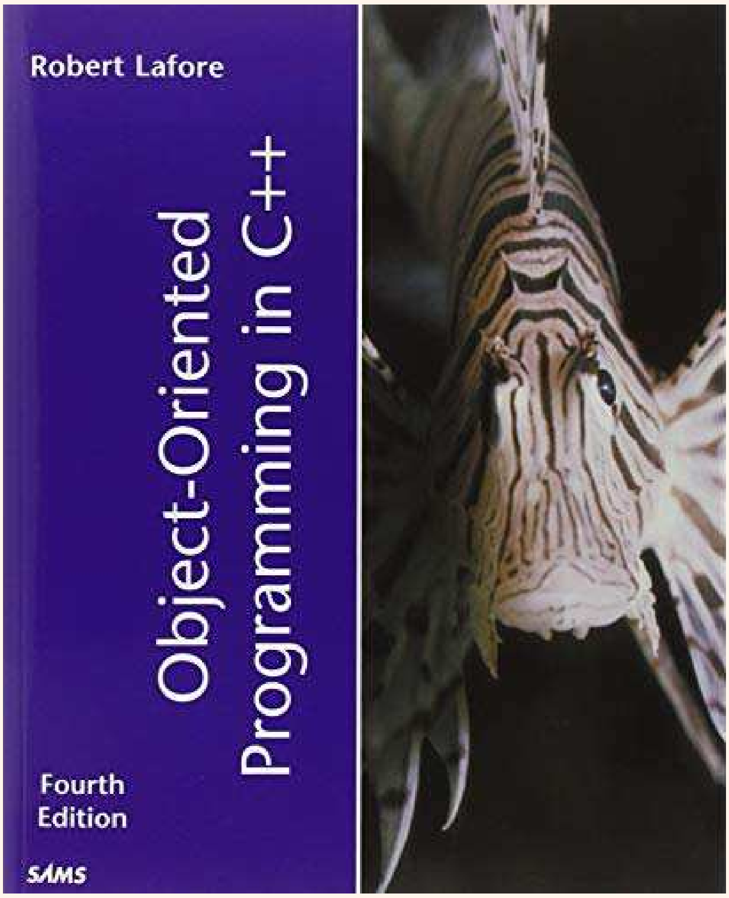


# Chapter 6

## Objects and Classes

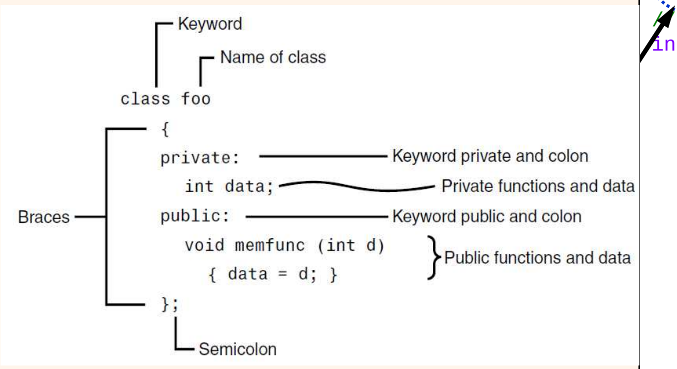
Animated Version

# Topics

- A Simple Class
- C++ Objects as Physical Objects
- C++ Objects as Data Types
- Constructors
- Objects as Function Arguments
- The Default Copy Constructor
- Returning Objects from Functions
- A Card-Game Example
- Structures and Classes 247
- Classes, Objects, and Memory
- Static Class Data
- `const` and Classes
- What Does It All Mean?

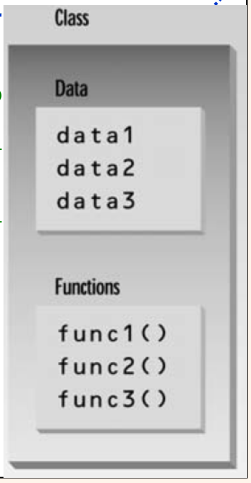
# A Simple Class

- Placing data and functions together into a single entity is a central idea in object-oriented programming.
- Classes and Objects
- Defining the Class



```
// smallobj.cpp
// demonstrates a small, simple object
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class smallobj                 //define a class
   {
   private:
      int somedata;            //class data
   public:
      void setdata(int d)      //member function to set data
         { somedata = d; }
      void showdata()      //member function to display data
         { cout << "Data is " << somedata << endl; }
   };
////////////////////////////////////////////
int main()
   {
   smallobj s1, s2; //define two objects o

   s1.setdata(1066);  //call member functi
   s2.setdata(1776);

   s1.showdata();     //call member functi
     data
   s2.showdata();
   return 0;
   }
```

```
Output:
Data is 1066
Data is 1776
```

