UCLA Stat 130D
Statistical Computing and Visualization in C++

Instructor: **Ivo Dinov, Asst. Prof. in Statistics / Neurology**

University of California, Los Angeles, Winter 2007

http://www.stat.ucla.edu/~dinov/courses_students.html

Precedence Rules

First: *unary* operators (e.g., -5)	+, -, ++, --	*Highest Precedence (done first)*
Next: *binary* arithmetic operators	*, /, %	↓
Next: *binary* arithmetic operators	+, -	
Next: *Relational* operators	< > <= >=	
Next: *Comparison* operators	== !=	
Next: *Boolean* operator	&&	
Next: *Boolean* operator	||	*Lowest Precedence (done last)*

Short-cut Evaluation

In evaluating a bool valued expression, some languages, including C++, evaluate the first expression. If, based on the value of this expression, the value of the expression can be decided, the second of the two expressions is not evaluated. This is called short-cut evaluation. C and C++ use short-cut evaluation.

The alternative of evaluating both expressions is called complete evaluation. Standard *Pascal* uses Compete Evaluation.

Short-cut evaluation is important where evaluating the second expression would be illegal.

Example:

```
if ( ( kids != 0 ) && ( ( pieces / kids ) >= 2) )
    cout << “each child may have two pieces.” << endl;
```

PITFALL: bool Expressions Convert to int values(1 of 2)

Suppose you want to use a bool valued expression to control an if-else. You want a bool valued expression to be *true* provided that a value of a variable *time* of type *int* is not greater than the value of a variable *limit*.

You might (but shouldn’t) write:

```
if (!time > limit)        ← For what we want, this is WRONG.
   Some_Statements;
else
   Some_Other_Statements;
```

Precedence requires that the expression (!time > limit) be evaluated
(!time) > limit

If *time* has value 36, this is the equivalent to “not 36”. This is at best unintuitive. C++ converts *time* to a bool value. Nonzero converts to *true*, zero converts to *false*.

Now things seem worse. We have
false > limit

This is also not intuitive.

C++ converts the bool value *false* to 0, which, when compared to the value of *limit*, which is 60, will have the value *false*.

PITFALL: bool Expressions Convert to int values (2 of 2)

The correct way to do this is

```
if ( !(time > limit) )
   Some_Statements;
else
   Some_Other_Statements;
```

An alternative way to write this is

```
if ( time <= limit )
   Some_Statements;
else
   Some_Other_Statements;
```

You can almost always avoid the ! operator and some programmers advocate avoiding the ! operator altogether. Excessive use of the ! operator can make your code hard to read. Clarity of expression is the important thing.

Function That Return a Boolean Value

- **You can have a function that returns a bool value just like a function can return any other type.**
- **A call to such a function can be used to control a loop, an if-else or anywhere else a bool valued expression can be used.**
- **Example:**

```
bool appropriate( int rate)
{
   return ((rate >=10) && (rate < 20)) || (rate == 0));
}
if (appropriate(rate))   // EQ. if (appropriate(rate) == true)
{
   ….
}
```

Enumeration Types(1 of 2)

An **Enumeration type** is a type whose values are defined by a list of constants of type *int*. An enumeration is like a list of declared constants except it has a type name associated with it.

Example:

```
enum MonthLength { JAN_LEN = 31, FEB_LEN = 28,
                   MAR_LEN = 31, APR_LEN = 30,
                   . . .
                   NOV_LEN = 30, DEC_LEN = 31};
```

Note that two or more identifiers can receive the same value.

Enumeration Types (2 of 2)

If you do not specify values, the first receives 0, the second 1, and so on.

Enum Direction { NORTH, SOUTH, EAST, WEST};

NORTH gets 0, SOUTH gets 1, EAST gets 2, WEST gets 3.

If you set a value for some identifier and don't set values for later identifiers, the later identifiers values are assigned values that increase by 1 each time.

- Example of enum values:
  enum MyEnum { ONE = 17, TWO, THREE, FOUR = -3, FIVE};

  ONE gets 17, TWO gets 18,
  THREE gets 19, FOUR gets -3,
  FIVE gets -2

Multi-way Branches

- A programming language construct that chooses between alternative action is called a **branching mechanism**.
- An **if-else** statement chooses between two alternatives.
- In this section we examine language constructs that choose from among more than two alternatives.

Multi-way Branches

Nested Statements

- Nested **if-else** statements can choose from among more that two alternatives.
- An if statement contains smaller statements within them, even another if statement.
- When an if statement contains another if statement you should indent each level of substatements.

An `if-else` Statement within An `if` Statement

```
if (count > 0)
    if (score > 5)
        cout << "count > 0 and score > 5\n";
    else
        cout << "count < 0 and score <= 5\n";
```

Programming Tip

Use Braces in Nested Statements

Which `if` to which the `else` is bound is unclear from reading this code fragment:

In the nested if-else the statement

```
if (fuel_gauge_reading < 0.75)
  if (fuel_gauge_reading > 0.25)
     cout << "Fuel very low. Caution \n";
  else
     cout << "Fuel over 3/4. Don't stop now!\n":
```

- The C++ language **requires** that the compiler bind the **else** to the **closest *unbound* if**.
- Our confusion is known as the **dangling else problem**.
- The fix is to use curly braces to ensure clarity

```
//          The Importance of Braces
#include <iostream>
using namespace std;
int main( )
{
   double fuel_gauge_reading;
   cout << "Enter fuel gauge reading: ";
   cin >> fuel_gauge_reading;
   cout << "First with braces:\n";
   if (fuel_gauge_reading < 0.75)
   {
      if (fuel_gauge_reading < 0.25)
         cout << "Fuel very low. Caution!\n";
   }
   else
   {
      cout << "Fuel over 3/4. Don't stop now!\n";
   }
   cout << "Now without braces:\n";
   if (fuel_gauge_reading < 0.75)
      if (fuel_gauge_reading < 0.25)
         cout << "Fuel very low. Caution!\n";
   else
      cout << "Fuel over 3/4. Don't stop now!\n";
   return 0;
}
```

Multiway `if-else` Statements

An if-else statement provides a two-way choice of actions.

Given a mutually exclusive list of conditions and corresponding actions, multiway selection can be implemented as:

```
if (guess > number)
   cout << "Too high.";
else if (guess < number)
   cout << "Too low.";
else if (guess == number)
   cout << "Correct!";
```

Note the difference in the indentation pattern from what we have seen. We advocate this pattern for multiway selection.

