Chapter 12
Pointers and Dynamic Arrays

Overview

• Pointers (12.1)
• Dynamic Arrays (12.2)
• Classes and Dynamic Arrays (12.3)

12.1 Pointers

• A pointer is the memory address of a variable
• Memory addresses can be used as names for variables
  • If a variable is stored in three memory locations, the address of the first can be used as a name for the variable.
  • When a variable is used as a call-by-reference argument, its address is passed

Pointers Tell Where To Find A Variable

• An address used to tell where a variable is stored in memory is a pointer
  • Pointers “point” to a variable by telling where the variable is located

Declaring Pointers

• Pointer variables must be declared to have a pointer type
  • Example: To declare a pointer variable p that can “point” to a variable of type double:
    double *p;
  • 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
The address of Operator

• The & operator can be used to determine the address of a variable which can be assigned to a pointer variable
  • Example: \[ p1 = \&v1; \]
    \[ p1 \text{ is now a pointer to } v1 \]
    \[ v1 \text{ can be called } v1 \text{ or "the variable pointed to by } p1" \]

Pointer Assignment

• The assignment operator = is used to assign the value of one pointer to another
  • Example: If \( p1 \) still points to \( v1 \) (previous slide) then \[ p2 = p1; \]
    causes \( *p2, *p1, \text{ and } v1 \text{ all to name the same variable} \)

The Dereferencing Operator

• C++ uses the * operator in yet another way with pointers
  • The phrase "The variable pointed to by \( p \)" is translated into C++ as "\( *p \)"
  • Here the * is the dereferencing operator
    • \( p \) is said to be dereferenced

Caution! Pointer Assignments

• Some care is required making assignments to pointer variables
  • \( p1 = p3; \text{ // changes the location that } p1 \text{ "points" to} \)
  • \( *p1 = *p3; \text{ // changes the value at the location that } \)
    \( p1 \text{ "points" to} \)

A Pointer Example

• \( v1 = 0; \]
  \[ p1 = \&v1; \]
  \[ *p1 = 42; \]
  \[ \text{cout} << v1 << \text{endl}; \]
  \[ \text{cout} << *p1 << \text{endl}; \]
  \[ \text{output:} \]
  \[ 42 \]
  \[ 42 \]

The new Operator

• Using pointers, variables can be manipulated even if there is no identifier for them
  • To create a pointer to a new "nameless" variable of type int:
    \[ p1 = \text{new int;} \]
  • The new variable is referred to as \( *p1 \)
  • \( *p1 \) can be used anywhere an integer variable can
    \[ \text{cin} >> *p1; \]
    \[ *p1 = *p1 + 7; \]
Dynamic Variables

• Variables created using the new operator are called dynamic variables
  • Dynamic variables are created and destroyed while the program is running
  • Additional examples of pointers and dynamic variables are shown in Display 12.2

An illustration of the code in Display 12.2 is seen in Display 12.3

The delete Operator

• When dynamic variables are no longer needed, delete them to return memory to the freestore
  • Example: delete p;
    The value of p is now undefined and the memory used by the variable that p pointed to is back in the freestore

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);

Dangling Pointers

• Using delete on a pointer variable destroys the dynamic variable pointed to
  • If another pointer variable was pointing to the dynamic variable, that variable is also undefined
  • Undefined pointer variables are called dangling pointers
    • Dereferencing a dangling pointer (*p) is usually disastrous

Basic Memory Management

• An area of memory called the freestore is reserved for dynamic variables
  • New dynamic variables use memory in the freestore
  • If all of the freestore is used, calls to new will fail
  • Unneeded memory can be recycled
    • When variables are no longer needed, they can be deleted and the memory they used is returned to the freestore

Automatic Variables

• Variables declared in a function are created by C++ and destroyed when the function ends
  • These are called automatic variables because their creation and destruction is controlled automatically
  • The programmer manually controls creation and destruction of pointer variables with operators new and delete
Global Variables
• Variables declared outside any function definition are global variables
• Global variables are available to all parts of a program
• Global variables are not generally used

Multiple Declarations Again
• Using our new pointer type defined as
typedef int* IntPtr;
• Prevent this error in pointer declaration:
  int *P1, P2; // Only P1 is a pointer variable
  with
  IntPtr P1, P2; // P1 and P2 are pointer variables

Type Definitions
• A name can be assigned to a type definition, then used to declare variables
• The keyword typedef is used to define new type names
  • Syntax: typedef Known_Type_Definition  New_Type_Name;
    known_Type_Definition can be any type

Pointer Reference Parameters
• A second advantage in using typedef to define a pointer type is seen in parameter lists
  • Example: void sample_function(IntPtr& pointer_var);
    is less confusing than
    void sample_function( int*& pointer_var);

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;

Section 12.1 Conclusion
• Can you
  • Declare a pointer variable?
  • Assign a value to a pointer variable?
  • Use the new operator to create a new variable in the freestore?
  • Write a definition for a type called NumberPtr to be a type for pointers to dynamic variables of type int?
  • Use the NumberPtr type to declare a pointer variable called my_point?
12.2 Dynamic Arrays

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

Creating Dynamic Arrays

- 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.

