Pointers and Dynamic Arrays

Modified from Chapter 9

Overview

- Pointers
- Dynamic Variables and Arrays

Pointers

Modified from Section 9.1

Pointers

- Pointers are variables that store addresses of variables or memory locations.
- Memory addresses can be used to identify variables
  - If a variable is stored in three memory locations (3 bytes), the address of the first can be used as an identifier for the variable.
  - When a variable is used as a call-by-reference argument, its address is passed

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
- 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 & (address-of) operator can be used to determine the address of a variable.
- The result of an & operation can be assigned to a pointer variable
  - Example: p1 = &v1;
  - After this assignment, v1 is 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

Pointer Assignment

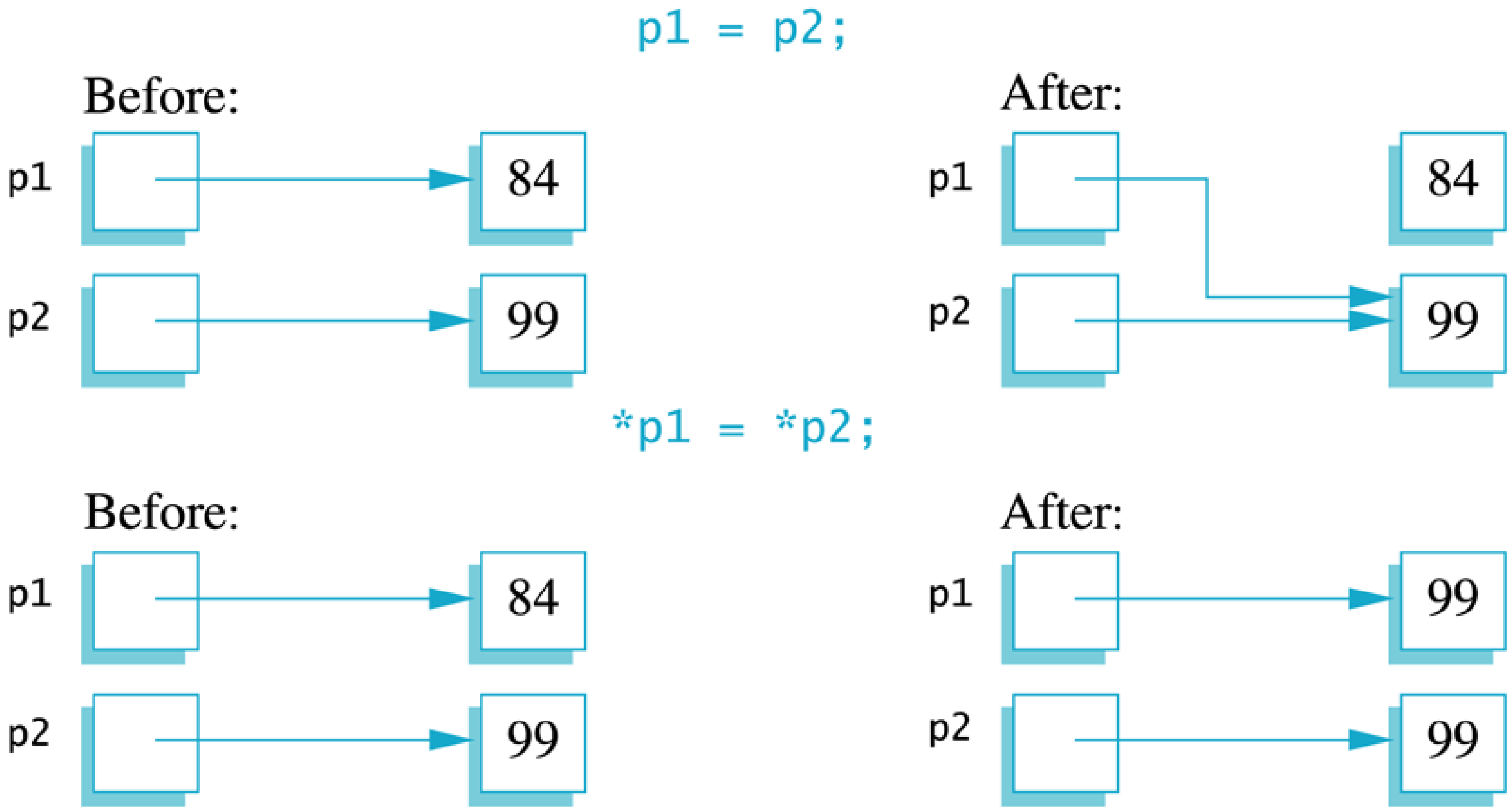
- The assignment operator = is used to assign the value of one pointer to another
  - Example: p2 = p1;
  - If p1 still points to v1, the above statement causes *p2, *p1, and v1 all to name the same variable
- Comparison with value assignment
  - p1= p2; // changes the location that p1 points to
  - *p1 = *p2; // changes the value at the location that
    // p1 "points" to

