CHAPTER

4

INTRODUCTION TO CLASSES

In this chapter you will learn

- About object-oriented programming and classes
- About information hiding
- How to use access specifiers
- About interface and implementation files
- How to use Visual C++ class tools
- How to prevent multiple conclusion
- How to work with member functions

Out of chaos comes order.

Friedrich Nietzsche (1844 - 1900)

PREVIEW: THE RETIREMENT PLANNER PROGRAM

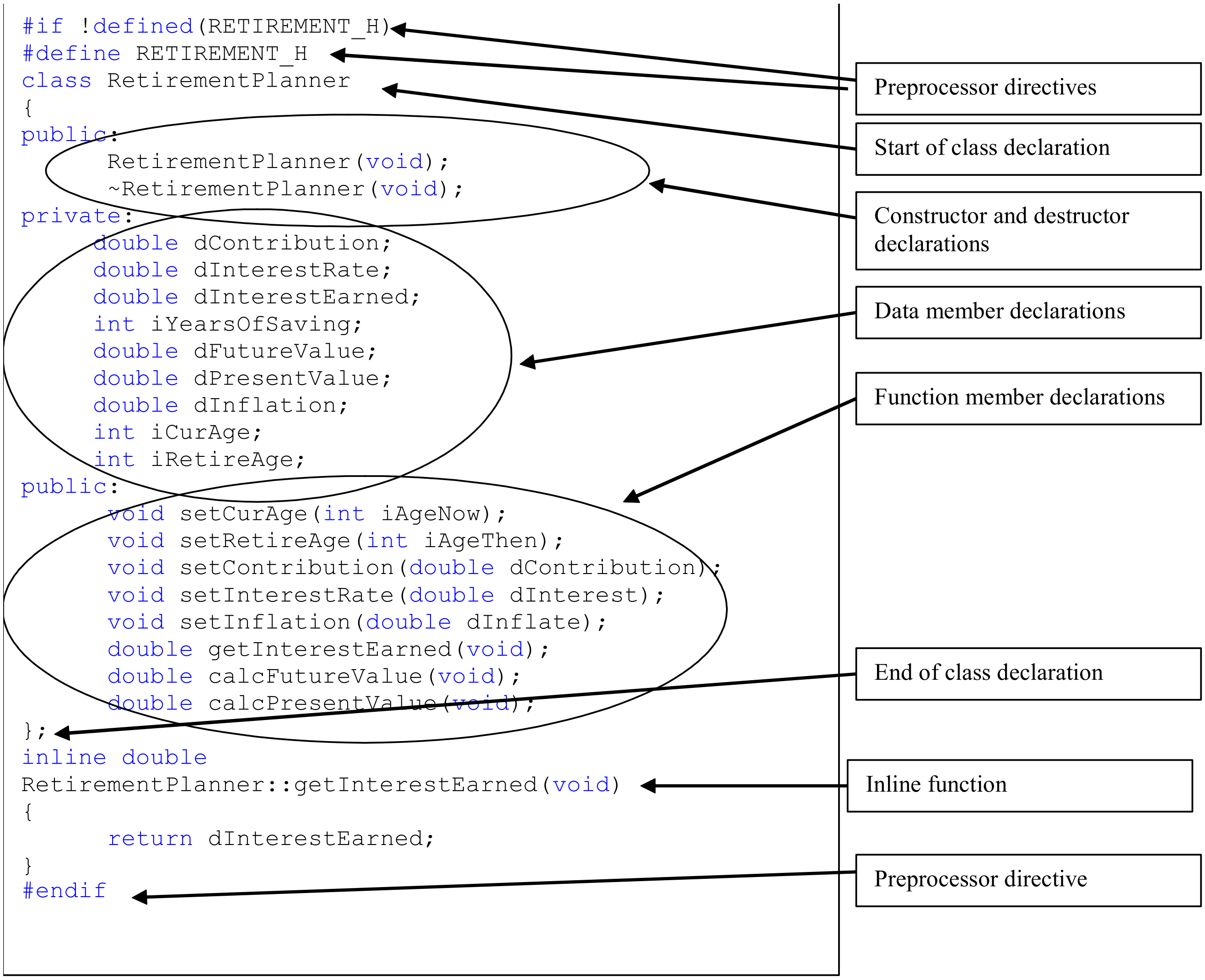
In this chapter and the next few chapters, you will study classes, which is perhaps the most important topic in C++ programming. Recall from Chapter 1, that classes are structures that contain code, methods, attributes, and other information. In this chapter you will create a Retirement Planner program to learn how to work with basic class techniques.

To preview the Retirement Planner program:

1. Create a **Chapter.04** folder in your Visual C++ Projects folder.
2. Copy the **Chapter4_RetirementPlanner** folder from the Chapter.04 folder on your Data Disk to the Chapter.04 folder in your Visual C++ Projects folder. Then open the **RetirementPlanner** project in Visual C++.
3. The Chapter4_RetirementPlanner folder in the Chapter.04 folder in your Visual C++ Projects folder contains two C++ source files and a C++ header file. First, open the **CalcSavings.cpp** file in the Code Editor window. This file is a Win32 console application with a `main()` function that displays and gathers information. You should be able to recognize most of the code. The CalcSavings.cpp file will be used for demonstrating how to work with the class that will provide the Retirement Planner program's functionality. Close the **CalcSavings.cpp** source file.

4. Open the **RetirementPlanner.h** header file. This file contains variable declarations and function prototypes, as illustrated in Figure 4-1. The file also contains some new preprocessor directives, along with two labels—public and private—that determine how functions and variables can be accessed outside of the class by other classes or programs. You will also see a function that includes the inline keyword, but that is declared outside of the class declaration. This is known as an `inline` function and is used with small functions to request that the compiler replace calls to a function with the function definition wherever the function is called in a program. Close the **RetirementPlanner.h** source file.

FIGURE 4-1: RetirementPlanner.h



```
#if !defined(RETIREMENT_H)
#define RETIREMENT_H
class RetirementPlanner
{
public:
      RetirementPlanner(void);
      ~RetirementPlanner(void);
private:
     double dContribution;
     double dInterestRate;
     double dInterestEarned;
     int iYearsOfSaving;
     double dFutureValue;
     double dPresentValue;
     double dInflation;
     int iCurAge;
     int iRetireAge;
public:
      void setCurAge(int iAgeNow);
      void setRetireAge(int iAgeThen);
      void setContribution(double dContribution);
      void setInterestRate(double dInterest);
      void setInflation(double dInflate);
      double getInterestEarned(void);
      double calcFutureValue(void);
      double calcPresentValue(void);
};
inline double
RetirementPlanner::getInterestEarned(void)
{
      return dInterestEarned;
}
#endif
```

5. Open the **RetirementPlanner.cpp** file. This file contains the actual definitions for the functions declared in the RetirementPlanner.h header file. Figure 4-2 shows a portion of the file. Notice that the file imports the RetirementPlanner.h file using an `#include` statement, but that the header file name is enclosed in quotation marks instead of brackets. Also notice that each function definition is preceded by RetirementPlanner and the scope resolution operator (::). RetirementPlanner is the name of the class itself. You use the class name and the scope resolution operator to define a function as being part of a particular class.

Figure 4-2: RetirementPlanner.cpp

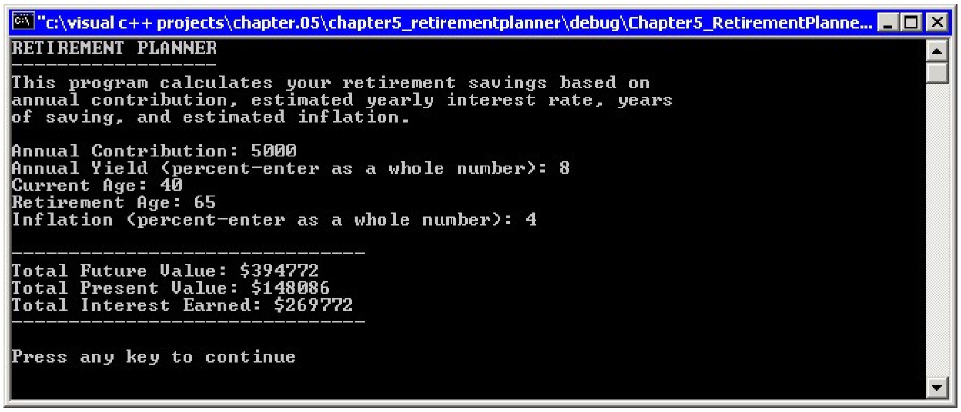
```
#include "retirementplanner.h"
RetirementPlanner::RetirementPlanner(void)
{
	dContribution = 0;
	dInterestRate = 0;
	dInterestEarned = 0;
	iYearsOfSaving = 0;
	dFutureValue = 0;
	dPresentValue = 0;
	dInflation = 0;
	iCurAge = 0;
	iRetireAge = 0;
	dInflation = 0;
}
RetirementPlanner::~RetirementPlanner(void)
{
}
void RetirementPlanner::setCurAge(int iAgeNow)
{
	iCurAge = iAgeNow;
}
...
```

Statement including the RetirementPlaner.h header file

Function names are preceded by class name and the scope resolution operator

6. Build and execute the Retirement Planner program. Then enter values for each of the variables to calculate retirement savings. Figure 4-3 shows how the program appears in the console window after entering some values

FIGURE 4-3: Retirement Planner console window


"c:\visual c++ projects\chapter.05\chapter5_retirementplanner\debug\Chapter5_RetirementPlanne...

```
RETIREMENT PLANNER
------------------
This program calculates your retirement savings based on
annual contribution, estimated yearly interest rate, years
of saving, and estimated inflation.

Annual Contribution: 5000
Annual Yield (percent-enter as a whole number): 8
Current Age: 40
Retirement Age: 65
Inflation (percent-enter as a whole number): 4

-------------------------------
Total Future Value: $394772
Total Present Value: $148086
Total Interest Earned: $269772
-------------------------------

Press any key to continue
```

7. Press any key to close the Retirement Planner program window.

8. Close the RetirementPlanner project by selecting Close Solution from the File menu.

OBJECT-ORIENTED PROGRAMMING AND CLASSES

Classes form the basis of object-oriented programming. Object-oriented programming is a way of designing and accessing code. The pieces of the programming puzzle—data types, variables, control structures, functions, and so on—are the same as in any other type of programming. What differs is how you assemble the puzzle. You first learned about classes in Chapter 1 in very general terms. In this chapter, you will learn about classes in detail.

Classes

Classes were defined in Chapter 1 as structures that contain code, methods, attributes, and other information. Now that you are familiar with the basics of a C++ program, let's refine this definition. In C++ programming, **classes** are data structures that contain variables along with functions for manipulating the variables. The functions and variables defined in a class are called **class members**. Class variables are referred to as **data members** or **member variables,** whereas class functions are referred to as **member functions** or **function members**. Functions that are not part of a class are referred to as **global functions**. To use the variables and functions in a class, you declare an object from that class. When you declare an object from a class, you are said to be **instantiating** an object. When you work with a class object, member functions are often referred to as methods, and data members are often referred to as properties.

Classes themselves are also referred to as user-defined data types or programmer-defined data types. These terms can be somewhat misleading, however, because they do not accurately reflect the fact that classes can contain member functions. Additionally, classes usually contain multiple data members of different data types, so calling a class a data type becomes even more confusing.

One reason classes are referred to as user-defined data types or programmer-defined data types is that you can work with a class as a single unit, or *object*, in the same way you work with a variable. In fact, C++ programmers use the terms *variable* and *object* interchangeably. The term *object-oriented programming* comes from the fact that you can bundle variables and functions together and use the result as a single unit (a *variable* or *object*). What this means will become clearer to you as you progress through this text. For now, think of a hand-held calculator as an example. A calculator could be considered an object of a Calculation class. You access all of the Calculation class functions (such as addition and subtraction) and its data members (operands that represent the numbers you are calculating) through your Calculator object. You never actually work with the Calculation class yourself, only with an object of the class (your calculator).

But why do you need to work with a collection of related variables and functions as a single object? Why not simply call each individual variable and function as necessary, without using all of this class business? The truth is you are not required to work with classes; you can create much of the same functionality without classes as you can by using classes. Some simple types of Visual C++ programs you write will probably not need to be created with classes. Classes help make complex programs easier to manage, however, by logically grouping related functions and data and by allowing you to refer to that grouping as a single object. Another reason for using classes is to hide information that users of a class do not need to access or know about. Information hiding helps minimize the amount of information that needs to pass in and out of an object, which helps increase program speed and efficiency. Classes also make it much easier to re-use code or distribute your code to others for use in their programs. (You will learn how to create your own classes and include them in your programs shortly.) Without a way to package variables and functions in classes and include those classes in a new program, you would need to copy and paste each segment of code you wanted to re-use (functions, variables, and so on) into any new program.

An additional reason to use classes is that instances of objects inherit their characteristics, such as class members, from the class upon which they are based. This inheritance allows you to build new classes based on existing classes without having to rewrite the code contained in the existing classes. You will learn more about inheritance in Chapter 7. For now, you should understand that an object has the same characteristics as its class.

There are two primary types of classes that you will work with in this text: classes declared with the `struct` keyword and classes declared with the `class` keyword. First, you will learn about classes declared with the `struct` keyword.

Creating Structures

So far you have worked with data types that store single values such as integers, floating-point numbers, and characters. You have also worked with arrays, which contain sets of data represented by a single variable name. One drawback to using arrays is that all elements in an array must be of the same data type. Suppose you have several pieces of related information that you want to be able to refer to as a single variable, similar to an array. An example may be the information related to a mortgage, including the property value (int), down payment (double), interest rate (double), terms (short), and a Boolean value indicating that the closing has been scheduled. You cannot use an array, however, because the individual pieces of information are of different data types. To store this type of information as a single variable, you use something called a structure. A **structure**, or **struct**, is an advanced, user-defined data type that uses a single variable name to store multiple pieces of related information. Remember that a user-defined data type is another way of referring to a class. This means that a structure is also a class. The individual pieces of information stored in a structure are called **elements**, **fields**, or **members**. You define a structure using the `struct` keyword and the following syntax:

```
struct structure_name {
     data_type field_name;
     data_type field_name;
     ...
} variable_name;
```

NOTE: You might also see structures referred to as record structures or data structures.

The *structure_name* portion of the structure definition is the name of the new, user-defined data type. You can use any name you like for a structure, as long as you follow the same naming conventions that you use when declaring variables and functions. Within the structure's curly braces, you declare the data type and field names for each piece of information stored in the structure, the same way you declare a variable and its data type. The *variable_name* portion of the structure declaration is optional and allows you to create a variable based on the new structure when the structure is first declared. If you omit *variable_name*, then you can later declare a new variable in your code using the structure name as the data type. The following code declares a structure for the mortgage information, but does not assign a variable name at declaration because *variable_name* is omitted:

```
struct Mortgage {
  char szPropertyLocation[50];
      int iPropertyValue;
      double dDownPayment;
      double dInterest;
  short siTerms;
  bool bClosingScheduled;
};
```

After creating the preceding structure, you declare a new variable of type mortgage using a statement similar to `Mortgage vacationHome;`. This statement instantiates a Mortgage object named `vacationHome`. Recall that the terms variable and object are used interchangeably. This means that a variable based on a structure is also an object. To access the fields inside a structure variable, you append a period to the variable name, followed by the field name using the syntax *`variable.field`*`;`. When you use a period to access an object's members, such as a structure's fields, the period is referred to as the **member selection operator**. You use the member selection operator to initialize or modify the value stored in an object field. For example, to assign or modify the value stored in a double field named `dInterest` in a Mortgage structure named `vacationHome`, you use a statement similar to `vacationHome.dInterest = .08;`.

The following code shows the same Mortgage structure definition followed by statements that declare a new Mortgage variable named `vacationHome` and assign values to the structure fields. Then the code shows statements that print the contents of each field:

```
#include <iostream>
#include <cstring>
struct Mortgage {
  char szPropertyLocation[50];
    int iPropertyValue;
    double dDownPayment;
    double dInterest;
  short siTerms;
  bool bClosingScheduled;
};
void main() {
```

Instantiated
Mortgage
object

```
  Mortgage vacationHome;

  strcpy(vacationHome.szPropertyLocation, "Miami, Florida");

     vacationHome.iPropertyValue = 250000;

  vacationHome.dDownPayment = .2;

  vacationHome.dInterest = .08;

  vacationHome.siTerms = 30;

  vacationHome.bClosingScheduled = true;

  cout << vacationHome.szPropertyLocation << endl;

  cout << vacationHome.iPropertyValue << endl;

  cout << vacationHome.dDownPayment << endl;

  cout << vacationHome.dInterest << endl;

  cout << vacationHome.siTerms << endl;

  cout << vacationHome.bClosingScheduled << endl;

}
```

You are not allowed to assign values to the fields inside the structure definition itself. For example, the following code causes a compile error:

```
struct Mortgage {

  char szPropertyLocation[50] = "Miami, Florida";

     int iPropertyValue = 250000;

     double dDownPayment = .2;

     double dInterest = .08;

  short siTerms = 30;

  bool bClosingScheduled = true;

};
```

You can, however, use an initializer list to assign values to a structure's fields when you declare a variable of the structure's type. An **initializer list** is a series of values that are assigned to an object at declaration. To use an initializer list, you must enclose the values you want assigned to the structure's fields within braces, separated by commas, and in the order in which the fields are declared in the structure definition. For example, the following code contains the employee structure definition, followed by the declaration of the `currentEmployee` variable, which assigns initial values to the fields:

```
#include <iostream>
#include <cstring>
struct Mortgage {
  char szPropertyLocation[50];
     int iPropertyValue;
     double dDownPayment;
     double dInterest;
  short siTerms;
  bool bClosingScheduled;
};
void main() {
     Mortgage vacationHome = {"Miami, Florida", 250000, .2,
.08, 30, true};
  ...
}
```

Initializer list

CAUTION: When using an initializer list to initialize a struct's members, you must list each value within the brackets in the order that each member is declared in the struct definition. If you do not, then the values you supply will be assigned to the wrong struct member. You will receive a compile error if you list more initialization values than there are members in the struct. If you do not supply enough initialization values, then the members you did not initialize will be assigned a value of zero.

