Chapter 11
Friends, Overloaded Operators, and Arrays in Classes

Overview
11.1 Friend Functions
11.2 Overloading Operators
11.3 Arrays and Classes
11.4 Classes and Dynamic Arrays

11.1 Friend Functions

Friend Function
- Class operations are typically implemented as member functions
- Some operations are better implemented as ordinary (nonmember) functions

Program Example: An Equality Function
- The DayOfYear class from Chapter 10 can be enhanced to include an equality function
- An equality function tests two objects of type DayOfYear to see if their values represent the same date
- Two dates are equal if they represent the same day and month

Declaration of The equality Function
- We want the equality function to return a value of type bool that is true if the dates are the same
- The equality function requires a parameter for each of the two dates to compare
- The declaration is
  ```
  bool equal(DayOfYear date1, DayOfYear date2);
  ```
- Notice that equal is not a member of the class DayOfYear
Defining Function equal

- The function equal, is not a member function
  - It must use public accessor functions to obtain the day and month from a DayOfYear object
- equal can be defined in this way:

```cpp
bool equal(DayOfYear date1, DayOfYear date2) {
    return (date1.get_month() == date2.get_month())
        &&
        date1.get_day() == date2.get_day();
}
```

- The equal function can be used to compare dates in this manner
  ```cpp
  if (equal(today, bach_birthday))
      cout << "It's Bach's birthday!";
  ```

A complete program using function equal is found in Display 11.1 (1)
Display 11.1 (2)
Display 11.1 (3)

Is equal Efficient?

- Function equal could be made more efficient
  - Equal uses member function calls to obtain the private data values
  - Direct access of the member variables would be more efficient (faster)

A More Efficient equal

- As defined here, equal is more efficient, but not legal
  ```cpp
  bool equal(DayOfYear date1, DayOfYear date2) {
      return (date1.month == date2.month)
              &&
              date1.day == date2.day;
  }
  ```

- The code is simpler and more efficient
- Direct access of private member variables is not legal!

Friend Functions

- Friend functions are not members of a class, but can access private member variables of the class
  - A friend function is declared using the keyword friend in the class definition
    - A friend function is not a member function
    - A friend function is an ordinary function
    - A friend function has extraordinary access to data members of the class
  - As a friend function, the more efficient version of equal is legal

Declaring A Friend

- The function equal is declared a friend in the abbreviated class definition here
  ```cpp
class DayOfYear {
    public:
        friend bool equal(DayOfYear date1, DayOfYear date2);
    // The rest of the public members
    private:
        // the private members
    }
```

Using A Friend Function

- A friend function is declared as a friend in the class definition
- A friend function is defined as a nonmember function without using the ‘::’ operator
- A friend function is called without using the ‘.’ operator

Friend Declaration Syntax

- The syntax for declaring friend function is
class class_name
{
  public:
    friend Declaration_for_Friend_Function_1
    friend Declaration_for_Friend_Function_2
    ...
    Member_Function_Declarations
  private:
    Private_Member_Declarations
};

Are Friends Needed?

- Friend functions can be written as non-friend functions using the normal accessor and mutator functions that should be part of the class
- The code of a friend function is simpler and it is more efficient

Choosing Friends

- How do you know when a function should be a friend or a member function?
  - In general, use a member function if the task performed by the function involves only one object
  - In general, use a nonmember function if the task performed by the function involves more than one object
  - Choosing to make the nonmember function a friend is a decision of efficiency and personal taste

Example: The Money Class (version 1)

- Display 11.3 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

Characters to Integers

- Notice how function input (Display 11.3) processes the dollar values entered
  - First read the character that is a $ or a –
    - If it is the -, set the value of negative to true and read the $ sign which should be next
  - Next read the dollar amount as a long
  - Next read the decimal point and cents as three characters
    - digit_to_int is then used to convert the cents characters to integers
digit_to_int

- digit_to_int is defined as
  ```
  int digit_to_int(char c)
  {
    return (int (c) - int ('0'));
  }
  ```
- A digit, such as '3' is parameter c;
- This is the character '3' not the number 3;
- The type cast int(c) returns the number that implements the character stored in c;
- The type cast int('0') returns the number that implements the character '0'.

int(c) – int ('0')?

