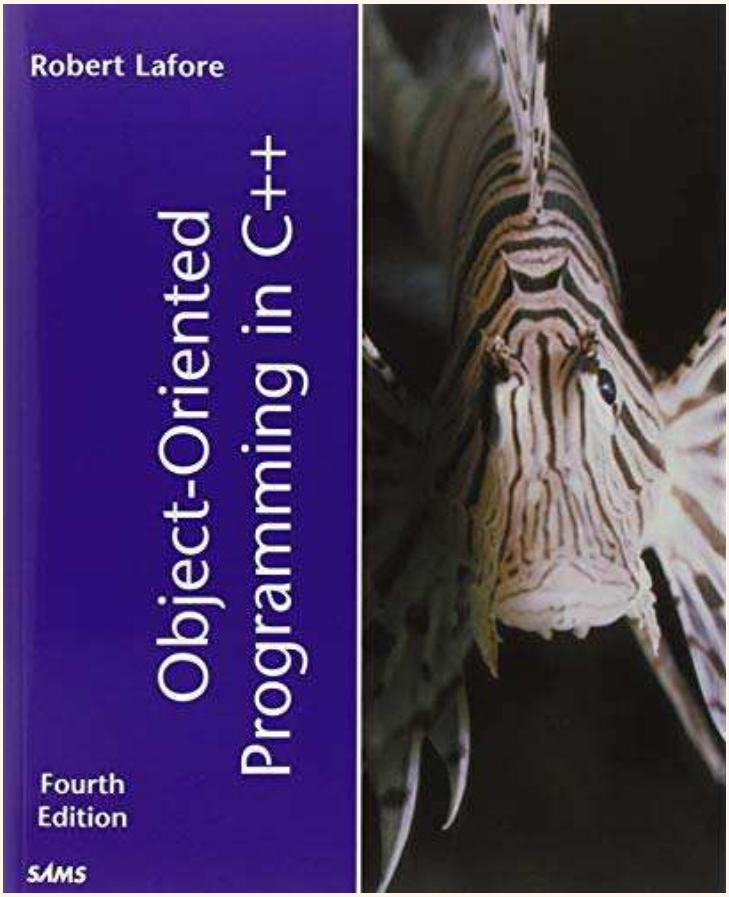


Chapter 8

Operator Overloading

Animated Version

Topics

- Overloading Unary Operators
- Overloading Binary Operators
- Data Conversion
- UML Class Diagrams
- Pitfalls of Operator Overloading and
- Conversion
- Keywords explicit and mutable

Introduction

- Operator overloading is one of the most exciting features of object-oriented programming.
- It can transform complex, obscure program listings into intuitively obvious ones.

```
d3.addobjects(d1, d2);
```

or the similar but equally obscure

```
d3 = d1.addobjects(d2);
```

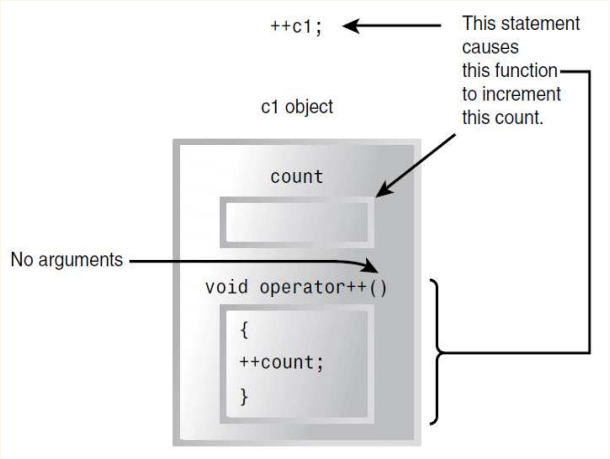
can be changed to the much more readable

```
d3 = d1 + d2;
```

- *Operator overloading* refers to giving the normal C++ operators, such as + , * , <= , and += , additional meanings when they are applied to user-defined data types.

