Unit 5 (ch11)

More about Classes

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat'l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *5.1 Friend Functions*
- 5.2 const Parameters in Classes
- 5.3 Overloading Operators
- 5.4 Arrays and Classes
- 5.5 Dynamic Variables in Classes
- 5.6 Using this Pointer in Classes

Why do We Need Friends??

- Theoretically, all required functions are included in the class itself
  - However, you may need outside help in some cases
  - **Friends can help !!**
- If you need help from ordinary (non-member) functions, access for internal data is necessary
  - Give special permission to those functions by declaring them as **FRIENDS**
  - Avoid extra overhead on calling accessors/mutators

Friend Functions

- Member functions can access private members
  - Non-member functions can only access private data through interface functions → inefficient !!
- Friend functions can directly access private members
  - No calls to accessor/mutators → efficient !!
- Friend functions are inherently dangerous
  - You should make friends very carefully !!
- Use keyword *friend* in front of function declaration
  - Should be specified **INSIDE** class definition
  - Encapsulation can still be achieved
- Use friends properly to get efficiency and keep safety

Example: An Equality Function

- The equality function tests two objects of type DayOfYear (in Ch10) and return a Boolean value
  - It is TRUE if two objects have the same day and month
- The *equal* function can be used to compare dates in this manner

```
if ( equal( today,  bach_birthday) )
     cout << "It's Bach's birthday!";
```

- The function *equal* cannot be a member function of any specific object
  - You have no access to the data of another object

Is the Function *equal* Efficient ?

- Without friend declaration, *equal* can be defined as:

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

- Function *equal* could be made more efficient
  - Using member function calls to obtain the private data values is not efficient
  - Direct access of the member variables would be more efficient (faster)

Non-Member Function (1/2)

```
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int theMonth, int theDay);
    DayOfYear( );
    void input( );
    void output( );
    int getMonth( );
    int getDay( );
private:
    void checkDate( );
    int month;
    int day;
};
bool equal(DayOfYear date1, DayOfYear date2);
```

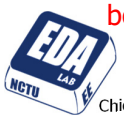
Declared as normal functions

```
int main( )
{
    DayOfYear today, bachBirthday(3, 21);

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

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

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



Non-Member Function (2/2)

```
DayOfYear::DayOfYear(int theMonth, int theDay)
              : month(theMonth), day(theDay)
{
    checkDate();
}

int DayOfYear::getMonth( )
{
    return month;
}

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

Sample Dialogue

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

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

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

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

Friend Functions

- The friend relationship is declared using the keyword ***friend*** in the class definition
  - A friend function is not a member function, but it has extra access to private members of the class
  - Friendship relation is neither symmetric nor transitive
- The friend function is a standalone function declared outside the class
  - Entire classes or member functions of other classes can also be friends of another class
- As a friend function, the more efficient version of function *equal* becomes possible

Declaring A Friend

- The function *equal* is declared a friend in the abbreviated class definition here

```
class DayOfYear
{
   public:
      friend bool equal(DayOfYear date1, DayOfYear date2);
      // The rest of the public members
   private:
      //  the private members
};
```

- Friend function is defined as a nonmember function without using the "::" operator
- Friend function is called without using the '.' operator

A More Efficient *equal*

- With friend relationship, direct access of private member variables is legal now !!
  - The code is simpler and more efficient

```
#include <iostream>
using namespace std;

class DayOfYear
{
 public:
   friend bool equal(DayOfYear date1,
                     DayOfYear date2);
   DayOfYear(int theMonth, int theDay);
   DayOfYear( );
   void input( );
   void output( );
   int getMonth( );
   int getDay( );
 private:
   void checkDate( );
   int month;
   int day;
};

int main( )
{
  //The main part of the program is the same
}

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

Choosing Friends

- How do you know when a function should be a friend or a member function?
  - In general, use a member function if its task involves only one object
  - In general, use a nonmember function if its task involves more than one object
- Choosing to make the nonmember function a friend is a decision of efficiency and personal taste
  - You can still access private members through the normal accessor and mutator functions of the class if need

Example: The Money Class

- This example demonstrates a class called *Money*
  - U.S. currency is represented
- Value is implemented as an integer representing the value as if converted to pennies
  - An integer allows exact representation of the value
  - Type long is used to allow larger values
- Two friend functions, *equal* and *add*, are used
  - Handle two different objects

Sample Dialogue

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09
One of us is richer.
$123.45 + $10.09 equals $133.54
```

Code for Money Class (1/3)

```
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class Money
{
 public:
    friend Money add(Money amount1,
                     Money amount2);
    friend bool equal(Money amount1,
                      Money amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money( );
    double getValue( );
    void input(istream& ins);
    void output(ostream& outs);
 private:
    long allCents;
};
```

```
Money::Money(long dollars, int cents)
{
    if(dollars*cents < 0)
    {
        cout << "Illegal values for dollars and cents.\n";
        exit(1);
    }
    allCents = dollars*100 + cents;
}

Money::Money(long dollars) : allCents(dollars*100)
{
    //Body intentionally blank.
}

Money::Money( ) : allCents(0)
{
    //Body intentionally blank.
}

double Money::getValue( )
{
   return (allCents * 0.01);
}
```

