Dynamic Array as Member Variable

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.

Program Example: A `StringType` Class

- We will define the class `StringType`
  - `StringType` objects will be "strings"
  - `StringType` objects use dynamic arrays whose size is determined when the program is running
  - The `StringType` class is similar to the `string` class in C++

Program Example: A `StringType` Class

- A very important question:

what are the member variable(s) of the `StringType` class?

  - **A dynamic C-string!**

Why dynamic?

What is the type of the variable that "holds" a dynamic C-string?

What else do we need?

The `StringType` Constructors

- The default `StringType` constructor could create an empty string
- Another `StringType` constructor takes an argument of type `int` which determines the maximum string length of the object (this constructor is actually not very useful)
- A useful `StringType` constructor that takes a C-string argument and…
  - sets length to the length of the C-string
  - copies the C-string into the object's string value

The `StringType` Interface

- In addition to constructors, the `StringType` interface includes:
  - Member functions

    ```
    int length( ); // returns the actual length
    ```

  - Friend function

    ```
    friend ostream& operator << (ostream& outs,
                                 const StringType& the_string);
    ```

  - Copy Constructor
  - Destructor

The `StringType` Implementation

- `StringType` uses a dynamic array to store its string
  - `StringType` constructors call `new` to create the dynamic array for member pointer variable `value`
  - '`\0`' is used to terminate the string
  - The size of the array is not determined until the array is declared
    - size could be determined by constructor arguments

Dynamic Variables

- Recall that dynamic variables do not "go away" unless `delete` is performed
  - Even if a local pointer variable goes away at the end of a function, the dynamic object it pointed to remains on "heap" unless `delete` is performed
  - A user (program) of the `StringType` 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 `StringType` object

~StringType

- The destructor in the `StringType` class must call `delete[]` to return the memory of the dynamic array (the member variable) to the heap

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

Pointers as Pass-by-Value Parameters

- Recall that using pointers as pass-by-value formal parameters yields results you might not expect
  - The value of the formal parameter is set to the value of the argument (actual parameter)
    - **The argument and the parameter hold the same address**
  - If the formal parameter is used to change the value pointed to, this is the same value pointed to by the actual parameter (argument)!
  - Recall the function `sneaky` we examined before …

`StringType` Copy Constructor

- The following is `StringType` copy constructor
  - Creates a new dynamic array for a copy of the argument
    - Why do we need it?
    - Making a new copy, protects the original from changes

```
StringType::StringType ( const StringType& string_object)
                              : max_length(string_object.length() )
{
      value = new char[max_length+ 1];
      strcpy(value, string_object.value);
}
```

The Need For a Copy Constructor

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

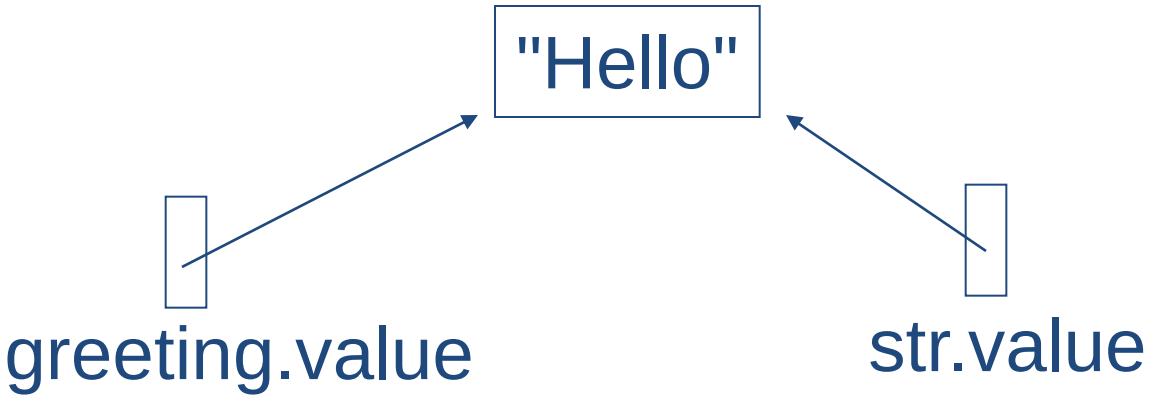
```
void do_sth(StringType str)
{…}

StringType greeting("Hello");
do_sth(greeting);
cout << greeting << endl;
```

  - When function `do_sth` is called, `greeting` is copied into `str`
  - `str.value` is set equal to `greeting.value`