Multiway if-else Statement (1 of 2)

```
//       Program to compute state income tax.
#include <iostream>
using namespace std;
double tax(int net_income);
//   Precondition: The formal parameter net_income is net income, rounded
//          to a whole number of dollars.
//   Postcondition: Returns the amount of state income tax due
//          computed as follows: no tax on income up to $15,000; 5% on income
//          between $15,001 and $25,000 plus 10% on income over $25,000.
int main( )
{
   int net_income;
   double tax_bill;
   cout << "Enter net income (rounded to whole dollars) $";
   cin >> net_income;
   tax_bill = tax(net_income);
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
   cout << "Net income = $" << net_income << endl
      << "Tax bill = $" << tax_bill << endl;
   return 0;
}
```

Multiway `if-else` Statement (2 of 2)

```
double tax(int net_income)
{
   double five_percent_tax, ten_percent_tax;
   if (net_income <= 15000)
      return 0;
   else if ((net_income > 15000) && (net_income <= 25000))
         //      return 5% of amount over $15,000
      return (0.05*(net_income - 15000));
   else     //      net_income > $25,000
   {
      //      five_percent_tax = 5% of income from $15,000 to $25,000.
      five_percent_tax = 0.05*10000;
      //      ten_percent_tax = 10% of income over $25,000.
      ten_percent_tax = 0.10*(net_income - 25000);
      return (five_percent_tax + ten_percent_tax);
   }
}
```

The *Switch* Statement (1 of 2)

We have used nested if-else statements for multiway branches.

The switch statement is another C++ construct for multiway branches.

Syntax:

```
switch (Controlling_Expression)
{
   case Constant_1: Statement_Sequence_1;   break;
   case Constant_1: Statement_Sequence_2;   break;
      .
      .
      .
   case Constant_n: Statement_Sequence_N;   break;
   default: Default_Statement_Sequence;     break;
}
```

The *Switch* Statement (2 of 2)

Example:

```
switch (vehicle_class)
{
   case 1:
         cout <<"passenger car" << endl;
         toll = 0.50;
         break;
   case 2:
         cout << "Bus: ";
         toll = 1.50;
         break;
   case 3:
         cout << "Truck: ";
         toll = 2.00;
         break;
   default:
         cout << "Unknown vehicle class: ";
         break;
}
```

The break exits the switch statement. Without it, the next case executes, making the passenger car pay $1.50.

A switch Statement (1 of 3)

```
//              Program to illustrate the switch-statement.
#include <iostream>
using namespace std;
int main( )
{
   char grade;
   cout << "Enter your midterm grade and press return: ";
   cin >> grade;
   switch (grade)
   {
      case 'A':
         cout << "Excellent. "
               << "You need not take the final.\n";
         break;
      case 'B':
         cout << "Very good. ";
         grade = 'A';
         cout << "Your midterm grade now is "
             << grade << endl;
         break;
```

19

A switch Statement (1 of 3)

```
//       Program to illustrate the switch-statement.

      case 'C':
         cout << "Passing.\n";
         break;
      case 'D':
      case 'F':
         cout << "Not good. "
               << "Go study.\n";
         break;
      default:
         cout << "That is not a possible grade.\n";
   }
   cout << "End of program.\n";
   return 0;
}
```

20

A switch Statement (1 of 3)

// Program to illustrate the switch-statement.

Sample Dialogue 1

```
Enter your midterm grade and press return: A
Excellent. You need not take the final.
End of Program.
```

Sample Dialogue 2

```
Enter your midterm grade and press return: B
Very good. Your midterm grade is now A.
End of Program.
```

Sample Dialogue 3

```
Enter your midterm grade and press return: D
Not good. Go study.
End of Program.
```

Sample Dialogue 4

```
Enter your midterm grade and press return: E
That is not a possible grade.
End of Program.
```

21

Pitfall: Forgetting a break in a switch statement.

Note that it is not an error to omit the break statement however,

If you forget the break statement, execution will proceed to the next case.

If we were to omit the break in case 1: of the example in the preceding slide, case 2 for a bus class vehicle will execute, making a passenger car pay $1.50 which is the toll for a bus.

22

Programming Tip

Use Function Calls in Branching Statements.

- **Multiway branching statements allow choice of several statements each branch.**
- **This can cause code to be hard to read, track bugs, test.**
- **To fix this, use single line function calls in the cases of the switch statement, or in the “true” clauses of a multiway if-else statement.**

23

Block with a Local Variable (1 of 3)

```
//      Program to compute bill for either a wholesale or a retail purchase.
#include <iostream>
using namespace std;
const double TAX_RATE = 0.05; //5% sales tax.
int main( )
{
   char sale_type;
   int number;
   double price, total;
   cout << "Enter price $";
   cin >> price;
   cout << "Enter number purchased: ";
   cin >> number;
   cout << "Type W if this is a wholesale purchase.\n"
        << "Type R if this is a retail purchase.\n"
        << "Then press return.\n";
   cin >> sale_type;
```

24

Block with a Local Variable (2 of 3)

```
if ((sale_type == 'W') || (sale_type == 'w'))
  {
     total = price * number;
  }
  else if ((sale_type == 'R') || (sale_type == 'r'))
  {
     double subtotal;  <-------------- Local to the block
     subtotal = price * number;
     total = subtotal + subtotal * TAX_RATE;
  }
  else
  {
     cout << "Error in input.\n";
  }
```

25

Block with a Local Variable (3 of 3)

```
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
   cout << number << " items at $" << price << endl;
   cout << "Total Bill = $" << total;
   if ((sale_type == 'R') || (sale_type == 'r'))
      cout << " including sales tax.\n";

   return 0;
}
```

26

Blocks

<u>Definitions</u>

- **A compound statement that has declarations is called a block or a local block.**
- **Variables declared within a block are said to be local to the block, or to have the block as their scope.**
- **There is no standard usage for a block that is not the body of a function. We introduce the term statement block for such a block.**

<u>Scope Rule for Nested Blocks</u>

- **A variable declared in the outer block and again defined with the same name in the inner block of nested blocks are two different variables.** **How does the compiler/loader distinguish between them?**

 - The variable defined in the inner block block is *only* defined in the inner block.

 - The other variable exists *only* in the outer block.

 - The two variables are distinct, changes made to one have no effect on the other.

27

Pitfall: Inadvertent local variables

- **A variable declared within a pair of braces, { } is local to that block.**
- **If you want a variable to be available outside the braces, you must declare it outside the braces.**

28

Increment and Decrement Operators Revisited

- The expressions n-- and --n have values as well.
- The expression n-- returns the value of n *before* to decrementing, then decrements the value of n.

  --n decrements the value of n, then returns the decremented value.