- The numbers implementing the 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') returns integer 0
- If c is '1',
  - int(c) – int ('0') returns integer 1

Leading Zeros

- Some compilers interpret a number with a leading zero as a base 8 number,
- Base 8 uses digits 0 – 7
- Using 09 to represent 9 cents could cause an error
- The digit 9 is not allowed in a base 8 number
- The ANSI C++ standard is that input should be interpreted as base 10 regardless of a leading zero

Parameter Passing Efficiency

- A call-by-value parameter less efficient than a call-by-reference parameter
  - The parameter is a local variable initialized to the value of the argument
    - This results in two copies of the argument
  - A call-by-reference parameter is more efficient
    - The parameter is a placeholder replaced by the argument
    - There is only one copy of the argument

Class Parameters

- It can be much more efficient to use call-by-reference parameters when the parameter is of a class type
- When using a call-by-reference parameter
  - If the function does not change the value of the parameter, mark the parameter so the compiler knows it should not be changed

const Parameter Modifier

- To mark a call-by-reference parameter so it cannot be changed:
  - Use the modifier const before the parameter type
  - The parameter becomes a constant parameter
  - const used in the function declaration and definition
**const Parameter Example**
- Example (from the Money class of Display 11.3):
  - 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 Considerations**
- When a function has a constant parameter, the compiler will make certain the parameter cannot be changed by the function
  - What if the parameter calls a member function?
    ```cpp
    Money add(const Money& amount1, const Money& amount2)
    {
      ...
      amount1.input( cin );
    }
    ```
    - The call to input will change the value of amount1!
    - The compiler will not accept this code

**const And Accessor Functions**
- Will the compiler accept an accessor function call from the constant parameter?
  ```cpp
  Money add(const Money& amount1, const Money& amount2)
  {
    ...
    amount1.output( cout );
  }
  ```
  - The compiler will not accept this code
  - There is no guarantee that output will not change the value of the parameter

**const Modifies Functions**
- If a constant parameter makes a member function call...
  - The member function called must be marked so the compiler knows it will not change the parameter
  - const is used to mark functions that will not change the value of an object
  - const is used in the function declaration and the function definition

**Function Declarations with const**
- To declare a function that will not change the value of any member variables:
  - Use const after the parameter list and just before the semicolon
    ```cpp
    class Money
    {
      public:
      ...
      void output( ostream& outs) const ;
      ...
    };
    ```

**Function Definitions with const**
- To define a function that will not change the value of any member variables:
  - Use const in the same location as the function declaration
    ```cpp
    void Money::output(ostream& outs) const
    {
      // output statements
    }
    ```
const Problem Solved

- Now that output is declared and defined using the const modifier, the compiler will accept this code
- Money add(const Money& amount1, const Money& amount2)
  
  
  
  amount1.output(cout);

const Wrapup

- Using const to modify parameters of class types improves program efficiency
- const is typed in front of the parameter's type
- Member functions called by constant parameters must also use const to let the compiler know they do not change the value of the parameter
- const is typed following the parameter list in the declaration and definition

Display 11.4

Use const Consistently

- Once a parameter is modified by using const to make it a constant parameter
- Any member functions that are called by the parameter must also be modified using const to tell the compiler they will not change the parameter
- It is a good idea to modify, with const, every member function that does not change a member variable

Section 11.1 Conclusion

- Can you
  - Describe the promise that you make to the compiler when you modify a parameter with const?
  - Explain why this declaration is probably not correct?

    class Money
    
    { …
    
    public:
    
    void input(istream& ins) const;
    
    …
    
    };

11.2

Overloading Operators

- In the Money class, function add was used to add two objects of type Money
- In this section we see how to use the '+' operator to make this code legal:

  Money total, cost, tax;

  …

  total = cost + tax;

  // instead of total = add(cost, tax);
Operators As Functions
- An operator is a function used differently than an ordinary function
- An ordinary function call enclosed its arguments in parenthesis
  
- With a binary operator, the arguments are on either side of the operator
- Example: add(cost, tax)
- cost + tax

Operator Overloading
- Operators can be overloaded
- The definition of operator + for the Money class is nearly the same as member function add
- To overload the + operator for the Money class
  - Use the name + in place of the name add
  - Use keyword operator in front of the +
  - Example:
    
    ```cpp
    friend Money operator + (const Money& amount1, const Money& amount2);
    ```

