Classes (Part3)

Prof. Seokin Hong

Agenda

- **Pointer**
- **Dynamic Array**
- **Dynamic Array and Classes**
  - Copy Constructor
  - Destructor
  - Overloading assignment operator

Pointers

Pointers

- **A pointer is the memory address of a variable**
  - Pointers "point" to a variable by telling where the variable is located

- **Declaring Pointers**
  - Example:

```
double  *p;  //declare a pointer variable p that can "point" to a variable of type double
             //the asterisk identifies p as a pointer variable
```

- **Multiple Pointer Declarations**
  - To declare multiple pointers in a statement, use the asterisk before each pointer variable
  - Example:

```
int *p1, *p2, v1, v2;   // p1 and p2 point to variables of type int
                        //v1 and v2 are variables of type int
```

Pointer Operators

- **& operator (“address of” Operator)**
  - used to determine the address of a variable which can be assigned to a pointer variable
    - Example:

```
p1 = &v1;      //p1 is now a pointer to v1
```

- *** operator (Dereferencing Operator)**
  - used to get the value that is stored in the memory location pointed by the pointer
    - Example:

```
v2 = *p1;
```

A Pointer Example

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

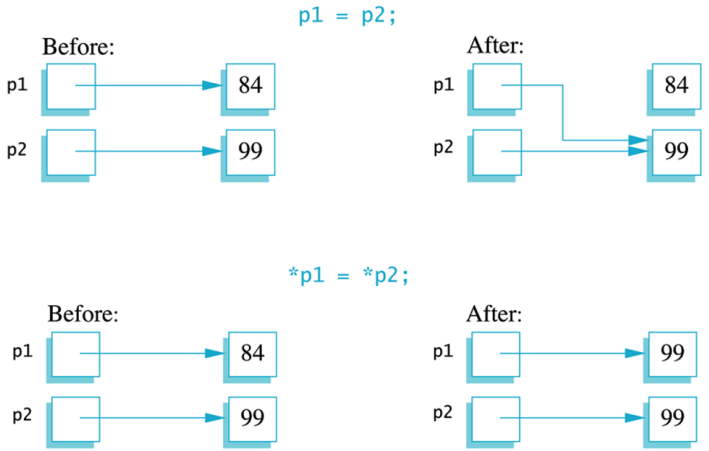
v1 and *p1 now refer to the same variable

```
output:
        42
        42
```

Pointer Assignment

- **The assignment operator = is used to assign the value of one pointer to another**
  - Example:

```
int *p1, *p2;
*p1=84;
*p2=99;
p1 = p2;        //causes *p2, *p1, and v1 points
                // to the same variable

*p1 = *p2;      //?
```



new and delete Operators

- **Dynamic memory allocation**
  - refers to performing memory allocation manually
  - dynamically allocated memory is allocated on Heap
  - **Why dynamic memory allocation?**
    - When it is impossible to know the required memory size at compile time.
    - When we want to allocate memory whenever we need and deallocate memory whenever we don’t need anymore
      - Example: linked list

- Dynamic memory allocation in C
  - malloc(), calloc()
  - free()

new and delete Operators (Cont.)

- **new operators**
  - Allocate memory and returns the address of the newly allocated memory if there is enough memory available
  - **Example**

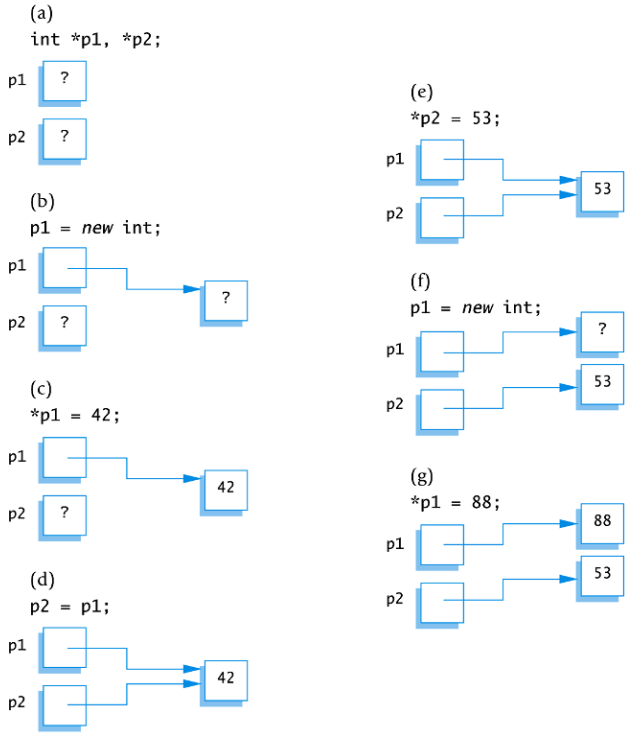
```
int* a = new int;
int* a = new int(10);              //allocate memory space and initialize it
float *q = new float(10.10);       //allocate memory space and initialize it
```