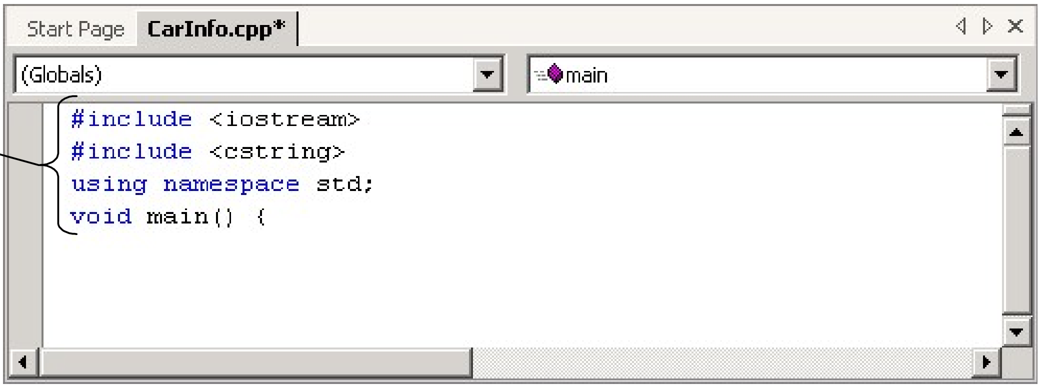
NOTE: Although you can use string class variables with structures, you cannot initialize a structure's string class variables using an initializer list. For this reason, any structure examples you see in this book will use C-style string variables instead of string class variables.

NOTE: structures are part of the C programming language. However, they are widely used in Windows programming; in fact, you will use structures extensively when you work with Windows programs later in this text. In order to familiarize you with defining structures, you will now create a simple console application that creates a structure named `sportsCar`, assigns values to a `sportsCar` variable, and then prints the variable's contents. The `sportsCar` structure will define fields of several different data types that will contain the specifications of a particular sports car.

To create a simple console application that creates the `sportsCar` structure, assigns values to a sportsCar variable, and then prints the variable's contents:

1. Create a new Win32 Project named **CarInfo** in the **Chapter.04** folder in your Visual C++ Projects folder. Be sure to clear the **Create directory for Solution** checkbox in the New Project dialog box. In the Application Settings tab of the Win32 Application Wizard dialog box, select **Console application** as the application type, click the **Empty project** checkbox, and then click the **Finish** button. Once the project is created, add a C++ source file named **CarInfo**.
2. As shown in Figure 4-4, type the preprocessor directives that give the program access to the iostream and cstring libraries along with the using directive that designates the std namespace. Also, type the opening header for the `main()` function.

FIGURE 4-4: Opening statements added to CarInfo.cpp

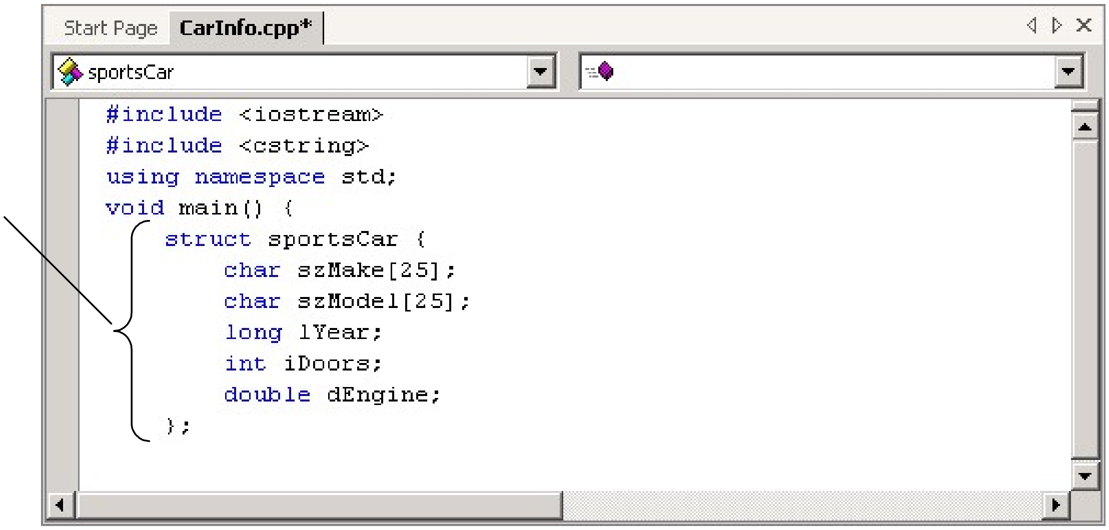


Add these four statements

```
#include <iostream>
#include <cstring>
using namespace std;
void main() {
```

4. In the `main()` function body, define the following `sportsCar` structure, as shown in Figure 4-5. Notice that the structure's fields are of different data types.

FIGURE 4-5: `sportsCar` struct added to the `main()` function

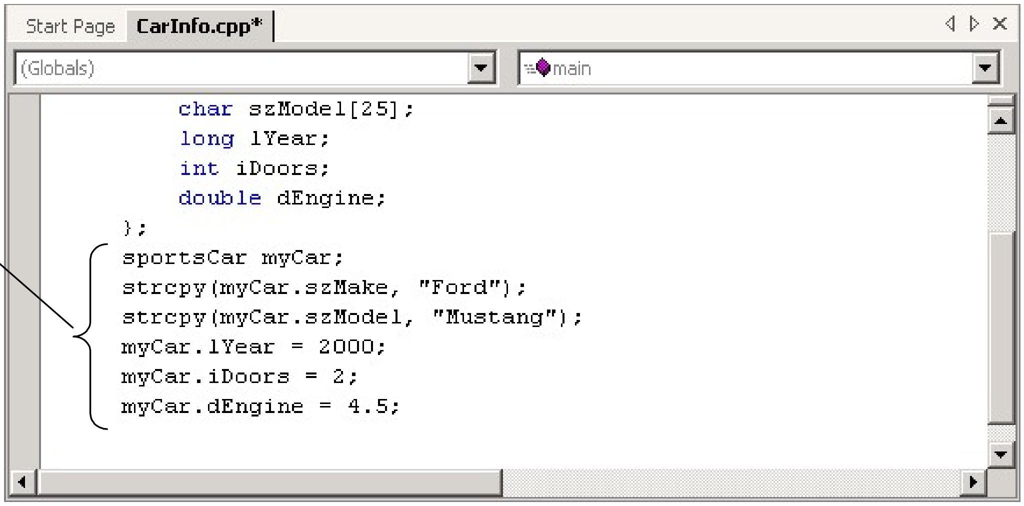


Add the `sportsCar` structure here

```
#include <iostream>
#include <cstring>
using namespace std;
void main() {
    struct sportsCar {
        char szMake[25];
        char szModel[25];
        long lYear;
        int iDoors;
        double dEngine;
    };
```

5. Type the statements shown in Figure 4-6, which declare a new `sportsCar` variable named `myCar` and assign values to the structure's fields.

FIGURE 4-6: `sportsCar` variable declared and values assigned to the structure's fields

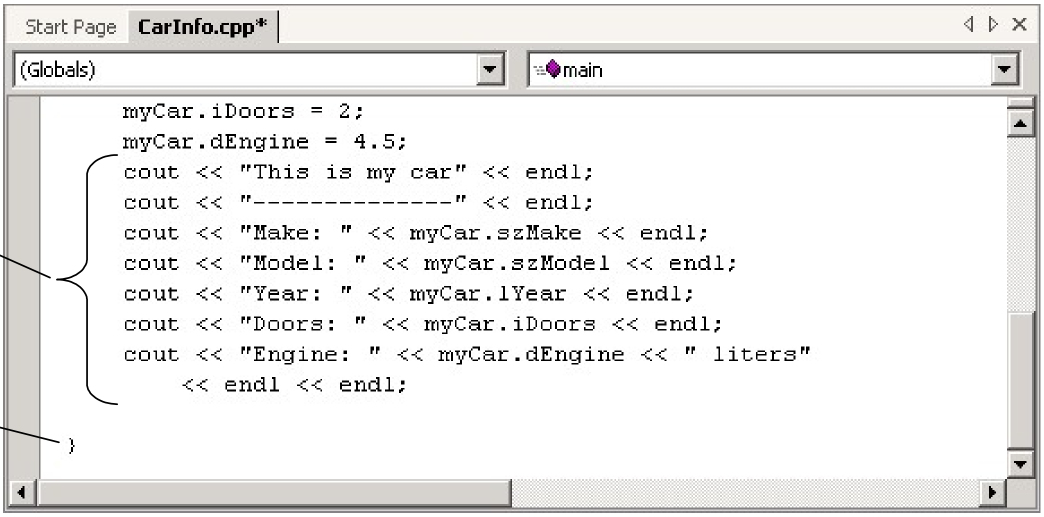


```
        char szModel[25];
        long lYear;
        int iDoors;
        double dEngine;
    };
    sportsCar myCar;
    strcpy(myCar.szMake, "Ford");
    strcpy(myCar.szModel, "Mustang");
    myCar.lYear = 2000;
    myCar.iDoors = 2;
    myCar.dEngine = 4.5;
```

Add these statements to declare the `sportsCar` variable and assign values to the structure's

6. Finally, add the statements shown in Figure 4-7, which print the values assigned to the structure's fields. Also, type the `main()` function's closing brace:.

FIGURE 4-7: Output statements and the `main()` function's closing brace added to CarInfo.cpp



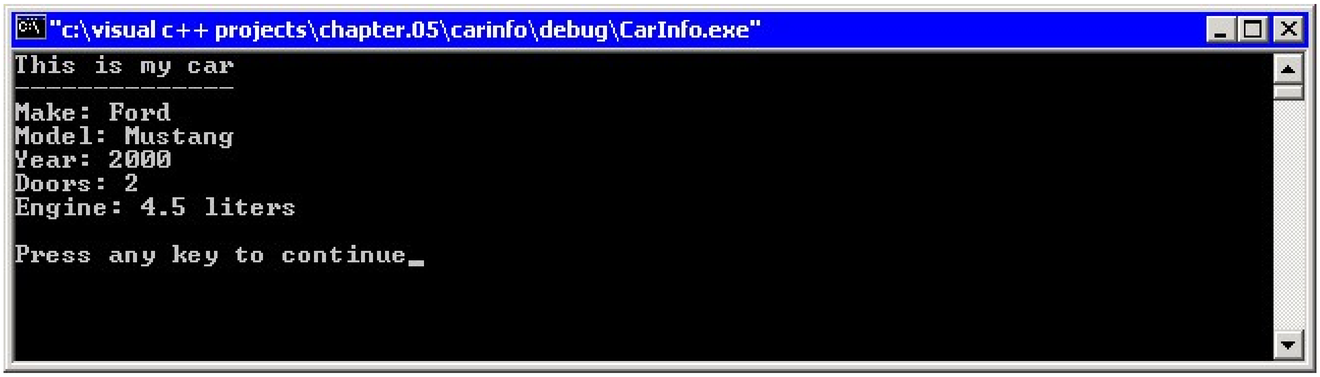
```
    myCar.iDoors = 2;
    myCar.dEngine = 4.5;
    cout << "This is my car" << endl;
    cout << "--------------" << endl;
    cout << "Make: " << myCar.szMake << endl;
    cout << "Model: " << myCar.szModel << endl;
    cout << "Year: " << myCar.lYear << endl;
    cout << "Doors: " << myCar.iDoors << endl;
    cout << "Engine: " << myCar.dEngine << " liters"
        << endl << endl;

}
```

Print the values assigned to the structure's fields

Add the `main()` function's closing brace

7. Build and execute the CarInfo program. Figure 4-8 shows the output.

FIGURE 4-8: Output of the Car Info program



"c:\visual c++ projects\chapter.05\carinfo\debug\CarInfo.exe"

```
This is my car
--------------
Make: Ford
Model: Mustang
Year: 2000
Doors: 2
Engine: 4.5 liters

Press any key to continue_
```

8. Press any key to close the command window.

9. Close the CarInfo project by selecting Close Solution from the File menu.

Creating Classes with the `class` Keyword

The most important type of class used in C++ programming is defined using the `class` keyword. For brevity, from this point forward classes defined with the `class` keyword will be referred to simply as classes. You define classes the same way you define structures, and you access a class's data members using the member selection operator. The following code shows an example of a class named `Stocks`:

```
class Stocks {
public:
     int iNumShares;
     double dPurchasePricePerShare;
     double dCurrentPricePerShare;
};
void main() {
     Stocks stockValue;
     stockValue.iNumShares = 500;
     stockValue.dPurchasePricePerShare = 10.785;
```

```
        stockValue.dCurrentPricePerShare = 6.5;
    }
```

The differences between the preceding class and the structure example you saw earlier are the use of the `class` keyword and the `public:` label. The `public:` label determines default accessibility to a class's members. In fact, default accessibility is one of the only differences between structures and classes. For now, you should understand that the accessibility to a class's members is what allows you to hide information, such as data members, from users of your class. You will learn more about accessibility when information hiding is discussed later in this chapter.

NOTE: Structures are left over from C programming. Classes are unique to C++ programming. In C programming, structures do not support encapsulation because C programming is primarily a procedural programming language, not object-oriented, as is C++. In C++ programming, however, structures do support encapsulation, making them virtually identical to classes. This means that you can substitute the `struct` keyword for any classes declared with the `class` keyword. However, most C++ programmers use the `class` keyword to clearly designate the programs they write as object-oriented C++ programs. And because you are studying C++, you will define your classes with the `class` keyword.

INFORMATION HIDING

One of the fundamental principals in object-oriented programming is the concept of information hiding. Information hiding gives an encapsulated object its black box capabilities so that users of a class can see only the members of the class that you allow them to see. Essentially, the principal of **information hiding** states that any class members that other programmers, sometimes called *clients*, do not need to access or know about should be hidden. Information hiding helps minimize the amount of information that needs to pass in and out of an object, which helps increase program speed and efficiency. Information hiding also reduces the complexity of the code that clients see, allowing them to concentrate on the task of integrating an object into their programs. For example, if a client wants to add to her Accounting program a Payroll object, she does not need to know the underlying details of the Payroll object's member functions, nor does she need to modify any local data members that are used by those functions. The client only needs to know which of the object's member functions to call and what data (if any) needs to be passed to those member functions.

Now consider information hiding on a larger scale. Professionally developed software packages are distributed in an encapsulated format, which means that the casual user—or even an advanced programmer—cannot see the underlying details of how the software is developed. Imagine what would happen if Microsoft distributed Excel without hiding the underlying programming details. Most users of the program would be bewildered if they accidentally opened the source files. Obviously, there is no reason why Microsoft would allow users to see the underlying details of Excel, because users do not need to understand how the underlying code performs the various types of spreadsheet calculations. Microsoft also has a critical interest in protecting proprietary information, as do you. The design and sale of software components is big business. You certainly do not want to spend a significant amount of time designing an outstanding software component, only to have an unscrupulous programmer steal the code and claim it as his or her own.

This same principal of information hiding needs to be applied in object-oriented programming. There are few reasons why clients of your classes need to know the underlying details of your code. Of course, you cannot hide *all* of the underlying code, or other programmers will never be able to integrate your class with their applications. But you need to hide most of it.

Information hiding on any scale also prevents other programmers from accidentally introducing a bug into a program by modifying a class's internal workings. Programmers are curious creatures and will often attempt to "improve" your code, no matter how well it is written. Before you distribute your classes to other programmers, your classes should be thoroughly tested and bug-free. With tested and bug-free classes, other programmers can focus on the more important task of integrating your code into their programs using the data members and member functions you designate.

To enable information hiding in your classes you must designate access specifiers for each of your class members. You must also place your class code into separate interface and implementation files. You will learn about these topics next.

Access Specifiers

The first step in hiding class information is to set access specifiers for class members. **Access specifiers** control a client's access to individual data members and member functions. There are four levels of access specifiers: `public`, `private`, `protected`, and `friend`. You will use the public, private, and friend access specifiers in this chapter. In Chapter 7, you will learn about the protected access specifier.

The **`public` access specifier** allows anyone to call a class's member function or to modify a data member. The **`private` access specifier** prevents clients from calling member functions or accessing data members and is one of the key elements in information hiding. Private access does not restrict a class's internal access to its own members; a class's member function can modify any private data member or call any private member function. Private access restricts clients from accessing class members. The `private` access specifier does not actually hide class member definitions; it only protects them. To hide class member definitions, you must separate classes into interface and implementation files, which you will learn about shortly.

NOTE: Both `public` and `private` access specifiers have what is called **class scope**: Class members of both access types are accessible by any of a class's member functions. In contrast, variables declared inside a member function have local scope to the function only and are not available outside the function, even if the function is declared with the `public` access specifier.

TIP: In Class View, the icons that represent private class members include a symbol that looks like a padlock.