- Example:

```
int k, j;
k = 2;
j = k--;  // j gets2, k's value is decremented to1.
k = 2;
j = --k;  // k's value is decremented to 1, j gets 1.
```

29

The for Statement

- Syntax:

```
for (Initialization_Action; Boolean_Action; UpdateAction)
 {  Body_Statements;  }
```

Example:

```
for (number = 100; number >= 0; number--)
   {   cout << number
             << " Bottles of beer on the shelf.\n";
   }
```

Equivalent Syntax:

```
number = 100
while (number >= 0)
{
    Body_Statement;
    number--;
}
```

30

PITFALL
Extra Semicolon in a for statement

```
for (int count = 1; count < 10; count++);    Causes a problem.
    cout << “Hello\n”;                        This is NOT an error.
```

The problem is that the null statement between the close parenthesis and the semicolon gets executed 10 times, then the cout << “Hello\n”; which was intended to be executed 10 times is executed only once.

What Kind of Loop to Use

The best answer to this is WAIT until design is completed.

Design using *pseudocode*, then as the pseudocode is translated into C++, this decision is easy.

E.g., “For every grade-score square the grade, print to std out and save to the Output file stream”

A loop that involves a numeric control variable that changes by a fixed amount each iteration, then a for loop is indicated.

Most other situations a while loop may be appropriate, unless you know that the loop is to be executed once regardless, then the do-while is appropriate.

The break Statement

- We saw that the break statement exits in switch.
- The break statement also may be used in a loop to terminate the loop.
- In nested loops, a break statement in an inner loop ends only the inner loop.

A break Statement

```
 //     Sums a list of 10 negative numbers.

#include <iostream>
using namespace std;
int main( )
{  int number, sum = 0, count = 0;
   cout << "Enter 10 negative numbers:\n";
   while (++count <= 10)
   {
       cin >> number;
       if (number >= 0)
       {     cout << "ERROR: positive number"
                  << " or zero was entered as the\n"
                  << count << "th number! Input ends "
                  << "with the " << count << "th number.\n"
                  << count << "th number was not added in.\n";
           break;
       }
       sum = sum + number;
   }
   cout << sum << " is the sum of the first "
        << (count - 1) << " numbers.\n";
   return 0;
}
```

Ending a Loop (1 of 5)

There are four common methods for terminating a loop:

1) List Headed by size (a known number of iterations)
2) Ask before iterating
3) List ended by a sentinel value
4) Running out of input (End of file).

Ending a Loop (2 of 5)

List Headed by size (a known number of iterations)

- If the program can determine how many interations before hand, the method is known as list headed by size.
- Example:

```
int sum = 0;
for(int count = 1; count <= this_many; count++)
{
   cin >> next;
   sum = sum + next;
}
```

Ending a Loop (3 of 5)

Example: Ask before iterating

```
sum = 0;
cout << “Are there numbers in the list?
        << “Y <cr> for yes, N <cr> for No.\n”;
char ans;
cin >> ans;
while ((ans == ‘Y’ || ans == ‘y’))
{   cout << “Enter number: “;
    cin >> number;
    sum = sum + number;
    cout << “Are thern any more numbers?”
            << “ Y <cr> for yes, N <cr> for No.\n”;
    cin >> ans;
}
```

Ending a Loop (4 of 5)

List ended by sentinel value

Here a non-data value is to be detected to end input.

```
cout << “Enter a list of nonnegative integers. \n”
        << “Terminate the list with a negative number.\n”;
sum = 0;
cin >> number;
while (number >= 0 )
{
    sum = sum + number;
    cin >> number;
}
```

Ending a Loop (5 of 5)

A list of more general techniques for ending a loop are:

1) Count controlled loops

2) Ask before iterating.

3) Exit on a flag condition.

We have seen #1 and #2.

The exit on sentinel value is a special case of technique #3, exit on a flag condition.

A flag is a variable that changes value to signal that some event has occurred.

Nested Loops

A loop body may contain statements of any kind, including another loop. A loop that contains another loop is called a nested loop. This is a very frequent programming practice when designing complex software packages.

The next two sets of slides contain examples of nested loops.

They do the same task, but the first is easier to understand than than the second.

Nicely Nested Loops (1 of 4)

```
//      Determines the total number of green-necked vulture eggs
//      counted by all conservationists in the conservation district.
#include <iostream>
using namespace std;
void instructions( );
void get_one_total(int& total);
//      Precondition: User will enter a list of egg counts
//                  followed by a negative number.
//     Postcondition: total is equal to the sum of all the egg counts.
int main( )
{
    instructions( );
    int number_of_reports;
    cout << "How many conservationist reports are there? ";
    cin >> number_of_reports;
```

Nicely Nested Loops (2 of 4)

```
int grand_total = 0, subtotal, count;
 for (count = 1; count <= number_of_reports; count++)
 {
     cout << endl << "Enter the report of "
            << "conservationist number " << count << endl;
     get_one_total(subtotal);
     cout << "Total egg count for conservationist "
         << " number " << count << " is "
         << subtotal << endl;
     grand_total = grand_total + subtotal;
 }

 cout << endl << "Total egg count for all reports = "
        << grand_total << endl;
 return 0;
}
```

Nicely Nested Loops (3 of 4)

```
//      Uses iostream:
void instructions( )
{
    cout << "This program tallies conservationist reports\n"
         << "on the green-necked vulture.\n"
         << "Each conservationist's report consists of\n"
         << "a list of numbers. Each number is the count of\n"
         << "the eggs observed in one"
         << " green-necked vulture nest.\n"
         << "This program then tallies"
         << " the total number of eggs.\n";
}
```

Nicely Nested Loops (4 of 4)

```
//          Uses iostream:
void get_one_total(int& total)
{
    cout << "Enter the number of eggs in each nest.\n"
         << "Place a negative integer"
         << " at the end of your list.\n";
    total = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
        total = total + next;
        cin >> next;
    }
}
```

Debugging Loops

- **A common loop error is the "off by one" error.**
- **The error is either running the loop one too many times or one too few times.**
- **A second error is an infinite loop.**
- **If your error isn't one of the common errors more sophisticated testing will be required.**
- **You can use a debugger and examine the loop variables or use output statements to print out these variables.**

Testing a Loop

A minimal set of test values is test every loop using values that will result in:

- **zero iterations**
- **one iteration**
- **the maximum number of iterations**
- **one less than the maximum number of iterations.**

Debugging a Very Bad Program

If your program is very bad, throw it out and start over.

This is a very hard thing to do! However, you will have learned much about your problem so starting over will be easier than the first time around.