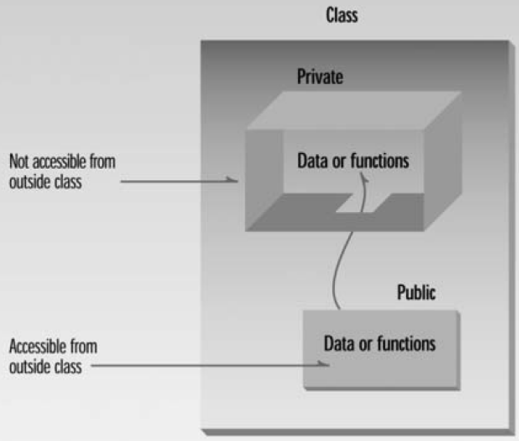


# A Simple Class (2)

- Data hiding:
  - data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class.
  - *Private* data or functions can only be accessed from within the class.
  - *Public* data or functions, on the other hand, are accessible from outside the class.
- Class data: data members
- Member functions
- Usually, functions are public, data is private
- Using the class
- Defining objects

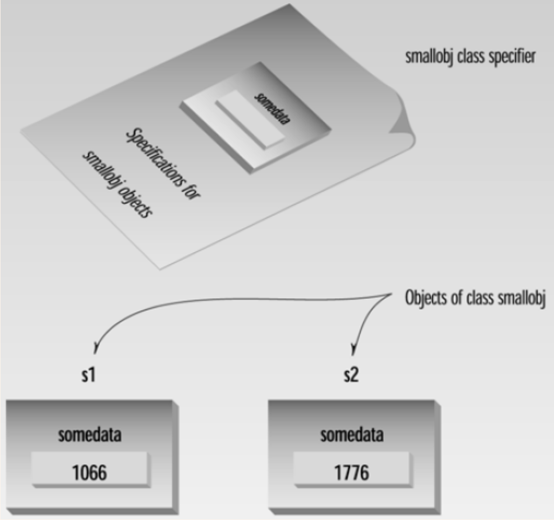
```
// smallobj.cpp
// demonstrates a small, simple object
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class smallobj                 //define a class
   {
   private:
      int somedata;            //class data
   public:
      void setdata(int d)      //member function to set data
         { somedata = d; }
      void showdata()      //member function to display data
         { cout << "Data is " << somedata << endl; }
   };
////////////////////////////////////////
int main()
   {
   smallobj s1, s2; //define two objects c

   s1.setdata(1066);
   s2.setdata(1776);

   s1.showdata();
   s2.showdata();
   return 0;
   }
```

```
Output:
Data is 1066
Data is 1776
```


Class
Private
Not accessible from outside class
Data or functions
Public
Accessible from outside class
Data or functions

# A Simple Class (3)

- Using the class
  - Defining objects
  - Calling Member Functions: *Messages*


smallobj class specifier
somedata
Specifications for smallobj objects
Objects of class smallobj
s1
s2
somedata
1066
somedata
1776

```
// smallobj.cpp
// demonstrates a small, simple object
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class smallobj               //define a class
   {
   private:
      int somedata;          //class data
   public:
      void setdata(int d)    //member function to set data
         { somedata = d; }
      void showdata()    //member function to display data
         { cout << "Data is " << somedata << endl; }
   };
////////////////////////////////////////////
int main()
   {
   smallobj s1, s2; //define two objects o

   s1.setdata(1066);  //call member function to set data
   s2.setdata(1776);

   s1.showdata();     //call member function to display data
   s2.showdata();
   return 0;
    }
```

Output:

```
Data is 1066
Data is 1776
```

# C++ Objects as Physical Objects

- Widget Parts as Objects

```
// objpart.cpp
// widget part as an object
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class part                //define a class
   {
   private:
      int modelnumber;    //ID number of widget
      int partnumber;     //ID number of widget part
      float cost;         //cost of part
   public:
      void setpart(int mn, int pn, float c)  //set data
         {
         modelnumber = mn;
         partnumber = pn;
         cost = c;
         }
      void showpart()                        //display data
         {
         cout << "Model "    << modelnumber;
         cout << ", part "   << partnumber;
         cout << ", costs $" << cost << endl;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   part part1;                           //define object
                                         //   of class part
   part1.setpart(6244, 373, 217.55F);    //call member function
   part1.showpart();                     //call member function
   return 0;
   }
```

