TimeHrMn and int operands, so the compiler invokes the second operator+ function ("B"). The third + involves two int operands, so the compiler invokes the built-in + operation. The fourth +, commented out, involves an int and TimeHrMn operands. Because no function has those operands ("B" has TimeHrMn and int, not int and TimeHrMn; order matters), that statement would generate a compiler error.

PARTICIPATION ACTIVITY 10.10.2: Determining which function is invoked.

Given:

```
Course crs1;
Course crs2;
int    num1;
int    num2;
```

num2 + crs2: **num1 + num2;** **crs1 + num1:** **crs1 + crs2:**

Error

Course Course::operator+(int val) {

Course Course::operator+(Course rhs) {

Built-in + operation

Reset

Exploring further:

- Overloadable operators from cplusplus.com. Provides a list of operators that can be overloaded, including a description of how to declare overloaded operator functions for operators with different operands like += and ++.

CHALLENGE ACTIVITY 10.10.1: Operator overloading.

Overload the + operator as indicated. Sample output for the given program:

```
First vacation: Days: 7, People: 3
Second vacation: Days: 12, People: 3
```

```
#include <iostream>
using namespace std;

class FamilyVacation{
   public:
      void  SetNumDays(int dayCount);
      void  SetNumPeople(int peopleCount);
      void  Print() const;
      FamilyVacation operator+(int moreDays);
   private:
      int   numDays;
      int   numPeople;
};

void FamilyVacation::SetNumDays(int dayCount) {
   numDays = dayCount;
   return;
}

```


Run

10.11 Abstract data types: Introduction

An ***abstract data type (ADT)*** is a data type whose creation and update are constrained to specific well-defined operations. A class can be used to implement an ADT. ***Information hiding*** is a key ADT aspect wherein the internal implementation of an ADT's data and operations are hidden from the ADT user. Information hiding allows an ADT user to be more productive by focusing on higher-level concepts. Information hiding also allows an ADT developer to improve or modify the internal implementation without requiring changes to programs using the ADT. Information hiding is also known as ***encapsulation***.

In the physical world, common kitchen appliances such as a coffee maker have a well-defined interface. The interface for most drip coffee makers include: a water tank for adding water, a basket for adding ground coffee, a carafe for the brewed coffee, and an on/off switch. A user can brew coffee with most coffee makers without understanding how the coffee maker works internally. The manufacturers of those coffee makers can make

improvements to how the coffee maker works, perhaps by increasing the size of the heating element to boil water faster. However, such improvements do not change how the user uses the coffee maker.

Programmers refer to separating an object's *interface* from its *implementation*; the user of an object need only know the object's interface (public member function declarations) and not the object's implementation (member function definitions and private data).

PARTICIPATION ACTIVITY 10.11.1: Public class members define the interface for an abstract data type.

Animation captions:

1. ADT's public member define interface for using ADT
2. ADT's implementation consists of private data and operations that are hidden from user

The string and vector types are examples of ADTs. As those are part of the standard C++ library, a programmer does not have (easy) access to the actual C++ code. Instead, programmers typically rely on web pages describing the ADT's public member function declarations.

PARTICIPATION ACTIVITY 10.11.2: Abstract data types.

1) All classes implement an abstract data type.
   - ○ True
   - ○ False

2) Programmers must understand all details of an ADT's implementation to use the ADT.
   - ○ True
   - ○ False

3) An ADT's interface is defined by the ADT's public member function declarations.
   - ○ True
   - ○ False

10.12 Vector ADT

The vector type is an Abstract Data Type (ADT) implemented as a class.[Tp] Below are commonly-used public member functions.

Table 10.12.1: Vector ADT functions.

Assumes: `vector<int> teamNums(5);`
size_type is an unsigned integer type. T represents the vector's type, such as int for vector<int>.

at()	`at(size_type n)`
	
	Accesses element n.
		`teamNums.at(3) =`
		`Assigns 99 to el`
		`x = teamNums.at(`
		`Assigns element`
		`x`

size()	`size_type size() const;`
	
	Returns vector's size.
		`if (teamNums.siz`
		`Size is 5 so con`
		`...`
		`}`

empty()	`bool empty() const;`
	
	Returns true if size is 0.
		`if (teamNums.emp`
		`is 5 so conditio`
		`...`
		`}`

clear()	Removes all elements. Vector size becomes 0.	`teamNums.clear()`
		`Vector now has n`
		`cout << teamNums`
		`Prints 0`
		`teamNums.at(3) =`
		`Error; element 3`

push_back()	`void push_back(const T& x);`
	
	Copies x to new element at vector's end, increasing size by 1. Parameter is pass by reference to avoid making local copy, but const to make clear not changed.
		`// Assume vector`
		`teamNums.push_ba`
		`Vector is: 77`
		`teamNums.push_ba`
		`Vector is: 77, 8`
		`cout << teamNums`
		`Prints 2`

erase()	`iterator erase (iteratorPosition);`
	
	Removes element from position. Elements from higher positions are shifted back to fill gap. Vector size decrements.
		`// Assume vector`
		`teamNums.erase(t`
		`+ 1); // Now 77,`
		`// (Strange posi`
		`explained below)`

insert()	`iterator insert(iteratorPosition, const T& x);`	`// Assume vector`
		`teamNums.insert(`
		`+ 1, 33); // Now`

	Copies x to element at position. Items at that position and higher are shifted over to make room. Vector size increments.