- **delete operators**
  - When dynamically allocated memory is no longer needed, delete them to return memory
  - Example

```
delete a, q;
```

new and delete Operators (Cont.)

```
int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Hope you got the point of this example!\n";
    return 0;
}
```



new and Class Types

- **Using operator new with class types calls a constructor as well as allocating memory**
  - If MyType is a class type, then

```
MyType *myPtr;                  // creates a pointer to a variable of type MyType
myPtr = new MyType;                 // calls the default constructor

myPtr  = new MyType (32.0, 17);     // calls  Mytype(double, int);
```

Defining Pointer Types

- **To avoid mistakes using pointers, define a pointer type name**
  - Example:

```
typedef int* IntPtr;      //Defines a new type, IntPtr, for pointer
                          //variables containing pointers to int variables
IntPtr p;                 // is equivalent to int *p;
```

- **Prevent this error in Multiple Declarations**
  - Example

```
int *P1, P2;       // Only P1 is a pointer variable
IntPtr P1, P2;     // P1 and P2 are pointer variables
```

Dynamic Arrays

Dynamic Arrays

- **A dynamic array is an array whose size is determined when the program is running, not when you write the program**

- **Normal arrays require that the programmer determine the size of the array when the program is written**
  - What if the programmer estimates too large?
    - Memory is wasted
  - What if the programmer estimates too small?
    - The program may not work in some situations

- **Dynamic arrays can be created with just the right size while the program is running**

Dynamic Arrays (Cont.)

- **Dynamic arrays are created using the new operator**
  - Example: To create an array of 10 elements of type double:

```
double* d = new double[10]; //Pointer variable d is a pointer to d[0]
```

- **When finished with the array, it should be deleted to return memory**
  - Example:

```
delete [ ] d;      //brackets tell C++ a dynamic array is being deleted
```

Dynamic Arrays (Cont.)

```
1   //Sorts a list of numbers entered at the keyboard.
2   #include <iostream>
3   #include <cstdlib>
4   #include <cstddef>
5
6   typedef int* IntArrayPtr;
7
8   void fill_array(int a[], int size);
9   //Precondition: size is the size of the array a.
10  //Postcondition: a[0] through a[size- 1] have been
11  //filled with values read from the keyboard.
12
13  void sort(int a[], int size);
14  //Precondition: size is the size of the array a.
15  //The array elements a[0] through a[size–1] have values.
16  //Postcondition: The values of a[0] through a[size–1] have been rearranged
17  //so that a[0] <= a[1] <= … <= a[size–1].
18
19  int main( )
20  {
21      using namespace std;
22      cout << "This program sorts numbers from lowest to highest.\n";
23
24      int array_size;
25      cout << "How many numbers will be sorted? ";
26      cin >> array_size;
27
28      IntArrayPtr a;
29      a = new int[array_size];
30
31      fill_array(a, array_size);
32      sort(a, array_size);
33
34      cout << "In sorted order the numbers are:\n";
35      for (int index = 0; index < array_size; index++)
36          cout << a[index] << " ";
37      cout << endl;
38
39      delete [] a;
40
41      return 0;
42  }
43
44  //Uses the library iostream:
45  void fill_array(int a[], int size)
46  {
```