Operator Overloading Rules
- At least one argument of an overloaded operator must be of a class type
- An overloaded operator can be a friend of a class
- New operators cannot be created
- The number of arguments for an operator cannot be changed
- The precedence of an operator cannot be changed
- At least one argument of an overloaded operator must be of a class type
- An overloaded operator can be a friend of a class
- New operators cannot be created
- The number of arguments for an operator cannot be changed
- The precedence of an operator cannot be changed
- , ::, *, and ? cannot be overloaded

Example: Overloading Operators
- The Money class with overloaded operators + and == is demonstrated in

Automatic Type Conversion
- With the right constructors, the system can do type conversions for your classes
- This code (from Display 11.5) actually works

  ```cpp
  Money base_amount(100, 60), full_amount;
  full_amount = base_amount + 25;
  ```

- The integer 25 is converted to type Money so it can be added to base_amount!
- How does that happen?

Type Conversion Event 1
- When the compiler sees base_amount + 25, it first looks for an overloaded + operator to perform

  ```cpp
  Money_object + integer
  ```

- If it exists, it might look like this

  ```cpp
  friend Money operator + (const Money& amount1, const int& amount2);
  ```
Type Conversion Event 2

- When the appropriate version of + is not found, the compiler looks for a constructor that takes a single integer
  - The Money constructor that takes a single parameter of type long will work
  - The constructor Money(long dollars) converts 25 to a Money object so the two values can be added!

Type Conversion Again

- Although the compiler was able to find a way to add
  base_amount + 25
  this addition will cause an error
  base_amount + 25.67
- There is no constructor in the Money class that takes a single argument of type double

A Constructor For double

- To permit base_amount + 25.67, the following constructor should be declared and defined

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

Overloading Unary Operators

- Unary operators take a single argument
- The unary – operator is used to negate a value
  \[ x = -y \]
- ++ and -- are also unary operators
- Unary operators can be overloaded
  - The Money class of Display 11.6 can include
    - A binary – operator
    - A unary – operator

Overloading -

- Overloading the – operator with two parameters allows us to subtract Money objects as in
  ```cpp
  Money amount1, amount2, amount3;
  ... 
  amount3 = amount1 – amount2;
  ```
- Overloading the – operator with one parameter allows us to negate a money value like this
  ```cpp
  amount3 = -amount1;
  ```

Overloading << and >>

- The insertion operator << is a binary operator
  - The first operand is the output stream
  - The second operand is the value following <<
  ```cpp
  cout << "Hello out there.\n";
  ```
- Overloading the >> operator with two parameters allows us to right shift a Money value as in
  ```cpp
  Operand1 >> Operand2
  ```

Display 11.6
Replacing Function output

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

Overloaded << Declaration

- Based on the previous example, << should return its first argument, the output stream.
- This leads to a declaration of the overloaded << operator for the Money class:
  ```
  class Money
  {
  ...
  friend ostream& operator << (ostream& outs,
                             const Money& amount);
  ...
  }
  ```

Overloaded << Definition

- The following defines the << operator
  ```
  ostream& operator <<(ostream& outs,
                       const Money& amount)
  {
    //Same as the body of Money::output in Display 11.3 (except all_cents is replaced with amount.all_cents) >
    return outs;
  }
  ```

Return ostream&?

- The & means a reference is returned.
  - So far all our functions have returned values.
  - The value of a stream object is not so simple to return.
    - The value of a stream might be an entire file, the keyboard, or the screen!
  - We want to return the stream itself, not the value of the stream.
  - The & means that we want to return the stream, not its value.

Explanation: Insertion operator

- The following defines the << operator
  ```
  ostream& operator <<(ostream& outs,
                       const Money& amount);
  ```

Money m1(1, 50), m2;
cout << m1 << m2;  // consecutive calls
// Equivalent to call
Operator<<( operator<<( cout, m1), m2);
Overloading >>
- Overloading the >> operator for input is very similar to overloading the << for output
- >> could be defined this way for the Money class

```cpp
istream& operator >> (istream& ins, 
Money& amount);
```

- {  *
- This part is the same as the body of  *
- Money::input in Display 11.3 (except that  *
- all_cents is replaced with amount.all_cents)  *
- return ins;
- }

Display 11.8 (1-4)