You place access specifiers in a class definition on a single line followed by a colon, similar to a `switch` statement's case labels. An access specifier that is placed on a line by itself followed by a colon is called an **access label**. The access privilege of any particular access label is applied to any class members that follow, up to the next label. For example, the following code contains a public and a private access label. The public label declares two public data members, `iNumShares` and `dPurchasePricePerShare`, and the private label declares a single private data member, `dCurrentPricePerShare`.

```
class Stocks {
public:
     int iNumShares;
     double dPurchasePricePerShare;
private:
     double dCurrentPricePerShare;
};
```

Access labels

Access labels can be repeated, although most programmers prefer to organize all of their public class members under a single public access label, and all of their private class members under a single private access label. However, the following code organization with its two public access labels is legal:

```
class Stocks {
public:
     int iNumShares;
private:
     double dCurrentPricePerShare;
public:
     double dPurchasePricePerShare;
};
```

TIP: It is common practice to list public class members first in order to clearly identify the parts of the class that can be accessed by clients.

The default access specifier for classes is private. If you exclude access specifiers from your class definition, then all class members in the definition are private by default. For example, because the following class definition does not include any access labels, the three data member definitions are private by default:

```
class Stocks {
     int iNumShares;
     double dPurchasePricePerShare;
     double dCurrentPricePerShare;
};
```

If you have some reason for making all of your class members private, you should include a private access label to make it clear how you intend for the class members to be used. Many programmers prefer to make all of their data members private to prevent clients from accidentally assigning the wrong value to a variable or from viewing the internal workings of their programs. Or, they simply want to prevent curious clients from modifying the various parts of their program. Even if you do not need to make it clear for yourself, you should include an access label in case other programmers need to modify your work.

NOTE: Default class member access is one of the major differences between classes and structures in C++. Access to classes is private by default. Access to structures is public by default.

Even if you make all data members in a class private, you can still allow clients of your program to retrieve or modify the value of data members by using accessor functions. **Accessor functions** are public member functions that a client can call to retrieve or modify the value of a data member. Because accessor functions often begin with the words *get* or *set*, they are also referred to as get or set functions. Get functions retrieve data member values; set functions modify data member values. To allow a client to pass a value to your program that will be assigned to a private data member, you include arguments in a set function's header definition. You can then write code in the body of the set function that validates the data passed from the client, prior to assigning values to private data members. For example, if you write a class named Payroll that includes a private data member containing the current state income-tax rate, then you could write a public accessor function named `getStateTaxRate()` that allows clients to retrieve the variable's value. Similarly, you could write a `setStateTaxRate()` function that performs various types of validation on the data passed from the client (such as making sure the value is not null, is not greater than 100%, and so on) prior to assigning a value to the private state tax rate data member.

Interface and Implementation Files

Although the first step in information hiding is to assign `private` access specifiers to class members, `private` access specifiers only designate which class members a client is not allowed to call or change. `Private` access specifiers do not prevent clients from seeing class code. To prevent clients from seeing the details of how your code is written, you place your class's interface code and implementation code in separate files. The separation of classes into interface and implementation files is a fundamental C++ software development technique because it allows you to hide the details of how your classes are written and makes it easier to modify programs.

Interface code refers to the data member and member function declarations inside a class definition's braces. Interface code does not usually contain definitions for member functions, nor does it usually assign values to the data members. Declarations are statements that only declare data members without assigning a value to them, such as `double dCurrentPricePerShare;`, or function prototypes such as `double getTotalValue();`. You create interface code in a header file with an .h extension. The interface code should be the only part of your class that a client can see and access. In effect, the interface is the "front door" to your program.

Implementation code refers to a class's function definitions and any code that assigns values to a class's data members. In other words, implementation code contains the actual member functions themselves and assigns values to data members. You add implementation code to standard C++ source files with an extension of .cpp. You give the implementation code access to the interface code by importing the header file into the C++ source file using an #include directive, just as you would import a header file from the C++ run-time library. However, instead of placing the header file name within a set of angle brackets (as you would with a header file from the C++ run-time library), you must place the name of a custom class within a set of quotation marks and include the .h extension. For example, you give the Stocks class implementation file access to the Stocks class interface file using the statement `"stocks.h"` (assuming the Stocks class interface code is saved in a file named stocks.h).

TIP: If you are familiar with the Java programming language, then you know that Java files must use the same name, including letter case, as the class they contain. In C++, however, you are not required to name your interface or implementation files with the same name and letter case as the class name.

C++ source files are distributed in compiled format, whereas header files are distributed as plain text files. Thus, clients who use your code can see only the names of data members and member functions. Clients can use and access public class members, but they cannot see the details of how your code is written. Without this ability to hide implementation details, clients could easily get around restrictions you place on class members with the `private` access specifier by copying your code into a new class file, and changing `private` access specifiers to public.

CAUTION: If you are using classes only to make your own code more efficient and have no intention of distributing your classes to others, you can place both the declarations and definitions into the same file. However, this is not considered to be good programming practice, because the separation of interface and implementation is a fundamental C++ software development technique. Additionally, if you change your mind and decide to distribute your class to others, you would need to go back and separate the class into interface and implementation files.

Now you will examine the RetirementPlanner.cpp implementation file that the Generic C++ Class Wizard automatically added for you. Note that some of the examples of implementation files you have seen in this chapter have included their own `main()` functions. Although you can add executable class code by including a `main()` function in an implementation file, you are not required to. For example, the RetirementPlanner.cpp implementation file will not include a `main()` function. Instead, the RetirementPlanner.cpp implementation file will be called by the CalcSavings.cpp file's `main()` function.

Modifying a Class

Hiding implementation details is reason enough for separating a class's interface from its implementation. But, another important reason for separating a class into interface and implementation files is to make it easier to modify a program at a later date. When you modify a class, interface code, such as class member declarations, should change the least. The implementation code normally changes the most when you modify a class. This rule of thumb is not carved in stone because you may find it necessary to drastically modify your class's interface. But for the most part, the implementation is what will change.

No matter what changes you make to your implementation code, the changes will be invisible to clients if their only entry point into your code is the interface—provided the interface stays the same. Designing your code so that modifications are made to the implementation code and not the interface code means that if you make any drastic changes or improvements to your class, you only need to distribute a new .cpp file to your clients, not a new interface file. Be aware, however, that if you modify class member declarations or add new declarations to expand the program's functionality, you may also need to distribute a new header file.

If the public interface class members stay the same, then clients do not need to make any changes to *their* code in order to work with your modified class. For instance, consider the Payroll object discussed earlier. The Payroll object may contain a public function member named `calcFederalTaxes()` that calculates a paycheck's federal tax withholding based on income-tax percentages published by the Internal Revenue Service. If the Internal Revenue Service changes any of the income-tax percentages, then you will need to modify the private data members within the `calcFederalTaxes()` function. Clients, however, do not need to be concerned with these details; they will continue to call the public `calcFederalTaxes()` function as usual. For these types of changes, you would not need to distribute a new interface file to your clients. You would need to distribute only a new implementation file containing the modified `calcFederalTaxes()` function.

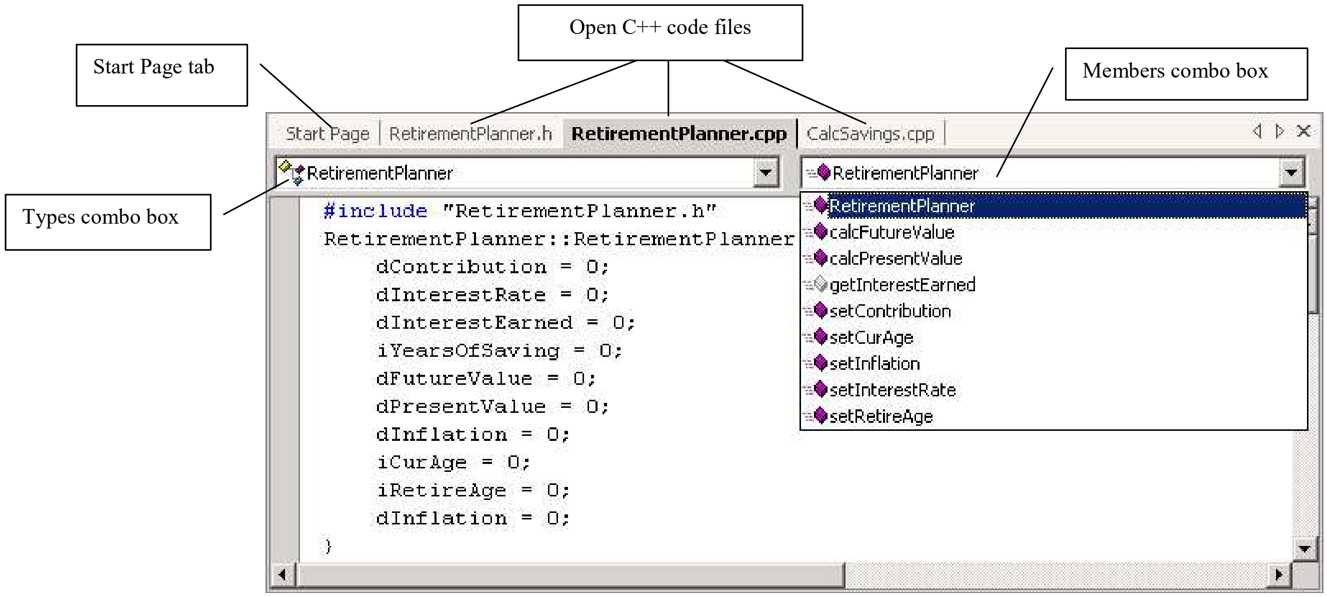
VISUAL C++ CLASS TOOLS

You can work with class header and source files using the Solution Explorer window, in the same manner that you work with C++ files that are not class-based. However, Visual C++ includes various tools that make it easier to work with the classes in your programs.

Although you have been working with the Code Editor window for some time now, it has a few additional class features that are worth examining. Figure 4-9 shows an example of the Retirement Planner project open in the Code Editor window. The Navigation bar at the top of the Code Editor window contains two combo boxes that you can use to navigate to a particular class or its members. The Types combo box at the left of the Navigator bar allows you to select a class name or *global*, which displays global declarations that are not associated with a particular class. The Members combo box at the right of the Navigator bar allows you to select a class member or global declaration, depending on what is selected in the Types combo box.

Once your projects begin to include multiple files, you will find it helpful to use tabs at the top of the Code Editor window to navigate to an open file. Figure 4-9 points out the tabs in the Code Editor window for each of the opened files, along with the tab for the Start Page window.

FIGURE 4-9: The Code Editor window with the contents of the Members combo box displayed



```
#include "RetirementPlanner.h"
RetirementPlanner::RetirementPlanner
    dContribution = 0;
    dInterestRate = 0;
    dInterestEarned = 0;
    iYearsOfSaving = 0;
    dFutureValue = 0;
    dPresentValue = 0;
    dInflation = 0;
    iCurAge = 0;
    iRetireAge = 0;
    dInflation = 0;
}
```

HELP: By default, the Start Page window also displays as a tab next to any open files in the Code Editor window, but you can easily close it by clicking its Close button.

TIP: Another Visual C++ tool that you may find useful is the Object Browser window, which allows you to examine class members and other programming elements that are contained within objects that are used by your program. To manually display the Object Browser window point to the Other Windows submenu on the View menu and click Object Browser, or press Ctrl+Alt+J.

Class View

The **Class View window** displays project files according to their classes and is similar to the Solution Explorer window. However, whereas Solution Explorer displays a hierarchical list of all projects, folders, and files in the solution, Class View displays hierarchical lists of classes and the members they contain. You primarily use Class View to navigate through the declarations and definitions of class members. An icon represents each of the various items displayed in Class View. Double clicking a data member's icon brings you to its declaration in the class header file. Double clicking a member function's icon brings you to its definition in the class source file.

[TIP] See the Class View and Object Browser Icons topic in the MSDN Library for a complete list of icons that display in Class View.

NOTE: You can also right-click a class member in Class View and select from the shortcut menu one of the navigation or sorting commands listed in Figure 4-10.

FIGURE 4-10: Navigation and Sorting commands available in Class View

Command	Description
Go To Definition	Opens the C++ file containing the member definition in the Code Editor window and places the insertion point in the definition statement
Go To Declaration	Opens the C++ file containing the member declaration in the Code Editor window and places the insertion point in the declaration statement
Browse Definition	Displays the selected class member in Object Browser
Quick Find Symbol	Searches for a symbol (which is an object such as a class or its

	members) according to criteria you previously entered in the Find Symbol Dialog box; You can open the Find Symbol dialog box by pointing to Find and Replace on the Edit menu and clicking Find Symbol, or by pressing Ctrl+Shift+Y
Sort Alphabetically	Sorts the items displayed in Class View alphabetically
Sort By Type	Sorts the items displayed in Class View by data type
Sort By Access	Sorts the items displayed in Class View according to their access specifier
Group By Type	Groups the items displayed in Class View by data type

TIP: The shortcut menu in Class View also includes a copy command, which allows you to copy a member declaration, and a Properties command, which displays the Properties window for the selected member.

HELP: The commands displayed on the shortcut menu in Class View will change, depending on the type of icon you select. The Go To Declaration command, for instance, is unavailable if you right-click a data member.

TIP: The Class View toolbar that displays at the top of the Class View window contains two buttons: Class View Sort By Type and New Folder. The Class View Sort By Type button displays the same sort commands that are listed in Figure 4-10. The New Folder button allows you to create "virtual folders" that you can use to organize the various items listed in the Class View window.

Next, you will use the Class View window to navigate through the class members in the Retirement Planner program you opened in the chapter preview.

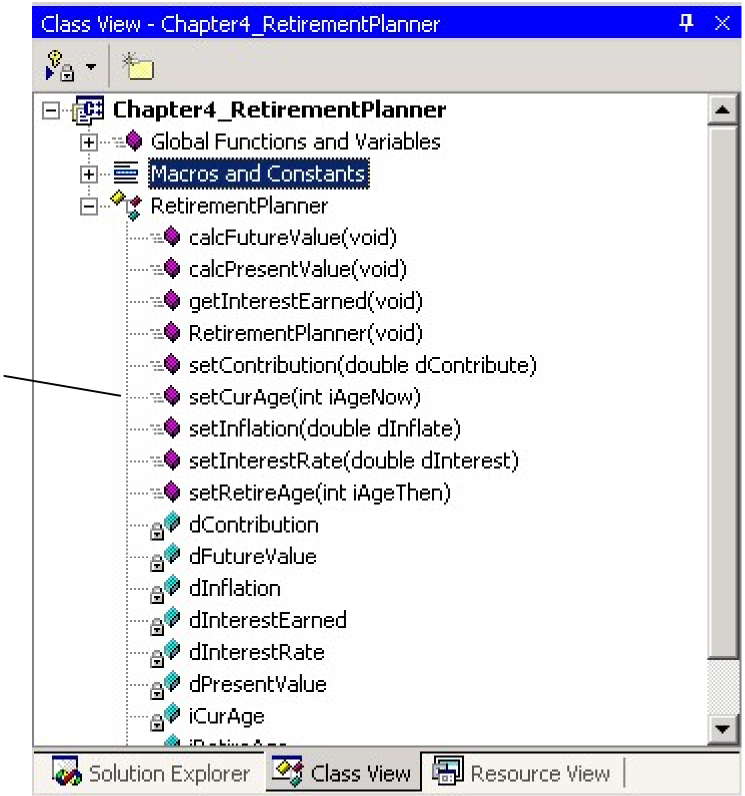
To use the Class View window to navigate through the class members in the Retirement Planner program you saw in the chapter preview:

1. Open the **RetirementPlanner** project from the **Chapter4_RetirementPlanner** folder in the Chapter.04 folder on your Data Disk.

2 Select **ClassView** from the **View** menu or press **Ctrl**+**Shift**+**C**. If ClassView is already open, you can simply click the ClassView tab next to the Solution Explorer tab.

3. The first item in the Class View window is a project icon for the RetirementPlanner project. Click the Plus box next to the project icon to expand its contents. You should see three items: Global Functions and Variables, Macros and Constants, and RetirementPlanner. The Global Functions and Variables item lists any global functions, such as the `main()` function, and global variables that you declare in a project. The Macros and Constants item lists any macros and constants that your project uses or defines. A macro represents C++ code, constants, and other programming elements and is defined using the #define preprocessor directive. You will learn about macros in Chapter 9. The RetirementPlanner item in ClassView represents the RetirementPlanner class. Click the Plus box next to the RetirementPlanner class icon. You should see a list of the RetirementPlanner class members, as shown in Figure 4-11.

FIGURE 4-11: RetirementPlanner class members in ClassView



Class View - Chapter4_RetirementPlanner

```
Chapter4_RetirementPlanner
    Global Functions and Variables
    Macros and Constants
    RetirementPlanner
        calcFutureValue(void)
        calcPresentValue(void)
        getInterestEarned(void)
        RetirementPlanner(void)
        setContribution(double dContribute)
        setCurAge(int iAgeNow)
        setInflation(double dInflate)
        setInterestRate(double dInterest)
        setRetireAge(int iAgeThen)
        dContribution
        dFutureValue
        dInflation
        dInterestEarned
        dInterestRate
        dPresentValue
        iCurAge
```

RetirementPlanner class members

4. Double-click the first class member, `calcFutureValue(void)`, which is a member function of the RetirementPlanner class. The RetirementPlanner.cpp source file should open in the Code Editor window and the definition for `calcFutureValue(void)` function should be selected. For now, do not worry about the coding syntax for the classes, statements, and functions; just focus on the class navigation techniques.

5. Next, right-click the `calcFutureValue(void)` member in Class View and select **Go To Declaration** from the shortcut menu. The RetirementPlanner.h header file should open in the Code Editor window and the declaration statement for the `calcFutureValue(void)` function should be selected.

6. Close the RetirementPlanner project by selecting **Close Solution** from the **File** menu. In the next exercise, you will start creating the Retirement Planner program from scratch.