Code for Money Class (2/3)

```
void Money::input(istream& ins)
{
   char oneChar, decimalPoint, digit1, digit2;
   long dollars;
   int cents;
   bool negative;
   ins >> oneChar;
   if (oneChar == '-')
   {
      negative = true;
      ins >> oneChar; //read '$'
   }
   else negative = false;
   ins >> dollars >> decimalPoint >> digit1
       >> digit2;
   if ( oneChar != '$' || decimalPoint != '.'
       || !isdigit(digit1) || !isdigit(digit2) )
    {
       cout << "Illegal input form\n";
       exit(1);
    }
   cents = digitToInt(digit1)*10 + digitToInt(digit2);

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

void Money::output(ostream& outs)
{
   long positiveCents, dollars, cents;
   positiveCents = labs(allCents);
   dollars = positiveCents/100;
   cents = positiveCents%100;
   if (allCents < 0)
      outs << "-$" << dollars << '.';
   else
      outs << "$" << dollars << '.';

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


Code for Money Class (3/3)

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

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

int digitToInt(char c)  // non-friend function
{
   return (static_cast<int>(c)
           - static_cast<int>('0') );
}

int main( )
{
   Money yourAmount, ourAmount;
   Money myAmount(10, 9);
   cout << "Enter an amount of money: ";
   yourAmount.input(cin);
   cout << "Your amount is ";
   yourAmount.output(cout);
   cout << endl;
   cout << "My amount is ";
   myAmount.output(cout);
   cout << endl;

   if (equal(yourAmount, myAmount))
      cout << "We have same amounts.\n";
   else cout << "One of us is richer.\n";

   ourAmount = add(yourAmount, myAmount);
   yourAmount.output(cout);
   cout << " + ";
   myAmount.output(cout);
   cout << " equals ";
   ourAmount.output(cout);
   cout << endl;
   return 0;
}
```

Dealing with Input Dollar Values

- The member function *input* processes the dollar values entered
  - Ex: $20.48, -$17.92, … (including both char and int)
- 1. Read the first character (can be $ or –)
  - If it is the minus sign (-), set the negative flag as TRUE and read the next $ sign
  - For others, set the negative flag as FALSE and do nothing (the $ sign has been read in already)
- 2. Read the first number (dollar amount) as a long
  - Stop at the period (.) because it is not a number
- 3. Read the decimal point and cents as 3 characters
  - You have only two digits for cents
  - *digitToInt* will convert the cents characters to integers

The Function digitToInt

- digitToInt is defined as

```
return ( static_cast<int> ( c ) – static_cast<int>( '0' ) );
```

- Input c is a character for one digit, such as '3'
  - This is the character '3' not the number 3
- The type cast static_cast<int>(c) returns the ASCII code that represents the character stored in c
- The ASCII codes for digits are in order
  - int('0') + 1 is equivalent to int('1')
  - int('1') + 1 is equivalent to int('2')
- If c is '0', int(c) - int('0') = integer 0
- If c is '0', int(c) - int('0') = integer 1

(ASCII) Code	
	$b_7 b_6 b_5$
$b_4 b_3 b_2 b_1$	011
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Overview

- 5.1 Friend Functions
- *5.2 const Parameters in Classes*
- 5.3 Overloading Operators
- 5.4 Arrays and Classes
- 5.5 Dynamic Variables in Classes
- 5.6 Using this Pointer in Classes

Class Parameters

- By default, class parameters are passed into a function by value
  - This results in two copies of the argument
  - However, a class can have many members in it …
- It is more efficient to use call-by-reference mechanism for class parameters
  - There is only one copy of the argument
- When using a call-by-reference parameter
  - If the function does not change the value of the parameter, mark the parameter as constant
  - Prevent the data from being changed unintentionally

const Parameter Modifier

- To mark a call-by-reference class parameter:
  - Use the modifier **const** before the parameter type (class name)
  - The class becomes a constant parameter
  - **const** used in both function declaration and definition
- Example (from the Money class):
  - A function declaration with constant parameters

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

  - A function definition with constant parameters

```
Money add(const Money& amount1,
          const Money& amount2)
{ … }
```

const and Accessor Functions

- When a function has a constant parameter, compiler has to protect the parameter from being changed
  - What if the parameter calls a member function?
  - It may change the internal of a constant object !!
- For example, there is an accessor function call from the constant parameter *amount1*

```
Money add(const Money& amount1,
          const Money& amount2)
{       …
        amount1.output(cout);
}
```

- The compiler will not accept this code
  - No guarantee that *output* will not modify the object

const Modifies Functions

- To allow a constant parameter make a member function call:
  - The called member function must be marked so the compiler knows it will not modify the object
  - Add **const** after the parameter list and just before the semicolon
  - **const** used in both function declaration and definition
- Example (from the Money class):
  - A function declaration without changing any member data

```
void output(ostream& outs) const ;
```

