## Chapter 10: Defining Classes

Instructor: Mark Edmonds

edmonds_mark@smc.edu

### Object-oriented Programming

- Up to this point, most of what we've learned also applies to C
- Object-oriented programming concept: group data and operations together logically and allow hierarchies of relationships
- Operations (methods): steer, accelerate, decelerate, fill up with gas, etc
- Data (attributes): speed, heading, gas tank status, odometer, etc.
- You don't have to know how the car precisely works to use one. You simply know that the pedal accelerates the car, brake decelerates, etc
- This is known as *information hiding*
- Another engineer has designed how the care operates, you just use the public (available) operations
  - But there are many more operations happening inside the car than you are aware
- We refer to the design of the car as the *class* (think of this like a blueprint)
- Once you have the design for a car, you can reuse it
  - Each "usage" is an *instantiation* of a car
- Similarly, once you have a design of a car, you can extend it to add new features
  - E.g. add upgrades, wings, etc
  - This is known as inheritance
- From the object-oriented programming view, programs are viewed as a collection of collaborating objects
  - This models the real world
  - Program structure is implemented via *classes* and *objects*

### Objects

- Consider the car as an object. It may have the following structure:

| Properties | Functionality |
|---|---|
| Make: Honda | play_music |
| Model: Prelude | toggle_left_blinker |
| Gas tank: Full | honk |

- An object has:
  1. State described via *attributes* (also known as *properties*)
  2. Behavior described via *methods*
  3. Identity described via *instances*
- Objects are instantiations of a specific blueprint (called classes in C++)
  - Each instantiation may have different values for their properties, even though they came from the same blueprint
  - For instance, different instances of a car may have different levels of gas in their tanks

## Classes

- Classes are like blueprints for objects
- We instantiate an *object* using the class's blueprint
  - Instances must be *declared* before they can be used, just like every other variable.
- Let's look at an example. Suppose we have the following `BankAccount` class design (this isn't C++ code yet):

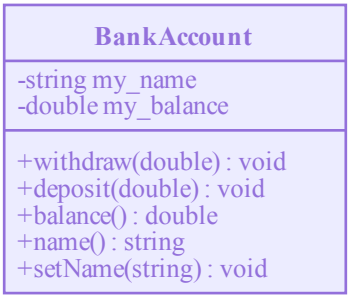


**Figure 1:** Box diagram for BankAccount class

- This diagram is called a **class block diagram**
  - The top panel indicates the class name, the middle pane indicates the member variables, and the bottom pane indicates the member methods
  - A - indicates a private member and a + indicates a public member (more on this in a bit)
  - For member methods (functions) they have the following form: `methodName(arg_types): return_type`. This is a bit different than how we write functions in C++, but this diagram format is the industry-standard in computer science and software engineering
- We can instantiate this class the way we do every other variable:

```
1 BankAccount my_bank;
```

- Since the `balance()` method returns a double, we can do the same thing that we've been doing with every other function:

```
BankAccount my_bank;
double balance = my_bank.balance();
```

- Notice that we use the `.` operation to access the `my_bank` instance's `balance()` function.

### Bank example

- The following example shows how to create and use a simple bank account class

#### Example Banker.cpp

```
// This program demonstrates how to make use of existing objects.
// This program uses a bankAccount class with the interface described
// in class.

#include <iostream>                // for std::cout
#include <string>                 // for string class
#include "BankAccount.h"              // for bankAccount class
using namespace std;              // supports cout

int main( )
{
  string name;
  double openingbalance;

  cout.setf( ios::fixed );
  cout.setf( ios::showpoint );
  cout.precision( 2 );

  cout << endl << endl << "\t\tWelcome to the Bank of SMC!" << endl;
  cout << "Please enter your name and opening bank balance: ";
  cin  >> name >> openingbalance;

  BankAccount account;
  account.setName( name );
  account.deposit( openingbalance );

  cout << "Thank you for banking with the Bank of SMC!" << endl;
  cout << account.name() << " has a balance of $" << account.balance();
```

```
29    cout << endl << endl;
30
31    return 0;
32  }
```