[NOTE] This project contains only one class named RetirementPlanner. If it contained additional classes, they would be located in an alphabetical list beneath the project icon.

Code Wizards

As you start building more complex projects, you may find it somewhat tedious to add all of the necessary code for a specific C++ programming element. For instance, to add a single member function to a class, you need to declare the function prototype in the header file and define the function in a separate source file. If your function prototype and the header in your function definition do not match exactly, you will receive a compile error when you attempt to build the project. In order to make it easier to add code to your projects, Visual C++ includes **code wizards**, which automate the task of adding specific types of code to your projects. You have already worked with several code wizards, including the Add New Item command. Listed below are some additional code wizards that are designed specifically for classes:

- Add Class Wizard
- Add Member Function Wizard
- Add Member Variable Wizard

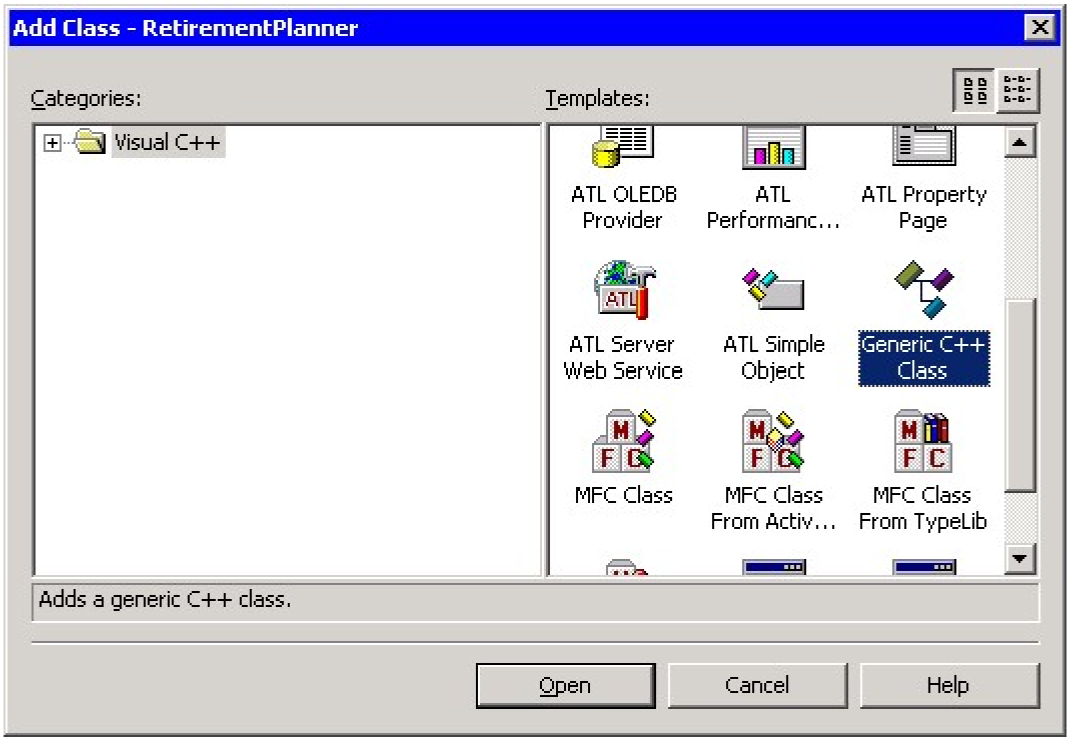
The Add Member Variable Wizard is of most use when working with MFC programs, so it will not be discussed until Chapter 9. The code wizards that will be examined in this chapter are the Add Class Wizard and Add Member Function Wizard.

CAUTION: Code wizards do not remove the need for you to understand how your code operates–they only assist you by adding the basic parts of each code element. It is still up to you to add the necessary code to give your program its functionality.

The Add Class Wizard

You can run the Add Class Wizard from anywhere in your project by selecting Add Class from the Project menu. Running the Add Class Wizard displays the Add Class dialog box shown in Figure 4-12.

FIGURE 4-12: The Add Class dialog box



Adds a generic C++ class.

When you use the Add Class Wizard to add a class to a Visual C++ project, Visual C++ should be selected by default in the Categories pane. The Templates pane displays a list of the various types of classes that can be added with the Add Class code wizard. The Templates option you will use in the next few chapters will be the Generic C++ Class option, which adds to a project a regular C++ class (as opposed to an MFC class or other type of class. Once you select a class type in the Add Class Wizard dialog box and press the Open button, the class wizard for your selected option executes. When you select the Generic C++ Class option, for instance, the Generic C++ Class Wizard dialog box appears.

Figure 4-13 shows an example of the Generic C++ Class Wizard dialog box. You type the name of your new class in the Class name text box. As you type the class name, the Generic C++ Class Wizard uses the class name you type as the suggested names for the header and source files. Although you can change the names of the header and source files if you want, it is usually easier to use the class name as the name of its header and source file. You use the Base class text box to designate another class upon which to base the new class. The Access combo box determines the accessibility (public, private, and so on) that the new class will have to the members in its base class. The Virtual destructor check box creates a virtual destructor in the new class, which helps ensure that the correct destructor executes when objects of classes that are based on other classes are deleted. The Inline check box creates the class definition code in the same file (with an extension of .h) as the declaration code. Although creating a class's declaration and definition code in the same file may make it easier to manage your class, it removes the ability to use information-hiding techniques.

FIGURE 4-13: The Generic C++ Class Wizard dialog box

NOTE: You will learn about base classes and destructors when you study inheritance techniques in Chapter 7.

Add Member Function Wizard

To run the Add Member Function Wizard, you click the Add Function command on the Project menu if you have a class icon selected in Class View. Alternatively, you can right-click a class icon in Class View, and select the Add Function command from the Add submenu on the shortcut menu.

After you run the Add Function command, the Add Member Function Wizard dialog box appears. You enter the name of the new function (without the parentheses) in the Function name text box. As you build the function, its declaration enters automatically into the Function signature text box at the bottom of the dialog box. You select a function's return type from the Return type combo box. To add a parameter, you select its type from the Parameter type combo box, type a name for the parameter in the Parameter name text box, and then click the Add button. The new parameter will appear in the Function signature text box at the bottom of the dialog box. You can then repeat the same steps to add additional parameters to the function. You can continue adding additional parameters, or remove a parameter by highlighting it in the Parameter list and clicking the Remove button. You select the function's access specifier from the Access combo box. The Static, Virtual, Pure, and Inline check boxes create advanced member function features that you will study later. The .cpp file box identifies the implementation file where the Add Member Function Wizard will create the function definition; by default, this is the .cpp file of the class to which the function is added.

TIP: For the Return type and Parameter type combo boxes, you can select a data type from the list or manually type an entry into the text portion of each combo box. You can also add a comment to the function using the Comment text box.

The function being added with the Add Member Function Wizard in Figure 4-14 does not include any parameters. However, notice that the definition in the Function signature text box contains the `void` keyword between the function's parentheses. A parameter value of void simply indicates that the function takes no parameters and is equivalent to leaving the function's parentheses empty. The Add Member Function Wizard adds the void keyword to make it explicitly clear that the function does not take parameters. You can add the void keyword in the parameter list for any functions you manually create that do not accept parameters, although it is not necessary to do so. However, keep in mind that any functions you add to your project with the Add Member Function Wizard that do not include functions *will* be created with the `void` keyword between the function's parentheses. For example, after clicking the Finish button in Figure 4-14, the Add Member Function Wizard creates the following function definition:

```
double Payroll::calcFederalTaxes(void)
{
  return 0;
}
```

One last thing to note is that when you create a function that returns a value, the Member Function Wizard automatically adds a default return statement for you. For example, the preceding function is automatically created with a return statement of `return 0;`. You change this statement to whatever value you need returned from your function. The `calcFederalTaxes()` function, for instance, may return a double variable named `dTaxResults`. Therefore, you would change the `return 0;` statement to `return dTaxResults;`.

Next, you will begin working on the Retirement Planner program. First you will create the project, and then you will use the Add Class Wizard to add the RetirementPlanner class to the project. Shortly, you will use the Add Member Function Wizard to add member functions to the RetirementPlanner class.

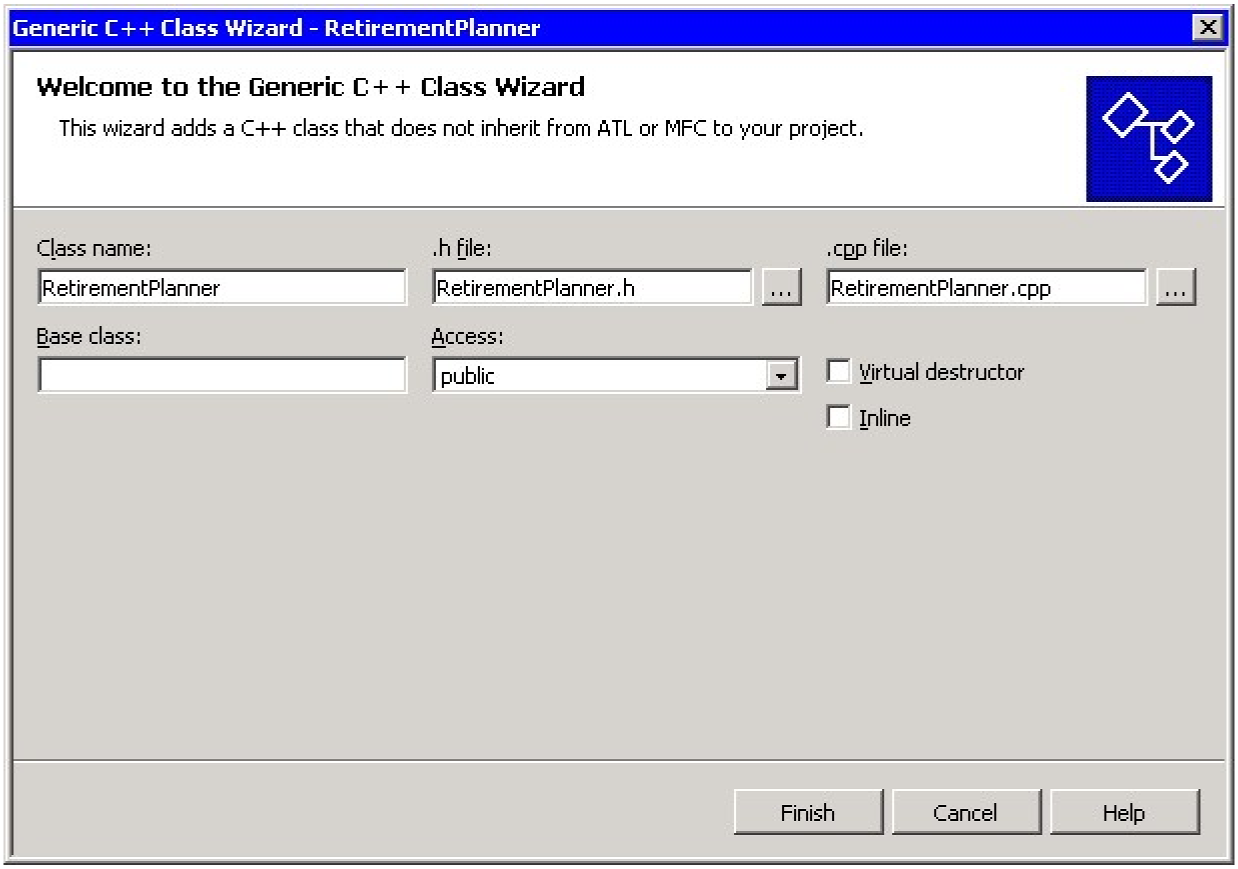
To begin working on the Retirement Planner program:

1. Return to Visual C++.
2. Create a new Win32 Project named **RetirementPlanner** in the **Chapter.04** folder in your Visual C++ Projects folder. Be sure to clear the **Create directory for Solution** checkbox in the New Project dialog box. In the Application Settings tab of the Win32 Application Wizard dialog box, select **Console application** as the application type, click the **Empty project** checkbox, and then click the **Finish** button.

3. Once the project is created, select **Add Class** from the **Project** menu to start the Add Class Wizard. The Add Class dialog box appears. If necessary, select Visual C++ in the Categories pane (it should be selected by default). The Templates pane displays a list of the various types of classes that can be added with the Add Class code wizard. Scroll through the list and select the **Generic C++ Class** option, then click the **Open** button.

4. After you click the Open button, the Generic C++ Class Wizard dialog box appears. Type **RetirementPlanner** in the Class name text box. Leave the Base class text box empty and the Access combo box set to its default setting of *public*. Make sure the Virtual destructor check box and Inline check boxes are cleared and click the **Finish** button. Figure 4-15 shows how the Generic C++ Class Wizard dialog box should appear before selecting the Finish button.

FIGURE 4-14: The Generic C++ Class Wizard dialog box when adding the RetirementPlanner class



Generic C++ Class Wizard - RetirementPlanner

Welcome to the Generic C++ Class Wizard

This wizard adds a C++ class that does not inherit from ATL or MFC to your project.

Class name: RetirementPlanner

.h file: RetirementPlanner.h

.cpp file: RetirementPlanner.cpp

Base class:

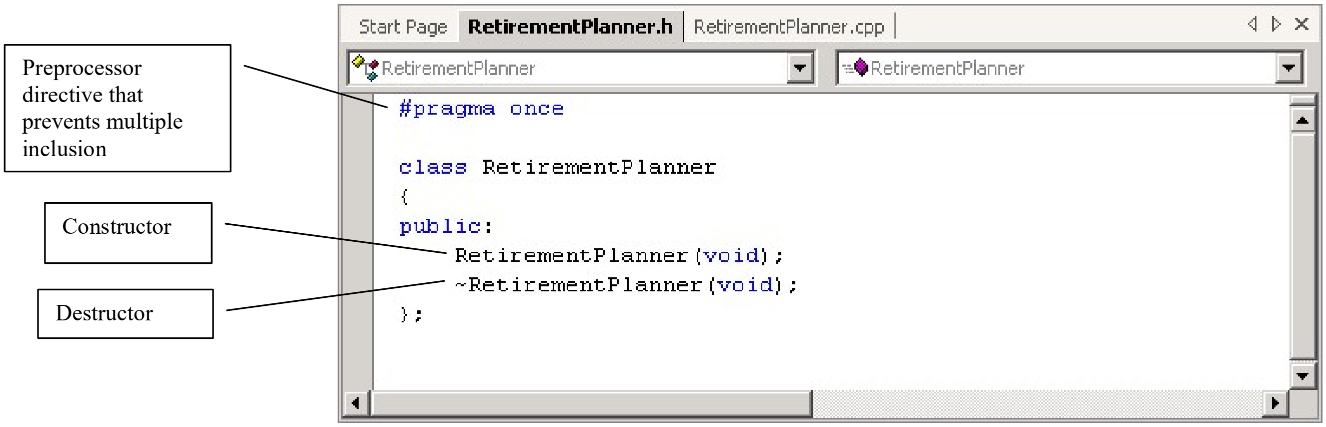
Access: public

□ Virtual destructor

□ Inline

5. After you click the Finish button, the Generic C++ Class Wizard creates the RetirementPlanner class header file (RetirementPlanner.h) and source file (RetirementPlanner.cpp). After the wizard finishes creating the files, the RetirementPlanner.h header file opens in the Code Editor. The first statement you see, `#pragma once`, is a preprocessor directive that prevents multiple instances of the same header file from being included when you compile the project. Multiple inclusion will be covered later in this chapter. The Generic C++ Class Wizard also creates the class declaration shown in Figure 4-15. The first statement within the public section of the class declaration is a constructor, which is a special function with the same name as its class that is called automatically when an object from a class is instantiated. The second statement within the public section of the class declaration is a destructor, which is another special function with the same name as its class, but preceded by a tilde (~), that is called automatically when an object from a class is destroyed. You will learn about constructors later in this chapter. You will learn about destructors in Chapter 7.

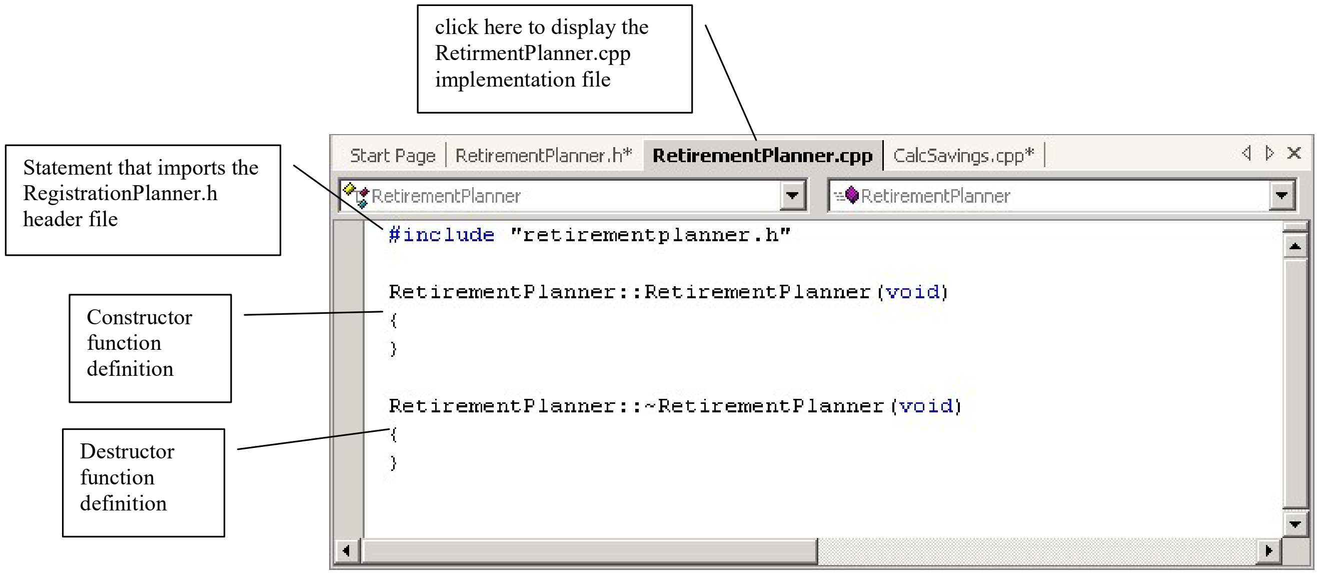
FIGURE 4-15: RetirementPlanner header file



```
#pragma once

class RetirementPlanner
{
public:
    RetirementPlanner(void);
    ~RetirementPlanner(void);
};
```

6. In the Code Editor window, click the RetirementPlanner.cpp tab. Your Code Editor window should appear the same as Figure 4-16. You can see that the Generic C++ Class Wizard automatically added the `#include "RetirementPlanner.h"` statement to import the RetirementPlanner.h header file. Also, notice that the wizard added empty definitions for the RetirementPlanner constructor and destructor functions. For now, do not worry about what the constructor and destructor functions do or how they are set up. Simply familiarize yourself with the code that the Generic C++ Class Wizard adds for you automatically. You will learn how to add member functions to a class shortly.