Overloading Unary Operators

- Unary operators: increment and decrement operators ++ and -- , and the unary minus
- The operator Keyword
- Operator Arguments




```
// countpp1.cpp
// increment counter variable with ++ operator
#include <iostream>
using namespace std;
class Counter
   {
   private:
      unsigned int count;              //count
   public:
      Counter() : count(0)             //constructor
         {  }
      unsigned int get_count()         //return count
         { return count; }
      void operator ++ ()              //increment (prefix)
         {
         ++count;
         }
   };
//////////////////
int main()
   {
   Counter c1, c2;
    initialize

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

   ++c1;                 //increment c1
   ++c2;                 //increment c2
   ++c2;                 //increment c2

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

Output:

```
c1=0    <------ counts are initially 0
c2=0
c1=1    <------ incremented once
c2=2    <------ incremented twice
```

Overloading Unary Operators (2)

- Operator Return Values
- Nameless Temporary Objects
  - creates an object with no name

```
Counter operator ++ ()
   {
   ++count;
   return Counter(count);
   }
```

```
// countpp2.cpp
// increment counter variable with ++ operator, return value
#include <iostream>
using namespace std;
class Counter
   {
   private:
      unsigned int count;       //count
   public:
      Counter() : count(0)      //constructor
         {  }
      unsigned int get_count() //return count
         { return count; }
      Counter operator ++ ()    //increment count
         {
         ++count;               //increment count
         Counter temp;          //make a temporary Counter
         temp.count = count;//give it same value as this obj
         return temp;           //return the copy
         }
   };
int main()
   {
   Counter c1, c2;

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

   ++c1;                                  //c1=1
   c2 = ++c1;                             //c1=2, c2=2

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

Output:
c1=0
c2=0
c1=2
c2=2

- Prefix: ++c
- Postfix: c++

Overloading Unary Operators (3)

- Postfix Notation

```
// postfix.cpp
// overloaded ++ operator in both prefix and postfix
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;          //count
   public:
      Counter() : count(0)         //constructor  no args
         {  }
      Counter(int c) : count(c)    //constructor, one arg
         {  }
      unsigned int get_count() const //return count
         { return count; }

      Counter operator ++ ()   //increment count (prefix)
         {                    //increment count, then return
         return Counter(++count); //an unnamed temporary object
         }                    //initialized to this count

      Counter operator ++ (int) //increment count (postfix)
         {                    //return an unnamed temporary
         return Counter(count++); //object initialized to this
         }                    //count, then increment count
   };
```

```
////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                        //c1=0, c2=0

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

   ++c1;                                  //c1=1
   c2 = ++c1;                             //c1=2, c2=2
     (prefix)

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

   c2 = c1++;                          //c1=3, c2=2 (postfix)

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

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

- This `int` isn't really an argument, and it doesn't mean integer. It's simply a signal to the compiler to create the postfix version of the operator.

Overloading Binary Operators

- Arithmetic Operators

```
dist3.add_dist(dist1, dist2);
```

```
dist3 = dist1 + dist2;
```

```
// englplus.cpp
// overloaded '+' operator adds two Distances
#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 operator + ( Distance ) const;  //add 2
     distances
   };
//--------------------------------------------------------------
```

```
                                  //add this distance to d2
Distance Distance::operator + (Distance d2) const  //return 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,
      {                           //then decrease inches
      i -= 12.0;                  //by 12.0 and
      f++;                        //increase feet by 1
      }                           //return a temporary Distance
   return Distance(f,i);          //initialized to sum
   }
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3, dist4;   //define distances
   dist1.getdist();                //get dist1 from user

   Distance dist2(11, 6.25);       //define, initialize dist2

   dist3 = dist1 + dist2;          //single '+' operator

   dist4 = dist1 + dist2 + dist3;  //multiple '+' operators
                                   //display all lengths
   cout << "dist1 = ";  dist1.showdist(); cout << endl;
   cout << "dist2 = ";  dist2.showdist(); cout << endl;
   cout << "dist3 = ";  dist3.showdist(); cout << endl;
   cout << "dist4 = ";  dist4.showdist(); cout << endl;
   return 0;
   }
```


Overloading Binary Operators

- Arithmetic Operators