Use of at(), size(), empty(), and clear() should be straightforward.

PARTICIPATION ACTIVITY 10.12.1: Vector functions at(), size(), empty(), and clear().

Given `vector<int> itemList(10);` Assume all elements have been assigned 0.

1) itemList().size returns 10.
   - True
   - False

2) itemList.size(10) returns 10.
   - True
   - False

3) itemList.size() returns 10.
   - True
   - False

4) itemList.at(10) returns 0.
   - True
   - False

5) itemList.empty() removes all elements.
   - True
   - False

6) After itemList.clear(), itemList.at(0) is an invalid access.
   - True
   - False

push_back() appends an item to the vector's end, automatically resizing the vector.

PARTICIPATION ACTIVITY 10.12.2: Vector push_back() member function.

Animation captions:

1. When initially declared, the vector dailySales has a size of 0.
2. The push_back() function appends a new element to the vector's end and automatically resizes the vector.

You can probably deduce that the vector class is implemented using several private data members, such as an array, a current size, etc. The power of an ADT, however, is that the user need not know how the ADT is implemented.

Below is an example using push_back(). The program assists a soccer coach scouting players, allowing the coach to enter the jersey number of players, and printing a list of those numbers when requested.

Figure 10.12.1: Using vector member functions: A player jersey numbers program.

```
Commands: 'a' add, 'p' print
   'q' quits
Command: p
Command: a
 Player number: 23
Command: a
 Player number: 47
Command: p
 23
 47
Command: a
 Player number: 19
Command: p
 23
 47
 19
Command: q
```

```
#include <iostream>
#include <vector>
using namespace std;

// Adds playerNum to end of vector
void PlayersAdd(vector<int>& players, int playerNum) {
   players.push_back(playerNum);
   return;
}

void PlayersPrint(const vector<int>& players) {
   unsigned int i = 0;
   for (i = 0; i < players.size(); ++i) {
      cout << " " << players.at(i) << endl;
   }
   return;
}

// Maintains vector of player numbers
int main() {
   vector<int> players;
   int playerNum = 0;
   char userKey = '?';

   cout << "Commands: 'a' add, 'p' print" << endl;
   cout << "    'q' quits" << endl;
   while (userKey != 'q') {
      cout << "Command: ";
      cin  >> userKey;
      if (userKey == 'a') {
         cout << " Player number: ";
         cin  >> playerNum;
         PlayersAdd(players, playerNum);
      }
      else if (userKey == 'p') {
         PlayersPrint(players);
      }
   }

   return 0;
}
```


Note that the programmer did not specify an initial vector size in main(), meaning the size is 0. Note from the output that the items are stored in the vector in the order they were added.

PARTICIPATION ACTIVITY 10.12.3: push_back() function.

Given: `vector<int> itemList;`
If appropriate type: Error
Answer the questions in order; each may modify the vector.

1) What is the initial vector's size?

Check Show answer

2) After itemList(0) = 99, what is the

vector's size?

Check Show answer

3) After itemList.push_back(99), what is the vector's size?

Check Show answer

4) After itemList.push_back(77), what are the vector's contents? Type element values in order separated by one space as in: 44 66

Check Show answer

5) After itemList.push_back(44), what is the vector's size?

Check Show answer

6) What does itemList.at(itemList.size()) return?

Check Show answer

The insert() function takes a position argument indicating where the new element should be inserted. However, position is not just a number like 1, but is rather: myVector.begin() + 1. The reason is beyond our scope here, but has to do with *iterators* that can be useful when iterating through a vector in a loop. The erase() function is similar.

PARTICIPATION ACTIVITY 10.12.4: insert() and erase() vector member functions.

Animation captions:

1. The erase() function takes a position argument indicating where the element should be removed. Elements from higher positions are shifted back. The vector size is automatically decremented.
2. The insert() function takes a position argument indicating where the element should be added and the element's value. Elements in that position and higher positions are shifted forward. The vector size is automatically incremented.
3. Note that the position argument is not an integer, but an iterator. The begin() function returns a iterator pointing to the first element of the vector. Then, an integer value is added to indicate the desired element's position.

The erase() function can be used to extend the player jersey numbers program with a player delete option, as shown below.