FIGURE 4-16: RetirementPlanner.cpp file in the Code Editor window



```
#include "retirementplanner.h"

RetirementPlanner::RetirementPlanner(void)
{
}

RetirementPlanner::~RetirementPlanner(void)
{
}
```

click here to display the RetirmentPlanner.cpp implementation file

Statement that imports the RegistrationPlanner.h header file

Constructor function definition

Destructor function definition

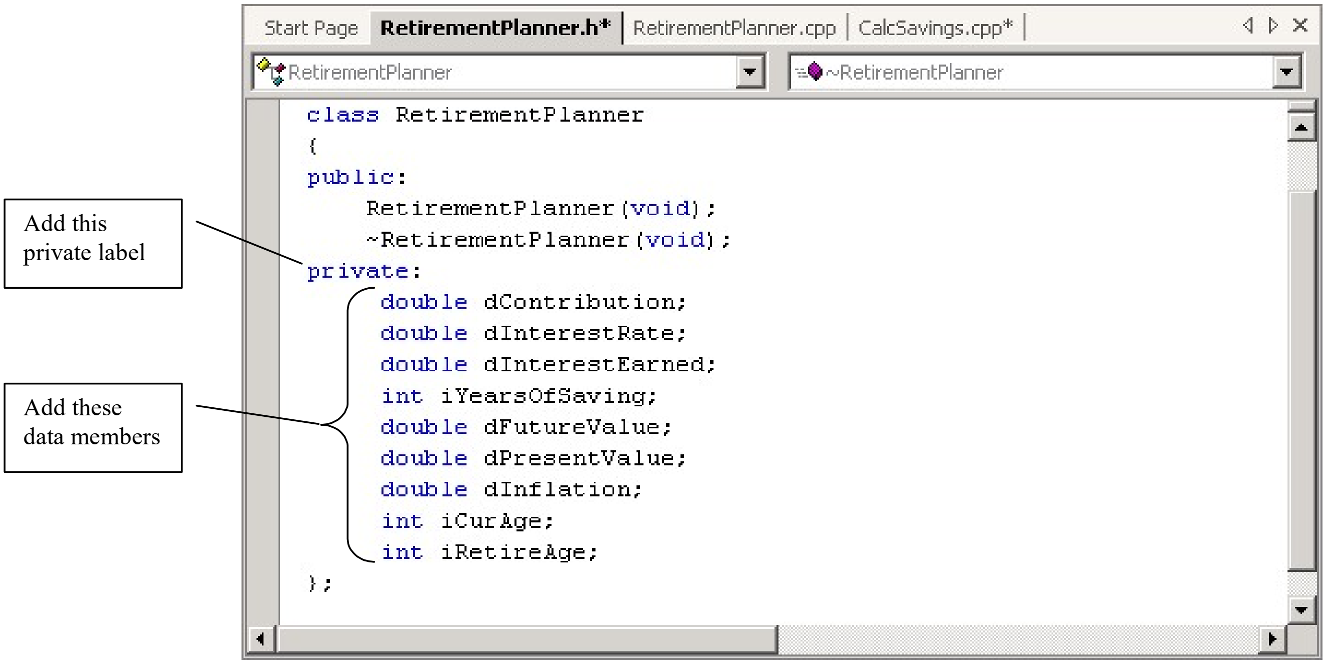
Next, you will add public and private labels to the RetirementPlanner.h file, along with some declarations for private data members.

To add private labels to the RetirementPlanner.h file, along with some declarations for private data members:

1. Return to the **RetirementPlanner.h** file in the Code editor window.

2. Add the private label and private data members to the RetirementPlanner.h file as shown in Figure 4-17. Later you will add member function declarations to the public section that will modify and retrieve the data members declared in the private section.

FIGURE 4-17: RetirementPlanner.h after adding private section and private data members



```
class RetirementPlanner
{
public:
    RetirementPlanner(void);
    ~RetirementPlanner(void);
private:
     double dContribution;
     double dInterestRate;
     double dInterestEarned;
     int iYearsOfSaving;
     double dFutureValue;
     double dPresentValue;
     double dInflation;
     int iCurAge;
     int iRetireAge;
};
```

Add this private label

Add these data members

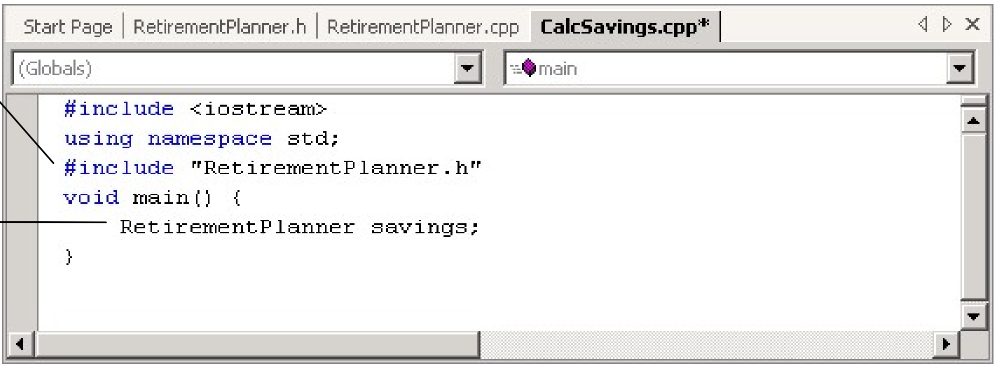
Next you will start creating the CalcSavings.cpp file, which will contain the program's `main()` function. In the `main()` function, you will declare a RetirementPlanner object and use that object to access data members and member functions in the RetirementPlanner class.

To start creating the CalcSavings.cpp file:

1. Add a new C++ source file named **CalcSavings.cpp** to the RetirementPlanner project.

2. Before you can instantiate an object of a class, you must first include the class's header file in your .cpp file, just as you would include any other header files you need in your program. With custom classes that you write yourself (as opposed to the runtime classes that are part of Visual C++), you must enclose the header file name within quotation marks instead of brackets, and include the file's .h extension. Type the statements shown in Figure 4-18 to include the iostream class and the RetirementPlanner class, along with the using directive that designates the std namespace.

FIGURE 4-18: CalcSavings.cpp file after adding header files, an `include` statement, and a `main()` function



```
#include <iostream>
using namespace std;
#include "RetirementPlanner.h"
void main() {
    RetirementPlanner savings;
}
```

Statement that includes the header file for the RetirementPlanner class

Statement that declares a RetirementPlanner object named savings

3. Also as shown in Figure 4-19, add a `main()` function that includes a single statement that declares a RetirementPlanner object named savings.

PREVENTING MULTIPLE INCLUSION

Larger class-based programs are sometimes composed of multiple interface and implementation files. With larger programs, you need to ensure that you do not include multiple instances of the same header file when you compile the program, because multiple inclusions will make your program unnecessarily large. Multiple inclusions of the same header usually occur when you include one header into a second header, and then include the second header in an implementation file.

Visual C++ generates an error if you attempt to compile a program that includes multiple instances of the same header. To prevent multiple inclusions prior to compilation, the Generic C++ Class Wizard adds the `#pragma once` statement to a class header file. A **pragma** is a special preprocessing directive that can execute a number of different compiler instructions. The **once pragma** instructs the compiler to include a header file only once, no matter how many times it encounters an #include statement for that header in other C++ files in the project. If you examine the RetirementPlanner.h header in your Code Editor window, you will see that the `#pragma once` statement is the first line of code in the file.

TIP: You can view a list of pragma directives that C++ supports in the Pragma Directives topic in the MSDN Library.

Pragmas are compiler specific; you will not find the same pragmas support in different C++ compilers. Visual C++, for instance, supports the pragma once directive. However, other C++ compilers consider the pragma once directive to be obsolete. To prevent multiple inclusion, these other compilers use the #define preprocessor directive with #if and #endif preprocessor directives in header files. You first learned how to use the #define preprocessor directive in Chapter 2 to define a constant. The **`#if`** and **`#endif` preprocessor directives** determine which portions of a file to compile depending on the result of a conditional expression. All statements located between the `#if` and `#endif` directives are compiled if the conditional expression evaluates to true. Each `#if` directive must include a closing `#endif` directive.

NOTE: The `#if` and `#endif` preprocessor directives are similar to the CODE]`if` statements you learned how to use in Chapter 3.

The following code shows the syntax for the `#if` and `#endif` preprocessor directives. Notice that unlike the conditional expression for standard `if` statements, the conditional expression for #if and #endif directives is not enclosed within parentheses.

```
#if conditional expression
     statements to compile;
#endif
```

To prevent multiple inclusions of header files, you use the #define directive to declare a constant representing a specific header file. Each time the compiler is asked to include that header file during the build process, it uses the defined constant expression with the #if directive to check if a specific header file's constant exists when you build a project. The **defined constant expression** returns a value of true if a particular identifier is defined or a value of false if it is not defined. The syntax for the defined constant expression is `#defined(`*`identifier`*`)`. To see if an identifier has *not* been defined, add the not operator (!) before the defined expression.

TIP: Common practice when defining a header file's constant is to use the header file's name in uppercase letters appended with _H. For example, the constant for the `stocks.h` header file is usually defined as `STOCKS_H`.

If a header file's constant has not been defined, statements between the `#if` and `#endif` directives define the constant, and any statements preceding the `#endif` directive are compiled. If a header file's constant has already been defined, however, all statements between the `#if` and `#endif` directives are skipped, preventing a multiple inclusion. Figure 4-19 shows how to add code to the header file that prevents multiple inclusions of the Stocks class.

FIGURE 4-19: Header file with preprocessor directives that prevent multiple inclusions

```
#if !defined(STOCKS_H)
#define STOCKS_H
class Stocks {
private:
     int iNumShares;
     double dPurchasePricePerShare;
     double dCurrentPricePerShare;
};
#endif
```

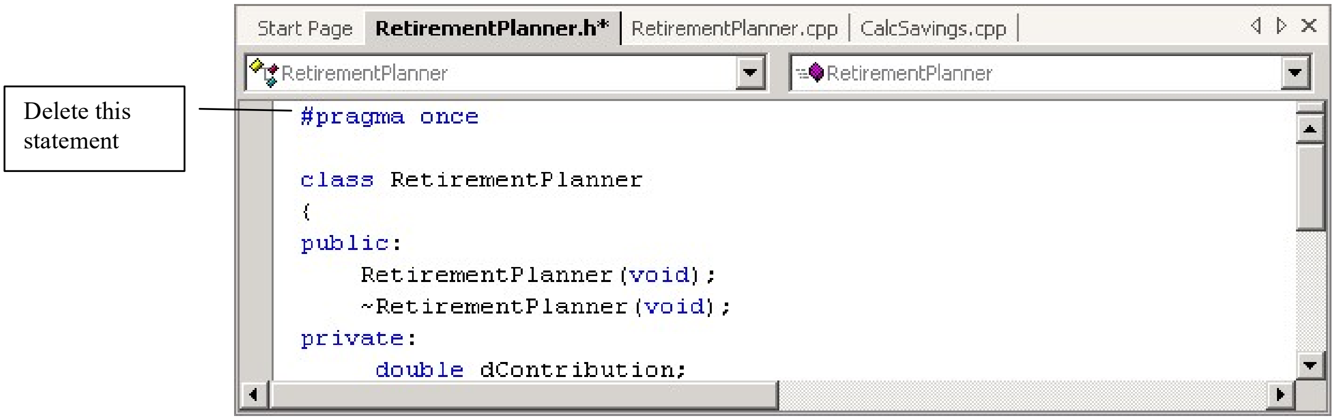
The once pragma is much easier to use than the #if and #endif directives. However, you should be familiar with how to use the #if and #endif directives to prevent multiple inclusion, especially if you ever work with a C++ compiler that does not support the once pragma. For practice you will replace the once pragma in the RetirementPlanner header file with #if and #endif directives to prevent multiple inclusion in the RetirementPlanner class.

To replace the once pragma in the RetirementPlanner header file with #if and #endif directives to prevent multiple inclusion in the RetirementPlanner class:

1. Return to the RetirementPlanner.h header file in the Code Editor window.

2. Delete the once pragma statement, as shown in Figure 4-20.

Figure 4-20: Delete the once pragma statement from the RetirementPlanner.h header file

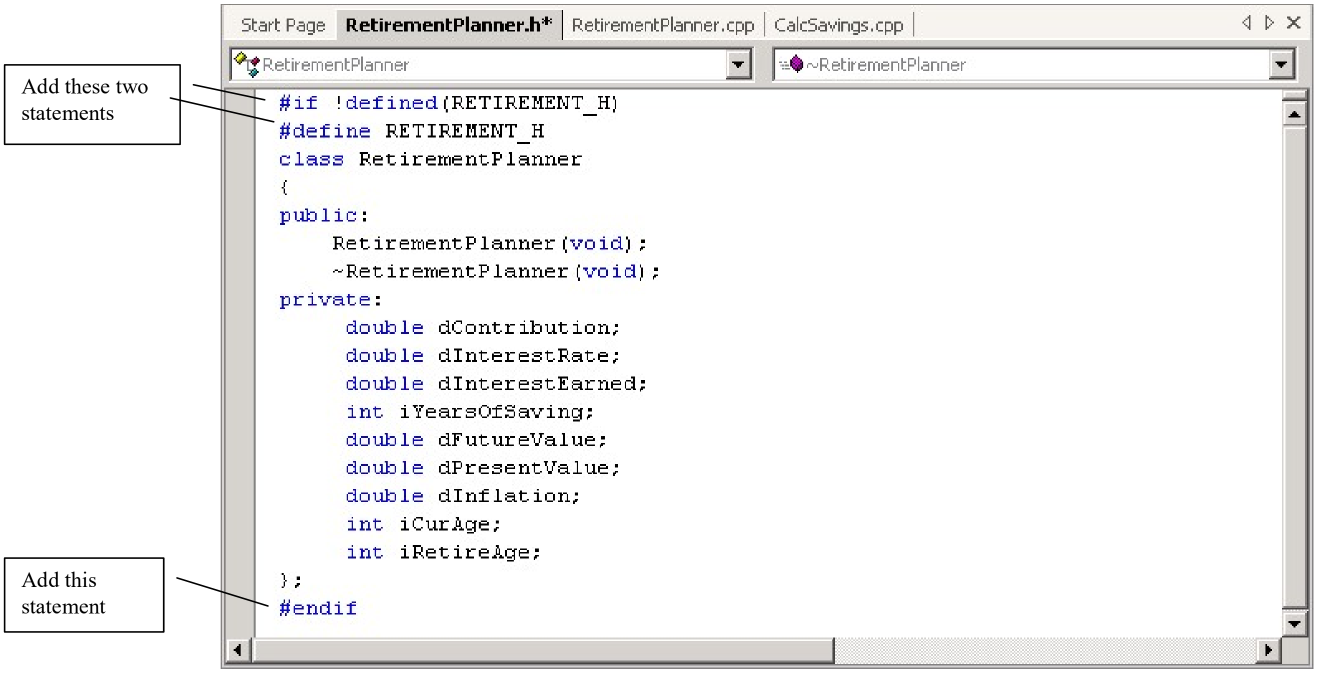


Delete this statement

```
#pragma once

class RetirementPlanner
{
public:
    RetirementPlanner(void);
    ~RetirementPlanner(void);
private:
     double dContribution;
```

2. Add the `#if` and `#endif` directives shown in Figure 4-21.

FIGURE 4-21: `#if` and `#endif` directives added to RetirementPlanner.h



Add these two statements

Add this statement

```
#if !defined(RETIREMENT_H)
#define RETIREMENT_H
class RetirementPlanner
{
public:
    RetirementPlanner(void);
    ~RetirementPlanner(void);
private:
     double dContribution;
     double dInterestRate;
     double dInterestEarned;
     int iYearsOfSaving;
     double dFutureValue;
     double dPresentValue;
     double dInflation;
     int iCurAge;
     int iRetireAge;
};
#endif
```

MEMBER FUNCTIONS

Because member functions perform most of the work in a class, you will learn about the various techniques associated with them. As you saw earlier, you declare functions in an interface file, but define them in an implementation file. Member functions are usually declared as public, but they can also be declared as private. Public member functions can be called by anyone, whereas private member functions can be called only by other member functions in the same class.