```
dist3.add_dist(dist1, dist2);
```

```
dist3 = dist1 + dist2;
```

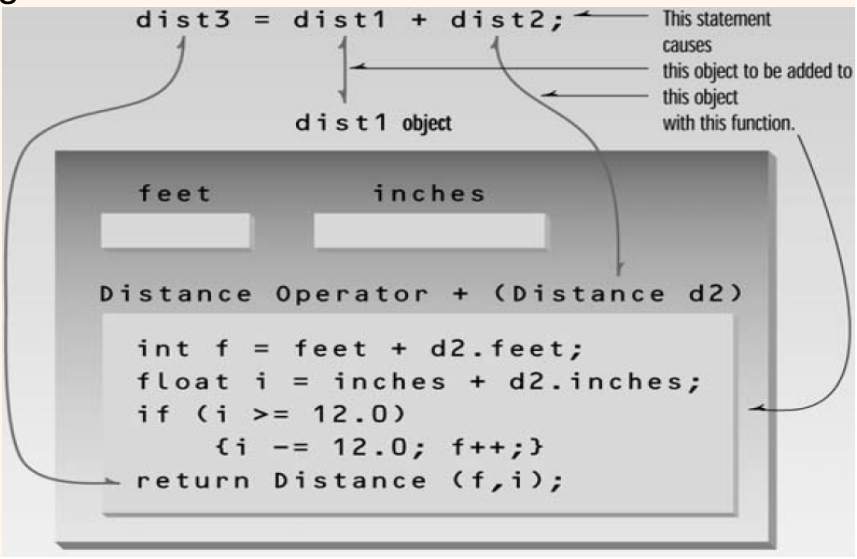
```
// englplus.cpp
// overloaded '+' operator adds two Distances
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                  //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                      //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 operator + ( Distance ) const;  //add 2
     distances
   };
//--------------------------------------------------------------
```

```
                                  //add this distance to d2
Distance Distance::operator + (Distance d2) const  //return 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,
      {                           //then decrease inches
      i -= 12.0;                  //by 12.0 and
      f++;                        //increase feet by 1
      }                           //return a temporary Distance
   return Distance(f,i);          //initialized to sum
   }
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist3, dist4;   //define distances
   dist1.getdist();                //get dist1 from user

   Distance dist2(11, 6.25);       //define, initialize dist2

   dist3 = dist1 + dist2;          //single '+' operator

   dist4 = dist1 + dist2 + dist3;  //multiple '+' operators
                                   //display all lengths
   cout << "dist1 = ";  dist1.showdist(); cout << endl;
   cout << "dist2 = ";  dist2.showdist(); cout << endl;
   cout << "dist3 = ";  dist3.showdist(); cout << endl;
   cout << "dist4 = ";  dist4.showdist(); cout << endl;
   return 0;
   }
```

```
Enter feet: 10
Enter inches: 6.5

dist1 = 10'-6.5"     <------ from user
dist2 = 11'-6.25"    <------ initialized in program
dist3 = 22'-0.75"    <------ dist1+dist2
dist4 = 44'-1.5"     <------ dist1+dist2+dist3
```

Overloading Binary Operators (2)

- Arithmetic Operators …
  - The argument on the left side of the operator (dist1 in this case) is the object of which the operator is a member.
  - The object on the right side of the operator (dist2) must be furnished as an argument to the operator.
  - The operator returns a value, which can be assigned or used in other ways; in this case it is assigned to dist3.



Overloading Binary Operators (3)

- Concatenating Strings

```
// strplus.cpp
// overloaded '+' operator concatenates strings
#include <iostream>
using namespace std;
#include <string.h>      //for strcpy(), strcat()
#include <stdlib.h>      //for exit()
////////////////////////////////////////////////////////////
/
class String             //user-defined string type
   {
   private:
      enum { SZ=80 };               //size of String objects
      char str[SZ];                 //holds a string
   public:
      String()                      //constructor, no args
         { strcpy(str, ""); }
      String( char s[] )            //constructor, one arg
         { strcpy(str, s); }
      void display() const          //display the String
         { cout << str; }
      String operator + (String ss) const  //add Strings
         {
         String temp;               //make a temporary String
         if( strlen(str) + strlen(ss.str) < SZ )
            {
            strcpy(temp.str, str);   //copy this string to temp
            strcat(temp.str, ss.str); //add the argument string
            }
         else
            { cout << "\nString overflow"; exit(1); }
         return temp;               //return temp String
         }
   };
```

```
////////////////////////////////////////////////////////////
int main()
   {
   String s1 = "\nMerry Christmas!  ";   //uses constructor 2
   String s2 = "Happy new year!";        //uses constructor 2
   String s3;                            //uses constructor 1

   s1.display();                         //display strings
   s2.display();
   s3.display();

   s3 = s1 + s2;                         //add s2 to s1,
                                         //   assign to s3
   s3.display();                         //display s3
   cout << endl;
   return 0;
   }
```