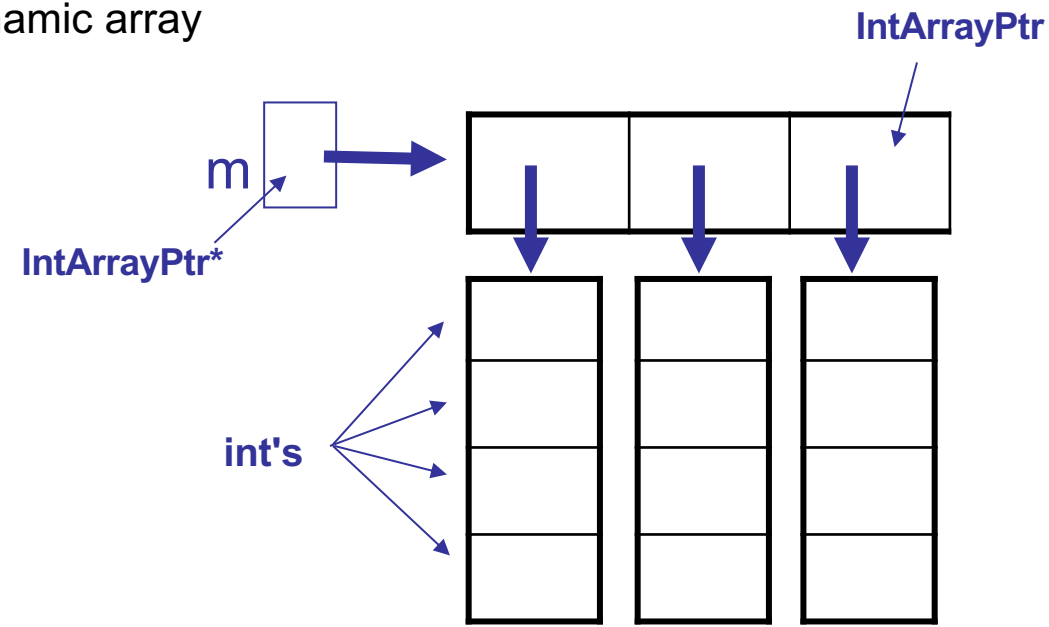
Ordinary array parameters

The dynamic array a is used like an ordinary array.

Multidimensional Dynamic Arrays

- **To create a 3x4 multidimensional dynamic array**
  - View multidimensional arrays as arrays of arrays
  - First create a one-dimensional dynamic array

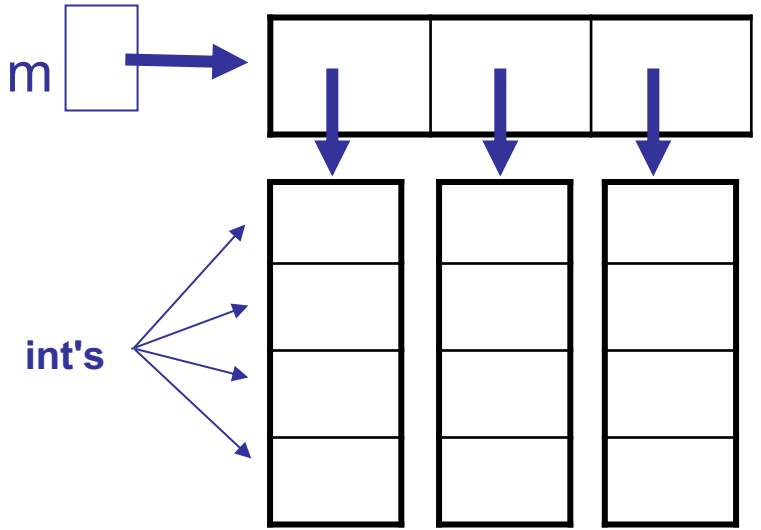
```
typedef int* IntArrayPtr;
IntArrayPtr *m = new IntArrayPtr[3];

for (int i = 0; i<3; i++)
   m[i] = new int[4];
```



Deleting Multidimensional Arrays

- **To delete a multidimensional dynamic array**
  - Each call to new that created an array must have a corresponding call to delete[ ]
  - Example:

```
for ( i = 0; i < 3; i++)
    delete [ ] m[i];   //delete the arrays of 4 int's
delete [ ] m;          // delete the array of IntArrayPtr's
```



Classes and Dynamic Arrays

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

Example: StringVar class

- **Class for string variables**
- **use dynamic arrays whose size is determined when the program is running**
- **Three Constructors**
  - The default constructor : creates an object with a maximum string length of 100
  - The second constructor : takes an argument of type int which determines the maximum string length of the object
  - The third 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

Example: StringVar class (Cont.)

- **Interface**
  - Member functions
    - int length( );
    - void input_line(istream& ins);
    - friend ostream& operator << (ostream& outs, const StringVar& theString);
  - Copy Constructor (discussed later)
  - Destructor (discussed later)

Example: StringVar class (Cont.)

```
class StringVar
{
public:
    StringVar(int size);
    //Initializes the object so it can accept string values up to size
    //in length. Sets the value of the object equal to the empty string.

    StringVar( );
    //Initializes the object so it can accept string values of length 100
    //or less. Sets the value of the object equal to the empty string.

    StringVar(const char a[]);
    //Precondition: The array a contains characters terminated with '\0'.
    //Initializes the object so its value 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 StringVar& stringObject);
    //Copy constructor.

    ~StringVar( );
    //Returns all the dynamic memory used by the object to the freestore.

    int length( ) const;
    //Returns the length of the current string value.

    void inputLine(istream& ins);
    //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.

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

<The definitions of the member functions and overloaded operators go here>

```
//Program to demonstrate use of the class StringVar.

void conversation(int maxNameSize);
//Carries on a conversation with the user.