You may wonder what good a private function member would be because a client of the program cannot access a private function. Suppose your program needs some sort of utility function that clients have no need to access. For example, your program may need to determine an employee's income-tax bracket by calling a function named `calcTaxBracket()`. To use your program, the client does not need to access the `calcTaxBracket()` function. By making the `calcTaxBracket()` function private, you protect your program and add another level of information hiding.

In order for your class to identify which functions in an implementation file belong to it (as opposed to global function definitions), you precede the function name in the function definition header with the class name and the scope resolution operator (::). For example, to identify the `getTotalValue()` function in an implementation file as belonging to the Stocks class, the function definition header should read `double Stocks:: getTotalValue(int iShares, double dCurPrice)` . Figure 4-22 shows both the interface and implementation files for the Stocks class. The `getTotalValue()` function's prototype is declared in an interface file named `stocks.h`, whereas the `getTotalValue()` function definition is placed in an implementation file named stocks.cpp.

FIGURE 4-22: Stocks class interface and implementation files

```
// stocks.h
class Stocks {
public:
     double getTotalValue(int iShares, double dCurPrice);
private:
     int iNumShares;
     double dCurrentPricePerShare;
```

Interface file

```
    double dCurrentValue;
};
```

Implementation file

```
// stocks.cpp
#include "stocks.h"
#include <iostream>
using namespace std;
double Stocks::getTotalValue(int iShares, double dCurPrice){
    iNumShares = iShares;
    dCurrentPricePerShare = dCurPrice;
    dCurrentValue = iNumShares * dCurrentPricePerShare;
    return dCurrentValue;
}
void main() {
    Stocks stockPick;
    ...
}
```

Even though the member functions of a class may be defined in an implementation file separate from the interface file, as long as the functions include the class's name and the scope resolution operator, they are considered to be part of the class definition. Just think of the declarations and definitions that compose your class as being spread across multiple files.

Next, you will use the Add Member Function Wizard to add member function declarations and definitions to the RetirementPlanner class. The RetirementPlanner class uses five functions for setting the values of private data members: `setContribution()`, `setInterestRate()`, `setCurAge()`, `setRetireAge()`, and `setInflation()`. Two other functions, `calcFutureValue()` and `calcPresentValue()`, perform the actual calculations that give the Retirement Planner program its functionality. The `calcFutureValue()` function returns the future value of an investment based on the amount invested each year, the yearly annual interest on the investment, and the number of years spent saving for retirement. The number of years spent saving for retirement is calculated by subtracting the age you started saving from the age you retire. The `calcPresentValue()` function adjusts the future value of an investment for inflation. An accessor function, `getInterestEarned()`, returns the total amount of interest earned on retirement savings. The `getInterestEarned()` function is a typical *get* function that returns to the client the value of a private data member. Note that you will not learn how the functions perform the calculations because algebra is not the purpose of your studies. However, if you examine the formulas closely, you will see that they are structured using typical C++ operators.

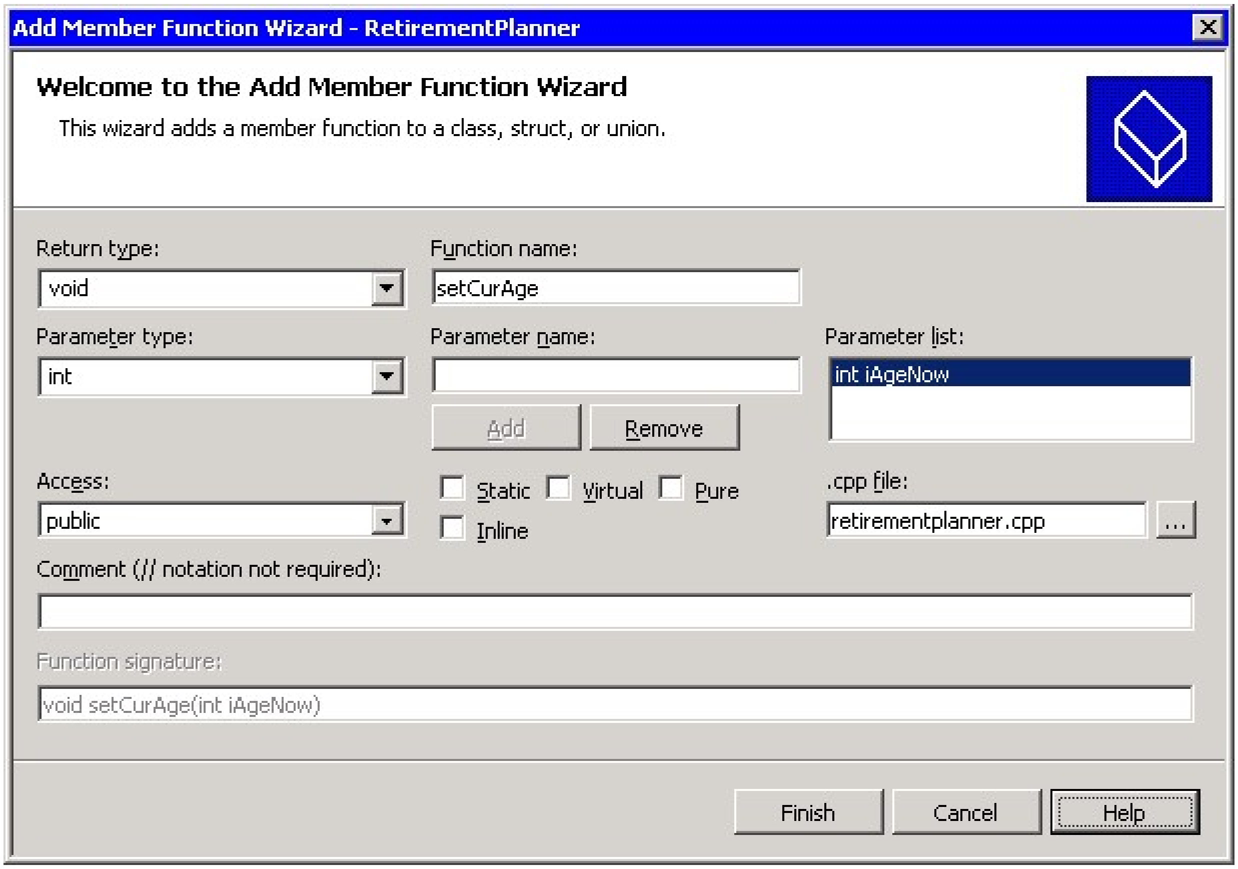
To use the Add Member Function Wizard to add member function declarations and definitions to the RetirementPlanner class:

1. Open Class View by selecting **Class View** from the **View** menu, or by pressing **Ctrl**+**Shift**+**C**.

2. First, you will add the setCurAge() function. Click the RetirementPlanner icon once and then select **Add Function** from the Project menu. Alternatively, you can right-click the **RetirementPlanner** icon and select **Add Function** from the **Add** submenu on the shortcut menu. The Add Member Function Wizard dialog box displays.

3. In the Add Member Function Wizard dialog box, select a return type of **void** from the Return type combo box and enter **setCurAge** (without the parentheses) in the Function name text box. Notice as you build the function that its declaration enters automatically into the Function signature text box at the bottom of the dialog box. The setCurAge() function includes a single int parameter named iAgeNow. To add this parameter, select **int** from the Parameter type combo box, type **iAgeNow** in the Parameter name text box, and then click the **Add** button. After you click the Add button, the new parameter appears in the Parameter list and in the Function signature text box. Leave the remainder of the dialog box options as they are. Because the iAgeNow function includes only a single parameter, you can click the **Finish** button. Figure 4-23 shows the Add Member Function Wizard dialog box as it should appear before you click the Finish button.

FIGURE 4-23: Adding the iAgeNow Function with the Add Member Function Wizard dialog box



Add Member Function Wizard - RetirementPlanner

Welcome to the Add Member Function Wizard

This wizard adds a member function to a class, struct, or union.

Return type: void

Function name: setCurAge

Parameter type: int

Parameter name:

Parameter list: int iAgeNow

Access: public

□ Static □ Virtual □ Pure

□ Inline

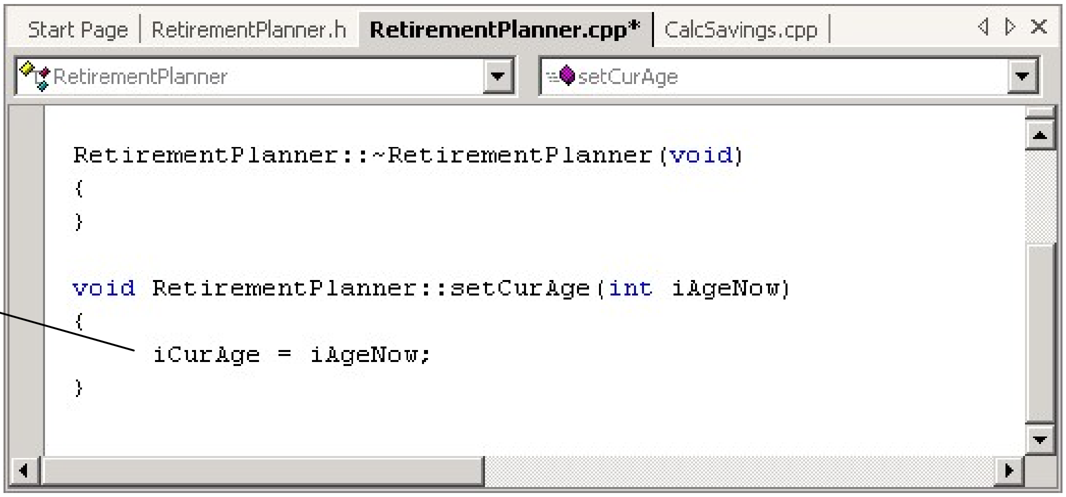
.cpp file: retirementplanner.cpp

Comment (// notation not required):

Function signature: void setCurAge(int iAgeNow)

4. After you click the Finish button, the Add Member Function Wizard creates the `setCurAge()` function declaration in the interface file (`RetirementPlanner.h`) and also creates its function definition in the implementation file (RetirementPlanner.cpp). The Code Editor window opens to the new empty `setCurAge()` function definition in the implementation file. Add the statement show in Figure 4-24 that assigns the value of the `iAgeNow` parameter to the iCurAge data member.

FIGURE 4-24: Assignment statement added to the `setCurAge()` member function



```
RetirementPlanner::~RetirementPlanner(void)
{
}

void RetirementPlanner::setCurAge(int iAgeNow)
{
    iCurAge = iAgeNow;
}
```

Add this statement

5. Use the Add Member Function Wizard to add the remainder of the set functions and `getInterestEarned()` member function, as follows:

```
void RetirementPlanner::setRetireAge(int iAgeThen) {
     iRetireAge = iAgeThen;
}

void RetirementPlanner::setContribution(double dContribute)
{
     dContribution = dContribute;
}
void RetirementPlanner::setInterestRate(double dInterest) {
     dInterestRate = dInterest;
}
void RetirementPlanner::setInflation(double dInflate) {
     dInflation = dInflate;
}
double RetirementPlanner::getInterestEarned() {
     return dInterestEarned;

}
```

6. Finally, use the Add Member Function Wizard to add the following `calcFutureValue()` and `calcPresentValue()` member functions:

```
// calcFutureValue() function
double RetirementPlanner::calcFutureValue() {
     iYearsOfSaving = iRetireAge - iCurAge;
     dFutureValue = 0;
     for (int i=0; i < iYearsOfSaving; i++) {
          dFutureValue += 1;
          dFutureValue *= (1+(dInterestRate/100));
}
     dFutureValue *= dContribution;
     dInterestEarned = dFutureValue -
          (dContribution * iYearsOfSaving);
     return dFutureValue;
}
// calcPresentValue() function
double RetirementPlanner::calcPresentValue() {
     double dFutureValue = calcFutureValue();
     for(int i = 0; i < iYearsOfSaving; i++) {
          dFutureValue /= (1 + (dInflation/100));
     }
     dPresentValue = dFutureValue;
     return dPresentValue;
}
```

Once you create a member function, you execute it from your implementation file by appending the function name to the to the object name with the member selection operator, in the same manner that you access data members. Unlike data members, however, you must also place a set of parentheses after the function name, containing any arguments required by the function. For example, the first statement in the following code declares a Stocks object named `stockPick`. The second statement executes the `getTotalValue()` function, passing to it the number of stocks and the current price per share.

```
void main() {
     Stocks stockPick;
     stockPick.getTotalValue(100, 10.875);
}
```

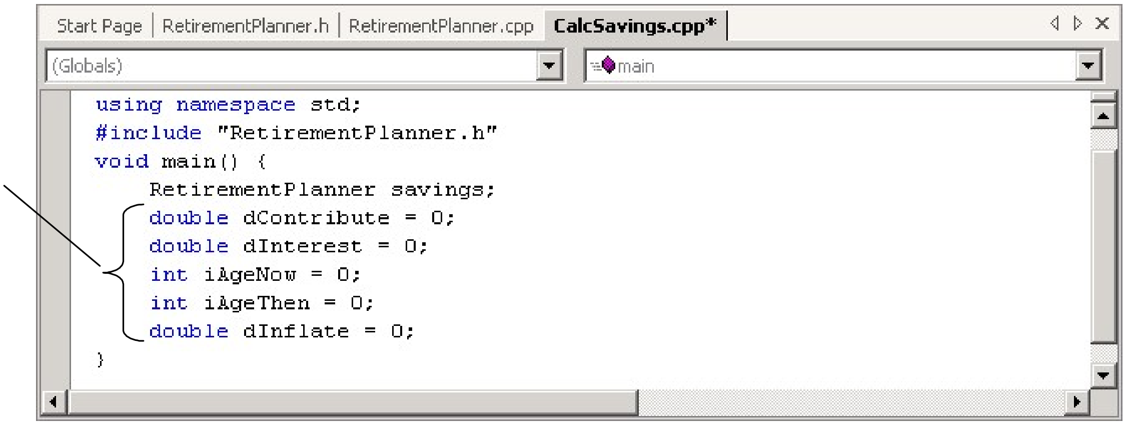
Next you will add code to the CalcSavings.cpp file that accesses the RetirementPlanner class's functions. You will use cout statements to display instructions to users and `cin` statements to gather data. You will assign the data returned from the `cin` statements to variables, which you will then pass to the RetirementPlanner class's function members. Finally, you will calculate and display the results using the `calcFutureValue()`, `calcPresentValue()`, and `getInterestEarned()` functions.

To add code to the CalcSavings.cpp file that accesses the RetirementPlanner class's functions:

1. Open the **CalcSavings.cpp** file in the Code Editor window.
2. Add the statements shown in Figure 4-25. The statements declare variables that you will use to hold the values retrieved from the user with the cin statements. The variables will then be passed to the RetirementPlanner class member functions.

FIGURE 4-25: Variable declarations added to CalcSavings.cpp

Add these variable declaration statements

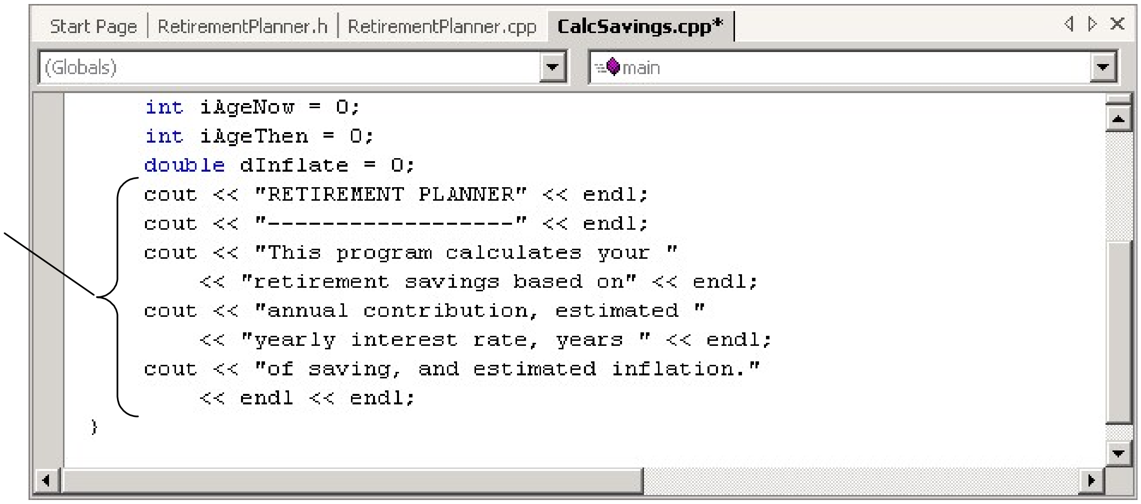


```
using namespace std;
#include "RetirementPlanner.h"
void main() {
    RetirementPlanner savings;
    double dContribute = 0;
    double dInterest = 0;
    int iAgeNow = 0;
    int iAgeThen = 0;
    double dInflate = 0;
}
```

3. Add the `cout` statements shown in Figure 4-26 that explain the program to the user:

FIGURE 4-26: `cout` statements added to CalcSavings.cpp that explain the program to the user

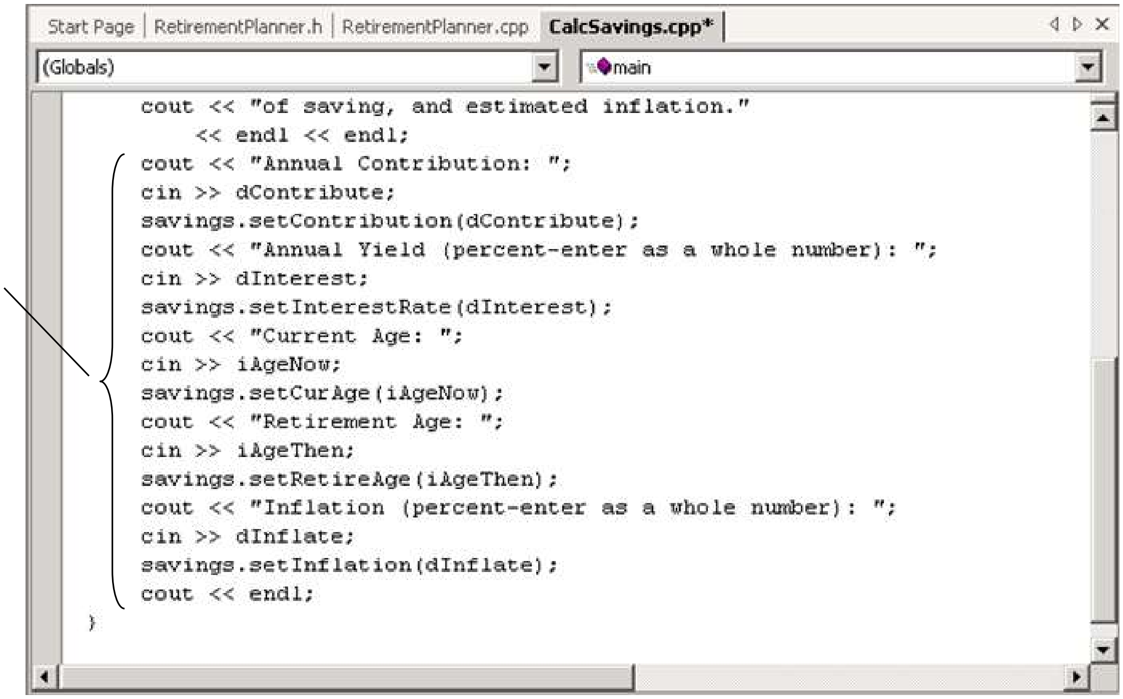
Add these cout statements



```
    int iAgeNow = 0;
    int iAgeThen = 0;
    double dInflate = 0;
    cout << "RETIREMENT PLANNER" << endl;
    cout << "------------------" << endl;
    cout << "This program calculates your "
        << "retirement savings based on" << endl;
    cout << "annual contribution, estimated "
        << "yearly interest rate, years " << endl;
    cout << "of saving, and estimated inflation."
        << endl << endl;
}
```

4. Add the statements shown in Figure 4-27 that gather values from the user, assign the values to variables, and then pass the variables to the member functions.