Pointer Variables and Array Variables

- Array variables are actually pointer variables that point to the first indexed variable.
  - Example: int a[10]; typedef int* IntPtr; IntPtr p;
    - Variables a and p are the same kind of variable.
    - Since a is a pointer variable that points to a[0], p = a; causes p to point to the same location as a.

Creating Dynamic Arrays

- Dynamic arrays are created using the new operator.
  - Example: To create an array of 10 elements of type double:
    typedef double* DoublePtr;
    DoublePtr d;
    d = new double[10];
    d can now be used as if it were an ordinary array!

Dynamic Arrays (cont.)

- Pointer variable d is a pointer to d[0].
- When finished with the array, it should be deleted to return memory to the freestore.
  - Example: delete [ ] d;
    - The brackets tell C++ a dynamic array is being deleted so it must check the size to know how many indexed variables to remove.
    - Forgetting the brackets, is not legal, but would tell the computer to remove only one variable.
Pointer Arithmetic (Optional)

- Arithmetic can be performed on the addresses contained in pointers
  - Using the dynamic array of doubles, \( d \), declared previously, recall that \( d \) points to \( d[0] \)
  - The expression \( d+1 \) evaluates to the address of \( d[1] \) and \( d+2 \) evaluates to the address of \( d[2] \)
  - Notice that adding one adds enough bytes for one variable of the type stored in the array

A Multidimensional Dynamic Array

- The dynamic array created on the previous slide could be visualized like this:

Pointer Arithmetic Operations

- You can add and subtract with pointers
  - The ++ and - - operators can be used
  - Two pointers of the same type can be subtracted to obtain the number of indexed variables between
    - The pointers should be in the same array!
  - This code shows one way to use pointer arithmetic:
    ```
    for (int i = 0; i < array_size; i++)
        cout << *(d + i) << "  " ;
    // same as cout << d[i] << "  " ;
    ```

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: To delete the dynamic array created on a previous slide:
    ```
    for ( i = 0; i < 3; i++)
        delete [ ] m[i]; //delete the arrays of 4 int's
    delete [ ] m; // delete the array of IntArrayPtr's
    ```

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 each pointer in \( m \), create a dynamic array of int's
    ```
    for (int i = 0; i<3; i++)
        m[i] = new int[4];
    ```

Section 12.2 Conclusion

- Can you
  - Write a definition for pointer variables that will be used to point to dynamic arrays? The array elements are of type char. Call the type CharArray.
  - Write code to fill array "entry" with 10 numbers typed at the keyboard?
    ```
    int * entry;
    entry = new int[10];
    ```
Classes and Dynamic Arrays 12.3

- 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.

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

Program 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

A StringVar Sample Program

- Using the StringVar interface of Display 12.7, 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 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 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

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!

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();

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

- 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; 
    }
    ```

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.)

- 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!

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
Copy Constructor Demonstration (cont.)

• When the_string goes out of scope, the destructor is called, returning the_string.value to the freestore

```
"Hello"
```

```greeting.value```
```the_string.value```

• greeting.value still exists and can be accessed or deleted without problems

The Assignment Operator

• Given these declarations:
  ```StringVar string1(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

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

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

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

Definition of =

• The definition of = for StringVar could be:
  ```
  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 =

- void StringVar::operator = (const StringVar& right_side)
  { delete [ ] value; int new_length = strlen(right_side.value); int 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 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
        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 12.3 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?
Chapter 12 -- End

Display 12.1

Uses of the Assignment Operator

Before:  
p1 = 84  
p2 = 99

After:  
p1 = 84  
p2 = 99

Display 12.2

Sample Dialogue:

Teacher: "What is the temperature in New York?"
Student: "84 degrees."
Teacher: "What is the temperature in Chicago?"
Student: "99 degrees."

Display 12.3

Display 12.4

Display 12.5

1/2