```
Merry Christmas!  Happy new year!      <------ s1, s2, and s3 (empty)
Merry Christmas!  Happy new year!      <------ s3 after concatenation
```

Overloading Binary Operators (4)

- Multiple Overloading is permitted.
- Comparison Operators:

```
// engless.cpp
// overloaded '<' operator compares two Distances
#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 << '\"'; }
      bool operator < (Distance) const; //compare distances
   };
```

```
//--------------------------------------------------------
                            //compare this distance with d2
bool Distance::operator < (Distance d2) const//return the sum
   {
   float bf1 = feet + inches/12;
   float bf2 = d2.feet + d2.inches/12;
   return (bf1 < bf2) ? true : false;
   }
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1;                  //define Distance dist1
   dist1.getdist();                 //get dist1 from user

   Distance dist2(6, 2.5);      //define and initialize dist2
                                    //display distances
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();

   if( dist1 < dist2 )              //overloaded '<' operator
      cout << "\ndist1 is less than dist2";
   else
      cout << "\ndist1 is greater than (or equal to) dist2";
   cout << endl;
   return 0;
   }
```

```
Enter feet: 5
Enter inches: 11.5
dist1 = 5'-11.5"
dist2 = 6'-2.5"
dist1 is less than dist2
```


Overloading Binary Operators (5)

- Comparing Strings

```
//strequal.cpp
//overloaded '==' operator compares strings
#include <iostream>
using namespace std;
#include <string.h>      //for strcmp()
////////////////////////////////////////////////////////////
    /
class String             //user-defined string type
   {
   private:
      enum { SZ = 80 };              //size of String objects
      char str[SZ];                  //holds a string
   public:
      String()                       //constructor, no args
         { strcpy(str, ""); }
      String( char s[] )             //constructor, one arg
         { strcpy(str, s); }
      void display() const           //display a String
         { cout << str; }
      void getstr()                  //read a string
         { cin.get(str, SZ); }
      bool operator == (String ss) const  //check for equality
         {
         return ( strcmp(str, ss.str)==0 ) ? true : false;
         }
   };
```

```
////////////////////////////////////////////////////////////
int main()
   {
   String s1 = "yes";
   String s2 = "no";
   String s3;

   cout << "\nEnter 'yes' or 'no': ";
   s3.getstr();                          //get String from user

   if(s3==s1)                            //compare with "yes"
      cout << "You typed yes\n";
   else if(s3==s2)                       //compare with "no"
      cout << "You typed no\n";
   else
      cout << "You didn't follow instructions\n";
   return 0;
   }
```

```
Enter 'yes' or 'no': yes
You typed yes
```

Overloading Binary Operators (6)

- Arithmetic Assignment Operators

```
// englpleq.cpp
// overloaded '+=' assignment operator
#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 << '\"'; }
      void operator += ( Distance );
   };
```

```
//----------------------------------------------------------
                                    //add distance to this one
void Distance::operator += (Distance d2)
   {
   feet += d2.feet;                 //add the feet
   inches += d2.inches;             //add the inches
   if(inches >= 12.0)               //if total exceeds 12.0,
      {                             //then decrease inches
      inches -= 12.0;               //by 12.0 and
      feet++;                       //increase feet
      }                             //by 1
   }
////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1;                  //define dist1
   dist1.getdist();                 //get dist1 from user
   cout << "\ndist1 = ";  dist1.showdist();

   Distance dist2(11, 6.25);        //define, initialize dist2
   cout << "\ndist2 = ";  dist2.showdist();

   dist1 += dist2;                  //dist1 = dist1 + dist2
   cout << "\nAfter addition,";
   cout << "\ndist1 = ";  dist1.showdist();
   cout << endl;
   return 0;
   }
```