FIGURE 4-27: Input statements added to CalcSavings.cpp

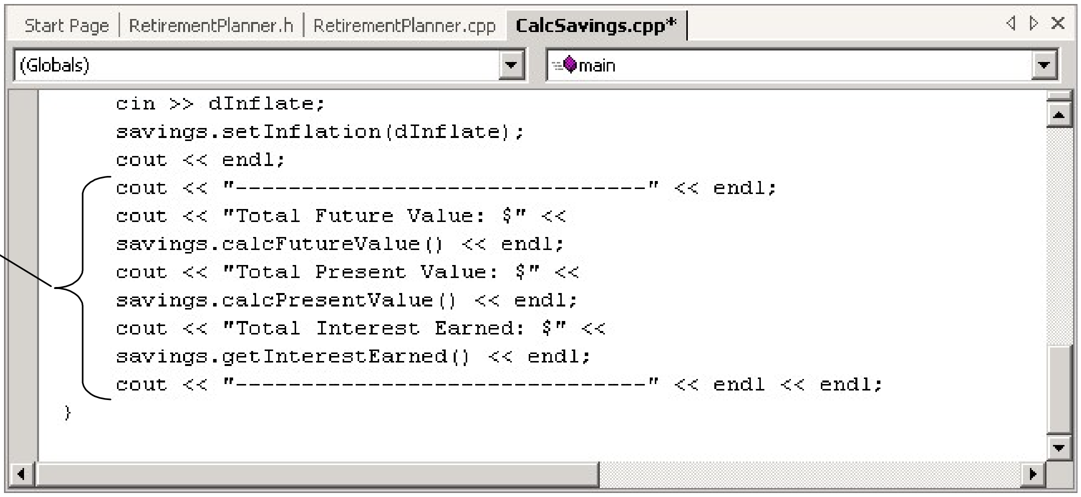


```
    cout << "of saving, and estimated inflation."
        << endl << endl;
    cout << "Annual Contribution: ";
    cin >> dContribute;
    savings.setContribution(dContribute);
    cout << "Annual Yield (percent-enter as a whole number): ";
    cin >> dInterest;
    savings.setInterestRate(dInterest);
    cout << "Current Age: ";
    cin >> iAgeNow;
    savings.setCurAge(iAgeNow);
    cout << "Retirement Age: ";
    cin >> iAgeThen;
    savings.setRetireAge(iAgeThen);
    cout << "Inflation (percent-enter as a whole number): ";
    cin >> dInflate;
    savings.setInflation(dInflate);
    cout << endl;
}
```

Add these statements that gather values from the user, assign the values to variables, and then pass the variables to the member functions

5. Finally, add the statements shown in Figure 4-28 that display the calculated savings results to the user:

FIGURE 4-28: Statements added to CalcSavings.cpp that display the calculated savings results to the user



```
    cin >> dInflate;
    savings.setInflation(dInflate);
    cout << endl;
    cout << "--------------------------------" << endl;
    cout << "Total Future Value: $" <<
    savings.calcFutureValue() << endl;
    cout << "Total Present Value: $" <<
    savings.calcPresentValue() << endl;
    cout << "Total Interest Earned: $" <<
    savings.getInterestEarned() << endl;
    cout << "--------------------------------" << endl << endl;
}
```

Add these statements that display the calculated savings results to the user

6. Build and execute the RetirementPlanner project. Then, test the program. When you enter percentages for either Annual Yield or Inflation, enter the numbers as whole numbers, not with a decimal point. For example, to enter 10%, type *10*, not *.10*.

Inline Functions

Although member functions are usually defined in an implementation file, they can also be defined in an interface file. Functions defined inside the class body in an interface file are called **inline functions**. To conform to information hiding techniques, only the shortest function definitions, such as accessor functions, should be added to the interface file. The following code shows an example of a public inline function definition in the Stocks class for a function member named `getTotalValue()`. The `getTotalValue()` function accepts two arguments from the client: the number of shares and the current price. The arguments are assigned to private data members, and then the price is calculated and returned.

Inline function

```
class Stocks {
public:
    double getTotalValue(int iShares, double dCurPrice){
        iNumShares = iShares;
        dCurrentPricePerShare = dCurPrice;
        dCurrentValue = iNumShares * dCurrentPricePerShare;
        return dCurrentValue;
    }
private:
    int iNumShares;
    double dCurrentPricePerShare;
    double dCurrentValue;
};
```

For functions that are not defined inside the class body, you can place the [BEGIN CODE]inline[END CODE] keyword at the start of a function header. An important point to remember is that you must place the function definition for a function declared with the `inline` keyword in the interface file—not the implementation file. For example, if the `getTotalValue()` function is defined within the Stocks interface file, but outside of the class body, you can mark it as an inline function using the statement `inline double getTotalValue(int iShares, double dCurPrice);`.

CATION: If you add the `inline` keyword to a function that is not declared within a class header file, you will receive a compile error.

When the compiler encounters either an inline function or a function declared with the `inline` keyword, it performs a cost/benefit analysis to determine whether replacing a function call with its function definition will increase the program's speed and performance. If the compiler's cost/benefit analysis determines that there will be no significant speed or performance gain by replacing a given function call with its function definition, then the inline function is executed as a normal function.

TIP: Because they are declared in the interface file, inline functions do not take advantage of information hiding. If you want to hide a function definition, be sure not to define it as an inline function.

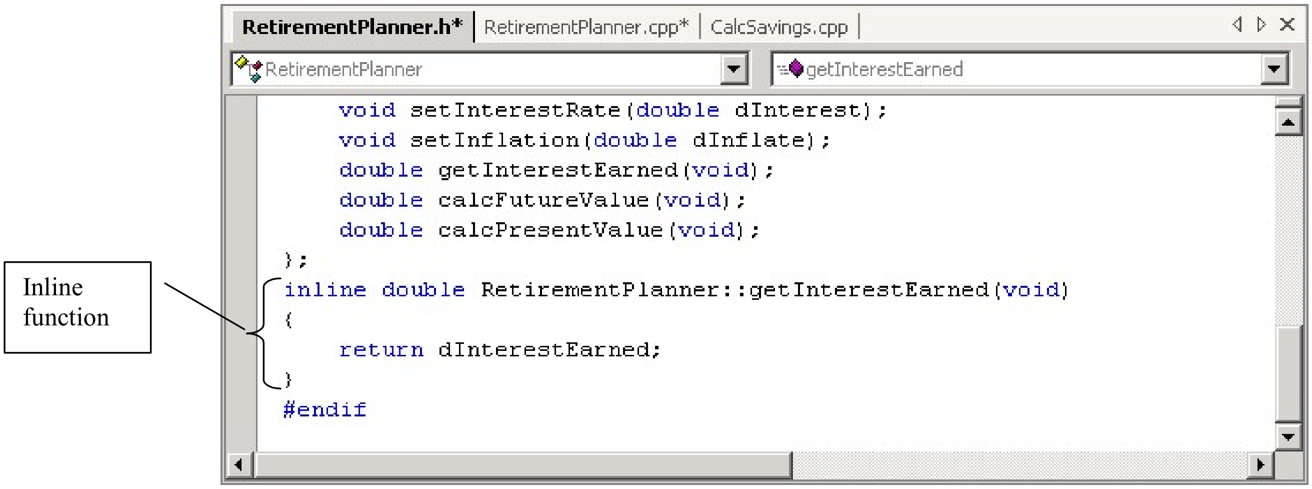
TIP: You can also use the `inline` keyword with global functions. As with member functions, however, the global functions you define as inline should be relatively small.

Next you will add the `inline` keyword to the `getInterestEarned()` function, which is small and stable enough that it can be defined as inline. You will also move the `getInterestEarned()` function definition to the header, or interface, file so that the program compiles correctly.

To define the getInterestEarned() function as inline:

1. Open the **RetirementPlanner.cpp** file in the Code Editor window.
2. Highlight the **getInterestEarned()** function definition and cut it to the Clipboard by selecting **Cut** from the **Edit** menu.
3. Open the **RetirementPlanner.h** file in the Code Editor window.
4. Paste the getInterestEarned() function definition after the class's closing brace and semicolon but above the #endif directive by selecting **Paste** from the **Edit** menu.
5. Modify the getInterestEarned() function definition so that it reads **inline** double RetirementPlanner:: getInterestEarned(). Your modified RetirementPlanner header file should look like Figure 4-29.

FIGURE 4-29: Inline function added to RetirementPlanner header



```
    void setInterestRate(double dInterest);
    void setInflation(double dInflate);
    double getInterestEarned(void);
    double calcFutureValue(void);
    double calcPresentValue(void);
};
inline double RetirementPlanner::getInterestEarned(void)
{
    return dInterestEarned;
}
#endif
```

6. Rebuild and execute the RetirementPlanner project. The program should function the same as it did before you added the inline keyword to the getInterestEarned() function.

Constructor Functions

When you first instantiate an object from a class, you will often want to assign initial values to data members or perform other types of initialization tasks, such as calling a function member that may calculate and assign values to data members. In a C++ program that does not use classes, you simply assign an initial value to a variable in the `main()` function using a statement such as `int iCount = 1;`, or you call a custom function using a statement such as [`double dInterest = calcInterest();`.

Although classes are "mini-programs," they do not include a `main()` function in which you can assign initial values to data members or call initialization functions. Instead, you use a constructor function. A **constructor function** is a special function with the same name as its class that is called automatically when an object from a class is instantiated. You define and declare constructor functions the same way you define other functions, although you do not include a return type because constructor functions do not return values. For example, the following inline constructor function for the Stocks class initializes the `iNumShares`, `dCurrentPricePerShare`, and `dCurrentValue` data members to zero:

Constructor function

```
class Stocks {
public:
  Stocks() {
        iNumShares = 0;
        dCurrentPricePerShare = 0;
        dCurrentValue = 0;
};
private:
     int iNumShares;
```

```
          double dCurrentPricePerShare;
          double dCurrentValue;
     };
```

You can also include just a function prototype in the interface file for the constructor function, and then create the function definition in the implementation file. The following code shows an example of how you implement the Stocks constructor function in an implementation file. It may look unusual, but when you define a constructor function in an implementation file, be sure to include the class name and scope resolution operator in order to identify the function as a class member.

```
Stocks::Stocks() {
  iNumShares = 0;
  dCurrentPricePerShare = 0;
  dCurrentValue = 0;
};
```

[NOTE] In Chapter 6 you will learn about some advanced constructor techniques, as well as how to use the opposite of a constructor, a *destructor*.

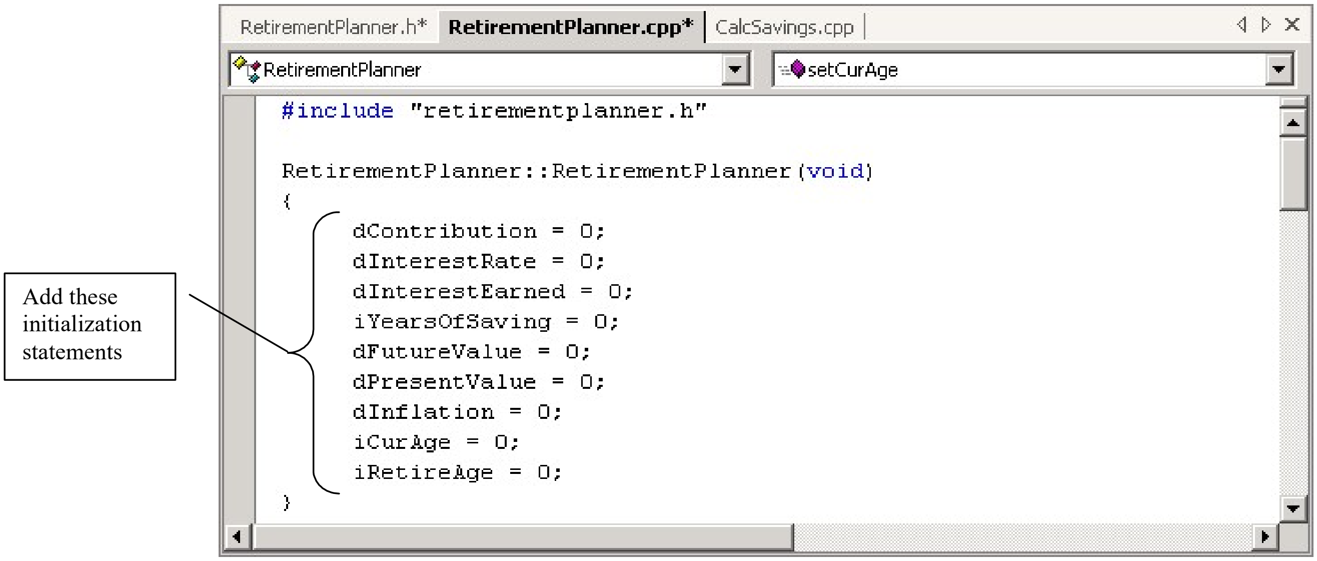
As you saw earlier, the Add Class Wizard automatically added an empty constructor function to the RetirementPlanner class for you. Next, you will modify the constructor so it initializes all of the private data members to 0. Initializing private data members to 0 ensures that the calculations within the member functions have a value to work with in the event that a client fails to provide one of the values when executing any of the member functions.

To add a constructor to the RetirementPlanner class:

1. Return to the **RetirementPlanner.cpp** file in the Code Editor window.

2. Modify the constructor function definition so it initializes the data members to 0, as shown in Figure 4-30:

FIGURE 4-30: Constructor function definition modified so it initializes the data members to 0



```
#include "retirementplanner.h"

RetirementPlanner::RetirementPlanner(void)
{
        dContribution = 0;
        dInterestRate = 0;
        dInterestEarned = 0;
        iYearsOfSaving = 0;
        dFutureValue = 0;
        dPresentValue = 0;
        dInflation = 0;
        iCurAge = 0;
        iRetireAge = 0;
}
```

Add these initialization statements

CAUTION: Be sure not to confuse the constructor function definition with the destructor function definition, which starts with a tilde (~).

5. Rebuild and execute the RetirementPlanner project. The program should work the same as it did before you added the constructor function.

friend **Functions and Classes**

When you use the public access modifier with a class member, the entire world has access to that class member. In contrast, only members of the same class can access private class members. What if you want to selectively allow access to class members, yet still maintain a level of information hiding? For example, another programmer in your department may have written a function in a separate program that needs to perform some calculations on your class's private data members. You could make the data members public (which removes information hiding), add the other programmer's function to your class (which may not make sense if the function is not useful to your class), or force the other programmer to use get methods to access the private data members (which could slow down his or her program). It would be easier to grant access to a class's private members only to a specific function or class. In these situations, the friend access modifier comes into play. The **friend** access modifier allows designated functions or classes to access a class's private members. Only a class itself can designate the function and class friends that can access its private members; external functions and classes cannot make themselves friends of a class. In other words, your class has to give friend access to external functions and classes. You declare a friend function by including the function's prototype in an interface file, preceded by the keyword `friend`.

TIP: You can place a `friend` declaration anywhere inside the class, except within a function definition. You can even place a friend declaration within the declarations for public and private class members. It is good practice, however, to keep all definitions for a specific type of access modifier together.

