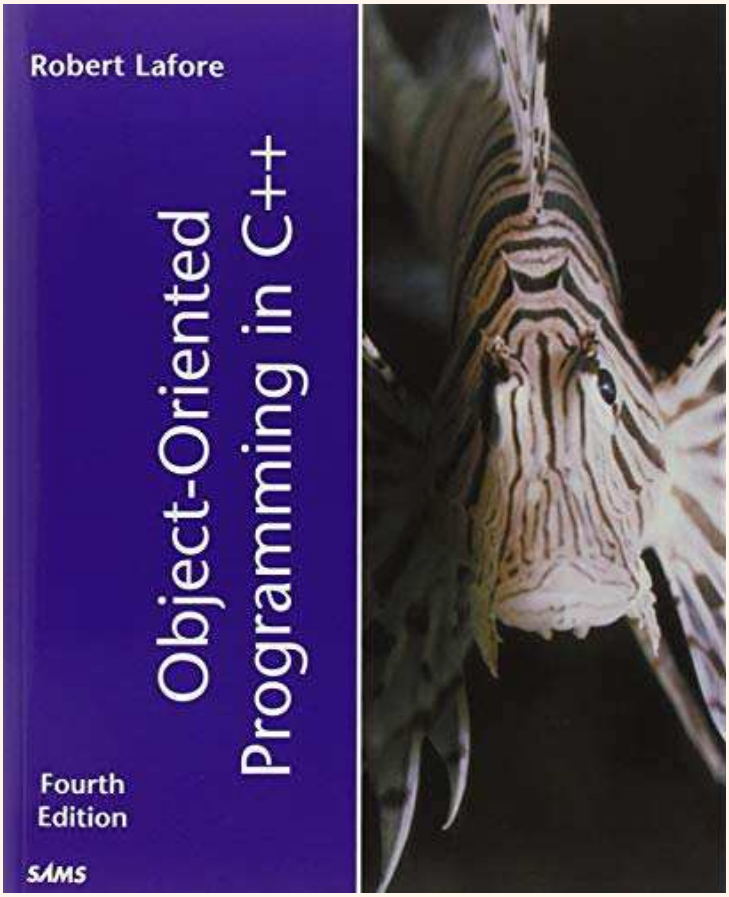


Chapter 11

Virtual Functions

Animated Version

Topics

- Virtual Functions
- Friend Functions
- Static Functions
- Assignment and Copy Initialization
- The `this` Pointer
- Dynamic Type Information

Virtual Functions

- means *existing in appearance but not in reality*. Use:
  - suppose a graphics program includes several different shapes: a triangle, a ball, a square, and so on. Each of these classes has a member function draw() that causes the object to be drawn on the screen.

```
shape* ptrarr[100];  // array of 100 pointers to shapes
```

  - If you insert pointers to all the shapes into this array, you can then draw an entire picture using a simple loop:

```
for(int j=0; j<N; j++)
   ptrarr[j]->draw();
```

  - This is an amazing capability:
    - Completely different functions are executed by the same function call. If the pointer in *ptrarr* points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle's function is called.
  - This is called polymorphism, which means different forms.

Virtual Functions (2)

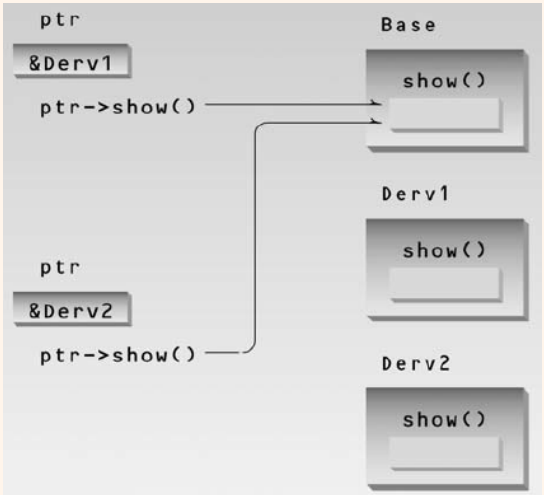
```
// notvirt.cpp
// normal functions accessed from pointer
#include <iostream>
using namespace std;
class Base                          //base class
   {
   public:
      void show()                   //normal function
         { cout << "Base\n"; }
   };
class Derv1 : public Base           //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
class Derv2 : public Base           //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
int main()
   {
   Derv1 dv1;              //object of derived class 1
   Derv2 dv2;              //object of derived class 2
   Base* ptr;              //pointer to base class

   ptr = &dv1;       //put address of dv1 in pointer
   ptr->show();            //execute show()

   ptr = &dv2;       //put address of dv2 in pointer
   ptr->show();            //execute show()
   return 0;
   }
```

```
Base
Base
```