Figure 10.12.2: Using the vector erase() function.

```
Commands: 'a' add, 'p' print, 'd' del
   'q' quits
Command: a
 Player number: 23
Command: a
 Player number: 47
Command: a
 Player number: 19
Command: p
 23
 47
 19
Command: d
 Player number: 23
Command: p
 47
 19
Command: d
 Player number: 19
Command: p
 47
Command: q
```

```
#include <iostream>
#include <vector>
using namespace std;

// Adds playerNum to end of vector
void PlayersAdd(vector<int>& players, int playerNum) {
   players.push_back(playerNum);
   return;
}

void PlayersPrint(const vector<int>& players) {
   unsigned int i = 0;
   for (i = 0; i < players.size(); ++i) {
      cout << " " << players.at(i) << endl;
   }
   return;
}

// Deletes all occurrences of playerNum from vector
void PlayersDel(vector<int>& players, int playerNum) {
   unsigned int i = 0;
   for (i = 0; i < players.size(); ++i) {
      if (players.at(i) == playerNum) {
         players.erase(players.begin() + i); // Delete
      }
   }
   return;
}

// Maintains vector of player numbers
int main() {
   vector<int> players;
   int playerNum = 0;
   char userKey = '?';

   cout << "Commands: 'a' add, 'p' print, 'd' del" <<
endl;
   cout << "    'q' quits" << endl;
   while (userKey != 'q') {
      cout << "Command: ";
      cin  >> userKey;
      if (userKey == 'a') {
         cout << " Player number: ";
         cin  >> playerNum;
         PlayersAdd(players, playerNum);
      }
      else if (userKey == 'p') {
         PlayersPrint(players);
      }
      else if (userKey == 'd') {
         cout << " Player number: ";
         cin  >> playerNum;
         PlayersDel(players, playerNum);
      }
   }

   return 0;
}
```


©zyBooks 04/05/18 21:46 261830
Julian Chan
WEBERCS2250ValleSpring2018

A common use of insert() is to insert a new item in sorted order.

PARTICIPATION ACTIVITY 10.12.5: Intuitive depiction of how to add items to a vector while maintaining items in ascending sorted order.

Animation captions:

1. The first number is added to the vector.
2. 44 is greater than 27 so it is added to the end of the vector.
3. 9 is less than 27. 44 and 27 are moved down so 9 can be added to front of the vector.
4. The rest of the numbers are added in the appropriate spots.

PARTICIPATION ACTIVITY 10.12.6: Insert in sorted order.

Run the program and observe the output to be: 55 4 250 19. Modify the numsInsert function to insert each item in sorted order. The new program should output: 4 19 55 250

Load default template... Run

```
1
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 void numsInsert(vector<int>& numsList, int newNum) {
7    unsigned int i = 0;
8    for (i = 0; i < numsList.size(); ++i) {
9       if (newNum < numsList.at(i)) {
10         // FIXME: insert newNum at element i
11         break; // Exits the for loop
12      }
13   }
14
15   // FIXME: change so executes if higher number NOT fou
16   // Change "true" to "i == ??" (determine what ?? shou
17   if (true) { // No higher number was found, so append
18      numsList.push_back(newNum);
19
```

PARTICIPATION ACTIVITY 10.12.7: The insert() and erase() functions.

Given: `vector<int> itemList;`
Assume itemList currently contains: 33 77 44.
Answer questions in order, as each may modify the vector.

1) itemList.at(1) returns 77.

- ○ True
- ○ False

2) itemList.insert(itemList.begin() + 1, 55) changes itemList to:

33 55 77 44.

- True
- False

3) itemList.insert(itemList.begin() + 0, 99) inserts 99 at the front of the list.

- True
- False

4) Assuming itemList is 99 33 55 77 44, then itemList.erase(itemList.begin() + 55) results in:
99 33 77 44

- True
- False

5) To maintain a list in ascending sorted order, a given new item should be inserted at the position of the first element that is greater than the item.

- True
- False

6) To maintain a list in descending sorted order, a given new item should be inserted at the position of the first element that is equal to the item.

- True
- False

Exploring further:

- Vectors at cplusplus.com
- Vectors at msdn.microsoft.com

CHALLENGE ACTIVITY 10.12.1: Modifying vectors.

Modify the existing vector's contents, by erasing the element at index 1 (initially 200), then inserting 100 and 102 in the shown locations. Use Vector ADT's erase() and insert() only, and remember that the first argument of those functions is special, involving an iterator and not just an integer. Sample output of below program:

```
100 101 102 103
```



```
#include <iostream>
#include <vector>
using namespace std;

void PrintVectors(vector<int> numsList) {
   int i;
   for (i = 0; i < numsList.size(); ++i) {
      cout << numsList.at(i) << " ";
   }
   cout << endl;
}

int main() {
   vector<int> numsList;

   numsList.push_back(101);
   numsList.push_back(200);
   numsList.push_back(103);

```

Run

(*Tp) Actually, the type is implemented as a class *template* that supports different types such as vector<int> or vector<string>, but we don't discuss templates here.