Tools for Defining ADTs

- **Defining ADT Operations**
  - **Friend Functions**
  - **Implementation of digit_to_int (Optional)**
  - **The const Parameter Modifier**
  - **Constructors for Automatic Type Conversion**
  - **Overloading Unary Operations**
  - **Overloading >> and <<**
- **Separate Compilation**
  - **ADTs Reviewed**
  - **Using #ifndef**
- **Namespaces**
  - **Namespaces and Using Directives**
  - **Creating a Namespace**
  - **Qualifying Names**

Tools for Defining ADTs

Defining ADT Operations

- So far we have implemented ADT operations as members of a class.
- For some operations, it is advantageous to implement the operations for ordinary functions with the ADT class parameters.
- In the next example we implement a DayOfYear class.
- To test for equality, we implement an ordinary function that returns a bool value and takes two arguments of type class DayOfYear.
- The prototype is:

  bool equal(DayOfYear date1, DayOfYear date2);

Equality Function (1 of 4)

```
//  Program to demonstrate the function equal. The class DayOfYear
#include <iostream>
using namespace std;
class DayOfYear
{
public:
    DayOfYear(int the_month, int the_day);
    //  Precondition: the_month and the_day form a possible date.
    //  Initializes the date according to the arguments.
    DayOfYear( );
    //   Initializes the date to January first.
    void input( );
    void output( );
    int get_month( );
    //   Returns the month, 1 for January, 2 for February, etc.
    int get_day( );
    //    Returns the day of the month.
private:
    int month;
    int day;
};
```

Equality Function (2 of 4)

```
bool equal(DayOfYear date1, DayOfYear date2);
//     Precondition: date1 and date2 have values.
//     Postcondition: returns true if date1 and date2
//         represent the same date,  otherwise returns false.
int main( )
{
    DayOfYear today, bach_birthday(3, 21);
    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );
    cout << "J. S. Bach's birthday is ";
    bach_birthday.output( );
    if ( equal(today, bach_birthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
```

Equality Function (3 of 4)

```
bool equal(DayOfYear date1, DayOfYear date2)
{
     return ( date1.get_month( ) == date2.get_month( ) &&
              date1.get_day( ) == date2.get_day( ) );
}
DayOfYear::DayOfYear(int the_month, int the_day)
{
    month = the_month;
    day = the_day;
}
DayOfYear::DayOfYear( )
{
    month = 1;
    day = 1;
}
int DayOfYear::get_month( )
{
    return month;
}
```

Equality Function (4 of 4)

```
int DayOfYear::get_day( )
{
    return day;
}
//     Uses iostream:
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}
//     Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

The Equality Function

- Notice that we could have made this function a member of the class, but equality is a symmetric operation.
- By implementing the function as a “stand-alone” function, we treat the two objects in the same way.
- The problem is that to decide whether two objects are equal, we need access to the data of the objects. A stand-alone function, defined outside a class, must use accessor functions.
- An ordinary function declared as a friend of the class gains the required access.

Friend Functions

- An ordinary function declared as a friend of the class gains the required access.
- The class grants friend status by declaring the function with the *friend* keyword.
- That is, the class grants friend status by placing the keyword friend followed by the prototype of the function in the class.
- A friend function is not a member of the class.
- When a friend function is defined, you do not use the class name and scope resolution operator as you do with members.

Equality Function (1 of 4)

```
//   In this version, the function equal is a friend of the class DayOfYear.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    friend bool equal(DayOfYear date1, DayOfYear date2);
    //   Precondition: date1 and date2 have values.
    //   Postcondition: returns true if date1 and date2
    //          represent the same date; otherwise returns false.

    DayOfYear(int the_month, int the_day);
    //    Precondition: the_month and the_day form a possible date.
    //        Initializes the date according to the arguments.

    DayOfYear( );
    //     Initializes the date to January first.

    void input( );
    void output( );
    int get_month( );
    //      Returns the month, 1 for January, 2 for February, etc.

    int get_day( );
    //      Returns the day of the month.
```

Here equal is granted friend status by the class

Equality Function (2 of 4)

```
private:
    int month;
    int day;
};

int main( )
{
    DayOfYear today, bach_birthday(3, 21);

    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );

    cout << "J. S. Bach's birthday is ";
    bach_birthday.output( );

    if ( equal(today, bach_birthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
```

// Equality Function (3 of 4)

```
bool equal(DayOfYear date1, DayOfYear date2)
{
   return ( date1.month == date2.month &&
                     date1.day == date2.day );
}

DayOfYear::DayOfYear(int the_month, int the_day)
{   month = the_month;
    day = the_day;
}

DayOfYear::DayOfYear( )
{   month = 1;
    day = 1;
}

int DayOfYear::get_month( )
{
    return month;
}
```

Here equal is defined exactly as any ordinary function is defined. Access to private data is enabled by friend status .

// Equality Function (4 of 4)

```
int DayOfYear::get_day( )
{
    return day;
}

//       Uses iostream:
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}

//        Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
        << ", day = " << day << endl;
}
```

Friend Functions (1 of 2)

A friend function of a class is an ordinary (non-member) function that has access granted to private members of a that class. To make a function a friend of a class you list that function prototype in the class with the keyword *friend*. The prototype may be placed in either the public or the private section of the class.

Syntax:

```
class Class_name
{
 public:
   friend Prototype_for_friend_Function_1;
   friend Prototype_for_friend_Function_1;
//     member function prototypes here.
 private:
//       private members here
};
```

Friend Functions (2 of 2)

Example:

```
class FuelTank
{
public:
   friend double need_to_fill(FuelTank (the_tank));  // !!!!!!!!!!
   FuelTank(double the_capacity, double the_level);
   FuelTank( );
   void input();
   void output();
private:
    double  capacity; // in liters
    double level;
};
```

A friend function is NOT a member function. A friend function is defined and called the same way as an ordinary function. You do not use the dot operator to call a friend function, and you do not use a type qualifier in the definition of a friend function.

Programming Tip:
Use both Member and Nonmember functions.

- Members and friends of a class do similar services for a class.
- To clarify whether a given task should be done by a friend or member, consider:
  - Use a member function if the task being performed by the function involves only one object.
  - Use a nonmember function if the task being performed involves more than one object.

  Example: The function equal involves two objects, so we make it a friend.
- Whether member or non-member is not always as simple as this rule suggests.
- A slightly less clear but more general rule is:
- Use a member if the task is related to ONE object,
- Use a non-member (friend) function if the task is symmetrically related to two objects.
- Clarity and readability is the cardinal rule for this decision.

PITFALL
Compilers without Friends

- In the process of meeting the ANSI C++ Standard, some compilers introduced problems that cause friends not to work correctly.
- On some of these compilers, friends deny access to private members.
- On others, there are problems private access with operator overloading.
- Techniques to workaround this difficulty:

  1. Make private members public. A poor solution, but it allows a quick check of code.

Programming Example
Money ADT (1 of 2)

The next slides contain the definition of a class Money, an ADT for amounts of US currency.

The value is implemented in a single int value that represents the number of cents. For example, $9.95 stores as 995

Negative amounts of money are represented as -$9.95

// Money ADT - Version 1 (1 of 8)

```
//     Program to demonstrate the class Money.
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

//      Class for amounts of money in U.S. currency.
class Money
{
public:
   friend Money add(Money amount1, Money amount2);
   //      Precondition: amount1 and amount2 have been given values.
   //      Returns the sum of the values of amount1 and amount2.

   friend bool equal(Money amount1, Money amount2);
   //      Precondition: amount1 and amount2 have been given values.
   //      Returns true if the amount1 and amount2 have the same value;
   //      otherwise, returns false.