Display 9.1

Display 9.1

Back Next

Uses of the Assignment Operator



```
p1 = p2;
```

```
*p1 = *p2;
```

Pointer Variables vs. Array Variables

- Array variables also hold memory addresses
  - An array variable (an array’s name) holds the address the first indexed variable
- An array variable can be used in a similar way to a pointer variable, except its value cannot be changed
  - Example: int a[10];
    int *p;
    p = a; // legal
    ~~a = p; // NOT legal~~

Pointer Arithmetic

- Arithmetic can be performed on the addresses contained in pointers
  - Using the pointer pointing to an array of integers, p, declared previously, recall that p points to a[0]
  - The expression p+1 evaluates to the address of a[1] and p+2 evaluates to the address of a[2]
  - Notice that adding one adds enough bytes for one variable of the type stored in the array
  - p can also be used as the array's name now
    - a[i] and p[i] can be used to access the same variable

Pointer Arithmetic

- 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!
  - Pointer arithmetic example:

```
for (int i = 0; i < 10; i++) {
      cout << *(p + i) << "  " ;
      // *(p+i) acts the same as a[i] or p[i]
}
```

Dynamic Variables and Arrays

Modified from Sections 9.1 and 9.2

The new Operator

- New expression syntax: new Type_Name;
  - Attempts to create a variable with dynamic memory, and returns a pointer to the newly-created variable
- Using pointers, variables can be manipulated even if there is no identifier for them
  - To create a pointer to a new "nameless“ int :

    p1 = new int;
  - The new variable is referred to as *p1
  - *p1 can be used anyplace an integer variable can
    - Examples: cin >> *p1; *p1 = *p1 + 7; cout << *p1;