10.13 Namespaces

This section explains why the programs in this material have the line `using namespace std;` at the top.

As programs become larger, carefully organizing the code becomes more important, in order to create correct code, to maintain code, to reuse code, to allow different people to develop different parts of the code, and more. Separating code into multiple files is one key method for keeping code well-organized. However, when programs become very large, names of classes (especially created by different programmers) may conflict with one another. For example, one programmer may develop a class named Seat that refers to a seat in an auditorium, while

another programmer may develop an entirely different class also named Seat that refers to the capital building of a county (the county seat). Now if another programmer wishes to create a reservation system for an auditorium in the capital building, the programmer may want to use both Seat classes, but the compiler will generate an error due to duplicate names.

A solution is to place all code related to the auditorium seat inside a "namespace" with a name like "auditorium", and all code related to the capital building inside a namespace like "capital_building". The syntax is simple:

Figure 10.13.1: Namespace syntax.

```
namespace auditorium {
  class Seat { ... }
  // Other class definitions ...
  // Function definitions ...
  // Etc.
}
```

Then a program that uses that class would declare a variable seat1 of type Seat as follows: `auditorium::Seat customerSeat;`. Likewise, the program could declare another variable as `capital_building::Seat buildingSeat;`. Note: The syntax involving "::" has a similar meaning to that used for defining member functions.

All items in the C++ standard library are enclosed in the namespace *std*. As such, to use an item like cout or string, a program must use the syntax `std::cout` or `std::string`. Such syntax can be quite distracting to someone first learning to program. C++ provides the `using namespace` construct to simplify the syntax. When the line `using namespace std;` appears near the top of a file, then for any names later in the file that are not otherwise declared (as a parameter, local variable, etc.), such as string, the compiler checks for `std::string`. This explains why `using namespace std;` appears at the top of this material's programs.

Other than using the std namespace, beginning programmers likely will not need to know more about namespaces until working on much larger projects. Namespaces are typically used for projects involving code written by multiple programmers. Also, note that most programming guidelines discourage use of the `using namespace` statement except perhaps for std.

PARTICIPATION ACTIVITY 10.13.1: Namespaces.

1) A namespace helps avoid name conflicts among classes, functions, and other items in a large program.
   - ○ True
   - ○ False

2) Standard library classes, functions, etc.,

are part of a namespace named std.

- True
- False

3) The line `using namespace std;` is required in any program.

- True
- False

4) Without `using namespace std`, a programmer can access cout via the notation std::cout.

- True
- False

5) Without any using namespace ___ line, if the compiler sees `cout << num1`, the compiler automatically checks in the standard library namespace for cout.

- True
- False

10.14 Separate files for classes

Programmers typically put all code for a class into two files, separate from other code.

Table 10.14.1: Typical two files per class.

ClassName.h	Contains the class definition, including data members and member function declarations.
ClassName.cpp	Contains member function definitions.

A file that uses the class, such as a main file or ClassName.cpp, must include ClassName.h. The .h file's contents are sufficient to allow compilation, as long as the corresponding .cpp file is

eventually compiled into the program too.

Figure 10.14.1: Using two separate files for a class.

File: StoreItem.h

```
#ifndef STOREITEM_H
#define STOREITEM_H

class StoreItem {
   public:
      void SetWeightOunces(int
ounces);
      void Print() const;
   private:
      int weightOunces;
};

#endif
```

File: StoreItem.cpp

```
#include <iostream>
using namespace std;

#include "StoreItem.h"

void StoreItem::SetWeightOunces(int ounces) {
   weightOunces = ounces;
   return;
}

void StoreItem::Print() const {
   cout << "Weight (ounces): " << weightOunces <<
endl;
   return;
}
```


File: main.cpp

```
#include <iostream>
using namespace std;

#include "StoreItem.h"

int main() {
   StoreItem item1;

   item1.SetWeightOunces(16);
   item1.Print();

   return 0;
}
```

Compilation example

```
% g++ -Wall StoreItem.cpp main.cpp
% a.out
Weight (ounces): 16
```

The figure shows how all the .cpp files might be listed when compiled into one program. Note that the .h file is *not* one of the listed files, as it is included in the appropriate .cpp files.

Sometimes multiple small related classes are grouped into a single file, to avoid a proliferation of files. But for typical classes, good practice is to create a unique .cpp and .h file for each class.

Earlier material used several classes that come standard with C++, by including those items' header files with lines like `#include <string>` for the string class.

For independent development and faster compilation, each class file is typically compiled individually into an object file, and then later linked with a main file. Such compilation is discussed in another section on modular compilation.