**Example BankAccount.h**

```
1  //-----------------------------------------------------------------------
2  // INTERFACE FILE: BankAccount.h
3  //
4  // Defines class BankAccount
5  //
6  //-----------------------------------------------------------------------
7
8  #ifndef BANKACCOUNT_H   // Avoid redeclaring class BankAccount.
9  #define BANKACCOUNT_H   // This code is compiled only once
10 #include <string>    // for class string
11 using namespace std;
12
13 //BankAccount class definition
14 class BankAccount {
15 public:  // class member functions
16
17     //--constructors
18     BankAccount();
19
20     // constructs a bank account with initName as the name and
          initBalance as the balance
21     BankAccount(string initName, double initBalance);
22
23     //--modifiers
24
25     void deposit(double depositAmount);
26
27     void withdraw(double withdrawalAmount);
28
29     //--accessors
30
31     double balance() const;
32
33     string name() const;
34
```

```
35      void setName( string initName );
36
37  private:
38      string my_name;     // Uniquely identify an object
39      double my_balance; // Store the current balance (non-persistent)
40  };  // NOTICE THE SEMICOLON AFTER THE CLOSING CURLY BRACE
41
42  #endif   // ifndef BANKACCOUNT_H
```

**Example BankAccount.cpp**

```
1   //-------------------------------------------------------------------------
2   // IMPLEMENTATION FILE: BankAccount.cpp
3   //
4   // Implements 1. class BankAccount member functions
5   //
6   //-------------------------------------------------------------------------
7   #include "BankAccount.h"  // allows for separate compilation if you
        want
8   #include <iostream>     // for ostream << and istream >>
9   #include <string>       // for class string
10  using namespace std;
11
12  //--constructors
13
14  BankAccount::BankAccount()
15  {
16    my_name = "?name?";
17    my_balance = 0.0;
18  }
19
20  BankAccount::BankAccount(string initName, double initBalance)
21  {
22    my_name = initName;
23    my_balance = initBalance;
24  }
25
26  //--modifiers
27
28  void BankAccount::deposit(double depositAmount)
29  {
30    my_balance = my_balance + depositAmount;
```

```
31 }
32
33 void BankAccount::withdraw(double withdrawalAmount)
34 {
35   my_balance = my_balance - withdrawalAmount;
36 }
37
38 //--accessors
39
40 double BankAccount::balance() const
41 {
42   return my_balance;
43 }
44
45 string BankAccount::name() const
46 {
47   return my_name;
48 }
49
50 void BankAccount::setName( string initName )
51 {
52   my_name = initName;
53 }
```

- One important thing to note: there's a semicolon placed after you declare a class (i.e. after the closing curly brace)
- Another important thing to note: in the implementation, we had to write `BankAccount::` before all of our method definitions.
  - This is to tell the compiler that this function is part of the `BankAccount` class, and gives the function *scope* inside the class. This allows it to access member variables and use other class methods.
- Also note that our header file `BankAccount.h` and our implementation file `BankAccount.cpp` have the same name except for the file extension. Follow this convention!
- We can instantiate multiple bank accounts, each with their own balance:

```
1 BankAccount bank1("bank1", 1000);
2 BankAccount bank2("bank2", 345);
3 double bank1_balance = bank1.balance();
4 double bank2_balance = bank2.balance();
```

- `bank1_balance` will be 1000 and `bank2_balance` will be 345.

- This shows the power of classes - we have one blueprint, `BankAccount` and can instantiate multiple instances of that class.
  - Each class has it's own copy of member variables (e.g. `bank1` and `bank2` have different values for the member variable `my_balance`)

### `private` vs. `public`

- In our `BankAccount.h` above, you may have noticed that we had one section under **`public`**: and another section under **`private`**:
  - This indicates whether or not this member variable or method is available for internal use (**`private`**) or internal or external use (**`public`**).
- **`public`** members can be accessed by the class itself or by anyone using the class. That's why we were able to call `bank1.balance()`; `balance()` is listed as a public method
- **`private`** members can only be access by the class itself. That means if we tried to directly access the member variable `my_balance`, we'd get an error. The function `balance()` is allowed to access `my_balance` because it is a method inside the class.
  - If you don't believe me, try to access `account.my_balance` directly in `Banker.cpp`. Compile and you'll get an error.
- This functionality is important because it allows the class designer to specify what a user can access in a class
  - For instance, if you were designing a class to model a car, you probably don't want to allow an ordinary user direct access to the transmission or internal engine components
- Some additional notes regarding **`public`** vs. **`private`**:
  - You can list them multiple times in one class header file
  - If you don't specify either, classes will default to **`private`**
  - Usually advisable to only have one block of **`public`** and one block of **`public`** per header
  - Every class method has access to **`private`** members, regardless of whether or not they are **`public`** or **`private`** (e.g. a **`public`** method can call a **`private`** method and access **`private`** member variables)

### When should you use `private`?

- Class member variables should be declared **`private`** as a general rule
- Class helper functions should also be declared **`private`**
  - For instance, suppose you were building a class for an automatic car
  - You wouldn't want to make the `shiftGears()` function be **`public`**, as in an automatic car, the user should not be able to call `shiftGears()` directly, but the car still needs to shift gears, so this should be a **`private`** method.