Output:

```
Model 6244, part 373, costs $217.55
```

# C++ Objects as Data Types

- This is similar to the Distance structure seen in examples in Chapter 4, but
- here the class Distance also has three member functions:
  - –setdist(), which uses arguments to set feet and inches;
  - –getdist(), which gets values for feet and inches from the user at the keyboard; and
  - –showdist(), which displays the distance in feet-and-inches format.

```cpp
// englobj.cpp
// objects using English measurements
#include <iostream>
using namespace std;
//////////////////////////////
class Distance
   {
   private:
      int feet;
      float inches;
   public:
      void setdist(int ft, float in)  //set Distance to args
         { feet = ft; inches = in; }

      void getdist()             //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }

      void showdist()            //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist2;        //define two lengths

   dist1.setdist(11, 6.25);      //set dist1
   dist2.getdist();              //get dist2 from user

                                 //display lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << endl;
   return 0;
   }
```

Output:

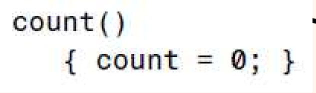
```
Enter feet: 10
Enter inches: 4.75
dist1 = 11’-6.25”
dist2 = 10’-4.75”
```

# Constructor

- Sometimes, it’s convenient if an object can initialize itself when it’s first created, without requiring a separate call to a member function.
- Automatic initialization is carried out using a special member function called a *constructor*.
- A constructor is a member function that is
  - – executed automatically whenever an object is created
  - – same name as class name
  - – no return type
  - – public

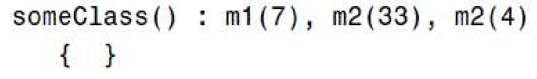
# Constructor (2)

- Automatic Initialization
- Same Name as the Class
- Initializer List
  - Same as:



```
count()
    { count = 0; }
```

  - If multiple members must be initialized, they're separated by commas.



```
someClass() : m1(7), m2(33), m2(4)
   {  }
```

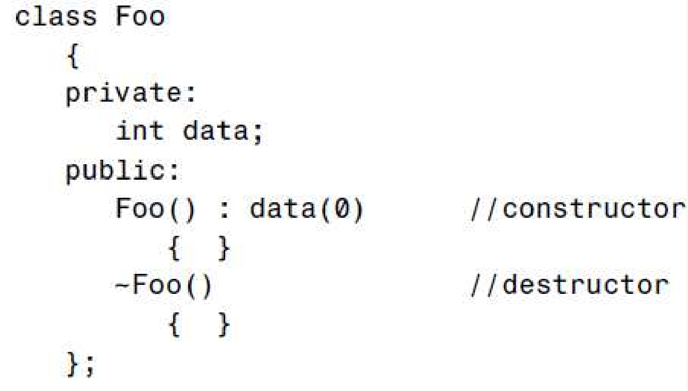
  - Data members are given a value before the constructor even starts to execute.
  - is the only way to initialize `const` member data and references.

```
// counter.cpp
// object represents a counter variable
#include <iostream>
using namespace std;
////////////////////////////
class Counter
   {
   private:
      unsigned int count;
   public:
      Counter() : count(0)                      //constructor
         { /*empty body*/ }
      void inc_count()                          //increment count
         { count++; }
      int get_count()                           //return count
         { return count; }
   };
/////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                       //define and initialize

   cout << "\nc1=" << c1.get_count();   //display
   cout << "\nc2=" << c2.get_count();

   c1.inc_count();                       //increment c1
   c2.inc_count();                       //increment c2
   c2.inc_count();                       //increment c2

   cout << "\nc1=" << c1.get_count();   //display again
   cout << "\nc2=" << c2.get_count();
   cout << endl;
   return 0;
   }
```

```
Output:
c1=0
c2=0
c1=1
c2=2
```

# Destructor

- Destructor function is called automatically when an object is destroyed.
- A destructor is a member function that is
  - executed automatically whenever an object is created
  - same name as class name preceded by a tilde (~)
  - no return type
  - public



```
class Foo
   {
   private:
      int data;
   public:
      Foo() : data(0)          //constructor
         {  }
      ~Foo()                   //destructor
         {  }
   };
```

# Objects as Function Arguments

```
// englcon.cpp
// constructors, adds objects using member function
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                     //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                         //constructor (no args)
      Distance() : feet(0), inches(0.0) // Default constructor
         {  }
                                   //constructor (two args)
      Distance(int ft, float in)  : feet(ft), inches(in)
         {  }

      void getdist()               //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }

      void showdist()              //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      void add_dist(Distance,  Distance) const;    //declaration
   };
```