  - A function definition without changing any member data

```
void Money::output(ostream& outs) const
{ … }
```

const Wrapup

- Using *const* to protect the call-by-reference class parameters improves efficiency and keeps safety
  - *const* is added in front of the class name
- Member functions called by constant parameters must also be constant
  - *const* is added following the parameter list
- If a member function will not change any member values, add *const* modifier to it

```
class Money
{
 public:
    friend Money add(const Money& amount1,
                     const Money& amount2);
    friend bool equal(const Money& amount1,
                      const Money& amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money( );
    double getValue( ) const;
    void input(istream& ins);
    void output(ostream& outs) const;
 private:
    long allCents;
};
```

Code Example of Constant Object

- Constant object can use constant functions only !!
  - Non-constant object has no restrictions

```
class DayOfYear
{
 public:
   friend bool equal(DayOfYear date1,
                     DayOfYear date2);
   DayOfYear(int theMonth, int theDay);
   DayOfYear( );
   void input( );
   void output( ) const;
   int getMonth( );   // make it non-const
   int getDay( ) const;
 private:
   void checkDate( );
   int month;
   int day;
};
```

```
int main( )
{
   DayOfYear date1;  // non-constant object
   const DayOfYear date2(3, 21);   // constant
     // non-constant object
   date1.input( );     // non-constant function
   date1.output( );   // constant function
   cout << date1.getMonth()   // non-const func
        << date1.getDay();     // constant func
     // constant object
X  date2.input( );     // non-constant function
√  date2.output( );   // constant function
X  cout << date2.getMonth()   // non-const func
√        << date2.getDay();     // constant func
   return 0;
}
```


Overview

- 5.1 Friend Functions
- 5.2 const Parameters in Classes
- *5.3 Overloading Operators*
- 5.4 Arrays and Classes
- 5.5 Dynamic Variables in Classes
- 5.6 Using this Pointer in Classes

Why Operator Overloading?

- All operators for built-in types are defined in C++
  - Ex: for *int*, we have +, −, *, /, =, %, ==, ...
- For user-defined types, e.g. the Money class, can we also use "operators" for them?
  - Function add was used to add two objects of type Money in previous examples

```
Money total, cost, tax;
...
total = add(cost, tax);
 // can we use total = cost + tax ?
```

  - Those operators are unknown without redefinition
- Operator overloading contributes to C++ extensibility
  - Make a program clearer than using function calls

Operators As Functions

- Operators, ex: +, −, *, /, %, are actually functions
- They are just invoked in different syntax
  - An ordinary function call enclosed its arguments in parenthesis, ex: add(cost, tax)
  - With a binary operator, the arguments are on either side of the operator, ex: cost + tax
  - answer = cost + tax ←→ answer = +(cost, tax)
- To overload the + operator for the Money class, the definition is nearly the same as function add
  - Use the name + in place of the name add
  - Use keyword ***operator*** in front of the +
  - Ex: friend Money operator + (const Money& amount1...

Operator Overloading Rules

- Operator overloading works on objects only !!
  - At least one argument must be of a class type
- An overloaded operator can be a friend of a class
- Most existing operators are allowed to be overloaded
  - You cannot invent a whole new operator !!
  - ., ::, *, and ternary operator (?:) cannot be overloaded
- The number of arguments for an operator cannot be changed
  - You cannot define a unary % or a ternary +
- The same precedence and associativity still hold
  - Ex: b = b + c * a; → b = (b + (c*a));
    even a, b, c are of type Money

Example for Operator Overloading (1/2)

```
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class Money
{
public:
   friend Money operator +(const Money&
          amount1, const Money& amount2);
   friend bool operator ==(const Money&
          amount1, const Money& amount2);
   Money(long dollars, int cents);
   Money(long dollars);
   Money( );
   double getValue( ) const;
   void input(istream& ins);
   void output(ostream& outs) const;
private:
   long allCents;
};
```

Two overloaded operators
+ and == are demonstrated

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

bool operator ==(const Money& amount1,
const Money& amount2)
{
   return (amount1.allCents ==
           amount2.allCents);
}
```

Example for Operator Overloading (2/2)

```
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;
}
```

Output

```
cost = $1.50
tax = $0.15
total bill = $1.65
Things seem normal.
```


Automatic Type Conversion

- Does this code actually work??

```
Money baseAmount(100, 60), fullAmount;
fullAmount = baseAmount + 25;
```

  - Integer 25 is not of type Money !!
- When the compiler sees baseAmount + 25, it first looks for an overloaded + operator to perform MoneyObject + integer
  - Ex: friend Money operator +(const Money& amount1, const int& amount2);
- If the appropriate version of + is not found, the compiler looks for a constructor that takes an *int*
  - The constructor Money(long dollars) converts 25 to a Money object so the two values can be added!

A Constructor for double

- (baseAmount + 25) is supported through proper constructor
- (baseAmount + 25.67) will cause an error !!
  - There is no constructor in the Money class that takes a single argument of type double
- To permit (baseAmount + 25.67), the following constructor should be declared and defined

```
class Money
{
    public:
            ...
            Money(double amount);
            // Initialize object so its value is $amount
            ...
```

Returning Constant Objects

- What's the difference of these two versions?
  - Money operator +(const Money& amount1, const Money& amount2)
  - **const** Money operator +(const Money& amount1, const Money& amount2)
- Consider the following example:

```
Money a(5), b(1,50), c(0,15);
if ((a+b) = c)   // an error version of (a+b) == c
...
```

  - (a+b)=c has no error in non-constant version, but causes compilation error in constant version
  - Returning constant object is preferred !!

可以避免不小心的失誤，能加就加

Overloading Unary Operators

- Unary operators can be overloaded, too
  - They take only one single argument
- The unary – operator is used to negate a value
- ++ and - - are also unary operators
  - Need special handling for prefix and postfix issues
  - Discussed later in another section
- In the next example, – operator is overloaded as two different versions (binary vs unary)
  - Two arguments: subtract two Money objects
    Ex: amount3 = amount1 – amount2;
  - One argument: negate the value in a Money object
    Ex: amount3 = -amount1;

Class Definition with Overloading

Same function name with different number of arguments

```
class Money
{
public:
   friend Money operator +(const Money&
          amount1, const Money& amount2);
   friend Money operator -(const Money&
          amount1, const Money& amount2);
   friend Money operator -(const Money& amount);
   friend bool operator ==(const Money&
          amount1, const Money& amount2);
   Money(long dollars, int cents);
   Money(long dollars);
   Money( );
   double getValue( ) const;
   void input(istream& ins);
   void output(ostream& outs) const;
private:
   long allCents;
};
```

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

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

// Other member functions are omitted
// they are the same as in previous example
```

Overloading << and >>

- The insertion operator << is a binary operator
  - The first operand is the output stream
  - The second operand is the value following <<

```
cout << "Hello out there.\n";
```

Operand 1 — Operator — Operand 2

- Overloading the << operator allows us to use << instead of Money's output function
  - Given the declaration: Money amount(100);
    amount.output(cout); → cout << amount;

What Does << Return?

- Because << is a binary operator

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

seems as if it could be grouped as

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

- To provide cout as an argument for << amount, (cout << "I have") must return cout (cout << "I have")
- Based on the previous example, << should return its first argument, the output stream

```
class Money
{
  public:
        ... cout
    friend ostream& operator << (ostream& outs,
                                 const Money& amount);
```

會return cout 這樣就可以一直執行，輸出下一個

Overloaded << Definition

- The following defines the << operator

```
ostream& operator <<(ostream& outs,
                     const Money& amount)
{
    // Same as the body of Money::output
    // Internal variable allCents is replaced with
    // amount.allCents from the given object

    return outs;
}
```

若為 istream
ifstream
ostream
ofstream
必須用 call by reference

- The & means a reference is returned instead of value
  - The value of a stream might be an entire file, the keyboard, or the screen!
  - Returning the stream itself is more efficient

Overloaded >>

- Overloading the >> operator for input is very similar to overloading the << for output
- >> could be defined this way for the Money class:

```
istream& operator >>(istream& ins, Money& amount)
{
    // Same as the body of Money::input
    // Internal variable allCents is replaced with
    // amount.allCents from the given object

    return ins;
}
```

Money Class with Operators (1/3)

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>
using namespace std;

int digitToInt(char c);
class Money
{
 public:
   friend Money operator +(const Money&
              amount1, const Money& amount2);
   friend Money operator -(const Money&
              amount1, const Money& amount2);
   friend Money operator -(const Money& amount);
   friend bool operator ==(const Money&
              amount1, const Money& amount2);
   Money(long dollars, int cents);
   Money(long dollars);
   Money( );
   double getValue( ) const;
   friend istream& operator >>(istream&
                        ins, Money& amount);
   friend ostream& operator <<(ostream&
                  outs, const Money& amount);
private:
   long allCents;
};

istream& operator >>(istream& ins, Money&
                                    amount)
{
   char oneChar, decimalPoint, digit1, digit2;
   long dollars;
   int cents;
   bool negative;

   ins >> oneChar;
   if (oneChar == '-')
   {
      negative = true;
      ins >> oneChar; //read '$'
   }
```


Money Class with Operators (2/3)

```
   else
      negative = false;

   ins >> dollars >> decimalPoint >> digit1
       >> digit2;
   if ( oneChar != '$' || decimalPoint != '.'
       || !isdigit(digit1) || !isdigit(digit2) )
   {
      cout << " Illegal input form\n";
      exit(1);
   }

   cents = digitToInt(digit1)*10 + digitToInt(digit2);
   amount.allCents = dollars*100 + cents;
   if (negative)
      amount.allCents = -amount.allCents;

   return ins;
}

ostream& operator <<(ostream& outs,
                        const Money& amount)
{
   long positiveCents, dollars, cents;
   positiveCents = labs(amount.allCents);
   dollars = positiveCents/100;
   cents = positiveCents%100;

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

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

   return outs;
}
```

Money Class with Operators (3/3)