```
Enter feet: 3
Enter inches: 5.75
dist1 = 3'-5.75"
dist2 = 11'-6.25"
After addition,
dist1 = 15'-0"
```

- the object that takes on the value of the sum is the object of which the function is a member.

Data Conversion

- Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object.
- But what happens when the variables on different sides of the = are of different types?
  - conversions between basic types and user-defined types, and
  - conversions between different user-defined types.
- Conversions Between Basic Types
  - Implicit: compiler automatically handles conversion
    - e.g: from float to double , char to float , and so on.
  - Explicit: force the compiler to convert one type to another
    - use the cast operator e.g:
      - `intvar = static_cast<int>(floatvar);`

Data Conversion (2)

- Conversions Between Objects and Basic Types

```
// englconv.cpp
// conversions: Distance to meters, meters to Distance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      const float MTF;            //meters to feet
      int feet;
      float inches;
   public:                        //constructor (no args)
      Distance() : feet(0), inches(0.0), MTF(3.280833F)
         {  }                     //constructor (one arg)
      Distance(float meters) : MTF(3.280833F)
         {                        //convert meters to Distance
         float fltfeet = MTF * meters;  //convert to float feet
         feet = int(fltfeet);           //feet is integer part
         inches = 12*(fltfeet-feet);    //inches is what's left
         }                        //constructor (two args)
      Distance(int ft, float in) : feet(ft),
                                   inches(in), MTF(3.280833F)
         {  }
      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      operator float() const      //conversion operator
         {                        //converts Distance to meters
         float fracfeet = inches/12;     //convert the inches
         fracfeet += static_cast<float>(feet); //add the feet
         return fracfeet/MTF;            //convert to meters
         }
   };
```

- From Basic to User-Defined

```
////////////////////////////////////////////////////////////
int main()
   {
   float mtrs;
   Distance dist1 = 2.35F;         //uses 1-arg constructor to
                                   //convert meters to Distance
   cout << "\ndist1 = "; dist1.showdist();

   mtrs = static_cast<float>(dist1);//uses conversion operator
                                   //for Distance to meters
   cout << "\ndist1 = " << mtrs << " meters\n";

   Distance dist2(5, 10.25);       //uses 2-arg constructor

   mtrs = dist2;                   //also uses conversion op
   cout << "\ndist2 = " << mtrs << " meters\n";

// dist2 = mtrs;                   //error, = won't convert
   return 0;
   }
```

```
dist1 = 7'-8.51949"      <------ this is 2.35 meters
dist1 = 2.35 meters      <------ this is 7'–8.51949"
dist2 = 1.78435 meters   <------ this is 5'–10.25"
```

- From User-Defined to Basic

Data Conversion (3)

- Conversion Between C-Strings and `String` Objects

```
// strconv.cpp
// convert between ordinary strings and class String
#include <iostream>
using namespace std;
#include <string.h>               //for strcpy(), etc.
////////////////////////////////////////////////////////////
class String                      //user-defined string type
   {
   private:
      enum { SZ = 80 };           //size of all String objects
      char str[SZ];               //holds a C-string
   public:
      String()                    //no-arg constructor
         { str[0] = '\0'; }
      String( char s[] )          //1-arg constructor
         { strcpy(str, s); }      //   convert C-string to String
      void display() const        //display the String
         { cout << str; }
      operator char*()            //conversion operator
         { return str; }          //convert String to C-string
   };
////////////////////////////////////////////////////////////
int main()
   {
   String s1;              //use no-arg constructor
                           //create and initialize C-string
   char xstr[] = "Joyeux Noel! ";

   s1 = xstr;                    //use 1-arg constructor
                           //   to convert C-string to String
   s1.display();                 //display String

   String s2 = "Bonne Annee!";   //uses 1-arg constructor
                                 //to initialize String
   cout << static_cast<char*>(s2); //use conversion operator
     cout << endl          //to convert String to C-string
     return 0;             //before sending to << op
   }
```

- From C-Strings to `String` Objects
- From `String` Objects to C-Strings

Data Conversion (4)

- Conversions Between Objects of Different Classes

- Two Kinds of Time

TABLE 8.1 12-Hour and 24-Hour Time

12-Hour Time	*24-Hour Time*
12:00 a.m. (midnight)	00:00
12:01 a.m.	00:01
1:00 a.m.	01:00
6:00 a.m.	06:00
11:59 a.m	11:59
12:00 p.m. (noon)	12:00
12:01 p.m.	12:01
6:00 p.m.	18:00
11:59 p.m.	23:59