Operator Overloading, Member Function
- Appendices 8, operator+, binary plus
- Appendices 6, operator[ ], array index
- Appendices 7, this pointer
- Appendices 5, inline functions

Equality Operator, Member function:
- Overload the equality operator for the same type
- Using friend from the example Money in text

```cpp
class DayOfYear
{
public:
    void output( );
    int month;
    int day;
    bool operator==(DayOfYear& rhs)
    {
        return month == rhs.month && day == rhs.day;
    }
};
```

Assignment Operator, Member function:
- Must use the member function to overload

```cpp
Money Money::operator =(const Money& amount2)
{
    return Money(0, all_cents = amount2.all_cents);
}
```

```cpp
Money m1, m2, m3(1, 50), m;
```

```cpp
m1 = m2 = m3; // consecutive calls
// Equivalent to call
m1.operator=(m2.operator=(m3));
```

Assignment Operator, Member function:
- Usually we overload the assignment operator for the same type, but you also can overload this operator between the different type if necessary

```cpp
struct B
{
    int i;
    void operator=(C& c) { i = c.i; }
};
struct C
{
    int i;
};
```

```cpp
int main()
{
    B b = {2};
    C c = {1};
    b = c;
}
```
Circular Reference in C++

```cpp
struct B {
    int i;
    void operator=(const C& c);
};
struct C {
    int i;
    void operator=(const B& b) { i = b.i; }
};
```

How to solve this?

Operators cannot be overloaded

- `?` (conditional)
- `.` (member selection)
- `*` (member selection with pointer-to-member)
- `::` (scope resolution)
- `sizeof` (object size information)

To understand why, read "Why can't I overload dot, ::, sizeof, etc.?" at the Bjarne Stroustrup's C++ Style and Technique FAQ

http://www.research.att.com/~bs/bs_faq2.html#overload-dot

---

Section 11.2 Conclusion

- Can you describe the purpose of making a function a friend?
- Describe the use of constant parameters?
- Identify the return type of the overloaded operators `<<` and `>>`?

---

Arrays and Classes

- Arrays can use structures or classes as their base types
- Example:
  ```cpp
  struct WindInfo {
      double velocity;
      char direction;
  }
  WindInfo data_point[10];
  ```

---

Accessing Members

- When an array's base type is a structure or a class...
  - Use the dot operator to access the members of an indexed variable
  ```cpp
  for (i = 0; i < 10; i++)
  {
      cout << "Enter velocity: ";
      cin >> data_point[i].velocity;
      ...;
  }
  ```
### An Array of Money
- The `Money` class of Chapter 11 can be the base type for an array.
- When an array of classes is declared:
  - The default constructor is called to initialize the indexed variables.
- An array of class `Money` is demonstrated in [Display 11.9 (1-3)](slide-11-57).