   Money(long dollars, int cents);
   //      Initializes the object so its value represents an amount with
   //      the dollars and cents given by the arguments. If the amount
   //      is negative,  then both dollars and cents should be negative.
```

Money ADT - Version 1 (2 of 8)

```
Money(long dollars);
//      Initializes the object so its value represents $dollars.00.

Money( );
//    Initializes the object so its value represents $0.00.

double get_value( );
//      Precondition: The calling object has been given a value.
//      Returns the money amount recorded in the calling object.

void input(istream& ins);
//     Precondition: If ins is a file input stream, then ins has
//       already been connected to a file. An amount of money,
//       including a dollar sign, has been entered in the input stream ins.
//       Notation for negative amounts is as in -$100.00.
//     Postcondition: The value of the calling object has been set to
//          the amount of money read from the input stream ins.

void output(ostream& outs);
//      Precondition: If outs is a file output stream, then outs has already
//             been connected to a file.
//      Postcondition: A dollar sign and the amount of money recorded
//        in the calling object have been sent to the output stream outs.
```

Money ADT - Version 1 (3 of 8)

```
private:
    long all_cents;
};

//   Prototype for use in the definition of Money::input:
int digit_to_int(char c);
//    Precondition: c is one of the digits '0' through '9'.
//     Returns the integer for the digit; e.g., digit_to_int('3') returns 3.

int main( )
{
    Money your_amount, my_amount(10, 9), our_amount;
    cout << "Enter an amount of money: ";
    your_amount.input(cin);
    cout << "Your amount is ";
    your_amount.output(cout);
    cout << endl;
    cout << "My amount is ";
    my_amount.output(cout);
    cout << endl;
```

Money ADT - Version 1 (4 of 8)

```
 if (equal(your_amount, my_amount))
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";

    our_amount = add(your_amount, my_amount);
    your_amount.output(cout);
    cout << " + ";
    my_amount.output(cout);
    cout << " equals ";
    our_amount.output(cout);
    cout << endl;
    return 0;
}

Money add(Money amount1, Money amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}
```

Money ADT - Version 1 (5 of 8)

```
bool equal(Money amount1, Money amount2)
{
    return (amount1.all_cents == amount2.all_cents);
}

Money::Money(long dollars, int cents)
{
    all_cents = dollars*100 + cents;
}

Money::Money(long dollars)
{
    all_cents = dollars*100;
}

Money::Money( )
{
    all_cents = 0;
}

double Money::get_value( )
{
    return (all_cents * 0.01);

}
```

Money ADT - Version 1 (6 of 8)

```
//      Uses iostream, cctype, cstdlib
void Money::input(istream& ins)
{
    char one_char, decimal_point,
         digit1, digit2;     //      digits for the amount of cents
    long dollars;
    int cents;
    bool negative;      //       set to true if input is negative.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //      if input is legal, then one_char == '$'
```

Money ADT - Version 1 (7 of 8)

```
        ins >> dollars >> decimal_point >> digit1 >> digit2;

        if ( one_char != '$' || decimal_point != '.'
             || !isdigit(digit1) || !isdigit(digit2) )
        {
            cout << "Error illegal form for money input\n";
            exit(1);
        }
        cents = digit_to_int(digit1)*10 + digit_to_int(digit2);

        all_cents = dollars*100 + cents;
        if (negative)
            all_cents = -all_cents;
    }
```

Money ADT - Version 1 (8 of 8)

```
//       Uses cstdlib and iostream:
void Money::output(ostream& outs)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (all_cents < 0)
        outs << "-$" << dollars << '.';
    else   outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;
}

int digit_to_int(char c)
{
    return ( int(c) - int('0') );
}
```

Implementation of digit_to_int

The function, digit_to_int from Display 8.3 is

```
int digit_to_int(char c)
{
    return (int (c)  - int ('0'));
}
```

The charater '3' is encoded by the decimal number 51.

The charater '0' is encoded by the decimal number 48.

The difference is the value of the digit 3.

The const Parameter Modifier

- For large objects, a call by reference is more efficient than a call by value, because the object must be copied.
- For any object you want not to modify, call by reference is unsafe.
- The `const` modifier allows you to promise the compiler that you won't change a parameter declared with a const keyword.
- In turn, the compiler will emit error messages when you compile code that does change the parameter.
- In effect, the compiler enforces your promise.
- Such a parameter is called a constant parameter

Example:

Money add(const Money& amount1, const Money& amount2);

Here any code written in the implementation of the add function that might change amount1 or amount2 will be marked as an error.

Display 8.4 contains prototypes of Money ADT with constant parameters.

PITFALL: Inconsistent use of const

- If you use the const parameter modifier in your prototype in your class, you MUST use the const modifier in exactly the same way in your definition, or you will get errors when you compile your code.

Example:

```
class foo
{
    public:
        void bar( const int &);
};
void foo::bar(int & x)
{
         // whatever bar does
}
```

The compiler will issue an error message that says "foo::bar(int&) is not a member of 'foo'".

Why? (Or How?) The compiler encodes the name along with the types and whether the parameters are const reference. If the compiler has a const parameter but the definition does not, the compiler considers them different functions, so issues an error message.

Overloading Operators(1 of 11)

- A principle in the design of C++ is to provide facilities to program ADTs that behave like built-in types (primitive types) as possible.
- The catch-phrase is "Operator overloading enables programming closer to the problem domain."
- Overloading operators is a facility that allows giving ADT's infix operators such as +, -, stream i/o using << and >> with behavior that the programmer specifies.

Overloading Operators(2 of 11)

- Operators are overloaded for the Money ADT to allow use of infix operator + :
  ```
  Money total, cost, tax;
  cout << "Enter cost  and tax amounts: ";
  cost.input(cin);
  tax.input(cin);
  total = cost + tax;
  ```
- Operator + is overloaded to allow this last line instead of the significantly more awkward syntax
  `total = cost.add(tax);`

Shortly, we will overload << and >> to provide stream io for even less awkward code (i.e., allow programming "closer to the problem domain.")

Overloading Operators(2 of 11)
Some Questions.

- How do we overload operators?
- What operators can we overload?
- What should an operator overloading return?
- What restrictions are there on operator overloading?
- What operator behavior can we NOT change?
- Where do the operands for operator overloading appear?
- How many parameters must my overloading operator function have?

Overloading Operators (4 of 11)
Some Answers

- How do we overload an operator? The keyword operator followed by the operator to be overloaded becomes the name of function that may be either standalone function or a member of a class.

An operator function can be, but is not required to be, a member of a class (but look for restrictions in a later slide).

Example -- Stand Alone operator overloading:

```
Money operator+(const Money & lhs, const Money & rhs)
{
     Money temp;
     temp.all_cents = lhs.all_cents + rhs.all_cents;
     return temp;
}
```

Here we create a local object, `temp`, set its private data then return it for use in an expression just like any built-in operator.

Overloading Operators (5 of 11)
Some Answers

Example -- Member function operator overloading:

```
Money Money::operator+(const Money & rhs)
{
     Money temp;
     temp.all_cents = all_cents + rhs.all_cents;
     return temp;
}
```

To use this, the class Money must have a member whose prototype is

```
Money operator+(const Money & rhs);
```

Note that there are TWO operands provided for both of these overloadings of the + operator. In the above case, the calling object is the left hand operand.

As in the stand alone version, we create local object, set its private data member, and return it. The member all_cents without qualification belongs to the calling object.

Overloading Operators (6 of 11)
Some Answers

- What operators can we overload?

We can overload all operators in C++ except

::	Scope Resolution Operator
.	Member Access Operator
?:	Ternary selection operator
.*	Member Access Through a pointer

To read about the reasons why overloading these operators is not allowed, see Stroustrup, The Design and Evolution of C++ Addison Wesley Longman, 1994. This and other references are listed in the text in the Annotated Bibliography on page 937.

Overloading Operators (7 of 11)
Some Answers

- What *should* an operator overloading function return?

Example: Member function operator overloading, continued:

Without comment, we used a Money return value for operator+.

(This is a partial answer to our question. More later.)

In the expression, suppose a, b and c are int variables:

a + b + c

This groups left to right, as if with these parentheses:

(a + b) + c

The first expression, a + c, returns an int value, which is used as an argument for the second + operator.

By analogy, if we overload an operator that is to behave like primitive objects, overloading +, -, *, and so on, should return an object of the same type as the operands, so the value that will be useful in a complex expression.

In the case of overloading + for Money, + should return a Money object.

Overloading Operators (8 of 11)
Some Answers

- What restrictions are there on operator overloading?
  - One of the operands must be a class object. If the operator overloading is done with a stand alone function, at least one of the parameters must have class type. Otherwise, the overloading function must be a member of a class.
  - This implies that you cannot change the behavior of built-in types.
  - We cannot create new operators.
- Four operators have special requirements for overloading.

These operators are:

  - assignment =, indexing [], member access through a pointer `->`, and function call ()
  - These operators must be members of a class. We discuss overloading [] and `->` later in the text.

Overloading Operators (9 of 11)
Some Answers

- What operator behavior can we NOT change?
  - We cannot change the number of operands that the operator requires.
    - If an operator requires one operand, we must provide one operand for the overloaded operator.
    - If an operator requires two operands, we must provide two operands for the overloaded operator.
    - For example, we cannot overload / or % to take one operand , and can we cannot overload ++ to take two operands.
  - We cannot change the precedence of operations. The * operator has precedence over + for built-in types, so with any overloaded + and overloaded * , the * operator will have precedence over the + operator.
  - We cannot change the way operators group their operands in an expression (associativity). Most operators group left to right, but these operators =, =+, =*, etc, group right to left.

Overloading Operators (10 of 11) Some Answers

- **Where do the operands for operator overloading appear?**
- **If we overload a binary operator as a stand alone function, whether as friend or not, the two operands for the infix operator appear as arguments for the two parameters in the operator function.**

Example: If our overloading is a stand-alone function such as:

```
Money operator+(const Money & lhs, const Money & rhs)
```

Then a programmer would write:

```
   final_amount = amount_1 + amount_2
```

The compiler translates programmer written code into this:

```
   final_amount = operator+(amount_1, amount_2);
```

Overloading Operators (11 of 11) Some Answers

- **How many parameters must my overloading operator function have?**

The answer depends on whether you are using a member operator function or standalone operator function to overload the operator, and how many arguments the operator has.

- **If the overloading operator function is a member of the class, the first operand (in the case of unary operators, the only operand) will be the calling object.**
  - **If the operator has two arguments, then the operator function must have one parameter.**
  - **If the operator has one argument, then the calling object is the argument for the operator. The operator function has no parameters.**
- **If the overloading operator function is a standalone function, there will be the same number of parameters as there are arguments for the operator.**

Overloading Operators (1 of 5)

```
//   To demonstrate the class Money. (An improved version of Display 8.4.)
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;
//      Class for amounts of money in U.S. currency.
class Money
{
public:
   friend Money operator +(const Money& amount1, const Money& amount2);
   //      Precondition: amount1 and amount2 have been given values.
   //      Returns the sum of the values of amount1 and amount2.
   friend bool operator ==(const Money& amount1, const Money& amount2);
   //      Precondition: amount1 and amount2 have been given values.
   //      Returns true if amount1 and amount2 have the same value;
   //          otherwise, returns false.
   Money(long dollars, int cents);
   Money(long dollars);
   Money( );
   double get_value( ) const;
   void input(istream& ins);
   void output(ostream& outs) const;
private:
   long all_cents;
};
```

Overloading Operators (2 of 5)

```
//       Prototype for use in the definition of Money::input:
int digit_to_int(char c);
//       Precondition: c is one of the digits '0' through '9'.
//       Returns the integer for the digit; e.g., digit_to_int('3') returns 3.
int main( )
{
   Money cost(1, 50), tax(0, 15), total;
   total = cost + tax;
   cout << "cost = ";
   cost.output(cout);
   cout << endl;
   cout << " tax = ";
   tax.output(cout);
   cout << endl;
   cout << " total bill = ";
   total.output(cout);
   cout << endl;
   if (cost == tax)
      cout << "Move to another state.\n";
   else
      cout << "Things seem normal.\n";
   return 0;
}
```

Overloading Operators (3 of 5)

```
Money operator +(const Money& amount1, const Money& amount2)
{
   Money temp;
   temp.all_cents = amount1.all_cents + amount2.all_cents;
   return temp;
}
bool operator ==(const Money& amount1, const Money& amount2)
{
   return (amount1.all_cents == amount2.all_cents);
}
Money::Money(long dollars, int cents)
{
   all_cents = dollars*100 + cents;
}
Money::Money(long dollars)
{
   all_cents = dollars*100;
}
Money::Money( )
{
   all_cents = 0;
}
double Money::get_value( ) const
{
   return (all_cents * 0.01);
}
```

Overloading Operators (4 of 5)