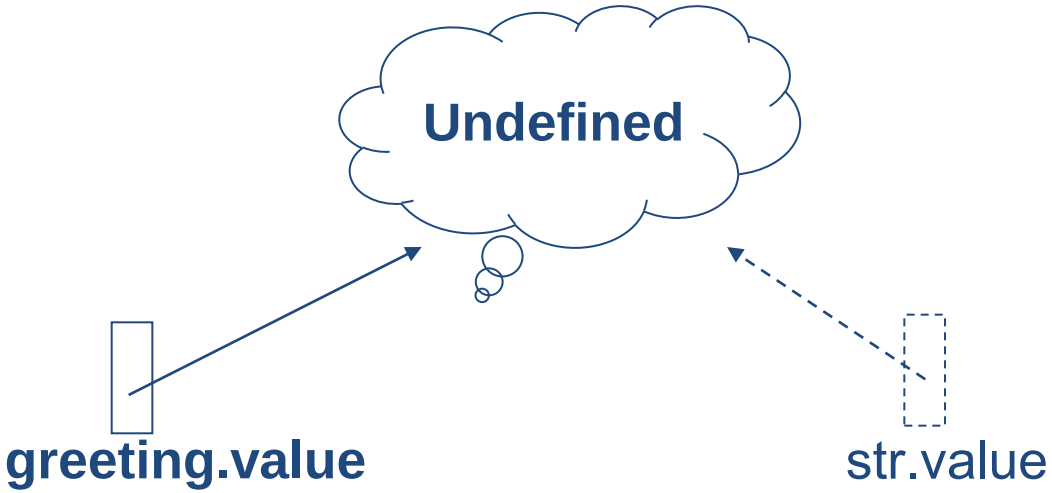
The Need For a Copy Constructor (cont)

- Since `greeting.value` and `str.value` are pointers, they now point to the same dynamic array



The Need For a Copy Constructor (cont.)

- When `do_sth` ends, the destructor for `str` executes, returning the dynamic array pointed to by `str.value` to the heap



- `greeting.value` now points to memory that has been deleted (given back to the heap)!

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
  - 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 `str.value` point to different locations in memory

Copy Constructor Demonstration (cont)

- When `str` goes out of scope, the destructor is called, returning `str.value` to the heap

  - `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 do not need copy constructors

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**

Shallow and Deep Copies

- Shallow copy
  - Assignment copies only member variable contents over
  - Default assignment and copy constructors
- Deep copy
  - Pointers, dynamic memory involved
  - Must dereference pointer variables to "get to" data for copying
  - Write your own assignment operator and copy constructor in this case!

The Assignment Operator

- Given these declarations:

```
StringType string1(10), string2(20);
```

  the statement

```
string1 = string2;
```

  is legal

- But, since `StringType`'s member value is a pointer, we have `string1.value` and `string2.value` pointing to the same memory location – again, problematic!

Overloading Assignment operator =

- The solution is to overload the assignment operator **=** so it works for `StringType`

  `operator =` is overloaded as a member function

  Example:

```
void operator=(const StringType& right_side);
```

  – Right_side is the argument from the right side of the **=** operator

Definition of the overloaded operator =

- The definition of = for `StringType` could be:

```
void StringType::operator=  (const StringType&   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';
}
```

`operator =` Details

- This version of **=** for `StringType`
  - Compares the lengths of the two `StringType` objects
  - Uses only as many characters as fit in the left hand `StringType` 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 though) attempt at overloading the assignment operator

Another Attempt at operator =

```
void StringType::operator=  (const StringType&   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

```
void StringType::operator = (const StringType& right_side)
 {
     int new_length = strlen(right_side.value);
     if (new_length > max_length)          //delete value only
     {                                     // if more space
             delete [ ] value;             // 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';
}
```

A Second Look at Overloading Operator =

- If assignment operator returns reference
  - then **assignment "chains"** are possible, e.g.

```
a = b = c;
```

    - Sets `a` and `b` equal to `c`
    - Note: the two assignment operators are evaluated right-to-left

- Operator **=** must return "same type" as its left-hand side
  - To allow chains to work
  - The `this` pointer will help with this!

A Second Look at Operator = (cont)

- Recall: assignment operator must be member of the class
  - **It has one parameter**
  - **Left-operand is calling object**

    ```
    s1 = s2;
    ```

    - Think of it like: `s1.=(s2);`

- `s1 = s2 = s3;`
  - Requires `s1 = (s2 = s3);`
  - So `(s2 = s3)` must return object of `s1`'s type, and pass to "`s1 =` ";

Now, question: can you modify the overloaded operator = for `StringType` to make `s1 = s2 = s3` work?