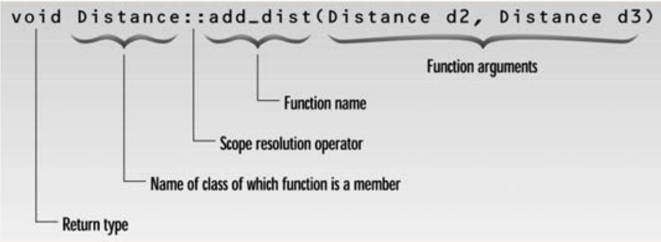
```
//--------------------------------------------------------------
                                   //add lengths d2 and d3
void Distance::add_dist(Distance d2,  Distance d3) const
   {
   inches = d2.inches + d3.inches; //add the inches
   feet = 0;                       //(for possible carry)
   if(inches >= 12.0)              //if total exceeds 12.0,
      {                            //then decrease inches
      inches -= 12.0;              //by 12.0 and
      feet++;                      //increase feet
      }                            //by 1
             .feet + d3.feet;      //add the feet
   }
////////////////////////////////////////////////////////////////
int main()
    {
   Distance dist1, dist3;          //define two lengths
   Distance dist2(11, 6.25);   //define and initialize dist2

   dist1.getdist();                //get dist1 from user
   dist3.add_dist(dist1, dist2);   //dist3 = dist1 + dist2

                                   //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << "\ndist3 = ";  dist3.showdist();
   cout << endl;
   return 0;
   }
```

```
Output:
Enter feet: 17
Enter inches: 5.75
dist1 = 17’-5.75”
dist2 = 11’-6.25”
dist3 = 29’-0”
```

- Overloaded Constructors
- Member Functions Defined Outside the Class

# Objects as Function Arguments (2)

- Member Functions Defined Outside the Class
  - –Scope resolution operator

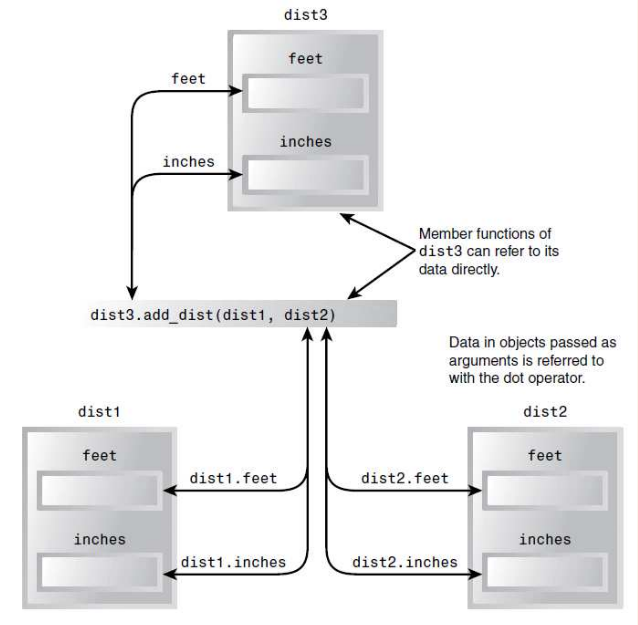

void Distance::add_dist(Distance d2, Distance d3)
Function arguments
Function name
Scope resolution operator
Name of class of which function is a member
Return type

```
//--------------------------------------------------------------
                                   //add lengths d2 and d3
void Distance::add_dist(Distance d2,  Distance d3) const
   {
   inches = d2.inches + d3.inches; //add the inches
   feet = 0;                       //(for possible carry)
   if(inches >= 12.0)              //if total exceeds 12.0,
      {                            //then decrease inches
      inches -= 12.0;              //by 12.0 and
      feet++;                      //increase feet
      }                            //by 1
   feet += d2.feet + d3.feet;      //add the feet
   }
////////////////////////////////////////////////////////////////
int main()
    {
   Distance dist1, dist3;          //define two lengths
   Distance dist2(11, 6.25);   //define and initialize dist2

   dist1.getdist();                //get dist1 from user
   dist3.add_dist(dist1, dist2);   //dist3 = dist1 + dist2

                                   //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << "\ndist3 = ";  dist3.showdist();
   cout << endl;
   return 0;
   }
```

```
Output:
Enter feet: 17
Enter inches: 5.75
dist1 = 17’-5.75”
dist2 = 11’-6.25”
dist3 = 29’-0”
```