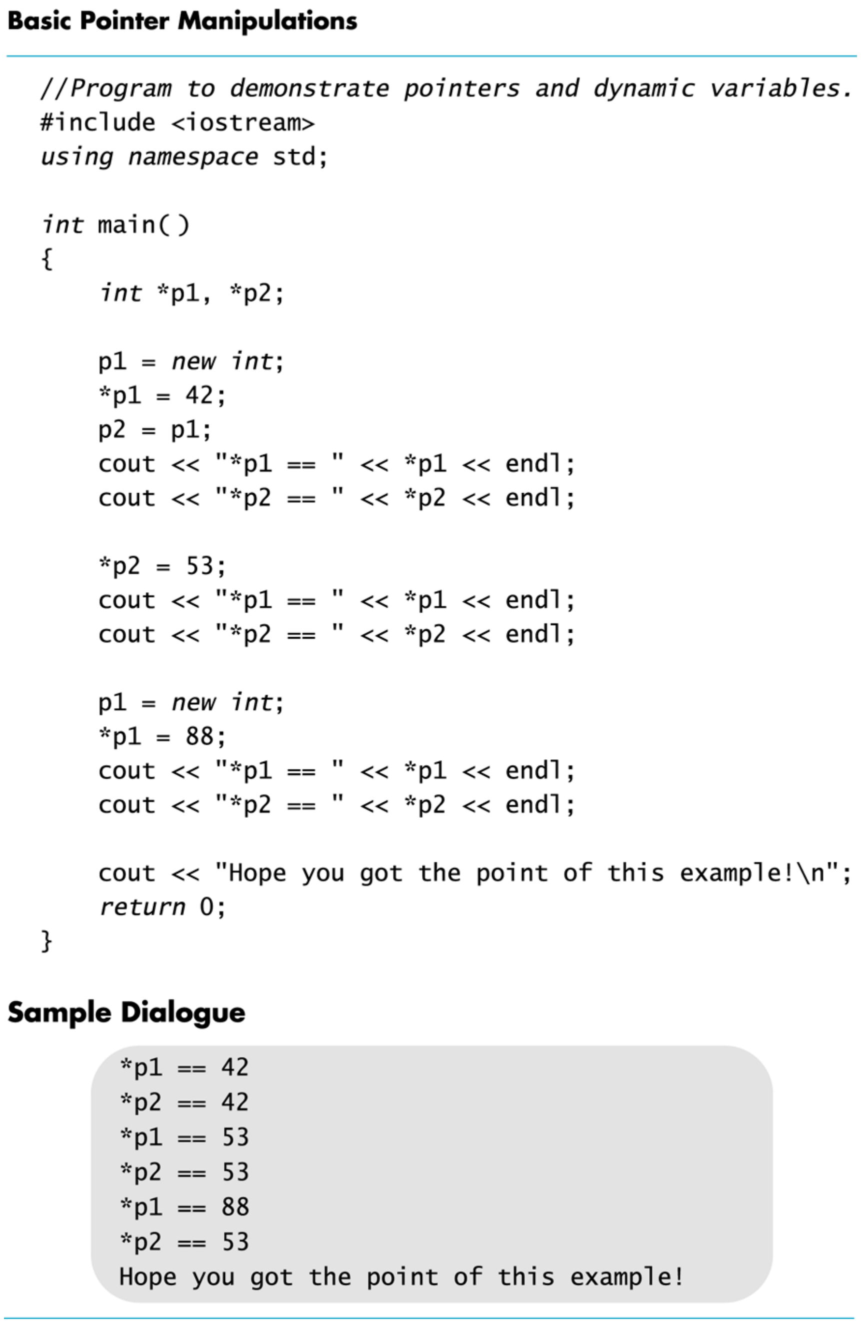
Dynamic Variables

- 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**

Displays 9.2 & 9.3

Back Next



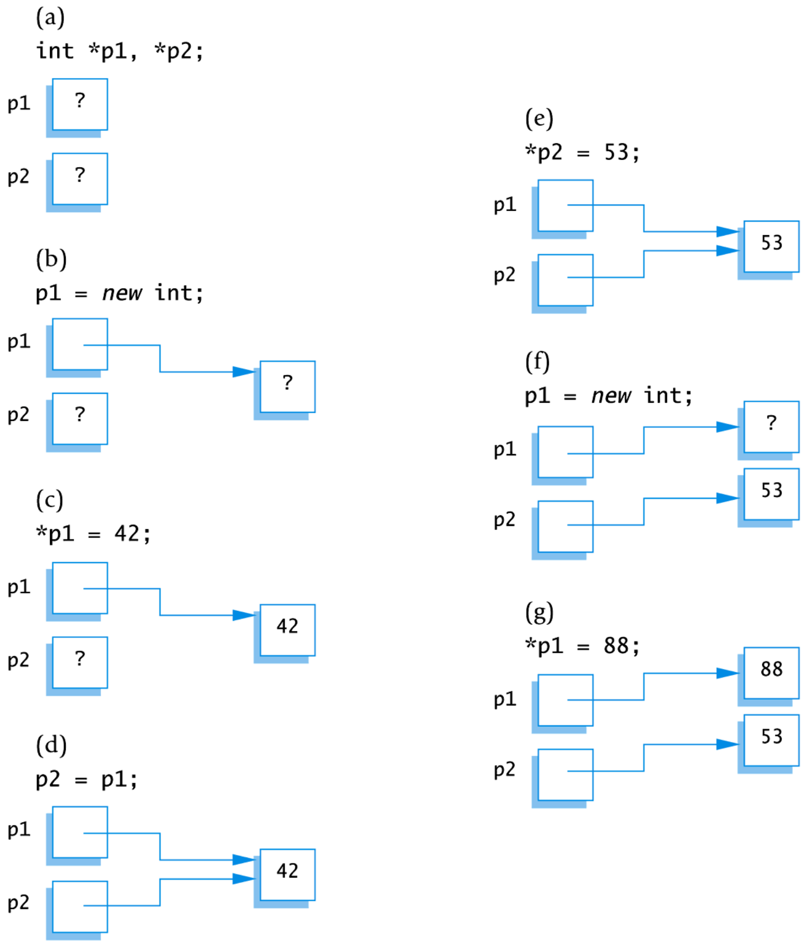
Basic Pointer Manipulations

```
//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;`

(b) `p1 = new int;`

(c) `*p1 = 42;`

(d) `p2 = p1;`

(e) `*p2 = 53;`

(f) `p1 = new int;`

(g) `*p1 = 88;`

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

- A dangling pointer is a pointer to storage that is no longer allocated.
  - p is a dangling pointer in following examples