- Normal Member Functions Accessed with Pointers
  - the function in the base class is always executed.
  - The compiler ignores the contents of the pointer *ptr* and chooses the member function that matches the type of the pointer.



Virtual Functions (3)

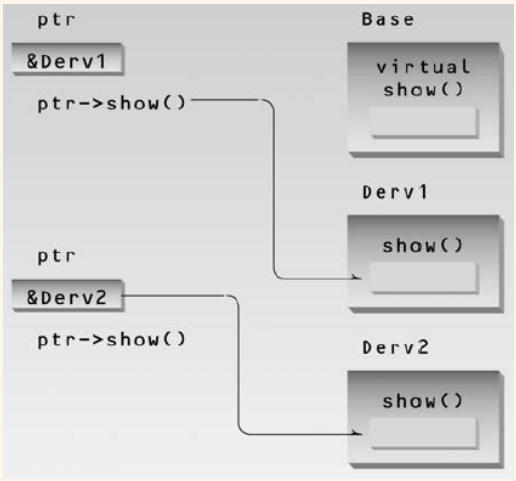
```
// virt.cpp
// virtual functions accessed from pointer
#include <iostream>
using namespace std;
class Base                          //base class
   {
   public:
     virtual void show()            //virtual function
         { cout << "Base\n"; }
   };
class Derv1 : public Base      //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
class Derv2 : public Base      //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
int main()
   {
   Derv1 dv1;            //object of derived class 1
   Derv2 dv2;            //object of derived class 2
   Base* ptr;            //pointer to base class

   ptr = &dv1;        //put address of dv1 in pointer
   ptr->show();          //execute show()

   ptr = &dv2;        //put address of dv2 in pointer
   ptr->show();          //execute show()
   return 0;
   }
```

```
Derv1
Derv2
```

- Virtual Member Functions Accessed with Pointers
  - member functions of the derived classes, not the base class, are executed.



Abstract Classes and Pure Virtual Functions

```
// virtpure.cpp
// pure virtual function
#include <iostream>
using namespace std;
class Base                          //base class
   {
   public:
     virtual void show() = 0;     //pure virtual
     function
   };
class Derv1 : public Base          //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
class Derv2 : public Base          //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
int main()
   {
// Base bad; //can't make object from abstract class
   Base* arr[2];    //array of pointers to base class
   Derv1 dv1;            //object of derived class 1
   Derv2 dv2;            //object of derived class 2

   arr[0] = &dv1;        //put address of dv1 in array
   arr[1] = &dv2;        //put address of dv2 in array
   arr[0]->show();    //execute show() in both objects
   arr[1]->show();
   return 0;
   }
```

```
Derv1
Derv2
```

- Pure Virtual Function
  - one with the expression =*0* added to the declaration
- Abstract Class
  - a class having a pure virtual function
  - or, a derived class that does not override a pure virtual function from base class
  - we can not instantiate objects of an abstract class
  - exists only to act as a parent of derived classes that will be used to instantiate objects
  - may also provide an interface for the class hierarchy.