# Objects as Function Arguments (2)

- Objects as Arguments


dist3
feet
inches
feet
inches
Member functions of dist3 can refer to its data directly.
dist3.add_dist(dist1, dist2)
Data in objects passed as arguments is referred to with the dot operator.
dist1
feet
inches
dist1.feet
dist1.inches
dist2
feet
inches
dist2.feet
dist2.inches

```
//--------------------------------------------------------------
                                    //add lengths d2 and d3
void Distance::add_dist(Distance d2,  Distance d3) const
   {
   inches = d2.inches + d3.inches; //add the inches
   feet = 0;                       //(for possible carry)
   if(inches >= 12.0)              //if total exceeds 12.0,
      {                            //then decrease inches
      inches -= 12.0;              //by 12.0 and
      feet++;                      //increase feet
      }                            //by 1
   feet += d2.feet + d3.feet;      //add the feet
   }
////////////////////////////////////////////////////////////////
int main()
    {
   Distance dist1, dist3;          //define two lengths
   Distance dist2(11, 6.25);   //define and initialize dist2

   dist1.getdist();                //get dist1 from user
   dist3.add_dist(dist1, dist2);   //dist3 = dist1 + dist2

                                   //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << "\ndist3 = ";  dist3.showdist();
   cout << endl;
   return 0;
   }
```

```
Output:
Enter feet: 17
Enter inches: 5.75
dist1 = 17’-5.75”
dist2 = 11’-6.25”
dist3 = 29’-0”
```

# The Default Copy Constructor

- A no-argument constructor can initialize data members to constant values, and
- a multi-argument constructor can initialize data members to values passed as arguments.
- another way to initialize an object: with another object of the same type.
  - one is already built into all classes. It’s called the *default copy constructor*.
  - It’s a one argument constructor whose argument is an object of the same class as the constructor.

```
// ecopycon.cpp
// initialize objects using default copy constructor
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance
   {
   private:
      int feet;
      float inches;
   public:                              //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }
      //Note: no one-arg constructor
                                     //constructor (two args)
      Distance(int ft, float in)  : feet(ft), inches(in)
         {  }

      void getdist()                 //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist()                //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1(11, 6.25);         //two-arg constructor
   Distance dist2(dist1);            //one-arg constructor
   Distance dist3 = dist1;           //also one-arg constructor
                                     //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << "\ndist3 = ";  dist3.showdist();
   cout << endl;
   return 0;
   }
```

```
Output:
dist1 = 11’-6.25”
dist2 = 11’-6.25”
dist3 = 11’-6.25”
```

# Returning Objects from Functions

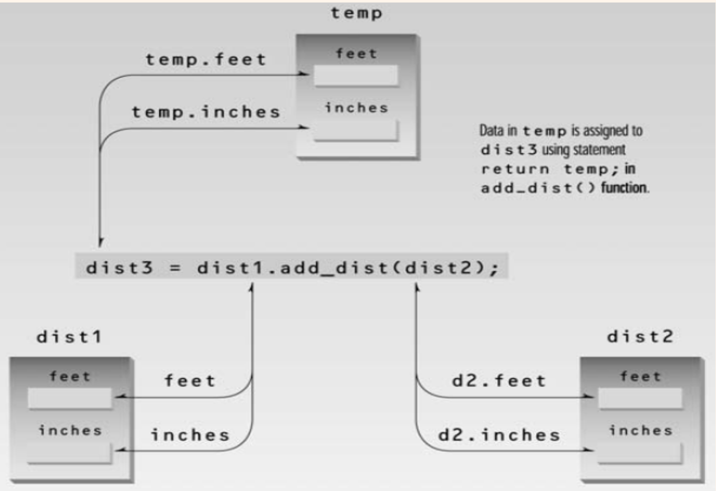
```
// englret.cpp
// function returns value of type Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                         //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                             //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }                          //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }

      void getdist()                   //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist()                  //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      Distance add_dist(Distance);     //add
   };
```

```
//-------------------------------------------------------------
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
   {
   Distance temp;                     //temporary variable
   temp.inches = inches + d2.inches;  //add the inches
   if(temp.inches >= 12.0)            //if total exceeds
12.0,
      {                               //then decrease inches
      temp.inches -= 12.0;            //by 12.0 and
      temp.feet = 1;                  //increase feet
      }                               //by 1
   temp.feet += feet + d2.feet;       //add the feet
   return temp;
   }
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3;          //define two lengths
   Distance dist2(11, 6.25);      //define, initialize dist2

   dist1.getdist();                //get dist1 from user
   dist3 = dist1.add_dist(dist2);  //dist3 = dist1 + dist2
                                   //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << "\ndist3 = ";  dist3.showdist();
   cout << endl;
   return 0;
   }
```