- The only things that should be declared **`public`** are functions you want the end-user to be do

#### Setters (mutators) and getters (accessors)

- Because we typically make class member variables **`private`**, we may need a way to allow the user to read or modify a member variable.
- A "getter" (also called an accessor) method will *get* the value of a member variable.
  - For instance, `balance()` and `name()` in the `BankAccount` class are both getter methods.
- A "setter" (also called a mutator) method will modify the value of a member variable.
  - For instance, `setName(string)` in the `BankAccount` class is a setter method.
- Why use setters and getters?
  - The main motivation for using getters is to hide the member address of a specific variable. This is for security.
    * Notice `balance()` returns a copy of `my_balance`. This prevents the user from knowing the exact memory address of `my_balance`
  - The main motivation for using setters is to allow the user to modify the variable *without* giving direct access to the member variable.
    * Setters can also implement additional functionality to check where or not the new value is valid. For instance, your data may have some constraints, like a telephone number that is expected to follow a certain form. By using a setter function, you can check and verify the input *before* assigning it to a member variable.

### Why use classes?

- Consider the `BankAccount` example again.
  - A user doesn't need to know exactly how the `BankAccount` class works internally; a user only needs to know how to use the class.
- This is called *information hiding*, and it makes complex things much more simple
  - A good example of this is the `toggle_left_blinker()` method from the car object
    * You don't need to know anything about electronics to successfully toggle the blinker. That's wonderful for all users of the class!
- Using **`private`** and **`public`** appropriately helps enforce information hiding.

### Why separate classes into header and implementation files?

- For the same reasons as before, but just to refresh our memory…
- The implementation file may be compiled in a *library* and distributed to others. For instance, the C++ Standard Library does this. When you include standard library functionality, like with

<iostream> the implementation file has never been on your computer. Instead, when you install a C++ compiler, it installs a precompiled version of the library on your computer.
  - However, the header file *does* exist somewhere on your machine. This is what enables you share your code with others *without* letting them see exactly how you implemented every function.
- When we give our class to others, we may want to hide exactly how we implemented each function for security or business reasons, but want to let others use the Application Programming Interface (API) we provide in header files. This is a great way to share your code with others without requiring them to compile every line of code you wrote

### Student example

- A good example of how challenging it is to write a well designed class
- But also an example of how easy it is to use a well-defined track

#### Example StudentDriver.cpp

```
1  // StudentDriver.cpp : Defines the entry point for the console
       application.
2  //
3
4  #include <iostream>
5  #include <cstdlib>
6
7  #include "Student.h"
8
9  using namespace std;
10
11 void dumpStudent( Student s );
12
13 int main(int argc, char* argv[])
14 {
15     Student s = Student();
16
17     cout << "====initial====" << endl;
18     dumpStudent( s );
19
20     s.addGrade( 4.0, 4.0 );   // add an A
21     cout << "====A====" << endl;
22     dumpStudent( s );
23
```

```
24      s.addGrade( 4.0, 3.0 );   // add a B
25      cout << "====A & B====" << endl;
26      dumpStudent( s );
27
28      s.addGrade( 4.0, 2.0 );   // add a C
29      cout << "====A & B & C====" << endl;
30      dumpStudent( s );
31
32      s.incYear();
33      cout << "====Sophomore====" << endl;
34      dumpStudent( s );
35
36      return( 0 );
37  }
38
39  void dumpStudent( Student s ) {
40      if (s.isFreshman()) {
41          cout << "Freshman -" ;
42      }
43      else if (s.isSophomore()) {
44          cout << "Sophomore - ";
45      }
46      else if (s.isJunior()) {
47          cout << "Junior - ";
48      }
49      else {
50          cout << "Senior - ";
51      }
52      cout << "units=" << s.getUnits();
53      cout << " gpa="  << s.getGPA();
54      cout << endl;
55  }
```