Here is a simple example that demonstrates how to declare a friend function. Assume that a computer manufacturer has a program with a class named Inventory that contains private data members that keep track of the company's inventory, along with accessor functions for each data member. (In a real-life application, the Inventory program would store inventory information in a database.) The Inventory class also contains a constructor that assigns some arbitrary values to each of the data members. (Again, in a real-life program, the number of units available for each item would normally be stored in and retrieved from a database.) You may also have an external function named `checkInventory()` that checks the number of units available for each item. The `checkInventory()` function compares the number of items available for each unit to a corresponding parameter that represents the number of items requested by a client. The function then returns a Boolean value indicating whether or not there is sufficient inventory to fulfill the order. Figure 4-33 shows the Inventory interface file and its implementation file. The class also includes a declaration for the `checkInventory()` function as a friend function. Although the `checkInventory()` function is defined in the class’s implementation file, it is not part of the class because its header declaration does not include the name of the class and the scope resolution operator. The implementation file also declares a global Inventory variable named `curOrder` and a `main()` function, which, again, is not part of the class itself. The `main()` function calls the `checkInventory()` member function, passing to it arguments representing the number of items ordered by the customer. If the `checkInventory()` function in Figure 4-31 were not declared as a friend function, the statements would be illegal because they directly access the private data members of the Inventory class. Figure 4-32 shows the program’s output.

FIGURE 4-31: Inventory class declaring and defining a friend function

```
// inventory.h
class Inventory {

private:
```

Friend function

```
  int iDesktopComputers;
  int iNotebookComputers;
  int iLaserPrinters;
  friend bool checkInventory(int, int, int);
public:
  Inventory();
  void setDesktopComputers(int);
  int getDesktopComputers();
  void setNotebookComputers(int);
  int getNotebookComputers();
  void setPrinters(int);
  int getPrinters();
};
// inventory.cpp
#include "inventory.h"
#include <iostream>
using namespace std;
Inventory::Inventory() {
  iDesktopComputers = 250;
  iNotebookComputers = 150;
  iLaserPrinters = 125;

}
void Inventory::setDesktopComputers(int iUnits) {
  iDesktopComputers = iUnits;
}
int Inventory::getDesktopComputers() {
  return iDesktopComputers;
}
```

```
void Inventory::setNotebookComputers(int iUnits) {
  iNotebookComputers = iUnits;
}
int Inventory::getNotebookComputers() {
  return iNotebookComputers;
}
void Inventory::setPrinters(int iUnits) {
  iLaserPrinters = iUnits;
}
int Inventory::getPrinters() {
  return iLaserPrinters;
}
bool checkInventory(int, int , int , int);
Inventory curOrder;
void main() {
  bool bRetValue = checkInventory(50, 25, 10);
  if (bRetValue == true)
        cout << "We can fill the order." << endl;
  else
        cout << "We cannot fill the order." << endl;
}
bool checkInventory(int iDesktops, int iNotebooks,
                          int iPrinters) {
  bool bFillOrder = true;
  if (curOrder.iDesktopComputers < iDesktops)
        bFillOrder = false;
  else if (curOrder.iNotebookComputers < iNotebooks)
        bFillOrder = false;
  else if (curOrder.iLaserPrinters < iPrinters)
```

```
            bFillOrder = false;
      return bFillOrder;
    }
```

FIGURE 4-32: Output of Inventory program

The `checkInventory()` friend function is part of the Inventory implementation file in Figure 4-33 for simplicity. However, in reality the `checkInventory()` friend function would probably be part of another class or program. To designate all functions within another class as friends of the current class, you create a declaration in the current class using the syntax `friend class` *`name;`*, replacing *name* with the name of the class containing the functions you want to mark as friends. For example, assume that the `checkInventory()` function is really a member of a class named Fulfillment. To allow all the functions in a Fulfillment class to access the private members in the Inventory class, you would add the declaration friend class Fulfillment; to the Inventory class's interface file as follows.

```
class Inventory {
private:
  int iDesktopComputers;
  int iNotebookComputers;
  int iLaserPrinters;
  bool checkInventory(int, int, int);
public:
  Inventory();
  void setDesktopComputers(int);
  int getDesktopComputers();
  void setNotebookComputers(int);
  int getNotebookComputers();
  void setPrinters(int);
  int getPrinters();
```

Declaring a friend class

```
    friend class Fulfullment;
};
```

CHAPTER SUMMARY

- In C++ programming, classes are structures that contain variables along with functions for manipulating that data.
- The functions and variables defined in a class are called class members.
- Class variables are referred to as data members or member variables, whereas class functions are referred to as member functions or function members.
- Functions that are not part of a class are referred to as global functions.
- When you declare an object from a class, you are said to be instantiating an object.
- Classes are referred to as user-defined data types or programmer-defined data types because you can work with a class as a single unit, or object, in the same way you work with variables.
- A structure, or struct, is an advanced, user-defined data type that uses a single variable name to store multiple pieces of related information.
- The individual pieces of information stored in a structure are referred to as elements, fields, or members.
- When you use a period to access an object’s members, such as a structure’s fields, the period is referred to as the member selection operator.
- An initializer list is a series of values that are assigned to an object at declaration.
- Most C++ programmers use the `class` keyword to clearly designate the programs they write as object-oriented C++ programs.

- The principal of information hiding states that any class members that other programmers, or clients, do not need to access or know about should be hidden.
- Access specifiers control a client’s access to data members and member functions. There are four levels of access specifiers: `public`, `private`, `protected`, and `friend`.
- The `public` access specifier allows anyone to call a class’s function member or to modify a data member.
- The `private` access specifier is one of the key elements in information hiding because it prevents clients from calling member functions or accessing data members.
- Both `public` and `private` access specifiers have what is called class scope in that class members of both access types are accessible by any of a class’s member functions.
- You place access specifiers in a class definition on a single line followed by a colon, similar to a switch statement’s case labels.
- An access specifier that is placed on a line by itself followed by a colon is referred to as an access label.
- Many programmers prefer to make all of their data members private in order to prevent clients from accidentally assigning the wrong value to a variable or from viewing the internal workings of their programs.
- Accessor functions are public member functions that a client can call to retrieve or modify the value of a data member.

- The separation of classes into separate interface and implementation files is considered to be a fundamental software development technique because it allows you to hide the details of how your classes are written and makes it easier to modify programs.
- The interface refers to the data member and function member declarations inside a class's braces.
- The implementation refers to a class's function definitions and any code that assigns values to a class's data members.
- The Navigation bar at the top of the Code Editor window contains two combo boxes, Types and Members, that you can use to navigate to a particular class or its members.
- The Class View window displays project files according to their classes.
- Code wizards automate the task of adding specific types of code to your projects.
- The Add Class wizard adds a new class to a project.
- The Add Function wizard adds a new member function to a class
- Multiple inclusions of the same header usually occur when you include one header into a second header, and then include the second header in an implementation file.
- To prevent multiple inclusions prior to compilation, the Generic C++ Class Wizard adds the `#pragma once` statement to a class header file.
- A pragma is a special preprocessing directive that can execute a number of different compiler instructions.
- The once pragma instructs the compiler to include a header file only once, no matter how many times it encounters an #include statement for that header in other C++ files in the project.

- The `#if` and `#endif` preprocessor directives determine which portions of a file to compile depending on the result of a conditional expression.

- The defined constant expression returns a value of true if a particular identifier is defined or a value of false if it is not defined.

- You can create member function definitions in either the interface file or the implementation file.

- Even though the member functions of a class may be defined in separate files from the class declarations, as long as the function includes the class's name and the scope resolution operator, then it is considered to be part of the class definition.

- Member function definitions in an interface file are referred to as inline functions.

- For small functions, you can use the `inline` keyword to request that the compiler replace calls to a function with the function definition wherever the function is called in a program.

- A constructor function is a special function with the same name as its class that is called automatically when an object from a class is instantiated.

- The `friend` access modifier allows designated functions or classes to access a class's hidden members.

REVIEW QUESTIONS

1. Which of the following terms refers to class functions?

   a. member functions

   b. global functions

   c. user-defined data types

d. programmer-defined data types

2. The term *object* is used interchangeably with the word(s) .

a. *function*

b. *variable*

c. *statement*

d. *data type*

3. You define a structure using the __________ keyword.

a. `structure`

b. `struct`

c. `record`

d. `data`

4. Which statement best describes how you store data types in a structure?

a. All variables in a structure must be of the same data type.

b. You must use all numeric data types or all character data types.

c. You can use any mix of data types.

d. Structures do not directly declare data types, only variable names that are later assigned a specific data type.

5. Which of the following is the correct syntax for declaring a variable named `accountingInfo` based on a structure named accounting?

a. `accounting accountingInfo;`

b. `currentEmployee accountingInfo;`

c. `accountingInfo currentEmployee();`

d. `accountingInfo = new currentEmployee();`

6. When you use a period to access an object's members, such as a structure's fields, the period is referred to as the __________.

   a. object selector

   b. member selection operator

   c. field indicator

   d. structure operand

7. Examine the following structure declaration and determine why it will cause a compiler error:

```
struct accounting {
     int iPeriod = 2;
     long lFiscalYear = 2000;
     double dCurTaxRate = .15;
};
```

   a. The name of the structure must be followed by parentheses, the same as a function definition.

   b. You are only allowed to use a single data type within a structure definition.

   c. You are not allowed to assign values to the fields inside the structure definition itself.

   d. The structure must be declared using the `structure` keyword.

8. What is the accessibility of the data members in the following class?

```
class Boat {
       int iLength;
       double dEngineSize;
       char cClass;
}
```

a. `public`

b. `private`

c. `friend`

d. `protected`

9. What is the accessibility of the data members in the following structure?

```
struct Boat {
      int iLength;
      double dEngineSize;
      char cClass;
}
```

a. `public`

b. `private`

c. `friend`

d. `protected`

10. Which of the following elements is not considered part of a class’s scope?

a. implementation files

b. interface files

c. constructor functions

d. a main() method

11. The _____ window displays project files according to their classes and is similar to the Solution Explorer window.

a. Solution Explorer

b. Class View

c. Properties

d. Implementation

12. Which of the following code wizards is not designed specifically for working with classes?

a. Add New Item

b. Add Class

c. Add Member Function

d. Add Member Variable

13. When you use a code wizard to add a class to your project, which of the following statements is automatically added to prevent multiple header file inclusion?

a. `#pragma once`

b. `#pragma twice`

c. `#inclusion false`

d. `#ifincludeonce`

14. Which of the following preprocessor directives is used for preventing multiple header file inclusion?

a. `#multiple`

b. `#define...#!define`

c. `#include...#stop_include`

d. `#if` and `#endif`

15. Member functions that are defined within an interface file are referred to as __________ functions.

    a. member

    b. inline

    c. embedded

    d. compiled

16. Which of the following keywords forces the compiler to replace calls to a function with the function definition wherever the function is called in a program?

    a. `include`

    b. `inline`

    c. `replace`

    d. `insert`

17. Which statement is the correct definition in an implementation file for the constructor function for a class named Boat, assuming the construct function does not accept any arguments?

    a. `Boat () {`

    b. `Boat:Boat () {`

    c. `function Boat::Boat () {`

    d. `class Boat::Boat () {`

18. The __________ access modifier allows designated functions or classes to access a class's hidden members.

    a. `public`

    b. `private`

c. `friend`

d. `protected`

PROGRAMMING EXERCISES

1. What is the difference between a class and a structure? How do the two class types differ between C and C++?

2. Rewrite the following structure as a class. Be sure to assign the same access to the data members as they have in the structure.

```
struct CourseInfo {
     double dTuition;
     int iCourseID;
     char cGrade;
}
```

3. To the `main()` function in the following code, add cout statements that print each of the `carInfo` object's data members.

```
struct Transportation {
     double dCarEngineSize;
     int iMotorcycleCCs;
     int iSemiNumberofAxels;
}
void main() {
     Transportation vehicleInfo;
     vehicleInfo.dCarEngineSize = 3.1;
     vehicleInfo.iMotorcycleCCs = 750;
     vehicleInfo.iSemiNumberofAxels = 6;
}
```

4. What is the difference between a class's interface file and its implementation file?

5. Recall that accessor functions, which assign values to and retrieve values from private data members, are often referred to as get and set functions. Write the appropriate implementation file for the following class declaration and create `get` and `set` functions so that they assign values to and retrieve values from the private data members.

```
#if !defined(MUTUALFUND_H)
#define MUTUALFUND_H
class MutualFund {

public:

     void setNumberOfShares(int iShares);

     void setAnnualYield(int iYield);

     int getNumberOfShares();

     double getAnnualYield();

private:

     int iNumberOfShares;

     double dAnnualYield;

}

#endif
```

6. Replace the code in the `MutualFund` interface file that prevents multiple inclusion with the correct pragma directive:

7. Write a `main()` function that sets, retrieves, and prints the values of the private data members in the `MutualFund` class.

8. Add a friend function to the `MutualFund` class. Design the friend function so that it sets, retrieves, and prints all of the `MutualFund` class's private data members.

9. Write the appropriate interface file for the following class implementation.

```
#include "DistanceConversion.h"

DistanceConversion:: DistanceConversion() {

     dMiles = 0;

     dKilometers = 0;

}

double DistanceConversion::milesToKilometers(

      double dMilesArg) {

     dMiles = dMilesArg;

     dKilometers = dMiles * 1.6;

     return dKilometers;

}

double DistanceConversion::kilometersToMiles(

     double dKiloArg) {

     dKilometers = dKiloArg;

     dMiles = dKilometers * .6;

     return dMiles;

}
```

10. Find two ways to modify the `DistanceConversion` class so that the member function is compiled inline. Create separate versions of the class for each of your solutions.

PROGRAMMING PROJECTS

1. Create a Movies class that determines the cost of a ticket to a cinema, based on the moviegoer's age. Assume that the cost of a full-price ticket is $10. Gather the user's age using a `cin` statement, and then assign the age to a private data member. Next, use a public member function to determine the ticket price, based on the following schedule:

Age	Price
under 5	free
5 to 17	half price
18 to 55	full price
over 55	$2 off

After you determine the ticket price, print the cost to the screen.

2. Create a BaseballTeam class with appropriate data members such as team name, games won, games lost, and so on. Write appropriate `get` and `set` functions for each data member. Instantiate a number of `BaseballTeam` objects and assign appropriate values to each private data member using the `set` functions. Finally, use the `get` statements to retrieve and print the values in each private data member.

3. A painting company estimates the cost of its jobs based on materials and labor costs. The cost of materials is .15 cents per square foot while the cost of labor is .25 cents per square foot. Write a Painting class that determines the cost of painting a house, by allowing a prospective customer to enter the estimated number of feet they need painted. Store the number of feet in a private data member, along with `get` and `set` functions for setting and retrieving the value of the private data member. Use a separate member function for determining the cost of materials and the cost of labor. Also, include a function that calls the member functions for the materials and labor costs in order to calculate the total cost of the job. Store the total estimate in a private data member and print the estimate to the screen.

4. Create an Automobile class. Include private data members such as `make`, `model`, `color`, and `engine`, along with the appropriate `get` and `set` functions for setting and retrieving private data members. Use `cin` and `cout` statements to gather and display information.

5. Create a class-based temperature conversion program that converts Fahrenheit to Celsius and Celsius to Fahrenheit. To convert Fahrenheit to Celsius subtract 32 from the Fahrenheit temperature, then multiply the remainder by .55. To convert Celsius to Fahrenheit, multiply the Celsius temperature by 1.8, then add 32. Use `cin` and `cout` statements to gather and display information.

6. A passenger train averages a speed of 50 mph. However, each stop of the train adds an additional five minutes to the trains schedule. Additionally, during bad weather the train can only average a speed of 40 mph. Write a Train class that allows a traveler to calculate how long it will take to reach their destination, based on speed, number of stops, and weather conditions. Use `cin` and `cout` statements to gather and display information. Save each piece of information you gather from the user in a private data member, and write the appropriate get and set functions for setting and retrieving each data member. Use a single `inline` function to calculate how long the traveler's trip will take and save the result in another private data member, and print the results to the screen.

7. Create a CompanyInfo class that includes private data members such as the company name, year incorporated, annual gross revenue, annual net revenue, and so on. Write `set` and `get` functions to store and retrieve values in the private data members. Also, create a `friend` function that calculates the company’s operating costs by subtracting net revenue from gross revenue. Use `cin` and `cout` statements to gather and display information.

8. Create a Change class that calculates the correct amount of change to return when perform a cash transaction. Allow the user (a cashier) to enter the cost of a transaction and the exact amount of money that the customer hands over to pay for the transaction. Use `set` and `get` functions to store and retrieve both amounts to and from private data members. Then use member functions to determine the largest amount of each denomination to return to the customer. Assume that the largest denomination a customer will give you is a $100 bill. Therefore, you will need to write member functions for $50, $20, $10, $5, and $1 bills, along with quarters, dimes, nickels and pennies. For example, if the price of a transaction is $5.65 and the customer hands the cashier $10, the cashier should return $4.35 to the customer. Print your results to the screen using `cout` statements.

9. Create a BankAccount class that allows users to calculate the balance in a bank account. The user should be able to enter a starting balance, and then calculate how that balance changes when they make a deposit, withdraw money, or enter any accumulated interest. Add the appropriate data members and member functions to the BankAccount class that will enable this functionality. Also, add code to the class that ensures that the user does not overdraw his or her account. Be sure that the program adheres to the information hiding techniques that were discussed in this chapter. Use `cin` and `cout` statements to gather and display information. You will need to use a decision-making structure that continually displays a menu from which the user can select commands to manage his or her account.