Virtual Functions (5)

- Virtual Functions and the person Class

```
// virtpers.cpp
// virtual functions with person class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////
class person                          //person class
   {
   protected:
      char name[40];
   public:
      void getName()
         { cout << "   Enter name: "; cin >> name; }
      void putName()
         { cout << "Name is: " << name << endl; }
      virtual void getData() = 0; //pure virtual func
      virtual bool isOutstanding() = 0;
                                    //pure virtual func
   };
////////////////////////////////////////////////////
class student : public person         //student class
   {
   private:
      float gpa;          //grade point average
   public:
      void getData()      //get student data from user
         {
         person::getName();
        cout << "   Enter student's GPA: "; cin >> gpa;
         }
      bool isOutstanding()
         { return (gpa > 3.5) ? true : false; }
   };
```

```
////////////////////////////////////////////////////
class professor : public person       //professor class
   {
   private:
      int numPubs;       //number of papers published
   public:
      void getData()     //get professor data from user
         {
         person::getName();
cout << "   Enter number of professor's publications: ";
         cin >> numPubs;
         }
      bool isOutstanding()
         { return (numPubs > 100) ? true : false; }
   };
```


Virtual Functions (6)

- Virtual Functions and the person Class

```
////////////////////////////////////////////////////
int main()
   {
   person* persPtr[100];//array of pointers to persons
   int n = 0;          //number of persons on list
   char choice;

   do {
      cout << "Enter student or professor (s/p): ";
      cin >> choice;
      if(choice=='s')               //put new student
         persPtr[n] = new student;     //   in array
      else                         //put new professor
         persPtr[n] = new professor;   //   in array
      persPtr[n++]->getData();  //get data for person
      cout << "   Enter another (y/n)? ";
                //do another person?
      cin >> choice;
      } while( choice=='y' );  //cycle until not 'y'

   for(int j=0; j<n; j++)          //print names of all
      {                            //persons, and
      persPtr[j]->putName();       //say if outstanding
      if( persPtr[j]->isOutstanding() )
         cout << "   This person is outstanding\n";
      }
   return 0;
   }  //end main()
```

```
Enter student or professor (s/p): s
   Enter name: Timmy
   Enter student's GPA: 1.2
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Brenda
   Enter student's GPA: 3.9
   Enter another (y/n)? y
Enter student or professor (s/p): s
   Enter name: Sandy
   Enter student's GPA: 2.4
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Shipley
   Enter number of professor's publications: 714
   Enter another (y/n)? y
Enter student or professor (s/p): p
   Enter name: Wainright
   Enter number of professor's publications: 13
   Enter another (y/n)? n

Name is: Timmy
Name is: Brenda
   This person is outstanding
Name is: Sandy
Name is: Shipley
   This person is outstanding
Name is: Wainright
```

Virtual Functions (7)

```
//vertdest.cpp
//tests non-virtual and virtual destructors
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class Base
   {
   public:
      ~Base()            //non-virtual destructor
//    virtual ~Base()  //virtual destructor
         { cout << "Base destroyed\n"; }
   };
////////////////////////////////////////////////
class Derv : public Base
  {
   public:
      ~Derv()
         { cout << "Derv destroyed\n"; }
   };
////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Derv;
   delete pBase;
   return 0;
   }
```

- Virtual Destructors

Output:

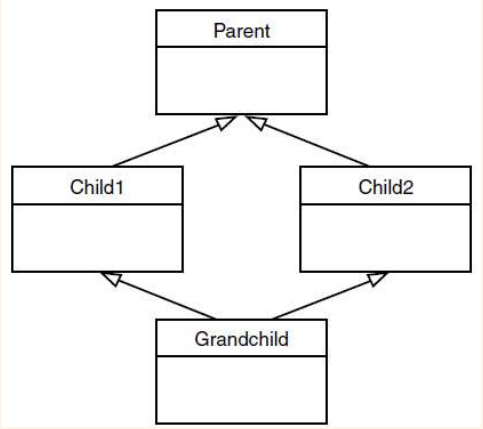
```
Base destroyed
```

If comment removed:

```
Derv destroyed
Base destroyed
```


Virtual Functions (7): Virtual Base Class

```
// normbase.cpp
// ambiguous reference to base class
class Parent
   {
   protected:
      int basedata;
   };
class Child1 : public Parent
   { };
class Child2 : public Parent
   { };
class Grandchild : public Child1, public Child2
   {
   public:
      int getdata()
         { return basedata; }   // ERROR: ambiguous
   };
```




```
// virtbase.cpp
// virtual base classes
class Parent
 {  protected:
    int basedata;
   };
class Child1 : virtual public Parent // shares copy of Parent
   { };
class Child2 : virtual public Parent // shares copy of Parent
   { };
class Grandchild : public Child1, public Child2
   {
   public:
      int getdata()
         { return basedata; }  // OK: only one copy of Parent
   };
```

Friend Functions

- Encapsulation and Data Hiding: nonmember functions should not be able to access an object's private or protected data.
- Imagine that you want a function to take objects of the two classes as arguments, and operate on their private data. In this situation there's nothing like a friend function.
- No breach of data integrity
  - A friend function must be declared as such within the class whose data it will access. Thus a programmer who does not have access to the source code for the class cannot make a function into a friend.
- However, should be used sparingly.

Friend Functions (2)

```
// friend.cpp
// friend functions
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////
class beta;        //needed for frifunc declaration
class alpha
   {
   private:
      int data;
   public:
      alpha() : data(3) {  }   //no-arg constructor
      friend int frifunc(alpha, beta);//friend function
   };
////////////////////////////////////////////////////////
class beta
   {
   private:
      int data;
   public:
      beta() : data(7) {  }  //no-arg constructor
      friend int frifunc(alpha, beta);//friend function
   };
////////////////////////////////////////////////////////
int frifunc(alpha a, beta b)  //function definition
   {
   return( a.data + b.data );
   }
//--------------------------------------------------
int main()
   {
   alpha aa;
   beta bb;
   cout << frifunc(aa, bb) << endl;//call the function
   return 0;
   }
```

- Friends as Bridges
- Breaching the Walls

Friend Functions (3)

- English Distance Example (without Friend function)

```
// nofri.cpp
// limitation to overloaded + operator
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////
class Distance               //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)
                             //constructor (no args)
         {  }                //constructor (one arg)
      Distance(float fltfeet)//convert float to Distance
         {                   //feet is integer part
         feet = static_cast<int>(fltfeet);
          inches = 12*(fltfeet-feet);
                             //inches is what's left
         }
      Distance(int ft, float in)//constructor (two args)
         { feet = ft; inches = in; }
      void showdist()        //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      Distance operator + (Distance);
   };
```

```
//-----------------------------------------------------
                              //add this distance to d2
Distance Distance::operator + (Distance d2)
                              //return the sum
   {
   int f = feet + d2.feet;        //add the feet
   float i = inches + d2.inches;  //add the inches
   if(i >= 12.0)             //if total exceeds 12.0,
      { i -= 12.0; f++;  }//less 12 inches, plus 1 foot
   return Distance(f,i); //return new Distance with sum
   }
////////////////////////////////////////////////////////
int main()
   {
   Distance d1 = 2.5;        //constructor converts
   Distance d2 = 1.25;       //float feet to Distance
   Distance d3;
   cout << "\nd1 = "; d1.showdist();
   cout << "\nd2 = "; d2.showdist();

   d3 = d1 + 10.0;           //distance + float: OK
   cout << "\nd3 = "; d3.showdist();
// d3 = 10.0 + d1;           //float + Distance: ERROR
// cout << "\nd3 = "; d3.showdist();
   cout << endl;
   return 0;
   }
```

```
d1 = 2'-6"
d2 = 1'-3"
d3 = 12'-6"
```


Friend Functions (4)