**Example Student.h**

```
1  #ifndef STUDENT_H
2  #define STUDENT_H
3  #include <iostream>
4  #include <cstdlib>
5
6  using namespace std;
7
```

```
8  class Student {
9  public:
10     Student();
11
12     double getGPA();
13     int    getYear();
14     void   incYear();
15     double getUnits();
16
17     bool   isFreshman();
18     bool   isSophomore();
19     bool   isJunior();
20     bool   isSenior();
21
22     void   addGrade( double unit, double gpa );
23 private:
24     double totalGPAValue;   // running total of GPA points earned
25     double unitValue;       // running total of units earned
26     int    yearValue;
27 };
28
29 #endif
```

**Example Student.cpp**

```
1  #include <iostream>
2  #include <cstdlib>
3
4  #include "Student.h"
5
6  using namespace std;
7
8
9  Student::Student() {
10     totalGPAValue  = 0.0;
11     unitValue = 0.0;
12     yearValue = 1;
13 }
14
15 // calculate the gpa from the running totals earned
16 double Student::getGPA() {
17     double gpa = 0.0;
```

```
18      if (totalGPAValue != 0.0) {
19          gpa = totalGPAValue / unitValue;
20      }
21      return( gpa );
22  }
23
24  int Student::getYear() {
25      return( yearValue );
26  }
27
28  void Student::incYear() {
29      ++yearValue;
30  }
31
32  double Student::getUnits() {
33      return( unitValue );
34  }
35
36  bool Student::isFreshman() {
37      return( yearValue == 1 );
38  }
39
40  bool Student::isSophomore() {
41      return( yearValue == 2 );
42  }
43
44  bool Student::isJunior() {
45      return( yearValue == 3 );
46  }
47
48  bool Student::isSenior() {
49      return( yearValue >= 4 );
50  }
51
52  // update the running totals earned
53  void Student::addGrade( double unit, double gpa ) {
54      unitValue += unit;
55      totalGPAValue  += gpa * unit;
56  }
```

### Class Constructors

- Constructors are called when instances are declared or initialized.
- They define how member variables should be initialized
- Constructors are special methods, but they can call any other method, or execute any other C++ code you need to execute when the class instance is being "constructed".
- Constructors cannot be called directly. They are only called when an instance is declared or initialized.
- Each class can define one constructor or multiple constructors. Each constructor must have a unique signature, meaning their parameter type and lengths must be unique.
  - For an example, the `BankAccount` class, there are two constructors. One, `BankAccount()` does not take any arguments, but `BankAccount(string initName, double initBalance)` takes two arguments
  - When we define multiple
  - These different constructors will be called automatically depending upon what arguments are passed. For instance:

```
BankAccount my_account1; // this will call the constructor with no parameters (BankAccount())
BankAccount my_account2("mark", 1000); // this will call the constructor with parameters (BankAccount(string initName, double initBalance))
cout << my_account1.name() << endl; // this will print the value ?name?
cout << my_account2.name() << endl; // this will print mark
```

### Radio example

- Another example

#### Example RadioDriver.cpp

```
#include <iostream>
#include <cstdlib>

#include "Radio.h"

using namespace std;

Radio::Radio() {
    tuneToFM();
```

```
10      setStation( 88.5 );
11      volume = 5;
12  }
13
14  Radio::Radio( bool   tuneToAM,
15                double station,
16                int    volumeValue ) {
17      tunedOnAM = tuneToAM,
18      setStation( station );
19      volume = volumeValue;
20  }
21
22  void Radio::tuneToAM() {
23      if (!tunedOnAM) {
24          tunedOnAM = true;
25          setStation( 590.0 );
26      }
27  }
28
29  void Radio::tuneToFM() {
30      if (tunedOnAM) {
31          tunedOnAM = false;
32          setStation( 88.5 );
33      }
34  }
35  bool Radio::isOnAM() {
36      return( tunedOnAM );
37  }
38
39  void Radio::setStation( double station ) {
40      stationValue = station;
41  }
42
43  double Radio::getStation() {
44      return( stationValue );
45  }
46
47  void Radio::incVolume() {
48      volume++;
49  }
50
51  void Radio::decVolume() {
52      volume--;
```

```
53 }
54
55 int Radio::getVolume() {
56     return( volume );
57 }
```