```
int digitToInt(char c)
{
    return ( static_cast<int>(c) –
             static_cast<int>('0') );
}

int main( )
{
    Money amount;
    ifstream inStream;
    ofstream outStream;

    inStream.open("infile.dat");
    if (inStream.fail( ))
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    outStream.open("outfile.dat");
    if (outStream.fail( ))
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }
    inStream >> amount;
    outStream << amount
              << " copied from the file infile.dat.\n";
    cout << amount
         << " copied from the file infile.dat.\n";
    inStream.close( );
    outStream.close( );

    return 0;
}
```


Overview

- 5.1 Friend Functions
- 5.2 const Parameters in Classes
- 5.3 Overloading Operators
- *5.4 Arrays and Classes*
- 5.5 Dynamic Variables in Classes
- 5.6 Using this Pointer in Classes

Arrays and Classes

- Arrays can use structures/classes as their base types
  - Example:

```
struct WindInfo
{
    double velocity;
    char direction;
};
WindInfo   dataPoint[10];
```

- Use the dot operator to access the members of an indexed variable
  - Example:

```
for (i = 0; i < 10; i++)
{
    cout << "Enter velocity: ";
    cin >> dataPoint[i].velocity;
}
```

dataPoint.velocity[i] has different meaning → explained later …

↔ different

An Array of Class Money

- Use default constructor to initialize each variable in array

```
#include <iostream>
using namespace std;

class Money
{  // same definition as in previous examples };

int main( )
{
    Money amount[5], max;
    int i;

    cout << "Enter 5 amounts of money:\n";
    cin >> amount[0];
    max = amount[0];
    for (i = 1; i < 5; i++)
    {
        cin >> amount[i];
        if (max < amount[i])
            max = amount[i];
    }

    Money difference[5];
    for (i = 0; i < 5; i++)
        difference[i] = max - amount[i];

    cout << "The highest amount is "
         << max << endl;
    cout << "The amounts and their\n"
         << "differences from the largest are:\n";
    for (i = 0; i < 5; i++)
    {
        cout << amount[i] << " off by "
             << difference[i] << endl;
    }

    return 0;
}
```

Sample Dialogue

```
Enter 5 amounts of money:
$5.00 $10.00 $19.99 $20.00 $12.79
The highest amount is $20.00
The amounts and their
differences from the largest are:
$5.00 off by $15.00
$10.00 off by $10.00
$19.99 off by $0.01
$20.00 off by $0.00
$12.79 off by $7.21
```

Arrays as Structure Members

- A structure can contain an array as a member
  - Example:

```
struct Data
{
    double time[10];
    int distance;
};
Data myBest;
```

  - *myBest* contains an array of type double
- To access the array elements within a structure
  - Use the dot operator to identify the array within the structure
  - Use the [ ]'s to identify the indexed variable desired
  - Example: **myBest.time[i]** → the ith variable of the array time in the structure object *myBest*

Arrays as Class Members

- Class TemperatureList includes an array
  - The array, named list, contains temperatures
  - Member variable size is the number of items stored

```
const int MAX_LIST_SIZE = 50;
class TemperatureList
{
public:
    TemperatureList( );
    void addTemperature(double temperature);
    bool full( ) const;
    friend ostream& operator <<(ostream& outs,
                const TemperatureList& theObject);
private:
    double list[MAX_LIST_SIZE];
    int size;
};
```

Overview of TemperatureList

- To create an object of type TemperatureList:
  - TemperatureList myData;
- To add a temperature to the list:
  - myData.add_temperature(77);
- A check is made to see if the array is full
  - The member function full()
- << is overloaded so output of the list is easy
  - cout << myData;

Member Functions of TemperatureList

```
TemperatureList::TemperatureList( ) : size(0)
{
      //Body intentionally empty.
}

bool TemperatureList::full( ) const
{
   return (size == MAX_LIST_SIZE);
}

ostream& operator <<(ostream& outs, const
TemperatureList& theObject)
{
   for (int i = 0; i < theObject.size; i++)
       outs << theObject.list[i] << " F\n";
   return outs;
}
```

```
void TemperatureList::addTemperature(double
                                     temperature)
{
   if ( full( ) )
   {
      cout << "Error: adding to a full list.\n";
      exit(1);
   }
   else
   {
      list[size] = temperature;
      size = size + 1;
    }
}
```

Overview

- 5.1 Friend Functions
- 5.2 const Parameters in Classes
- 5.3 Overloading Operators
- 5.4 Arrays and Classes
- *5.5 Dynamic Variables in Classes*
- 5.6 Using this Pointer in Classes

Classes and Dynamic Arrays

- A dynamic array can have a class as its base type
- A class can have a member variable that is a dynamic array
- Ex: the class StringVar
  - StringVar objects will be string variables
  - StringVar objects use dynamic arrays whose size is determined when the program is running
  - The StringVar class is similar to the string class discussed earlier

The StringVar Implementation

- The size of the array is not determined until the array is declared
- StringVar constructors call *new* to create the dynamic array for member variable values
  - Default constructor: creates an object with maximum string length (100)
  - 2nd constructor: takes an integer argument which determines the maximum string length of the object
  - 3rd constructor: takes a C-string argument and creates an object with the same string length and contents
  - 4th constructor (copy constructor): discussed later
- '\0' is added automatically to terminate the string

The StringVar Interface

- Its interface includes these member functions:
  - int length( );
  - void input_line(istream& ins);
  - friend ostream& operator << (ostream& outs, const StringVar& theString);
- Two special functions will be discussed later …
  - Copy Constructor
  - Destructor