PARTICIPATION ACTIVITY 10.14.1: Separate files.

1) Commonly a class definition and

associated function definitions are contained entirely in their own .h file.

- True
- False

2) The .cpp file for a class should #include the associated .h file.

- True
- False

3) A drawback of the separate file approach is longer compilation times.

- True
- False

10.15 Classes with classes

Creating a new program may start by determining how to decompose the program into objects. The programmer considers what "things" or objects exist, and what each object contains and does. Sometimes this results in creating multiple classes where one class uses another class.

PARTICIPATION ACTIVITY 10.15.1: Creating a program as objects.

Animation captions:

1. Thinking of a program as objects that have other objects
2. Coaches and players are people with similar info, create "Person" class. Private: name, age. Public: getters/setters, print.
3. Create class for a Team, has Person members

Above, the programmer realized that a "Person" object would be useful to represent coaches and players. The programmer sketches a Person class. Each Person will have private (indicated by "-") data like name and age (other data omitted for brevity). Each Person will have public (indicated by "+") functions like get/set name, get/set age, print, and more.

Next, the programmer realized that a "Team" object would be useful. The programmer sketches a Team class with private and public items. Note that the Team class uses the Person class.

PARTICIPATION ACTIVITY 10.15.2: Class using a class.

1) There is exactly one way to decompose a program into objects.
   - True
   - False

2) The - in the above sketch indicates a class' private item.
   - True
   - False

3) The + in the above sketch indicates additional private items.
   - True
   - False

4) The Team class uses the Person class.
   - True
   - False

5) The Person class uses the Team class.
   - True
   - False

Figure 10.15.1: A class using a class: SoccerTeam has TeamPersons as data.

TeamPerson.h	TeamPerson.cpp

```
#ifndef TEAMPERSON_H
#define TEAMPERSON_H

#include <string>
using namespace std;

class TeamPerson {
   public:
      void   SetFullName(string
firstAndLastName);
      void   SetAgeYears(int ageInYears);
      string GetFullName() const;
      int    GetAgeYears() const;
      void   Print() const;

   private:
      string fullName;
      int    ageYears;
};

#endif
```

```
#include <iostream>
#include <string>
using namespace std;

#include "TeamPerson.h"

void TeamPerson::SetFullName(string
firstAndLastName) {
   fullName = firstAndLastName;
   return;
}

void TeamPerson::SetAgeYears(int ageInYears)
{
   ageYears = ageInYears;
   return;
}

string TeamPerson::GetFullName() const {
   return fullName;
}

int TeamPerson::GetAgeYears() const {
   return ageYears;
}

void TeamPerson::Print() const {
   cout << "Full name: "   << fullName <<
endl;
   cout << "Age (years): " << ageYears <<
endl;
}
```


SoccerTeam.h

```
#ifndef SOCCERTEAM_H
#define SOCCERTEAM_H

#include "TeamPerson.h"

class SoccerTeam {
   public:
      void SetHeadCoach(TeamPerson
teamPerson);
      void SetAssistantCoach (TeamPerson
teamPerson);

      TeamPerson GetHeadCoach() const;
      TeamPerson GetAssistantCoach() const;

      void Print() const;

   private:
      TeamPerson headCoach;
      TeamPerson assistantCoach;
      // Players omitted for brevity
};

#endif
```

SoccerTeam.cpp

```
#include <iostream>
using namespace std;

#include "TeamPerson.h"
#include "SoccerTeam.h"

void SoccerTeam::SetHeadCoach(TeamPerson
teamPerson) {
   headCoach = teamPerson;
   return;
}

void
SoccerTeam::SetAssistantCoach(TeamPerson
teamPerson) {
   assistantCoach = teamPerson;
   return;
}

TeamPerson SoccerTeam::GetHeadCoach() const
{
   return headCoach;
}

TeamPerson SoccerTeam::GetAssistantCoach()
const {
   return assistantCoach;
}

void SoccerTeam::Print() const {
   cout << "HEAD COACH: " << endl;
   headCoach.Print();
   cout << endl;

   cout << "ASSISTANT COACH: " << endl;
   assistantCoach.Print();
   cout << endl;
   return;
}
```


main.cpp

```
#include <iostream>
using namespace std;

#include "SoccerTeam.h"
#include "TeamPerson.h"

int main() {
   SoccerTeam teamCalifornia;
   TeamPerson headCoach;
   TeamPerson asstCoach;

   headCoach.SetFullName("Mark Miwerds");
   headCoach.SetAgeYears(42);
   teamCalifornia.SetHeadCoach(headCoach);

   asstCoach.SetFullName("Stanley Lee");
   asstCoach.SetAgeYears(30);

teamCalifornia.SetAssistantCoach(asstCoach);

   teamCalifornia.Print();

   return 0;
}
```

```
HEAD COACH:
Full name: Mark Miwerds
Age (years): 42

ASSISTANT COACH:
Full name: Stanley Lee
Age (years): 30
```

Note that each file only includes needed header files. SoccerTeam.h has a TeamPerson member so includes TeamPerson.h. SoccerTeam.cpp includes both SoccerTeam.h and TeamPerson.h. main.cpp declares objects of both types so also includes both .h files. A common error is to include unnecessary .h files, which misleads the reader. Note that only .h files are included, never .cpp files.