# Returning Objects from Functions

```
// englret.cpp
// function returns value of type Distance
//-------------------------------------------------------------
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
   {
   Distance temp;                     //temporary variable
   temp.inches = inches + d2.inches;  //add the inches
   if(temp.inches >= 12.0)            //if total exceeds 12.0,
      {                               //then decrease inches
      temp.inches -= 12.0;            //by 12.0 and
      temp.feet = 1;                  //increase feet
      }                               //by 1
   temp.feet += feet + d2.feet;       //add the feet
   return temp;
   }
```

```
// englcon.cpp
// constructors, adds objects using member function
//////////////////////////////////////////////////////////////
 //add lengths d2 and d3
void Distance::add_dist(Distance d2,  Distance d3) const
   {
   inches = d2.inches + d3.inches; //add the inches
   feet = 0;                       //(for possible carry)
   if(inches >= 12.0)              //if total exceeds 12.0,
      {                            //then decrease inches
      inches -= 12.0;              //by 12.0 and
      feet++;                      //increase feet
      }                            //by 1
   feet += d2.feet + d3.feet;      //add the feet
   }
```



Data in temp is assigned to dist3 using statement return temp; in add_dist() function.

# A Card-Game Example

```
// cardobj.cpp
// cards as objects
#include <iostream>
using namespace std;

const int jack = 11;          //from 2 to 10 are
const int queen = 12;         //integers without names
const int king = 13;
const int ace = 14;
enum Suit { clubs, diamonds, hearts, spades };
////////////////////////////////////////////////////////////////
class card
   {
   private:
      int number;             //2 to 10, jack, queen, king, ace
      Suit suit;              //clubs, diamonds, hearts, spades
   public:
         card ()                  //constructor (no args)
        {  }
                              //constructor (two args)
      card (int n, Suit s) : number(n), suit(s)
         {  }
      void display();         //display card
      bool isEqual(card);     //same as another card?
   };
```

```
//------------------------------------------------------------
void card::display()          //display the card
   {
   if( number >= 2 && number <= 10 )
      cout << number << " of ";
   else
      switch(number)
         {
         case jack:  cout << "jack of ";  break;
         case queen: cout << "queen of "; break;
         case king:  cout << "king of ";  break;
         case ace:   cout << "ace of ";   break;
         }
   switch(suit)
      {
      case clubs:    cout << "clubs"; break;
      case diamonds: cout << "diamonds"; break;
      case hearts:   cout << "hearts"; break;
      case spades:   cout << "spades"; break;
      }
   }
//------------------------------------------------------------
bool card::isEqual(card c2)    //return true if cards equal
   {
   return ( number==c2.number && suit==c2.suit ) ? true :
false;
   }
```

# A Card-Game Example (2)

```
////////////////////////////////////////////////////////////////
int main()
   {
   card temp, chosen, prize;      //define various cards
   int position;

   card card1( 7, clubs );        //define & initialize card1
   cout << "\nCard 1 is the ";
   card1.display();               //display card1

   card card2( jack, hearts );    //define & initialize card2
   cout << "\nCard 2 is the ";
   card2.display();               //display card2

   card card3( ace, spades );     //define & initialize card3
   cout << "\nCard 3 is the ";
   card3.display();               //display card3

   prize = card3;                 //prize is the card to guess

   cout << "\nI'm swapping card 1 and card 3";
   temp = card3; card3 = card1; card1 = temp;

   cout << "\nI'm swapping card 2 and card 3";
   temp = card3; card3 = card2; card2 = temp;

   cout << "\nI'm swapping card 1 and card 2";
   temp = card2; card2 = card1; card1 = temp;
```

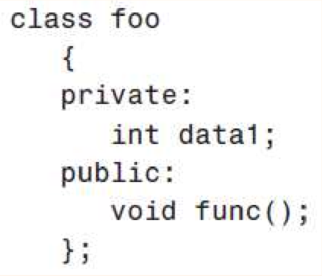
```
   cout << "\nNow, where (1, 2, or 3) is the ";
   prize.display();               //display prize card
   cout << "? ";
   cin >> position;               //get user's guess of
position

   switch (position)
      {                           //set chosen to user's
choice
      case 1: chosen = card1; break;
      case 2: chosen = card2; break;
      case 3: chosen = card3; break;
      }
   if( chosen.isEqual(prize) )    //is chosen card the
prize?
      cout << "That's right!  You win!";
   else
      cout << "Sorry. You lose.";
   cout << "  You chose the ";
   chosen.display();              //display chosen card
   cout << endl;
   return 0;
   }
```