- English Distance Example (using Friend function)

```
// frengl.cpp
// friend overloaded + operator
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////
class Distance          //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      Distance()             //constructor (no args)
         { feet = 0; inches = 0.0; }
      Distance( float fltfeet )//constructor (one arg)
         {                  //convert float to Distance
         feet = int(fltfeet); //feet is integer part
         inches = 12*(fltfeet-feet);
                            //inches is what's left
         }
      Distance(int ft, float in)//constructor (two args)
         { feet = ft; inches = in; }
      void showdist()           //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      friend Distance operator + (Distance, Distance);
    //friend
   };
```

```
//-----------------------------------------------------
Distance operator + (Distance d1, Distance d2)
                                   //add D1 to d2
   {
   int f = d1.feet + d2.feet;        //add the feet
   float i = d1.inches + d2.inches;  //add the inches
   if(i >= 12.0)                     //if inches exceeds 12.0,
      { i -= 12.0; f++;  }//less 12 inches, plus 1 foot
   return Distance(f,i);//return new Distance with sum
   }
//-----------------------------------------------------
int main()
   {
   Distance d1 = 2.5;        //constructor converts
   Distance d2 = 1.25;       //float-feet to Distance
   Distance d3;
   cout << "\nd1 = "; d1.showdist();
   cout << "\nd2 = "; d2.showdist();

   d3 = d1 + 10.0;           //distance + float: OK
   cout << "\nd3 = "; d3.showdist();
   d3 = 10.0 + d1;           //float + Distance: OK
   cout << "\nd3 = "; d3.showdist();
   cout << endl;
   return 0;
   }
```

Friend Functions (5)

- friend s for Functional Notation (with member function)

```
// misq.cpp
// member square() function for 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 showdist()               //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      float square();               //member function
   };
```

```
//-----------------------------------------------------
float Distance::square()            //return square of
   {                                //this Distance
   float fltfeet = feet + inches/12; //convert to float
   float feetsqrd = fltfeet * fltfeet;//find the square
   return feetsqrd;                 //return square feet
   }
////////////////////////////////////////////////////
int main()
   {
   Distance dist(3, 6.0); //two-arg constructor (3'-6")
   float sqft;

   sqft = dist.square();  //return square of dist
                          //display distance and square
   cout << "\nDistance = "; dist.showdist();
   cout << "\nSquare = " << sqft << " square feet\n";
   return 0;
   }
```

```
Distance = 3'-6"
Square = 12.25 square feet
```


Friend Functions (6)

- friend s for Functional Notation (with friend function)

```
// frisq.cpp
// friend square() function for Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////
class Distance                      //English Distance
    class
   {
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)
                                    //constructor (no args)
         {  }
                                    //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void showdist()               //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      friend float square(Distance); //friend function
   };
```

```
//-----------------------------------------------------
float square(Distance d)            //return square of
   {                                //this Distance
 float fltfeet = d.feet + d.inches/12;//convert to float
 float feetsqrd = fltfeet * fltfeet;  //find the square
   return feetsqrd;                 //return square feet
   }
////////////////////////////////////////////////////////
int main()
   {
   Distance dist(3, 6.0);//two-arg constructor (3'-6")
   float sqft;

   sqft = square(dist);   //return square of dist
                          //display distance and square
   cout << "\nDistance = "; dist.showdist();
   cout << "\nSquare = " << sqft << " square feet\n";
   return 0;
   }
```

Friend Functions (7)

```
// friclass.cpp
// friend classes
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class alpha
   {
   private:
      int data1;
   public:
      alpha() : data1(99) {  }   //constructor
      friend class beta;       //beta is a friend class
   };
////////////////////////////////////////////////
class beta
   {                          //all member functions can
   public:                    //access private alpha data
      void func1(alpha a)  { cout << "\ndata1=" <<
    a.data1; }
      void func2(alpha a)  { cout << "\ndata1=" <<
     a.data1; }
   };
////////////////////////////////////////////////
int main()
   {
   alpha a;
   beta b;

   b.func1(a);
   b.func2(a);
   cout << endl;
   return 0;
   }
```