PARTICIPATION ACTIVITY 10.15.3: Team and Person classes and includes.

Indicate which .h files should be included in each file.

1) TeamPerson.h
   - TeamPerson.h
   - SoccerTeam.h
   - No .h file needed

2) TeamPerson.cpp
   - TeamPerson.h
   - SoccerTeam.h
   - No .h file needed

3) SoccerTeam.h
   - TeamPerson.h
   - SoccerTeam.h
   - No .h file needed

4) SoccerTeam.cpp
   - TeamPerson.h
   - SoccerTeam.h
   - TeamPerson.cpp
   - TeamPerson.h and SoccerTeam.h

©zyBooks 04/05/18 21:46 261830
Julian Chan
WEBERCS2250ValleSpring2018

5) main.cpp
   - main.h
   - TeamPerson.h
   - TeamPerson.h and SoccerTeam.h

TeamPerson.cpp

SoccerTeam.cpp

10.16 C++ example: Salary calculation with classe

PARTICIPATION ACTIVITY 10.16.1: Calculate salary: Using classes.

The program below uses a class, TaxTableTools, which has a tax table built in. The main function prompts for a salary, then uses a TaxTableTools function to get the tax rate. The program then calculates the tax to pay and displays the results to the user. Run the program with annual salaries of 10000, 50000, 50001, 100001 and -1 (to end the program) and note the output tax rate and tax to pay.

1. Modify the TaxTableTools class to use a setter function that accepts a new salary and tax rate table.
2. Modify the program to call the new function, and run the program again, noting the same output.

Note that the program's two classes are in separate tabs at the top.

Current file: **IncomeTaxMain.cpp**

```
#include <iostream>
#include <limits>
#include <vector>
#include <string>
#include "TaxTableTools.h"

int GetInteger(const string userPrompt) {
   int inputValue = 0;

   cout << userPrompt << ": " << endl;
   cin >> inputValue;

   return inputValue;
}

// ************************************************************************

int main() {
   const string PROMPT_SALARY = "\nEnter annual salary (-1 to exit)";
```


10000 50000 50001 100001 -1

Run

PARTICIPATION ACTIVITY 10.16.2: Salary calculation: Overloading a constructor.

The program below calculates a tax rate and tax to pay given an annual salary. The program uses a class, TaxTableTools, which has the tax table built in. Run the program with annual salaries of 10000, 50000, 50001, 100001 and -1 (to end the program) and note the output tax rate and tax to pay.

1. Overload the constructor.
   a. Add to the TaxTableTools class an overloaded constructor that accepts the base salary table and corresponding tax rate table as parameters.
   b. Modify the main function to call the overloaded constructor with the two tables (vectors) provided in the main function. Be sure to set the nEntries value, too.
   c. Note that the tables in the main function are the same as the tables in the TaxTableTools class. This sameness facilitates testing the program with the same annual salary values listed above.
   d. Test the program with the annual salary values listed above.
2. Modify the salary and tax tables
   a. Modify the salary and tax tables in the main function to use different salary ranges and tax rates.
   b. Use the just-created overloaded constructor to initialize the salary and tax tables.
   c. Test the program with the annual salary values listed above.

Current file: **IncomeTaxMain.cpp**


```
#include <iostream>
#include <limits>
#include <vector>
#include <string>
#include "TaxTableTools.h"
using namespace std;

int main() {
   const string PROMPT_SALARY = "\nEnter annual salary (-1 to exit)";
   int annualSalary = 0;
   double taxRate = 0.0;
   int taxToPay = 0;
   int i = 0;
```

```
14    vector<int> salaryBase(5);
15    vector<double> taxBase(5);
16
17    salaryBase.at(0) = 0;
18    salaryBase.at(1) = 20000;
```

```
10000
50000
50001
```


Run

10.17 C++ example: Domain name availability with

PARTICIPATION ACTIVITY 10.17.1: Domain name availability: Using classes.

The program below uses a class, DomainAvailabilityTools, which includes a table of registered domain names. The main function prompts for domain names until the user presses Enter at the prompt. The domain name is checked against a list of the registered domains in the DomainAvailabilityTools class. If the domain name is not available, the program displays similar domain names.

1. Run the program and observe the output for the given input.
2. Modify the DomainAvailabilityClass's function named GetSimilarDomainNames so that some unavailable domain names do not get a list of similar domain names. Run the program again and observe that unavailable domain names with TLDs of .org or .biz do not have similar names.