```
//      Uses iostream, cctype, cstdlib
void Money::input(istream& ins)
{
   char one_char, decimal_point, digit1, digit2;   //   digits for the amount of cents
   long dollars;
   int cents;
   bool negative;//set to true if input is negative.
   ins >> one_char;
   if (one_char == '-')
   {
      negative = true;
      ins >> one_char; //read '$'
   }
   else
      negative = false;
   //      if input is legal, then one_char == '$'
   ins >> dollars >> decimal_point >> digit1 >> digit2;
   if ( one_char != '$' || decimal_point != '.'
        || !isdigit(digit1) || !isdigit(digit2) )
   {
      cout << "Error illegal form for money input\n";
      exit(1);
   }
   cents = digit_to_int(digit1)*10 + digit_to_int(digit2);
   all_cents = dollars*100 + cents;
   if (negative)
      all_cents = -all_cents;
}
```

Overloading Operators (5 of 5)

```
//      Uses cstdlib and iostream:
void Money::output(ostream& outs) const
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (all_cents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';
    if (cents < 10)
        outs << '0';
    outs << cents;
}

int digit_to_int(char c)
{
    return ( int(c) - int('0') );
}
```

91

Overloading a Unary Operator (1 of 2)

```
//    This is not a complete program and cannot be run.
//    Class for amounts of money in U.S. currency.
class Money
{  public:
   friend Money operator +(const Money& amount1, const Money& amount2);
   friend Money operator -(const Money& amount1, const Money& amount2);
   //    Precondition: amount1 and amount2 have been given values.
   //    Returns amount 1 minus amount2.
   friend Money operator -(const Money& amount); // The - operator has one argument so
   //Precondition: amount has been given a value.This friend HAS ONE parameter
   //Returns the negative of the value of amount.
   friend bool operator ==(const Money& amount1, const Money& amount2);
   Money(long dollars, int cents);     // The ==, +, and = operators have two arguments
   Money(long dollars);                // so the friend overloading have two parameters.
   Money( );
   double get_value( ) const;
   void input(istream& ins);
   void output(ostream& outs) const;
private:
   long all_cents;
};
```

92

Overloading a Unary Operator (2 of 2)

<Any additional prototypes as well as the main part of the program go here.>

```
Money operator -(const Money& amount1, const Money& amount2)
{
   Money temp;
   temp.all_cents = amount1.all_cents - amount2.all_cents;
   return temp;
}

Money operator -(const Money& amount)
{
   Money temp;
   temp.all_cents = -amount.all_cents;
   return temp;
}
```

<The other function definitions are the same>

93

Overloading >> and << (1 of 4)

- **We pointed out that one of the design principles for C++ is to provide facilities to program ADTs that behave like built-in types (primitive types) as possible.**
- **Overloading the istream insertion >> operator and ostream extractor << operator make the behavior of programmer defined types closer to the built-in types in C++.**
- **We remind you that “Operator overloading enables programming closer to the problem domain.”**
- **After overloading >> and << for our Money class, we can write**

```
Money amount;
    cout << “Enter an amount of money: “ ;
    cin >> amount;
    cout << “I have “ << amount << “ of money in my purse.\n”;
```

just as we would any built-in type object.

94

Overloading >> and << (2 of 4)

- **What value should be returned from a call to operator<< ?**

We examine a chain of output statements:

```
cout << "I have " << amount << " in my purse.\n ";
```

The operator << associates left to right, as if grouped this way:

```
((cout << "I have ") << amount) << " in my purse.\n ";
```

Within the inside parentheses, colored () here, we have the code

```
cout  << " I have "
```

which inserts the cstring "I have " into the ostream variable cout.

When the compiler sees this, it replaces this code with a call to the overloading function for <<, that is, operator<<, with arguments cout, and "I have ":

```
operator<< (cout, “I have “)
```

The operator << function returns a reference to cout, which is the stream we passed to the overloading function operator <<. This is used as the left argument for the next << operator. We insert the next entry, amount, into this stream object.

95

Overloading >> and << (3 of 4)

The compiler sees

```
(cout << amount)
```

This is << with an ostream on the left and a Money object on the right. The compiler finds our operator overloading that has these types as parameters, so our operator function is called.

The *next* call is critical. We have whatever is returned by the expression within () on the left of an insertion operator, <<, and a string on the right. If our overloading returns its ostream argument, things work as expected:

```
cout  << " in my purse.\n "
```

- **We conclude that our overloading of operator<< should return a reference to the ostream that was sent to this function as its left argument, so that will be available for insertion of the next object in the chain.**

96

Overloading >> and << (4 of 4)

An extract from class Money:

```
class Money
{
public:
    . . .
    friend ostream& operator<< (ostream& outs, const Money& amount);
    // Pre:   If outs is a file output stream, then it has been connected to a file.
    // Post: A dollar sign ($) and the amount of money in calling object have
    // been sent to the output stream outs.
   . . .
};                             The argument for outs is the left argument of <<.
ostream & operator<< (ostream& outs, const Money& amount)
{                               The argument for amount is the right argument of <<.
    long positive_cents, dollars, cents;
    positive_cents = labs(amount.all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (amount.all_cents < 0)
      outs << "-$" << dollars << '.';
   else
      outs << "$" << dollars << '.';
   if (cents < 10)
      outs << '0';
   outs << cents;
   return outs;   We return the parameter that has a reference to the
}                 to the ostream argument from the left of the << operator.
```

Separate Compilation

- A large program will usually have to be worked on by several programmers, usually, teams of programmers.
- To do this effectively requires that a program be divided into parts that are kept in separate files.
- C++ provides facilities to separately compile and link these separate parts of a program.
- Here are a few efficiencies gained by the process:
  - We can compile an ADT once then use it with several applications.
  - We can modify one file, compile just this part, and relink the pieces, avoiding the otherwise long process of compiling all of a large program.
  - We can separate the definition from the specification. This enables us to change the implementation without changing the application that depends only on the interface, not on the implementation.

PROGRAMMING TIP: Hiding the Helping Functions

We defined helping functions for the ADT DigitalTime, but we hid them.

Helping functions are defined to be used in one or more member functions of the class, but are not intended for client programmer use. Such functions are normally made inaccessible by placing them in the private section of the class definition.

The advantages of separate compilation are that we can factor out common actions between member functions, and we can revise the definition of the member functions without requiring the client to change the application.

In ADT DigitalTime we defined read_hour and read_minute to support operator>>.

Using #ifndef (1 of 2)

- **Suppose we have interface and implementation spread across many files. It is possible to include a particular function definition or class definition more than once. This is illegal, and will be caught by every compiler.**
- **We can use part of the compiler to avoid this error.**
  The C++ command
  #include
  is called a preprocessor directive, because the part of the compiler that processes this directive is called the preprocessor.
- **This preprocessor directive defines the preprocessor symbol DTIME_H**
  #define DTIME_H
- **The preprocessor has a table of names that have been defined by preprocessor directives. This command places DTIME_H in that list.**

Using #ifndef (2 of 2)

- **This directive tests whether DTIME_H has been defined, and if it HAS been defined, skips all lines of code between this directive and the next occurrence of**
  #endif
- **Consider the sequence of preprocessor commands:**

```
#ifndef DTIME_H
#define DTIME_H
class foo
{
   // members
};
#endif
```

- **If these commands are in dtime.h, no matter how many times**
  #include "dtime.h"
- **is present in a file, the class foo will be defined only once.**

Avoiding Multiple Definitions

```
//This is the HEADER FILE dtime.h. This is the INTERFACE for the class
// DigitalTime.
//Values of this type are times of day. The values are input and output in
//24 hour notation as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.