**Example Radio.h**

```
1  #ifndef RADIO_H
2  #define RADIO_H
3  #include <iostream>
4  #include <cstdlib>
5
6  using namespace std;
7
8  class Radio {
9  public:
10     Radio();
11     Radio( bool   tuneToAM,
12            double station,
13            int    volumeValue );
14
15     void   tuneToAM();
16     void   tuneToFM();
17     bool   isOnAM();
18     void   setStation( double station );
19     double getStation();
20     void   incVolume();
21     void   decVolume();
22     int    getVolume();
23 private:
24     int    volume;
25     double stationValue;
26     bool   tunedOnAM;
27 };
28
29 #endif
```

**Example Radio.cpp**

```
#include <iostream>
#include <cstdlib>

#include "Radio.h"

using namespace std;

Radio::Radio() {
    tuneToFM();
    setStation( 88.5 );
    volume = 5;
}

Radio::Radio( bool   tuneToAM,
              double station,
              int    volumeValue ) {
    tunedOnAM = tuneToAM,
    setStation( station );
    volume = volumeValue;
}

void Radio::tuneToAM() {
    if (!tunedOnAM) {
        tunedOnAM = true;
        setStation( 590.0 );
    }
}

void Radio::tuneToFM() {
    if (tunedOnAM) {
        tunedOnAM = false;
        setStation( 88.5 );
    }
}
bool Radio::isOnAM() {
    return( tunedOnAM );
}

void Radio::setStation( double station ) {
    stationValue = station;
}

double Radio::getStation() {
```

```
44      return( stationValue );
45  }
46
47  void Radio::incVolume() {
48      volume++;
49  }
50
51  void Radio::decVolume() {
52      volume--;
53  }
54
55  int Radio::getVolume() {
56      return( volume );
57  }
```

### I/O Streams

- Let's refer back to the Chapter 6: I/O Streams notes
- One thing you may not have noticed was that `ifstream` and `ofstream` are actually classes
  - For instance, we wrote:

```
1  ifstream in_stream;
2  in_stream.open("infile.dat");
```

- We've actually been using classes all along…
  - `cout` and `cin` are special instances of stream objects.
    * >> and << are operators in C++, similar to + or −…but more on that later

### Structures

- Structures are extremely similar to classes
- In C, structures can only hold data (i.e. you cannot add methods to a structure, only member variables)
- In C++, you can add member variables to a structure, but most people use a class in this case.
  - There is another more subtle difference between classes and structures: structures declare everything as **`public`** by default and classes declare everything as **`private`** by default.
- Let's declare a structure that defines a CD account at our bank:

```
1  struct CDAccount
2  {
```

```
3    double balance;
4    double interestRate;
5    int term;  //months to maturity
6  };  // notice we also end the declaration of the struct with a
       semicolon
```

- The keyword `struct` indicates we are declaring a struct and `CDAccount` is the structure's type
- We can use this similar to how we've used classes up to this point:

```
1  CDAccount my_account;
2  my_account.balance = 10;
```

- As a general rule, use structures when you DON'T need to encapsulate functionality (methods) with data (member variables), and use classes when you DO need to encapsulate functionality (methods) with data (member variables).

### Enumerations

- Mappings between labels and integers
- Enumerations are not composed of any data types only labels.
- Example enum to represent colors

```
1  enum color
2  {
3      red,
4      orange,
5      yellow,
6      green,
7      cyan,
8      blue,
9      purple,
10 };
```

- Under the hood, red will be assign the value `0`, orange `1`.
- How do you use them?

```
1  enum color value = red;
2  if(value == green){
3    cout << "We should never execute this statement\n";
4  }
```

- We can also use them in switch statements conveniently:

```cpp
#include <iostream>
#include <string>

enum color {
  red,
  orange,
  yellow,
  green,
  cyan,
  blue,
  purple,
};

int main(){
  enum color value = red;
  switch(value){
    case red:
      cout << "color is red\n";
      break;
    case orange:
      cout << "color is orange\n";
      break;
    case yellow:
      cout << "color is yellow\n";
      break;
    case green:
      cout << "color is green\n";
      break;
    case cyan:
      cout << "color is cyan\n";
      break;
    case blue:
      cout << "color is blue\n";
      break;
    case purple:
      cout << "color is purple\n";
      break;
  }
}
```

- Example usage: suppose we wanted to compare against the day of the week. We could use string comparisons for everything, but this is clunky, hard to read and string comparison is

computationally expensive.
  - Because enumerations are really just labeled integers, we can make code more readable by using them!