Current file: **DomainAvailabilityMain.cpp**

```
1 #include <iostream>
2 #include <string>
3 #include <cctype>
4 #include "DomainAvailabilityTools.h"
5 using namespace std;
6
7 // ***************************************************************************
8
```

```
// prompts user string. Returns string.
string GetString(string prompt) {
   string userInput = "";

   cout << prompt << endl;
   cin >> userInput;

   return userInput;
}
// ************************************************************************
```



```
programming.com
apple.com
oracle.com
```

Run

PARTICIPATION ACTIVITY 10.17.2: Domain validation: Using classes (solution).

A solution to the above problem follows.

Current file: **DomainAvailabilityMain.cpp**

```
#include <iostream>
#include <string>
#include <cctype>
#include "DomainAvailabilityTools.h"
using namespace std;

// ************************************************************************

// Prompts user for input string and returns the string
string GetString(string prompt) {
   string userInput = "";

   cout << prompt << endl;
   cin >> userInput;

   return userInput;
}
// ************************************************************************

```



```
programming.com
apple.com
oracle.com
```

Run

10.18 Ch 7 Warm up: Online shopping cart (C++)

(1) Create three files to submit:

- ItemToPurchase.h - Class declaration
- ItemToPurchase.cpp - Class definition
- main.cpp - main() function

Build the ItemToPurchase class with the following specifications:

- Default constructor
- Public class functions (mutators & accessors)
- SetName() & GetName() (2 pts)
- SetPrice() & GetPrice() (2 pts)
- SetQuantity() & GetQuantity() (2 pts)
- Private data members
- string itemName - Initialized in default constructor to "none"
- int itemPrice - Initialized in default constructor to 0
- int itemQuantity - Initialized in default constructor to 0

(2) In main(), prompt the user for two items and create two objects of the ItemToPurchase class. Before prompting for the second item, call **cin.ignore()** to allow the user to input a new string. (2 pts)

Ex:

```
Item 1
Enter the item name:
Chocolate Chips
Enter the item price:
3
Enter the item quantity:
1

Item 2
Enter the item name:
Bottled Water
```

```
Enter the item price:
1
Enter the item quantity:
10
```

(3) Add the costs of the two items together and output the total cost. (2 pts)

Ex:

```
TOTAL COST
Chocolate Chips 1 @ $3 = $3
Bottled Water 10 @ $1 = $10

Total: $13
```

LAB ACTIVITY 10.18.1: Ch 7 Warm up: Online shopping cart (C++) 0 / 10

Submission Instructions

Deliverables

`main.cpp` , `ItemToPurchase.cpp` and `ItemToPurchase.h` *You must submit these f*

Compile command

`g++ main.cpp ItemToPurchase.cpp -Wall -o a.out` *We will use this command to compi*

Submit your files below by dragging and dropping into the area or choosing a file on your hard drive

main.cpp	Drag file here or **Choose on hard drive.**	**ItemTo...e.cpp**	Drag file here or **Choose on hard drive.**	**ItemTo...**

Submit for grading

Latest submission

No submissions yet

10.19 Ch 7 Program: Online shopping cart (continu

This program extends the earlier "Online shopping cart" program. (Consider first saving your earlier program).

(1) Extend the ItemToPurchase class per the following specifications:

- Parameterized constructor to assign item name, item description, item price, and item quantity (default values of 0). (1 pt)
- Public member functions
- SetDescription() mutator & GetDescription() accessor (2 pts)
- PrintItemCost() - Outputs the item name followed by the quantity, price, and subtotal
- PrintItemDescription() - Outputs the item name and description
- Private data members
- string itemDescription - Initialized in default constructor to "none"

Ex. of PrintItemCost() output:

```
Bottled Water 10 @ $1 = $10
```

Ex. of PrintItemDescription() output:

```
Bottled Water: Deer Park, 12 oz.
```

(2) Create three new files:

- ShoppingCart.h - Class declaration
- ShoppingCart.cpp - Class definition
- main.cpp - main() function (Note: main()'s functionality differs from the warm up)

Build the ShoppingCart class with the following specifications. Note: Some can be function stubs (empty functions) initially, to be completed in later steps.