#ifndef DTIME_H          // This asks "Have I seen this before?
#define DTIME_H          // if so, skip down to #endif

#include <iostream>       // otherwise, this is new, compile...
using namespace std;

class DigitalTime

{

//The class members are  the same as in dtime.h

};


#endif //DTIME_H         // end of our "Have I seen this before?" question.
```

Programming Tip:
Choosing ADT Operations

We should supply:

1. Constructors for setting and changing the data value(s) of an object of the ADT. Include a default constructor.
2. Some way to test whether two objects represent the same data value(s). Typically this is done by overloading the == operator.
3. An input method for objects of the ADT type. Typically this is done by overloading the extraction operator, >>, and deciding on an external representation.
4. A method for outputting data value(s) of an object of the ADT. Typically this is done by overloading the insertion operator, <<.
5. Some functions to perform basic operations. For example, the (overloaded) function to advance the time in the DigitalTime ADT is an example of such functions. The selection will necessarily depend on the particular ADT.

Programming Tip:
Defining Other Libraries

If we have a family of related tasks, you may wish to define a library of our functions. We can place the prototypes of the functions of our library in a header file along with prototype comments. If we write a library, we should write extended documentation for the library and place this in a standard system place for documentation.

Where you place the interface, the implementation and the documentation is system dependent.

Namespaces
Namespace and Using Directives

- We have been using the namespace `std`. This namespace contains all the names that come from headers of the standard library (remember -- these header have no .h extension).
- For example,

```
#include <iostream>
```

places all the names defined in this header into namespace std.

To access them we write

```
using namespace std;
```

- Any name not specifically placed in a named namespace is placed in the global namespace. Names from the global namespace are available without directive or qualification.
- It is an error to define a name twice in a scope.
- It is an error to included the same name from two different namespaces.

Namespace and Using Directives -- An Example

Example:

```
{// sibling block to the next block
   using namespace ns1;
   my_function();  // my_function from namespace ns1
}
{  // sibling block to the first block
   using namespace ns2;
   my_function(); // my_function from namespace ns2
}
```

- There is no name clash here because:
- A using directive a the start of a block applies to the block.

Remember,

- A using directive at the start of a file applies to the whole file.
- Typically Using directives are placed near the start of a file or the start of a block.

Namespaces
Creating a Namespace

- A namespace is created by placing the definition of the name in a namespace grouping in a header file, say

```
namespace Stat130D
{
   void greeting();
}
```

- A namespaces may be composed of several namespace groupings spread across several files.
- To access the names, we must make the names available with a using directive:

```
using namespace Stat130D;
```

Namespaces
Placing Our Namespaces in a Header File

- To create a namespace with functions for use in several files, you place the namespace grouping of the prototypes in in a header file:

```
// This is file Stat130D.h
namespace Stat130D
{
   void greeting();
}
```

- To make these names available we must use this include directive and a using directive:

```
#include “Stat130D.h”
using namespace Stat130D;
// code that accesses the functions.
```

Namespaces
Linking to Functions Defined in Our Namespaces

- We must also make the function available to the linker as well.
- To use the function whose prototype is in the header file Stat130D.h, it is necessary to make the names accessible, as in the previous slide, then place the function definition in same namespace as the prototype, namely, Stat130D :

```
// This is file Stat130D.cxx
namespace Stat130D
{
   void greeting()
   {
      cout << “Hi there!\n”;
    }
}
```

- Finally, we must compile the file Stat130D.cxx with our program. (We could compile separately then link the object files.)

109

Namespaces
Qualifying Names(1 of 2)

- If we have two namespaces each of which have the same name defined, then the following will cause a name clash:

```
using namespace ns1;  // defines name1
using namespace ns2;  // defines name1; causes a name clash
```

- If we need other names, say fun1 from namespace ns1 and fun2 from ns2, we must prevent the clash of the name1 names.

  The answer is:

```
using ns1::fun1;
using ns2::fun2;
```

- This works because these directives make only the specified names available.

110

Namespaces
Qualifying Names (2 of 2)

- If we need to use both the names name1 from namespace1 and namespace2 in one block, different solution, namely qualified names:
- int sum = ns1::name1 + ns2::name1;
  - ns1::name1 means name1 from namespace1
  - ns2::name1 means name1 from namespace2
- In a function that uses a parameter or return type that is name from namespace, we can qualify the name, as in

```
#include <iostream>
std::ostream& operator<<(std::ostream& outs, My_Class this_object);
```

111

Namespace Demonstration (1 of 2)

```
#include <iostream>
using namespace std;
namespace Stat130D_a
{
   void greeting();
}
namespace Stat130D_b
{
   void greeting();
}
void big_greeting();
int main( )
{
   {
      using namespace Stat130D_b;
      greeting();
   }
   {
      using namespace Stat130D_a;
      greeting();
   }
   big_greeting();
   return 0;
}
```

112

Namespace Demonstration (2 of 2)

```
namespace Stat130D_a
{
   void greeting()
   {
      cout << "Hello from namespace Stat130D_a.\n";
   }
}
namespace Stat130D_b
{
   void greeting()
   {
      cout << "Greetings from namespace Stat130D_b.\n";
   }
}
void big_greeting()
{
   cout << "A Big Global Hello!\n";
}
```

113

A Programming Example
Placing a class in a Namespace (Header File)

```
//This is the HEADER FILE dtime.h. This is the INTERFACE for the class
//DigitalTime.
//Values of this type are times of day. The values are input and output in
//24 hour notation as in 9:30 for 9:30 AM and 14:45 for 2:45 PM.
#ifndef DTIME_H
#define DTIME_H
#include <iostream>
using namespace std;
namespace dtime
{
   class DigitalTime
    {
     <The definitions of the class Digital Time is the same as before>
    };
}//end dtime

#endif //DTIME_H
```

114

A Programming Example

Placing a class in a Namespace (Implementation File)

```
//This is the IMPLEMENTATION FILE: dtime.cxx (Your system may require some
//suffix other than .cxx). This is the IMPLEMENTATION of the ADT DigitalTime.
//The interface for the class DigitalTime is in the header file dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

namespace dtime
{
    <all prototypes and function definitions go here.>
}//end dtime
```