```
//times1.cpp
//converts from time24 to time12 using operator in time24
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////
    /
class time12
   {
   private:
      bool pm;                      //true = pm, false = am
      int hrs;                      //1 to 12
      int mins;                     //0 to 59
   public:                          //no-arg constructor
      time12() : pm(true), hrs(0), mins(0)
         {  }
                                    //3-arg constructor
      time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m)
         {  }
      void display() const          //format: 11:59 p.m.
         {
         cout << hrs << ':';
         if(mins < 10)
            cout << '0';            //extra zero for "01"
         cout << mins << ' ';
         string am_pm = pm ? "p.m." : "a.m.";
         cout << am_pm;
         }
   };
```


Data Conversion (5)

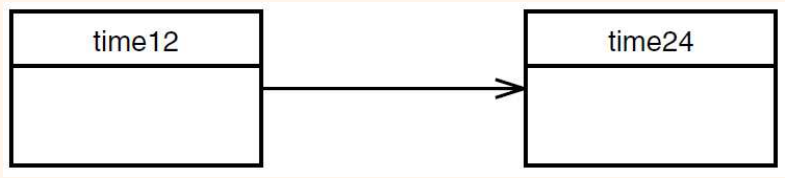
```
////////////////////////////////////////////////////////////
class time24
   {
   private:
      int hours;                    //0 to 23
      int minutes;                  //0 to 59
      int seconds;                  //0 to 59
   public:                          //no-arg constructor
      time24() : hours(0), minutes(0), seconds(0) {  }
      time24(int h, int m, int s) :  //3-arg constructor
            hours(h), minutes(m), seconds(s)    {  }
      void display() const {          //format: 23:15:01
         if(hours < 10)    cout << '0';
                cout << hours << ':';
         if(minutes < 10)  cout << '0';
                cout << minutes << ':';
         if(seconds < 10)  cout << '0';
                cout << seconds;   }
      operator time12() const;      //conversion operator
   };
//--------------------------------------------------------------
time24::operator time12() const           //conversion operator
   {
   int hrs24 = hours;
   bool pm = hours < 12 ? false : true;   //find am/pm
                                          //round secs
   int roundMins = seconds < 30 ? minutes : minutes+1;
   if(roundMins == 60)                    //carry mins?
      {
      roundMins=0;
      ++hrs24;
      if(hrs24 == 12 || hrs24 == 24)      //carry hrs?
         pm = (pm==true) ? false : true;  //toggle am/pm
      }
   int hrs12 = (hrs24 < 13) ? hrs24 : hrs24-12;
   if(hrs12==0)                           //00 is 12 a.m.
      { hrs12=12; pm=false; }
   return time12(pm, hrs12, roundMins);
   }
```

```
////////////////////////////////////////////////////////////
int main()
   {
   int h, m, s;

   while(true)
      {                             //get 24-hr time from user
      cout << "Enter 24-hour time: \n";
      cout << "   Hours (0 to 23): "; cin >> h;
      if(h > 23)                    //quit if hours > 23
         return(1);
      cout << "   Minutes: ";  cin >> m;
      cout << "   Seconds: ";  cin >> s;

      time24 t24(h, m, s);          //make a time24
      cout << "You entered: ";      //display the time24
      t24.display();

      time12 t12 = t24;           //convert time24 to time12

      cout << "\n12-hour time: "; //display equivalent time12
      t12.display();
      cout << "\n\n";
      }
   return 0;
   }
```