### Student example with enumerations

- Here is an example of our Student class using enumerations for the year and grades

#### Example EnumerationDriver.cpp

```
1  // EnumerationDriver.cpp : Defines the entry point for the console
       application.
2  //
3
4  #include <iostream>
5  #include <cstdlib>
6
7  #include "Student.h"
8
9  using namespace std;
10
11 void dumpStudent( Student s );
12
13 int main(int argc, char* argv[])
14 {
15     Student s = Student();
16
17     cout << "====initial====" << endl;
18     dumpStudent( s );
19
20     s.addGrade( 4.0, 4.0 );   // add an A
21     cout << "====A====" << endl;
22     dumpStudent( s );
23
24     s.addGrade( 4.0, 3.0 );   // add a B
25     cout << "====A & B====" << endl;
26     dumpStudent( s );
27
28     s.addGrade( 4.0, 2.0 );   // add a C
29     cout << "====A & B & C====" << endl;
30     dumpStudent( s );
31
```

```
32      s.incYear();
33      cout << "====Sophomore====" << endl;
34      dumpStudent( s );
35
36      return( 0 );
37  }
38
39  void dumpStudent( Student s ) {
40      switch( s.getYear() ) {
41      case Student::FRESHMAN:
42              cout << "Freshman -" ;
43              break;
44      case Student::SOPHOMORE:
45              cout << "Sophomore -";
46              break;
47      case Student::JUNIOR:
48              cout << "Junior -";
49              break;
50      case Student::SENIOR:
51              cout << "Senior -";
52              break;
53      }
54      cout << " units=" << s.getUnits();
55      cout << " gpa="  << s.getGPA();
56      cout << " letterGrade=";
57
58      if (s.testGPA( Student::A_AVG )) {
59          cout << "A";
60      }
61      else if (s.testGPA( Student::B_AVG )) {
62          cout << "B";
63      }
64      else if (s.testGPA( Student::C_AVG )) {
65          cout << "C";
66      }
67      else if (s.testGPA( Student::D_AVG )) {
68          cout << "D";
69      }
70      else if (s.testGPA( Student::F_AVG )) {
71          cout << "F";
72      }
73      else if (s.testGPA( Student::NO_AVG )) {
74          cout << "No Average Earned Yet";
```

```
75      }
76
77
78      cout << endl;
79  }
```

**Example Student.h**

```
1   #ifndef STUDENT_H
2   #define STUDENT_H
3   #include <iostream>
4   #include <cstdlib>
5
6   using namespace std;
7
8   class Student {
9   public:
10      Student();
11
12      enum Year { FRESHMAN = 100, SOPHOMORE = 200,
13                  JUNIOR   = 300, SENIOR    = 400 };
14
15      enum Average { A_AVG = 10, B_AVG = 20, C_AVG = 30,
16                     D_AVG = 40, F_AVG = 50, NO_AVG= 60 };
17
18      double getGPA();
19      int    getYear();
20      void   incYear();
21      double getUnits();
22
23      bool   testYear( Year year );
24      bool   testGPA( Average avg );
25
26      void   addGrade( double unit, double gpa );
27  private:
28      double totalGPAValue;   // running total of GPA points earned
29      double unitValue;       // running total of units earned
30      int    yearValue;
31  };
32
33  #endif
```

**Example Student.cpp**

```cpp
#include <iostream>
#include <cstdlib>

#include "Student.h"

using namespace std;


Student::Student() {
    totalGPAValue  = 0.0;
    unitValue = 0.0;
    yearValue = Student::FRESHMAN;
}

// calculate the gpa from the running totals earned
double Student::getGPA() {
    double gpa = 0.0;
    if (totalGPAValue != 0.0) {
        gpa = totalGPAValue / unitValue;
    }
    return( gpa );
}

int Student::getYear() {
    return( yearValue );
}

void Student::incYear() {
    switch( yearValue ) {
    case Student::FRESHMAN:
        yearValue = Student::SOPHOMORE;
        break;
    case Student::SOPHOMORE:
        yearValue = Student::JUNIOR;
        break;
    case Student::JUNIOR:
        yearValue = Student::SENIOR;
        break;
    }
}