### Arrays as Structure Members
- A structure can contain an array as a member.
  - Example:
    ```cpp
default struct Data{
    double time[10];
    int distance;
    }
```
- `Data my_best;` is declared.
- `my_best` contains an array of type `double`.

### Accessing Array Elements
- 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: `my_best.time[i]` references the i'th indexed variable of the variable `time` in the structure `my_best`.

### Arrays as Class Members
- Class `TemperatureList` includes an array.
  - The array, named `list`, contains temperatures.
  - Member variable `size` is the number of items stored.
  ```cpp
class TemperatureList{
  public:
    TemperatureList();
    //Member functions
  private:
    double list[MAX_LIST_SIZE];
    int size;
  }
```
- To create an object of type `TemperatureList`:
  ```cpp
  TemperatureList my_data;
  ```
- To add a temperature to the list:
  ```cpp
  my_data.add_temperature(77);
  ```
- A check is made to see if the array is full.
- `<<` is overloaded so output of the list is:
  ```cpp
  cout << my_data;
  ```

### Overview of TemperatureList
- To create an object of type `TemperatureList`:
  ```cpp
  TemperatureList my_data;
  ```
- To add a temperature to the list:
  ```cpp
  my_data.add_temperature(77);
  ```
- `<<` is overloaded so output of the list is:
  ```cpp
  cout << my_data;
  ```

### Section 11.3 Conclusion
- Can you
  - Declare an array as a member of a class?
  - Declare an array of objects of a class?
  - Write code to call a member function of an element in an array of objects of a class?
  - Write code to access an element of an array of integers that is a member of a class?
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
- In this section you will see a class using a dynamic array as a member variable.

Example: A String Variable Class

- We will define 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 Constructors

- The default StringVar constructor creates an object with a maximum string length of 100
- Another StringVar constructor takes an argument of type int which determines the maximum string length of the object
- A third StringVar constructor takes a C-string argument and...
  - sets maximum length to the length of the C-string
  - copies the C-string into the object's string value

The StringVar Interface

- In addition to constructors, the StringVar interface includes:
  - Member functions
    - int length();
    - void input_line(istream& ins);
    - friend ostream& operator << (ostream& outs, const StringVar& the_string);
  - Copy Constructor …discussed later
  - Destructor …discussed later

A StringVar Sample Program

- Using the StringVar interface of Display 11.11, we can write a program using the StringVar class
  - The program uses function conversation to
    - Create two StringVar objects, your_name and our_name
    - your_name can contain any string max_name_size or shorter in length
    - our_name is initialized to "Borg" and can have any string of 4 or less characters
The StringVar Implementation

- StringVar uses a dynamic array to store its string
- StringVar constructors call new to create the
dynamic array for member variable value
- '\0' is used to terminate the string
- The size of the array is not determined until the
array is declared
  - Constructor arguments determine the size

Dynamic Variables

- Dynamic variables do not "go away" unless
delete is called
- Even if a local pointer variable goes away at
the end of a function, the dynamic variable it
pointed to remains unless delete is called
- A user of the StringVar class could not know
that a dynamic array is a member of the class,
so could not be expected to call delete when
finished with a StringVar object

Destructors

- A destructor is a member function that is called
automatically when an object of the class goes
out of scope
  - The destructor contains code to 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

- The destructor in the StringVar class must call
delete [] to return the memory of any dynamic
variables to the freestore
  - Example: StringVar::~StringVar( )
    {
      delete [] value;
    }

Pointers as Call-by-Value Parameters

- Using pointers as call-by-value parameters yields
results you might not expect
  - Remember that parameters are local variables
    - No change to the parameter should cause a change to the
    argument
  - The value of the parameter is set to the value of the
  argument (an address is stored in a pointer variable)
    - The argument and the parameter hold the same address
  - If the parameter is used to change the value pointed to,
this is the same value pointed to by the argument!

Copy Constructors

- The problem with using call-by-value parameters
  with pointer variables is solved by the
  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
StringVar Copy Constructor

- This code for 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& string_object) :
    max_length(string_object.length())
    { 
      value = new char[max_length+ 1];
      strcpy(value, string_object.value);
    }

Calling a Copy Constructor

- A copy constructor can be called as any other constructor when declaring an object
- The copy constructor is called automatically
  - When a class object is defined and initialized by an object of the same class
  - When a function returns a value of the class type
  - When an argument of the class type is plugged in for a call-by-value parameter

The Need For a Copy Constructor

- This code (assuming no copy constructor) illustrates the need for a copy constructor
  - void show_string(StringVar the_string) 
    { … }
  - StringVar greeting("Hello");
  - show_string(greeting);
  - cout << greeting << endl;
  - When function show_string is called, greeting is copied into the_string
    - the_string.value is set equal to greeting.value

The Need For a Copy Constructor (cont.)

- Since greeting.value and the_string.value are pointers, they now point to the same dynamic array
  - "Hello"
  - greeting.value
  - the_string.value

The Need For a Copy Constructor (cont.)

- Two problems now exist for object greeting
  - Attempting to output greeting.value is likely to produce an error
    - In some instances all could go OK
  - When greeting goes out of scope, its destructor will be called
    - Calling a destructor for the same location twice is likely to produce a system crashing error
Copy Constructor Demonstration
- Using the same example, but with a copy constructor defined
  - greeting.value and the_string.value point to different locations in memory

```
"Hello"       "Hello"
greeting.value  the_string.value
```

Copy Constructor Demonstration (cont.)
- When the_string goes out of scope, the destructor is called, returning the_string.value to the freestore