```
Enter 24-hour time:
   Hours (0 to 23): 17
   Minutes: 59
   Seconds: 45
You entered: 17:59:45
12-hour time: 6:00 p.m.
```

UML Class Diagrams

- Offers a new way of looking at object-oriented programs



- Associations:
  - time12 is associatd with class time24 because we are converting objects of one class into objects of the other. Here, an object of the `time12` class, called `t12` , calls the conversion routine operator `time12()` in the object `t24` of the `time24` class.
  - Drivers are related to cars, books are related to libraries, race horses are related to race tracks -> Associations.
  - A class association actually implies that objects of the classes, rather than the classes themselves, have some kind of relationship.
    - Typically, two classes are associated if an object of one class calls a member function (an operation) of an object of the other class. An association might also exist if an attribute of one class is an object of the other class.
- Navigability: Arrow directed to `time24` Because `time12` calls `time24`

Pitfalls of Operator Overloading and Conversion

- Operator overload can make code more intuitive and readable. It can also make code more obscure and hard to understand.
- Guidelines:
  - Use Similar Meanings e.g: + should not deduct
  - Use Similar Syntax
    - If you overload one arithmetic operator, you may for consistency want to overload all of them. This will prevent confusion.
  - Show Restraint:
    - overloaded operators should be intuitive. Use overloaded operators sparingly, and only when the usage is obvious. When in doubt, use a function instead of an overloaded operator, since a function name can state its own purpose.
  - Avoid Ambiguity: avoid doing the same conversion in more than one way.
  - Not All Operators Can Be Overloaded:
    - member access or dot operator (.), the scope resolution operator (::), and the conditional operator (?:).
    - Also, the pointer-to-member operator (->),cannot be overloaded.
    - Only existing operators can be overloaded.

Keywords `explicit` and `mutable`

- Preventing Conversions with **`explicit`**
  - You should actively discourage any conversion that you don't want to prevents unpleasant surprises.
  - Placing `explicit` just before the declaration of a one-argument constructor prevents overloading it to obstruct implicit conversion.

```
//explicit.cpp
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                  //English Distance class
   {
   private:
      const float MTF;          //meters to feet
      int feet;
      float inches;
   public:                      //no-args constructor
      Distance() : feet(0), inches(0.0), MTF(3.280833F)
         {  }
                                //EXPLICIT one-arg constructor
      explicit Distance(float meters) : MTF(3.280833F)
         {
         float fltfeet = MTF * meters;
         feet = int(fltfeet);
         inches = 12*(fltfeet-feet);
         }
      void showdist()           //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
```

```
////////////////////////////////////////////////////////////
int main()
   {
   void fancyDist(Distance);    //declaration
   Distance dist1(2.35F);       //uses 1-arg constructor to
                                //convert meters to Distance

// Distance dist1 = 2.35F;      //ERROR if ctor is explicit
   cout << "\ndist1 = "; dist1.showdist();

   float mtrs = 3.0F;
   cout << "\ndist1 ";
// fancyDist(mtrs);             //ERROR if ctor is explicit

   return 0;
   }
//------------------------------------------------------------
void fancyDist(Distance d)
   {
   cout << "(in feet and inches) = ";
   d.showdist();
   cout << endl;
   }
```


Keywords `explicit` and `mutable`

- Changing **`const`** Object Data Using **`mutable`**
  - It's like what happens when your bank sells your mortgage to another bank; all the terms of the mortgage are the same, but the owner is different.

```
//mutable.cpp
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////
class scrollbar
   {
   private:
      int size;                 //related to constness
      mutable string owner;     //not relevent to constness
   public:
      scrollbar(int sz, string own) : size(sz), owner(own)
         {  }
      void setSize(int sz)              //changes size
         { size = sz; }
      void setOwner(string own) const   //changes owner
         { owner = own; }
      int getSize() const               //returns size
         { return size; }
      string getOwner() const           //returns owner
         { return owner; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   const scrollbar sbar(60, "Window1");

// sbar.setSize(100);           //can't do this to const obj
   sbar.setOwner("Window2");      //this is OK
                                  //these are OK too
   cout << sbar.getSize() << ", " << sbar.getOwner() << endl;
   return 0;
   }
```

Summary (1)

- In this chapter we've seen how the normal C++ operators can be given new meanings when applied to user-defined data types.
  - The keyword operator is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.
- Closely related to operator overloading is the issue of type conversion. Some conversions take place between user-defined types and basic types. Two approaches are used in such conversions:
  - A one-argument constructor changes a basic type to a user-defined type, and
  - a conversion operator converts a user-defined type to a basic type.
  - When one user-defined type is converted to another, either approach can be used.

TABLE 8.2 Type Conversions

	Routine in Destination	*Routine in Source*
Basic to basic	**(Built-In Conversion Operators)**	
Basic to class	Constructor	N/A
Class to basic	N/A	Conversion operator
Class to class	Constructor	Conversion operator

Summary (2)

- A constructor given the keyword explicit cannot be used in implicit data conversion situations. A data member given the keyword mutable can be changed, even if its object is `const`.
- UML class diagrams show classes and relationships between classes. An association represents a conceptual relationship between the real-world objects that the program's classes represent. Associations can have a direction from one class to another; this is called navigability.