Output:

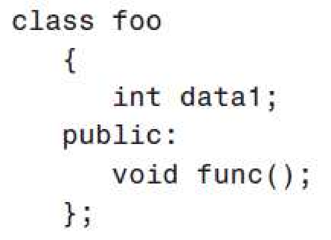
```
Card 1 is the 7 of clubs
Card 2 is the jack of hearts
Card 3 is the ace of spades
I’m swapping card 1 and card 3
I’m swapping card 2 and card 3
I’m swapping card 1 and card 2
Now, where (1, 2, or 3) is the ace of
spades? 1
Sorry, you lose. You chose the 7 of clubs
```

# Structures and Classes

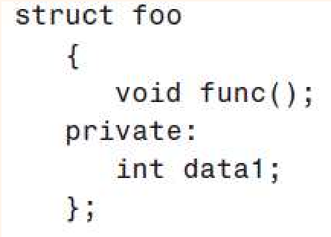
- In fact, you can use structures in almost exactly the same way that you use classes.
- The only formal difference between `class` and `struct` is that in a class the members are *private* by default, while in a structure they are *public* by default.



```
class foo
   {
   private:
      int data1;
   public:
      void func();
   };
```



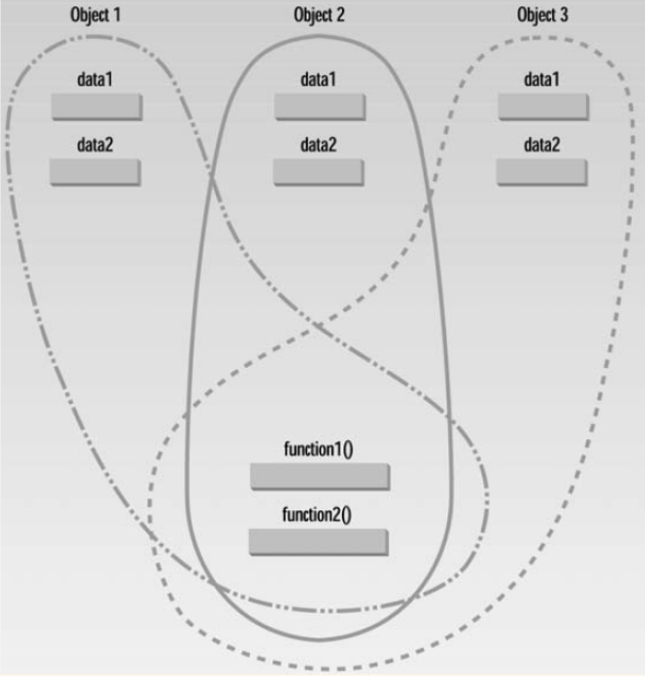
```
class foo
   {
      int data1;
   public:
      void func();
   };
```



```
struct foo
   {
      void func();
   private:
      int data1;
   };
```

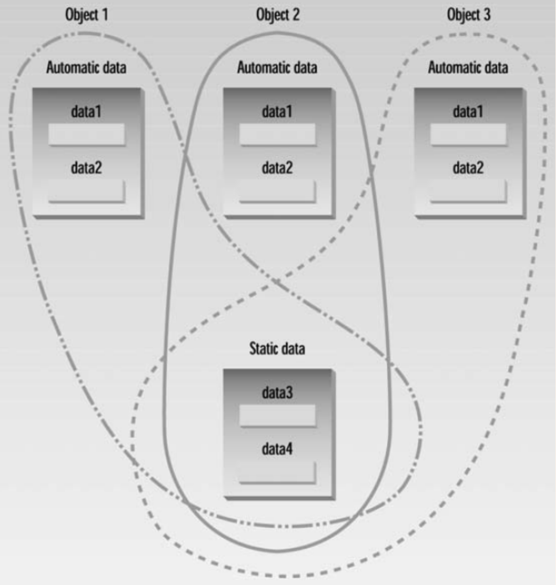
# Classes, Objects, and Memory

- each object has its own separate data items.
- all the objects in a given class use the same member functions.



# Static Class Data

- If a data item in a class is declared as static , only one such item is created for the entire class, no matter how many objects there are.
  - useful when all objects of the same class must share a common item of information.
  - A static member variable is visible only within the class, but its lifetime is the entire program.
  - It continues to exist even if there are no objects of the class.
  - used to share information among the objects of a class.