```
"Hello"       undefined
   greeting.value  the_string.value
```
- 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", include a copy constructor
- Classes that do not involve pointers and dynamically allocated memory usually do not need copy constructors created by yourself

The Big Three
- The big three include
  - The copy constructor
  - The assignment operator
  - The destructor
- If you need to define one, you need to define all

The Assignment Operator
- Given these declarations:
  `StringVar string(10), string2(20);`
  the statement `string1 = string2;` is legal
- But, since StringVar’s member value is a pointer, we have string1.value and string2.value pointing to the same memory location

Overloading =
- The solution is to overload the assignment operator = so it works for StringVar
  - `operator =` is overloaded as a member function
  - Example: `operator =` declaration

```
void operator=(const StringVar& right_side);
```
- Right_side is the argument from the right side of the = operator
Definition of =

- The definition of = for StringVar could be:

```cpp
void StringVar::operator= (const StringVar& right_side) {
  int new_length = strlen(right_side.value);
  if ((new_length) > max_length)  
    new_length = max_length;

  for(int i = 0; i < new_length; i++)
    value[i] = right_side.value[i];

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

= Details

- This version of = for StringVar
  - Compares the lengths of the two StringVar’s
  - Uses only as many characters as fit in the left hand StringVar object
  - Makes an independent copy of the right hand object in the left hand object

Problems with =

- The definition of operator = has a problem
  - Usually we want a copy of the right hand argument regardless of its size
  - To do this, we need to delete the dynamic array in the left hand argument and allocate a new array large enough for the right hand side's dynamic array
  - The next slide shows this (buggy) attempt at overloading the assignment operator

Another Attempt at =

```cpp
void StringVar::operator= (const StringVar& right_side)
{
  delete [] value;
  int new_length = strlen(right_side.value);
  max_length = new_length;
  value = new char[max_length + 1];
  for(int i = 0; i < new_length; i++)
    value[i] = right_side.value[i];

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

A New Problem With =

- The new definition of operator = has a problem
  - What happens if we happen to have the same object on each side of the assignment operator?
    - `my_string = my_string;`
  - This version of operator = first deletes the dynamic array in the left hand argument.
  - Since the objects are the same object, there is no longer an array to copy from the right hand side!

A Better = Operator

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

  for(int i = 0; i < new_length; i++)
    value[i] = right_side.value[i];

  value[new_length] = '\0';
}
```
Section 11.4 Conclusion

- Can you explain why an overloaded assignment operator is not needed when the only data consist of built-in types?
- Explain what a destructor does?
- Explain when a copy constructor is called?

**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!
DISPLAY 11.1 Program Using the StringVar Class (part 1 of 3)

```cpp
1 // This is the definition for the class StringVar.
2 // Whole values are strings. An object is declared as follows.
3 // Note that you use max_size, not max[;size]
4 // StringVar(0, max_size); // StringVar(0, max_size);
5 // where max_size is the longest string length allowed.
6 #include <iostream>
7 using namespace std;
8
9 class StringVar
10 {
11 public:
12 StringVar(int size);
13 // Initializes the object so it can accept string values up to size
14 // (in length). Sets the value of the object equal to the empty string.
15
16 (continued)
```

Display 11.1 (2/3)

```cpp

display 11.11 (2/3)

1 // This is only a demonstration function.
2 void StringVar::operator<< (ostream & outs, const StringVar the_string)
3 {
4 outs << the_string.value;
5 return outs;
6}
```

Display 11.1 (3/3)

```cpp

display 11.11 (3/3)
```

Display 11.12 Implementation of StringVar (part 2 of 2)

```cpp
46 void StringVar::input_line(istream & ins)
47 { 48 ins.getline(value, max_length + 1);
49 } 50
51 // Uses ofstream:
52 ofstream operator<<(ofstream outs, const StringVar the_string)
53 {
54 outs << the_string.value;
55 return outs;
56 }
```

Display 11.13 A Callable Parameter
DISPLAY 11.14  The Function Call sneaky(p):

1. Before call to sneaky:
   \[ p \rightarrow 77 \]
   \[ \text{temp} \rightarrow 99 \]

2. Value of \( p \) is plugged in for \( \text{temp} \):
   \[ p \rightarrow 77 \]
   \[ \text{temp} \rightarrow 99 \]

3. Change made to \(*\text{temp}\):
   \[ p \rightarrow 77 \]
   \[ \text{temp} \rightarrow 99 \]

4. After call to sneaky:
   \[ p \rightarrow 99 \]
   \[ \text{temp} \rightarrow 99 \]