```
class StringVar
{
 public:
    StringVar(int size);
    StringVar( );
    StringVar(const char a[]);
    StringVar(const StringVar& stringObject);
    ~StringVar( );
    int length( ) const;
    void input_line(istream& ins);
    friend ostream& operator <<(ostream&
                  outs, const StringVar& theString);
 private:
    char *value;
    int max_length;
};
```

Demo Class StringVar (1/2)

```
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include <cstring>
using namespace std;

StringVar::StringVar(int size) : maxLength(size)
{
    value = new char[maxLength + 1];
    //+1 is for '\0'
    value[0] = '\0';   // initially an empty string
}

StringVar::StringVar( ) : maxLength(100)
{
    value = new char[maxLength + 1];
    //+1 is for '\0'
    value[0] = '\0';   // initially an empty string
}
```

```
StringVar::StringVar(const char a[])
                        : maxLength(strlen(a))
{
    value = new char[maxLength + 1];
    //+1 is for '\0'
    strcpy(value, a);
}

StringVar::StringVar(const StringVar& stringObject)
                    : maxLength(stringObject.length( ))
{
    value = new char[maxLength + 1];
    //+1 is for '\0'
    strcpy(value, stringObject.value);
}

StringVar::~StringVar( )
{
    delete [] value;
}
```

Demo Class StringVar (2/2)

```
int StringVar::length( ) const
{
    return strlen(value);
}

//Uses iostream:
void StringVar::input_line(istream& ins)
{
    ins.getline(value, maxLength + 1);
}

//Uses iostream:
ostream& operator <<(ostream& outs,
                     const StringVar& theString)
{
    outs << theString.value;
    return outs;
}
```

```
void conversation(int maxNameSize);

int main( )
{
    using namespace std;
    conversation(30);
    cout << "End of demonstration.\n";
    return 0;
}

void conversation(int maxNameSize)
{
    using namespace std;
    StringVar your_name(maxNameSize);
    StringVar our_name("Borg");
    cout << "What is your name?\n";
    your_name.input_line(cin);
    cout << "We are " << our_name << endl;
    cout << "We will meet again " << your_name
         << endl;
}
```

Destructors

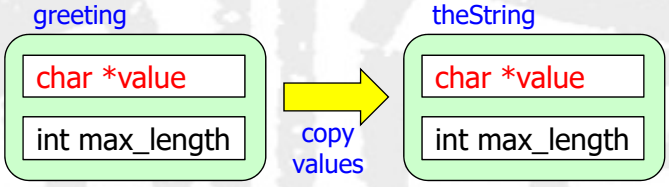
- Dynamic variables do not "go away" unless deleted
  - Even if a local pointer variable goes away at the end of a function, the allocated memory space still remains
- A destructor is a member function that is called automatically when an object goes out of scope
  - Delete all dynamic variables created by the object
  - A class has only one destructor with no arguments
  - The name of the destructor is distinguished from the default constructor by the tilde symbol ~
  - Example:

```
StringVar::~StringVar( )
{
    delete [ ] value;
}
```

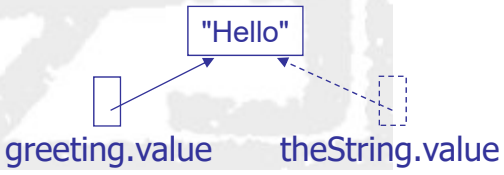
Return the memory space for whole array

Why Need a Copy Constructor? (1/2)

- The default assignment operator will copy values of a object into another
  - Even the memory address in a pointer variable
- For example: StringVar theString = greeting;

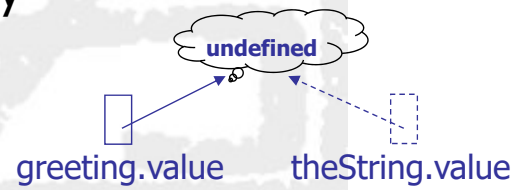


- Since the two pointers have the same memory address, they now point to the same dynamic array
  - Do you really want to share the same data between two separated objects ??



Why Need a Copy Constructor? (2/2)

- Given a function that prints out the given object
  - void showString(StringVar theString)
    { …}
- When function showString is called, *greeting* is copied into *theString*, including memory address
  - StringVar greeting("Hello");
    showString(greeting);
    cout << greeting << endl;
- When showString ends, the destructor for theString will be executed automatically
  - Delete the dynamic memory pointed by greeting.value
- Get trouble for next cout !!



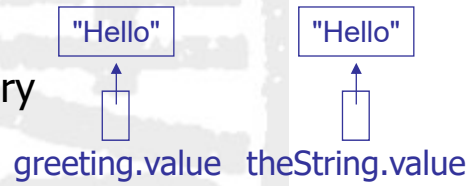
StringVar Copy Constructor

- A copy constructor is a constructor with one parameter of the same type as the class
  - The parameter is a call-by-reference parameter
  - The parameter is usually a constant parameter
  - The constructor creates a complete, independent copy of its argument
- The StringVar copy constructor creates a new dynamic array for a copy of the argument
  - Making a new copy, protects the original from changes

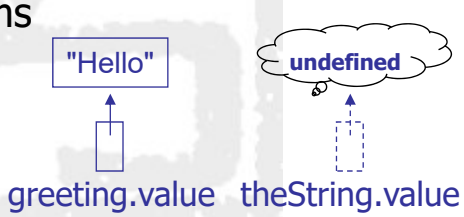
```
StringVar::StringVar(const StringVar& stringObject)
                    : maxLength(stringObject.length())
{
        value = new char[maxLength+ 1];
        strcpy(value, stringObject.value);
}
```

Copy Constructor Demonstration

- Using the same example, but with a copy constructor defined
  - greeting.value and theString.value point to different locations in memory



- When theString goes out of scope, the destructor is called, returning theString.value to the freestore
  - greeting.value still exists and can be accessed or deleted without problems



When To Include a Copy Constructor

- When a class definition involves pointers and dynamically allocated memory using "new"
  - Classes that do not involve pointers and dynamically allocated memory do not need copy constructors
- The big three include
  - The copy constructor
  - The assignment operator
  - The destructor
- If you need to define one, you need to define all

有 new 的 class => need copy constructor
else => overloading "="

Overloading = (Assignment)

- Given two objects, the following assignment is legal but encounters the pointer issue
  - StringVar string(10), string2(20);
    string1 = string2;
- The solution is to overload the assignment operator = so it works for StringVar
  - operator = is overloaded as a member function
  - Ex: void operator=(const StringVar& rightSide);
    - rightSide is the argument from the right side of the = operator

First Draft of = for StringVar

- Compares the lengths of the two StringVar's
- If there are too many characters, only copy as many characters as the max capacity in the left-hand object
- Makes an independent copy of the right-hand object in the left-hand object