# Static Class Data (2)

- Uses of Static Class Data
- Separate Declaration and Definition because:
  - memory space for such data is allocated only once and one static member variable is accessed by an entire class
  - If static member data were defined inside the class (as it actually was in early versions of C++), it would violate the idea that a class definition is only a blueprint and does not set aside any memory.

```
// statdata.cpp
// static class data
#include <iostream>
using namespace std;
///////////////////////////////////////////////////////////////
class foo
   {
   private:
      static int count;    //only one data item for all objects
                           //note: *declaration* only!
   public:
      foo()                //increments count when object created
         { count++; }
      int getcount()       //returns count
         { return count; }
   };
//
int foo::count = 0;        //*definition* of count
///////////////////////////////////////////////////////////////
int main()
   {
   foo f1, f2, f3;         //create three objects

   cout << "count is " << f1.getcount() << endl;  //each object
   cout << "count is " << f2.getcount() << endl;  //sees the
   cout << "count is " << f3.getcount() << endl;  //same value
   return 0;
   }
```

```
Output:
count is 3
count is 3
count is 3
```

# const and Classes: const Member Functions

– A const member function guarantees that it will never modify any of its class's member data.

– const Member Function Arguments

```
// engConst.cpp
// const member functions and const arguments to member funcions
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                           //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                               //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }                            //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }

      void getdist()                     //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const              //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      Distance add_dist(const Distance&) const;     //add
   };
//--------------------------------------------------------------
```

```
//add this distance to d2, return the sum
Distance Distance::add_dist(const Distance& d2) const
   {
   Distance temp;                    //temporary variable
// feet = 0;                    //ERROR: can't modify this
// d2.feet = 0;                 //ERROR: can't modify d2
   temp.inches = inches + d2.inches;  //add the inches
   if(temp.inches >= 12.0)           //if total exceeds
     12.0,
      {                              //then decrease inches
      temp.inches -= 12.0;           //by 12.0 and
      temp.feet = 1;                 //increase feet
      }                              //by 1
   temp.feet += feet + d2.feet;      //add the feet
   return temp;
   }
////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3;       //define two lengths
   Distance dist2(11, 6.25);    //define, initialize dist2

   dist1.getdist();                //get dist1 from user
   dist3 = dist1.add_dist(dist2);  //dist3 = dist1 + dist2
                                   //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << "\ndist3 = ";  dist3.showdist();
   cout << endl;
   return 0;
   }
```

# const and Classes: const Objects

- When an object is declared as const, you can't modify it.
- Can use only const member functions with it, because they're the only ones that guarantee not to modify it.
- When you're designing classes it's a good idea to make const any function that does not modify any of the data in its object.
  - allows the user of the class to create const objects.
  - These objects can use any const function, but cannot use any non-const function.

```
// constObj.cpp
// constant Distance objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                      //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                          //2-arg constructor
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()                //user input; non-const func
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const         //display distance; const func
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   const Distance football(300, 0);

// football.getdist();              //error: getdist() not const
   cout << "football = ";
   football.showdist();             //OK
   cout << endl;
   return 0;
   }
```

# Few Advantages of Using OOP

- close correspondence between the real-world things being modeled by the program and the C++ objects in the program.
- You figure out what parts of the problem can be most usefully represented as objects, and then put all the data and functions connected with that object into the class.
  - procedural program don’t form such single, easily grasped unit.
- The larger the program, the greater the benefit.

# Summary (1)

- A class is a specification or blueprint for a number of objects.
  - Objects consist of both data and functions that operate on that data.
  - In a class definition, the members—whether data or functions—can be
    - » private , meaning they can be accessed only by member functions of that class,
    - » or public , meaning they can be accessed by any function in the program.
- A member function is a function that is a member of a class. Member functions have access to an object’s private data, while non-member functions do not.
- A constructor is a member function, with the same name as its class, that is executed every time an object of the class is created.
  - has no return type but can take arguments.
  - often used to give initial values to object data members.
  - Constructors can be overloaded, so an object can be initialized in different ways.
- A destructor is a member function with the same name as its class but preceded by a tilde ( ~ ).
  - It is called when an object is destroyed.
  - A destructor takes no arguments and has no return value.

# Summary (2)

- In the computer’s memory there is a separate copy of the data members for each object that is created from a class, but there is only one copy of a class’s member functions.
- You can restrict a data item to a single instance for all objects of a class by making it static .
- One reason to use OOP is the close correspondence between real-world objects and OOP classes.
  - Deciding what objects and classes to use in a program can be complicated.
    - For small programs, trial and error may be sufficient.
    - For large programs, a more systematic approach is usually needed.