```

```
42 double Student::getUnits() {
43     return( unitValue );
44 }
45 
46 bool Student::testYear( Year y ) {
47     return( yearValue == y );
48 }
49 
50 bool Student::testGPA( Average avg ) {
51     bool result = false;
52     double gpa = getGPA();
53     switch( avg ) {
54     case Student::A_AVG:
55         result = (gpa >= 3.50);
56         break;
57     case Student::B_AVG:
58         result = (gpa >= 2.50);
59         break;
60     case Student::C_AVG:
61         result = (gpa >= 1.50);
62         break;
63     case Student::D_AVG:
64         result = (gpa >= 0.50);
65         break;
66     case Student::F_AVG:
67         result = (unitValue != 0.0);
68         break;
69     case Student::NO_AVG:
70         result = (unitValue == 0.0);
71         break;
72     }
73     return( result );
74 }
75 
76 // update the running totals earned
77 void Student::addGrade( double unit, double gpa ) {
78     unitValue += unit;
79     totalGPAValue  += gpa * unit;
80 }
```

### Default values

- Any function can be supplied with default values
- This allows the user to omit certain arguments and use the defaults instead
- All default arguments must be grouped at the end of a function declaration
- Here are some examples using default values; notice that once we start using a default value, all arguments proceeding the first use a default value also must use a default value

```
1 void func(double d=0.0);
2 void calc(int arg1,
3           int arg2,
4           int arg3=0,
5           int arg4=1);
```

- The following are all valid invocations of these functions:

```
1 func();
2 func(12.5);
3 calc(12, 1);
```

### Radio example with default values

- Next, let's look at our Radio class, but rewrite it to use some default values
- Notice how we can mix default arugments with normal parameters

#### Example DefaultDriver.cpp

```
1  // DefaultDriver.cpp : Defines the entry point for the console
       application.
2  // test test
3
4  #include <iostream>
5  #include <cstdlib>
6
7  #include "Radio.h"
8
9  using namespace std;
10
11 void dumpRadio( Radio r );
12
13 int main(int argc, char* argv[])
```

```
14 {
15     Radio myRadio = Radio();
16     Radio yourRadio = Radio( true, 90.9, 10 );
17 
18     cout << "===myRadio===" << endl;
19     dumpRadio( myRadio );
20     cout << "===yourRadio===" << endl;
21     dumpRadio( yourRadio );
22 
23     return( 0 );
24 }
25 
26 void dumpRadio( Radio r ) {
27     if (r.isOnAM()) {
28         cout << "AM:";
29     }
30     else {
31         cout << "FM:";
32     }
33     cout << r.getStation();
34     cout << "\tvolume=" << r.getVolume();
35     cout << endl;
36 }
```

**Example Radio.h**

```
1  #ifndef RADIO_H
2  #define RADIO_H
3  #include <iostream>
4  #include <cstdlib>
5  
6  using namespace std;
7  
8  class Radio {
9  public:
10     // note use of default parameter values
11     Radio( bool   tuneToAM = false,
12            double station = 88.5,
13            int    volumeValue = 5);
14 
15     void   tuneToAM();
16     void   tuneToFM();
```

```
17      bool   isOnAM();
18      void   setStation( double station );
19      double getStation();
20      void   incVolume();
21      void   decVolume();
22      int    getVolume();
23  private:
24      int    volume;
25      double stationValue;
26      bool   tunedOnAM;
27  };
28
29  #endif
```

**Example Radio.cpp**

```
1   #include <iostream>
2   #include <cstdlib>
3
4   #include "Radio.h"
5
6   using namespace std;
7
8   // note that default parameter values are not
9   // specified in the implementation of the function call
10  Radio::Radio( bool   tuneToAM,
11                double station,
12                int    volumeValue ) {
13      tunedOnAM = tuneToAM,
14      setStation( station );
15      volume = volumeValue;
16  }
17
18  void Radio::tuneToAM() {
19      if (!tunedOnAM) {
20          tunedOnAM = true;
21          setStation( 590.0 );
22      }
23  }
24
25  void Radio::tuneToFM() {
26      if (tunedOnAM) {
```

```
27          tunedOnAM = false;
28          setStation( 88.5 );
29      }
30  }
31  bool Radio::isOnAM() {
32      return( tunedOnAM );
33  }
34
35  void Radio::setStation( double station ) {
36      stationValue = station;
37  }
38
39  double Radio::getStation() {
40      return( stationValue );
41  }
42
43  void Radio::incVolume() {
44      volume++;
45  }
46
47  void Radio::decVolume() {
48      volume--;
49  }
50
51  int Radio::getVolume() {
52      return( volume );
53  }
```