```
void StringVar::operator=(const StringVar&  rightSide)
{
      int newLength = strlen(rightSide.value);
      if (( newLength) > maxLength)
         newLength = maxLength;

      for(int i = 0; i < newLength; i++)
         value[i] = rightSide.value[i];

      value[newLength] = '\0';
}
```

Another Attempt of = for StringVar

- Usually we want a copy of the right-hand argument regardless of its size
- We need to delete the array in the left-hand argument and allocate a new array large enough for source array

```
void StringVar::operator=(const StringVar&  rightSide)
{
     delete [ ] value;
     int newLength = strlen(rightSide.value);
     maxLength = newLength;

     value = new char[maxLength + 1];

     for(int i = 0; i < newLength; i++)
         value[i] = rightSide.value[i];

     value[newLength] = '\0';
}
```

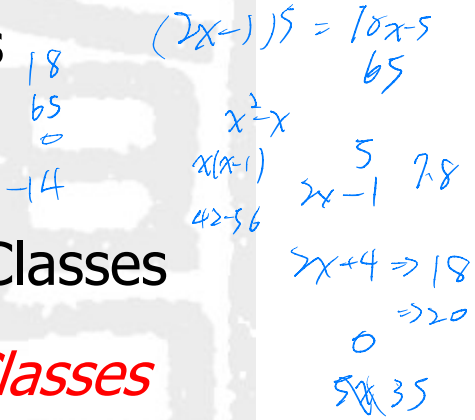
A Better Version of = for StringVar

- What happens if we happen to have the same object on each side of the assignment operator?
  - myString = myString;
- If the array in the left-hand argument is deleted, we will have no source array to copy from
  - Pre-check condition is required

```
void StringVar::operator=(const StringVar&  rightSide)
{
    int newLength = strlen(rightSide.value);
    if (newLength > maxLength)     //delete value only
    {                              // if more space is needed
        delete [ ] value;
        maxLength = newLength;
        value = new char[maxLength + 1];
    }
    ……
```

Overview

- 5.1 Friend Functions
- 5.2 const Parameters in Classes
- 5.3 Overloading Operators
- 5.4 Arrays and Classes
- 5.5 Dynamic Variables in Classes
- *5.6 Using this Pointer in Classes*



Using the *this* Pointer

- Do member functions know which object's data members to manipulate?
  - How to represent the concept of "this" object?
- In C++, every object has access to its own address through a pointer called *this* (a reserved keyword)
  - The *this* pointer is an implicit argument to each of the object's non-static member functions
- Objects can use the *this* pointer to reference their members implicitly or explicitly (ex: this->member)
  - Open a backdoor for your object?? → should be careful
- The type of the this pointer depends on the object
  - If the member function using this pointer is declared const, it is also treated as a pointer to constant object

Demo the *this* Pointer

```
#include <iostream>
using namespace std;

class Test
{
 public:
   Test(int value=0);
   void print( ) const;
private:
   int x;
};

Test::Test(int value) : x(value)
{
    //Body intentionally blank.
}
```

```
void Test::print() const
{
    // directly access the member x
    cout << "        x = " << x << endl;

    // use this pointer to access the member x
     cout << "  this->x = " << this->x << endl;