- friend Classes

Static Functions

```
// statfunc.cpp
// static functions and ID numbers for objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class gamma
   {
   private:
      static int total; //total objects of this class
                        //   (declaration only)
      int id;           //ID number of this object
   public:
      gamma()           //no-argument constructor
         {
         total++;       //add another object
         id = total;    //id equals current total
         }
      ~gamma()          //destructor
         {
         total--;
        cout << "Destroying ID number " << id  << endl;
         }
      static void showtotal()  //static function
         {
         cout << "Total is " << total << endl;
         }
      void showid()             //non-static function
         {
         cout << "ID number is " << id << endl;
         }
   };
//-----------------------------------------------
int gamma::total = 0;      //definition of total
```

- Accessing static Functions
- Investigating Destructors

```
////////////////////////////////////////////////
int main()
   {
   gamma g1;
   gamma::showtotal();

   gamma g2, g3;
   gamma::showtotal();

   g1.showid();
   g2.showid();
   g3.showid();
   cout << "----------end of program----------\n";
   return 0;
   }
```

```
Total is 1
Total is 3
ID number is 1
ID number is 2
ID number is 3
----------end of program--------
Destroying ID number 3
Destroying ID number 2
Destroying ID number 1
```

The `this` Pointer

- magic pointer *this* points to the object itself.
- Accessing Member Data with this:

```
// where.cpp
// the this pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class where
   {
   private:
      char charray[10];   //occupies 10 bytes
   public:
      void reveal()
     { cout << "\nMy object's address is " << this; }
   };
////////////////////////////////////////////////
int main()
   {
   where w1, w2, w3;       //make three objects
   w1.reveal();            //see where they are
   w2.reveal();
   w3.reveal();
   cout << endl;
   return 0;
   }
```

```
My object's address is 0x8f4effec
My object's address is 0x8f4effe2
My object's address is 0x8f4effd8
```

```
// dothis.cpp
// the this pointer referring to data
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class what
   {
   private:
      int alpha;
   public:
      void tester()
         {
         this->alpha = 11;     //same as alpha = 11;
         cout << this->alpha;  //same as cout << alpha;
         }
   };
////////////////////////////////////////////////
int main()
   {
   what w;
   w.tester();
   cout << endl;
   return 0;
   }
```

The `this` Pointer (2)

- Using *this* for Returning Values

```
//assign2.cpp
// returns contents of the this pointer
#include <iostream>
using namespace std;
class alpha
   {
   private:
      int data;
   public:
      alpha()              //no-arg constructor
         { }
      alpha(int d)         //one-arg constructor
         { data = d; }
      void display()       //display data
         { cout << data; } //overloaded = operator
      alpha& operator = (alpha& a)
         {
         data = a.data;    //not done automatically
         cout << "\nAssignment operator invoked";
         return *this;     //return copy of this alpha
         }
   };
int main()
   {
   alpha a1(37);
   alpha a2, a3;

   a3 = a2 = a1;         //invoke overloaded =, twice
   cout << "\na2="; a2.display();  //display a2
   cout << "\na3="; a3.display();  //display a3
   cout << endl;
   return 0;
   }
```

```
Assignment operator invoked
Assignment operator invoked
a2=37
a3=37
```

Summary (1)

- Virtual functions provide a way for a program to decide while it is running what function to call. Ordinarily such decisions are made at compile time.
  - Virtual functions make possible greater flexibility in performing the same kind of action on different kinds of objects. In particular, they allow the use of functions called from an array of type pointer-to-base that actually holds pointers (or references) to a variety of derived types. This is an example of polymorphism.
  - Typically a function is declared virtual in the base class, and other functions with the same name are declared in derived classes.
- The use of one or more pure virtual functions in a class makes the class abstract, which means that no objects can be instantiated from it.
- A friend function can access a class's private data, even though it is not a member function of the class.
  - This is useful when one function must have access to two or more unrelated classes and when an overloaded operator must use, on its left side, a value of a class other than the one of which it is a member. *f*

Summary (2)

- A static function is one that operates on the class in general, rather than on objects of the class. In particular it can operate on static variables. It can be called with the class name and scope-resolution operator.
- The this pointer is predefined in member functions to point to the object of which the function is a member. The this pointer is useful in returning the object
- The UML object diagram shows the relationship of a group of objects at a specific point in a program's operation. of which the function is a member.