- Default constructor
- Parameterized constructor which takes the customer name and date as parameters (1 pt)
- Private data members
- string customerName - Initialized in default constructor to "none"
- string currentDate - Initialized in default constructor to "January 1, 2016"
- vector < ItemToPurchase > cartItems

- Public member functions
- GetCustomerName() accessor (1 pt)
- GetDate() accessor (1 pt)
- AddItem()
  - Adds an item to cartItems vector. Has parameter ItemToPurchase. Does not return anything.
- RemoveItem()
  - Removes item from cartItems vector. Has a string (an item's name) parameter. Does not return anything.
  - If item name cannot be found, output this message: `Item not found in cart. Nothing removed.`
- ModifyItem()
  - Modifies an item's description, price, and/or quantity. Has parameter ItemToPurchase. Does not return anything.
  - If item can be found (by name) in cart, check if parameter has default values for description, price, and quantity. If not, modify item in cart.
  - If item cannot be found (by name) in cart, output this message: `Item not found in cart. Nothing modified.`
- GetNumItemsInCart() (2 pts)
  - Returns quantity of all items in cart. Has no parameters.
- GetCostOfCart() (2 pts)
  - Determines and returns the total cost of items in cart. Has no parameters.
- PrintTotal()
  - Outputs total of objects in cart.
  - If cart is empty, output this message: `SHOPPING CART IS EMPTY`
- PrintDescriptions()
  - Outputs each item's description.

Ex. of PrintTotal() output:

```
John Doe's Shopping Cart - February 1, 2016
Number of Items: 8

Nike Romaleos 2 @ $189 = $378
Chocolate Chips 5 @ $3 = $15
Powerbeats 2 Headphones 1 @ $128 = $128

Total: $521
```


Ex. of PrintDescriptions() output:

```
John Doe's Shopping Cart - February 1, 2016
```

```
Item Descriptions
Nike Romaleos: Volt color, Weightlifting shoes
Chocolate Chips: Semi-sweet
Powerbeats 2 Headphones: Bluetooth headphones
```

(3) In main(), prompt the user for a customer's name and today's date. Output the name and date. Create an object of type ShoppingCart. (1 pt)

Ex.

```
Enter customer's name:
John Doe
Enter today's date:
February 1, 2016

Customer name: John Doe
Today's date: February 1, 2016
```

(4) Implement the PrintMenu() function. PrintMenu() has a ShoppingCart parameter, and outputs a menu of options to manipulate the shopping cart. Each option is represented by a single character. Build and output the menu within the function.

If the an invalid character is entered, continue to prompt for a valid choice. *Hint: Implement Quit before implementing other options.* Call PrintMenu() in the main() function. Continue to execute the menu until the user enters q to Quit. (3 pts)

Ex:

```
MENU
a - Add item to cart
d - Remove item from cart
c - Change item quantity
i - Output items' descriptions
o - Output shopping cart
q - Quit

Choose an option:
```


(5) Implement Output shopping cart menu option. (3 pts)

Ex:

```
OUTPUT SHOPPING CART
John Doe's Shopping Cart - February 1, 2016
Number of Items: 8

Nike Romaleos 2 @ $189 = $378
Chocolate Chips 5 @ $3 = $15
Powerbeats 2 Headphones 1 @ $128 = $128

Total: $521
```


(6) Implement Output item's description menu option. (2 pts)

Ex.

```
OUTPUT ITEMS' DESCRIPTIONS
John Doe's Shopping Cart - February 1, 2016

Item Descriptions
Nike Romaleos: Volt color, Weightlifting shoes
Chocolate Chips: Semi-sweet
Powerbeats 2 Headphones: Bluetooth headphones
```

(7) Implement Add item to cart menu option. (3 pts)

Ex:

```
ADD ITEM TO CART
Enter the item name:
Nike Romaleos
Enter the item description:
Volt color, Weightlifting shoes
Enter the item price:
189
Enter the item quantity:
2
```


(8) Implement remove item menu option. (4 pts)

Ex:

```
REMOVE ITEM FROM CART
Enter name of item to remove:
Chocolate Chips
```

(9) Implement Change item quantity menu option. *Hint: Make new ItemToPurchase object and use ItemToPurchase modifiers before using ModifyItem() function.* (5 pts)

Ex:

```
CHANGE ITEM QUANTITY
Enter the item name:
Nike Romaleos
Enter the new quantity:
3
```

LAB ACTIVITY 10.19.1: Ch 7 Program: Online shopping cart (continued) (C++) 0 / 31

Submission Instructions

Deliverables

`ItemToPurchase.cpp` , `ShoppingCart.h` , `ItemToPurchase.h` , `main.cpp` a

`ShoppingCart.cpp`

Compile command

```
g++ ItemToPurchase.cpp main.cpp ShoppingCart.cpp -Wall -o a.out
```

We will u

Submit your files below by dragging and dropping into the area or choosing a file on your hard drive

ItemTo...e.cpp	Drag file here or Choose on hard drive.	**Shoppi...art.h**	Drag file here or Choose on hard drive.	**ItemTo...**	
main.cpp	Drag file here or Choose on hard drive.	**Shoppi...t.cpp**	Drag file here or Choose on hard drive.		

Submit for grading

Latest submission

No submissions yet

©zyBooks 04/05/18 21:46 261830
Julian Chan
WEBERCS2250ValleSpring2018