    // use this pointer to access the member x
     cout << "(*this).x = " << (*this).x << endl;
}

int main()
{
    Test testObject(12);
    testObject.print();
}
```

```
        x = 12
  this->x = 12
(*this).x = 12
```


Cascaded Member Function Calls

- Another use of the *this* pointer is to enable cascaded member-function calls
  - Return the reference of current object as the subject for the next member function call
  - Invoking multiple functions in the same statement
- In next example, the class Time's set functions are modified to return a reference to a Time object
- How does it work?
  - Remember the dot operator (.) associates from left to right
  - After t.setHour(18) is executed, it returns a reference to the object t
  - t.setHour(18).setMinute(30) becomes t.setMinute(30)

Demo Cascaded Function Call (1/3)

```
#include <iostream>
#include <iomanip>
using namespace std;

class Time
{
 public:
   Time( int = 0, int = 0, int = 0 );
 // set functions that enable cascading
   Time &setTime( int, int, int ); // set all variables
   Time &setHour( int );      // set hour
   Time &setMinute( int );    // set minute
   Time &setSecond( int );    // set second
 // get functions declared const
   int getHour() const;       // return hour
   int getMinute() const;     // return minute
   int getSecond() const;     // return second
 // print functions declared const
   void printUniversal() const; // universal time
   void printStandard() const; // standard time
 private:
   int hour;    // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
};  // end class Time

 // default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
}  // end Time constructor

 // set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s )
{
   setHour( h );
   setMinute( m );
   setSecond( s );
   return *this;  // enables cascading
}  // end function setTime
```


Demo Cascaded Function Call (2/3)

```
 // set hour value
Time &Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   return *this;  // enables cascading
}  // end function setHour

 // set minute value
Time &Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   return *this;  // enables cascading
}  // end function setMinute

// set second value
Time &Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0;
   return *this;  // enables cascading
}  // end function setSecond

 // get hour value
int Time: :getHour() const
{
   return hour;
}  // end function getHour

 // get minute value
int Time: :getMinute() const
{
   return minute;
}  // end function getMinute

 // get second value
int Time: :getSecond() const
{
   return second;
}  // end function getSecond
```

Demo Cascaded Function Call (3/3)

```
// print in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
   cout « setfill( '0' ) « setw( 2 ) « hour « ":"
   « setw( 2 ) « minute « ":" « setw( 2 )
   « second;
} // end function printUniversal

// print in standard-time format (AM or PM)
void Time::printStandard() const
{
   cout « ( ( hour == 0 || hour == 12 )
         ? 12 : hour % 12 )
   « ":" « setfill( '0' ) « setw( 2 ) « minute
   « ":" « setw( 2 ) « second
   « ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

```
int main()
{
   Time t;  // create Time object

// cascaded function calls
   t.setHour(18).setMinute(30).setSecond(22);

// output time in universal and standard formats
   cout « "Universal time: ";
   t.printUniversal();
   cout « "\nStandard time: ";
   t.printStandard();

// cascaded function calls
   cout « "\n\nNew standard time:
   t.setTime( 20, 20, 20 ).printStandard();
   cout « endl;
} // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```


Handle Prefix/Postfix Operators

- As you may know, increment/decrement operators can be prefix or postfix
  - Prefix: ++x, --x
  - Postfix: y++, y--
- All prefix/postfix operators need lvalues
  - ++i: ok, the integer variable i can appear on the left-hand side of an assignment operator
  - ++5: error, 5 is even not a variable, which cannot appear on the left-hand side of an assignment operator
- What is returned in a prefix/postfix operation?
  - Prefix increment/decrement operators return lvalues
  - Postfix increment/decrement operators don't
    - They return constant object instead

Demo Prefix/Postfix Operators

```
class LLint    // class for long precision integer
{
 public:
    LLint( int );
    LLint( );
    // increment & decrement operators
    LLint & operator ++( ); // prefix ++
    const LLint operator ++( int );  // postfix ++
    LLint & operator --( ); // prefix --
    const LLint operator --( int );  // postfix –
    // other overloaded operators, such as +=
 private:
    // implementation dependent, ex: int array
}; // end class Llint
```

Just a mark for postfix. Not a real int

```
 // prefix increment operator
LLint &LLint::operator++( )
{
    *this += 1;  // use overloaded +=
    return *this;
}  // end prefix increment
```

No calls to copy ctor!!

```
 // postfix increment operator
const LLint LLint::operator++( int )
{
    LLint old(*this); // invoke copy constructor
    ++(*this);        // invoke prefix ++
    return old;       // return the previous value
}  // end postfix increment

// You can implement -- in similar way

int main()
{
    LLint i=10;
    ++i;       // i = 11, --> i.operator++()
    i++;       // i = 12, --> i.operator++(0)
    ++++i;   // i = 14, -->
               // (i.operator++()).operator++()
    i++++;  // error
}
```

LLint can act just like int !!

Summary of Prefix/Postfix Ops

- Both unary ++/-- operators can be prefix or postfix
  - Postfix operator has a redundant int in its argument
- Conventions
  - They are overloaded by non-static member functions
  - Prefix ++/-- return *this
  - Postfix ++/-- return const object
- Typically, prefix is more efficient than postfix
  - No extra call to copy constructor
  - Prefer using prefix ++/-- whenever possible
- Implementing new functions based on old functions improves maintainability
  - Ex: prefix++ uses +=; postfix++ uses prefix++