C++ for Science and Engineering COSC3000/6000

2018 Spring Semester

Part XVI
Dynamic Arrays and Classes

1 Dynamic Arrays as Class Member Variables

- A class can have a member variable that is a dynamic array

1.1 Program Example: A String Variable Class

```
class StringVar
{
public:
    /// Initializes the object so it can accept string values of length 100
    /// or less. Set the value of the object euqal to empty string.
    StringVar();

    /// Initializes the object so it can accept string values up to size
    /// in length. Set the value of the object euqal to empty string.
    StringVar(int size);

    /// Precondition: The array a contains charactors terminated with '\0'.
    /// Initializes the object so its cvalue is the string stored in a and
    /// so that it can later be set to string values up to strlen(a) in
    /// length.
    StringVar(const char a[]);

    /// Copy constructor
    StringVar(const StringVar &string_object);

    /// Destructor
    ~StringVar();

    /// Returns the length of the current string values.
    size_t length() const;

    /// Precondition: If ins is a file input stream, then ins has been
    /// connected to a file.
    /// Action: The next text in the input stream ins, up to '\n', is
    /// copied to the calling object. If there is not sufficient room,
    /// then only as much as will fit is copied.
    void input_line(std::istream &ins);
```

```
    /// Overload the << operator so it can be used to putput values
    /// of type StringVar
    /// Precondition: If outs is a file output stream, then outs has
    /// already been connected to  file.
    friend std::ostream& operator <<(std::ostream &outs, const StringVar &string_object);
private:
    char *value;///< pointer to dynamic array that holds the string value.
    size_t max_length;///<declared max length of any string value.
};
```

1.1.1 The StringVar Constructors

- The default **StringVar** constructor creates an object with a maximum string length of 100

```
StringVar::StringVar():StringVar(100)// calling other constructor
{
}
```

- Another **StringVar** constructor takes an argument of **type int** which determines the maximum string length of the object

```
StringVar::StringVar(int size):max_length(size)
{
    value = new char[max_length + 1];
    value[0] = '\0';
}
```

- 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

```
StringVar::StringVar(const char a[]):max_length(strlen(a))
{
    value = new char[max_length + 1];
    strcpy(value,a);
}
```

- **Copy Constructors :** See below

1.1.2 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 **SringVar** 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

1.2 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 ~**

- The destructor in the **StringVar** class must call **delete [ ]** to return the memory of any dynamic variables to the freestore

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

1.3 Copy Constructors

- 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

```
    /// Copy constructor
    StringVar(const StringVar &string_object);
```

1.3.1 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);
}
```

1.3.2 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

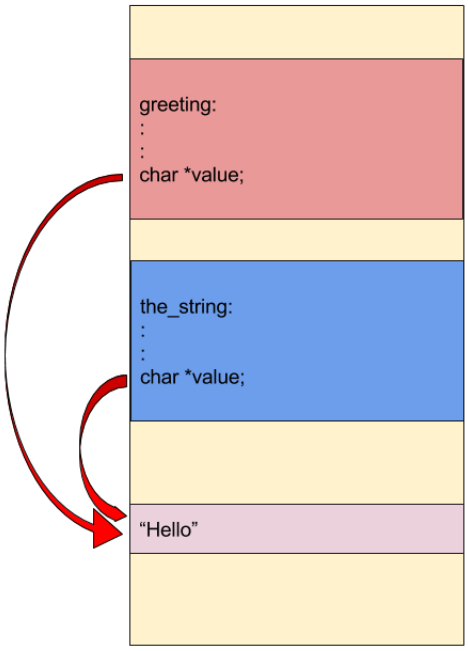
1.3.3 The Need For a Copy Constructor

- This code (**assuming no copy constructor**) illustrates the need for a copy constructor

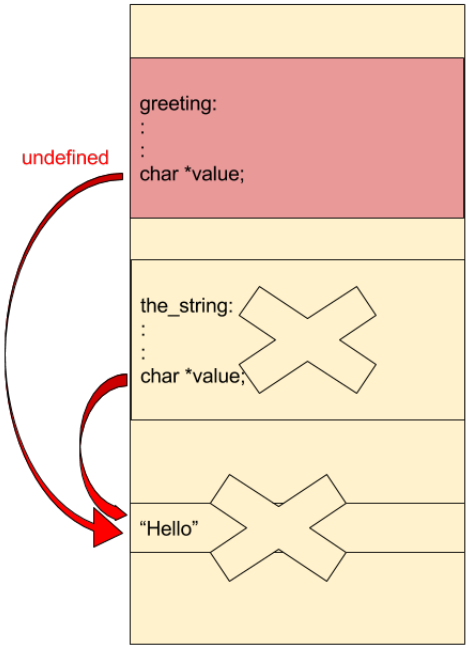
```
void show_string(StringVar the_string)
{
  // do something
}

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**



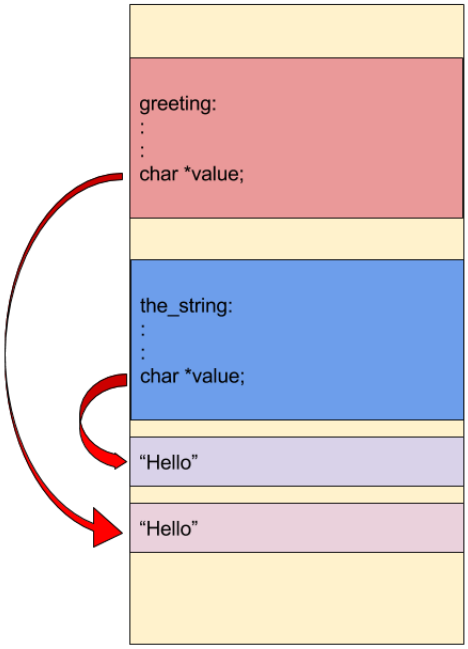
- Since **greeting.value** and **the_string.value** are pointers, they now point to the same dynamic array
- When **show_string** ends, the destructor for **the_string** executes, returning the dynamic array pointed to by **the_string.value** to the freestore
- **greeting.value now points to memory that has been given back to the freestore!**



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

1.3.4 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 do not need copy constructors