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

Memory is returned to the freestore when the function call ends.

```
//This is only a demonstration function:
void conversation(int maxNameSize)
{
    using namespace std;

    StringVar yourName(maxNameSize), ourName("Borg");

    cout << "What is your name?\n";
    yourName.inputLine(cin);
    cout << "We are " << ourName << endl;
    cout << "We will meet again " << yourName << endl;
}
```

Determines the size of the dynamic array

Example: StringVar class (Cont.)

```
//This is the implementation of the class StringVar.
//The definition for the class StringVar is in  Display 11.11.
#include <cstdlib>
#include <cstddef>
#include <cstring>

//Uses cstddef and cstdlib:
StringVar::StringVar(int size) : maxLength(size)
{
    value = new char[maxLength + 1];//+1 is for '\0'.
    value[0] = '\0';
}

//Uses cstddef and cstdlib:
StringVar::StringVar( ) : maxLength(100)
{
    value = new char[maxLength + 1];//+1 is for '\0'.
    value[0] = '\0';
}

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

Copy constructor

```
//Uses cstring, cstddef, and cstdlib:
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;
}
```

Destructor

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

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

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

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 by the tilde symbol ~
    - Example:

      ~StringVar( );

Destructors (Cont.)

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

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

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
- **Example**

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

Calling a Copy Constructor

- **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 object of the class is passed (to a function) by value as an argument.

The Need For a Copy Constructor

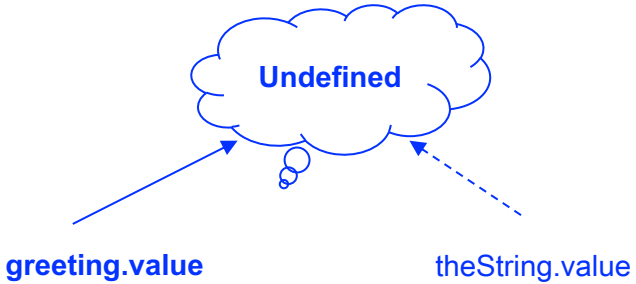
- **This code (assuming no copy constructor) illustrates the need for a copy constructor**

```
void showString(StringVar theString)
 { …}

StringVar greeting("Hello");
showString(greeting);
cout << greeting << endl;
```

  - When function showString is called, greeting is copied into theString
    - theString.value is set equal to greeting.value

The Need For a Copy Constructor (cont.)

- **When showString ends, the destructor for theString executes and deallocates the dynamic array pointed to by theString.value**



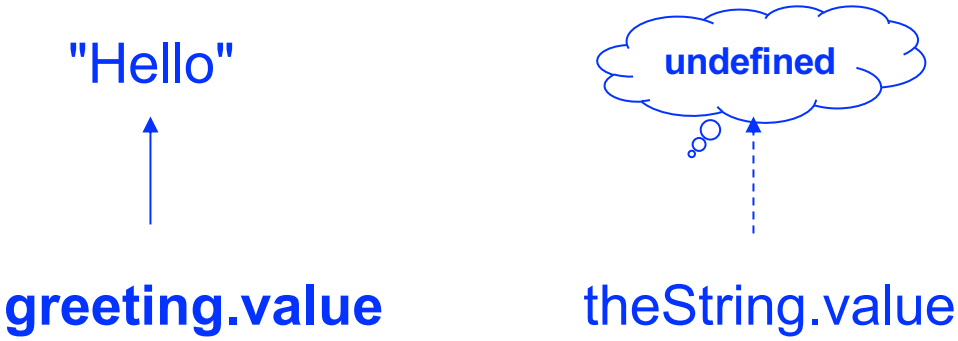
- **greeting.value now points to memory that has been deallocated**
- **Two problems exist for the object greeting**
  - Attempting to output greeting.value is likely to produce an error
  - When greeting goes out of scope, its destructor will be called
    - Calling a destructor for the same location 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 theString.value point to different locations in memory

Copy Constructor Demonstration (cont.)

- **When theString goes out of scope, the destructor is called, and the memory pointed to by theString.value is deallocated**
  - greeting.value still exists and can be accessed or deleted without problems



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:

```
void operator=(const StringVar& rightSide);
```

    - rightSide is the argument from the right side of the = operator

Overloading = (Cont.)

- **Example**

```
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';
}
```

A problem

- Usually, we want a copy of the right-hand argument regardless of its size

Overloading = (Cont.)

- **Example: Another implementation**

```
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 problem

What happens if we happen to have the same object on each side of the assignment operator?

myString = myString;

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!

Overloading = (Cont.)

- **Example: Better implementation**

```
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];
       }

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

NEXT ?

Namespace
Linked-list, Stack, Queue