Chapter 9

Pointers and Dynamic Arrays
Overview

9.1   Pointers

9.2   Dynamic Arrays
9.1 Pointers
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:

```c
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: \( p1 = \&v1; \)

  \( p1 \) is now a pointer to \( v1 \)
  \( v1 \) can be called \( v1 \) or "the variable pointed to by \( p1 \)"
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
A Pointer Example

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

output:

  42
  42

v1 and *p1 now refer to the same variable
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, and v1 all to name the same variable
Caution! Pointer Assignments

- Some care is required making assignments to pointer variables
  - p1= p3; // changes the location that p1 "points" to
  - *p1 = *p3; // changes the value at the location that p1 "points" to
Uses of the Assignment Operator

Before:  
\[ \text{p1} \rightarrow 84 \]
\[ \text{p2} \rightarrow 99 \]

After:  
\[ \text{p1} \rightarrow 84 \]
\[ \text{p2} \rightarrow 99 \]

\[ \text{p1} = \text{p2}; \]

\[ \ast\text{p1} = \ast\text{p2}; \]
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 9.2

An illustration of the code in Display 9.2 is seen in Display 9.3
Basic Pointer Manipulations

//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using namespace std;

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;
}

Sample Dialogue

*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
DISPLAY 9.3  Explanation of Display 9.2

(a)
int *p1, *p2;
p1
\[\]
p2
\[\]

(b)
p1 = new int;
p1
\[\]
p2
\[\]

(c)
*p1 = 42;
p1
\[42\]
p2
\[\]

(d)p2 = p1;
p1
\[\]
p2
\[\]

(e)
*p2 = 53;
p1
\[\]
p2
\[53\]

(f)p1 = new int;
p1
\[\]
p2
\[\]

(g)
*p1 = 88;
p1
\[88\]
p2
\[53\]
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);
  ```
Basic Memory Management

- An area of memory called the **freestore** or the **heap** 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
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
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.
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
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
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;`
Multiple Declarations Again

- Using our new pointer type defined as:
  ```c
  typedef int* IntPtr;
  ```

- Prevent this error in pointer declaration:
  ```c
  int *P1, P2;  // Only P1 is a pointer variable
  ```

  with:
  ```c
  IntPtr P1, P2;  // P1 and P2 are pointer variables
  ```
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);
Section 9.1 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?
9.2

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.
Array variables are actually pointer variables that point to the first indexed variable

Example:

```c
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],

```c
p = a;
```

causes p to point to the same location as a
Pointer Variables
As Array Variables

- Continuing the previous example:
  Pointer variable p can be used as if it were an array variable

- Example: p[0], p[1], …p[9]
  are all legal ways to use p

- Variable a can be used as a pointer variable except the pointer value in a cannot be changed
  - This is not legal: IntPtr p2;
    … // p2 is assigned a value
    a = p2 // attempt to change a
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
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!

This could be an integer variable!
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
// Sorts a list of numbers entered at the keyboard.
#include <iostream>
#include <cstdlib>
#include <cstddef>

typedef int* IntArrayPtr;

void fill_array(int a[], int size);
// Precondition: size is the size of the array a.
// Postcondition: a[0] through a[size-1] have been
// filled with values read from the keyboard.

void sort(int a[], int size);
// Precondition: size is the size of the array a.
// Postcondition: The values of a[0] through a[size-1] have been rearranged
// so that a[0] <= a[1] <= ... <= a[size-1].

int main()
{
    using namespace std;
    cout << "This program sorts numbers from lowest to highest.\n";

    int array_size;
    cout << "How many numbers will be sorted? \n";
    cin >> array_size;

    IntArrayPtr a;
    a = new int[array_size];

    fill_array(a, array_size);
    sort(a, array_size);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < array_size; index++)
        cout << a[index] << " ";
    cout << endl;

    delete [] a;

    return 0;
}

(continued)
//Uses the library iostream:
void fill_array(int a[], int size) {
    using namespace std;
    cout << "Enter " << size << " integers.\n";
    for (int index = 0; index < size; index++)
        cin >> a[index];
}

void sort(int a[], int size)

<Any implementation of sort may be used. This may or may not require some additional function definitions. The implementation need not even know that sort will be called with a dynamic array. For example, you can use the implementation in Display 7.12 (with suitable adjustments to parameter names).>
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
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:

```c++
for (int i = 0; i < array_size; i++)
  cout << *(d + i) << "  " ;
// same as cout << d[i] << "  " ;
```
Multidimensional Dynamic Arrays

- To create a 3x4 multidimensional dynamic array
  - View multidimensional arrays as arrays of arrays
  - First create a one-dimensional dynamic array
    - Start with a new definition:
      
      ```
      typedef int* IntArrayPtr;
      ```
    - Now create a dynamic array of pointers named m:
      
      ```
      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];
      ```
A Multidimensional Dynamic Array

- The dynamic array created on the previous slide could be visualized like this:
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:
    ```cpp
    for ( i = 0; i < 3; i++)
        delete [ ] m[i]; //delete the arrays of 4 int's
    delete [ ] m; // delete the array of IntArrayPtr's
    ```

Display 9.7 (1)  Display 9.7 (2)
Section 9.2 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?

```c
int * entry;
entry = new int[10];
```
Arrays and Pointer Variables

//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main()
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}

Output

0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10

Note that changes to the array p are also changes to the array a.
(a)  
```c
IntPtr p;
int a[10];
```

```c
int a[10];
```

<table>
<thead>
<tr>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)  
```c
for (index = 0; index < 10; index++)
    a[index] = index;
```

```c
a = 0 1 2 3 4 5 6 7 8 9
```

<table>
<thead>
<tr>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c)  
```c
p = a;
```

```c
int a[10];
```

<table>
<thead>
<tr>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d)  
```c
for (index = 0; index < 10; index++)
    p[index] = p[index] + 1;
```

```c
a = 0 1 2 3 4 5 6 7 8 9
```

<table>
<thead>
<tr>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Output
```

```
0 1 2 3 4 5 6 7 8 9
```

Iterating through `p` is the same as iterating through `a`

```
Output
```

```
0 1 2 3 4 5 6 7 8 9
```

Iterating through `a` is the same as iterating through `p`
A Two-Dimensional Dynamic Array (part 1 of 2)

```cpp
#include <iostream>
using namespace std;

typedef int* IntArrayPtr;

int main() {
  int d1, d2;
  cout << "Enter the row and column dimensions of the array:\n";
  cin >> d1 >> d2;

  IntArrayPtr *m = new IntArrayPtr[d1];
  int i, j;
  for (i = 0; i < d1; i++)
    m[i] = new int[d2];
  //m is now a d1 by d2 array.

  cout << "Enter " << d1 << " rows of 
    " << d2 << " integers each:\n";
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++)
      cin >> m[i][j];

  cout << "Echoing the two-dimensional array:\n";
  for (i = 0; i < d1; i++)
  {
    for (j = 0; j < d2; j++)
      cout << m[i][j] << " ";
    cout << endl;
  }

  delete[] m;
}
```
A Two-Dimensional Dynamic Array (part 2 of 2)

```cpp
for (i = 0; i < d1; i++)
    delete[] m[i];
delete[] m;

return 0;
}
```

Sample Dialogue

Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2

Note that there must be one call to delete [] for each call to new that created an array. (These calls to delete [] are not really needed since the program is ending, but in another context it could be important to include them.)