```
int *p;
{
    int n = 45;
    p = &n;
}
```

```
int *p, *q;
q = new int;
*q = 2;
p = q;
delete q;
```

- Dereferencing a dangling pointer (*p) is usually disastrous

Dynamic Arrays

- A dynamic array is an array whose size is determined when the program is running, not when you write the program
- Dynamic arrays are created with the new operator
  - Syntax: new Type_Name[Array_Size];
  - The new operator returns the memory address of the first element, which is usually assigned to a pointer
  - Array_Size can be a non-constant expression for dynamic arrays
  - Example: int *p = new int[10];
    - p can be used as the array's name

Type Definitions (optional)

- A type definition creates an alias that can be used in place of a (possibly complex) type name.
- The keyword typedef is used to define new type names
  - Syntax: typedef Known_Type_Name New_Type_Name;
  - Example: typedef int* IntPtr;
    - Defines an alias, IntPtr, for pointers to int variables
    - IntPtr p; becomes equal to int *p;

Deleting Dynamic Arrays

- When finished with the array, it should be deleted to return memory to the freestore
  - Example: delete [ ] p;
    - 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 does not produce syntax errors, but would tell the computer to remove only one variable (the first one in this array)

Multidimensional Dynamic Arrays

- To create a 3x4 dynamic array of integers
  - View n-dimensional arrays as arrays of pointers, where each pointer points to an (n-1)-dimensional array
  - First create a dynamic array of integer pointers

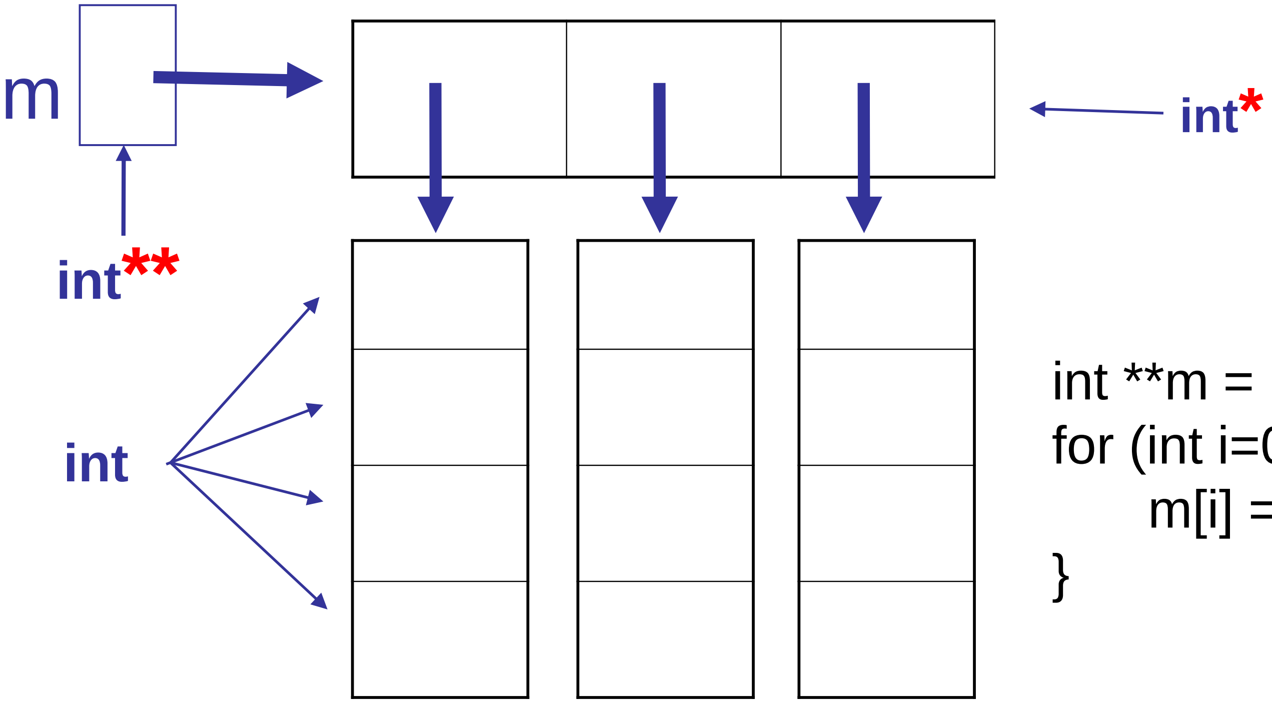
```
int **m = new int*[3];
```

    - "int **m" declares a pointer pointing to integer pointers
    - "int*" in "new int*[3]" means elements in this array are integer pointers
  - Next, for each pointer in m, create a dynamic array

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

A Multidimensial Dynamic Array

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



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

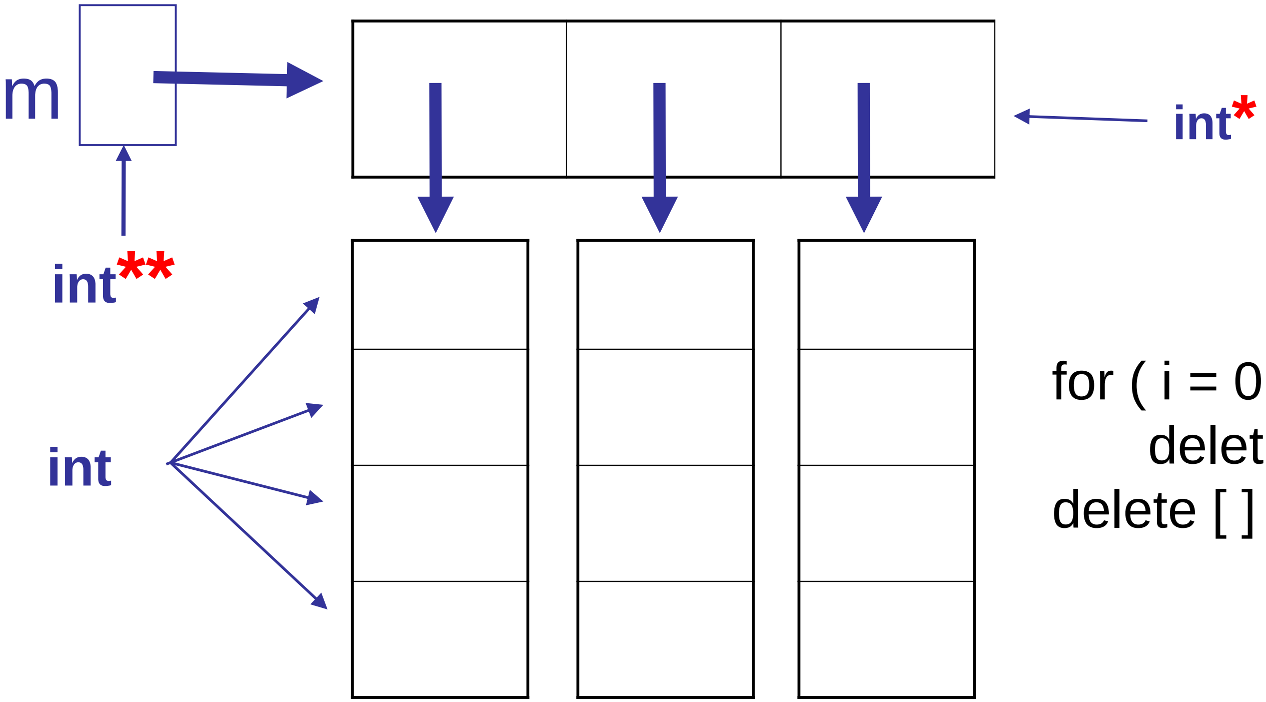
Deleting Multidimensional Arrays

- To delete a multidimensional dynamic array
  - Each call to new that created an array must have a corresponding call to delete[ ]
  - Example: To delete a 3x4 dynamic array of integers

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

A Multidimensial Dynamic Array

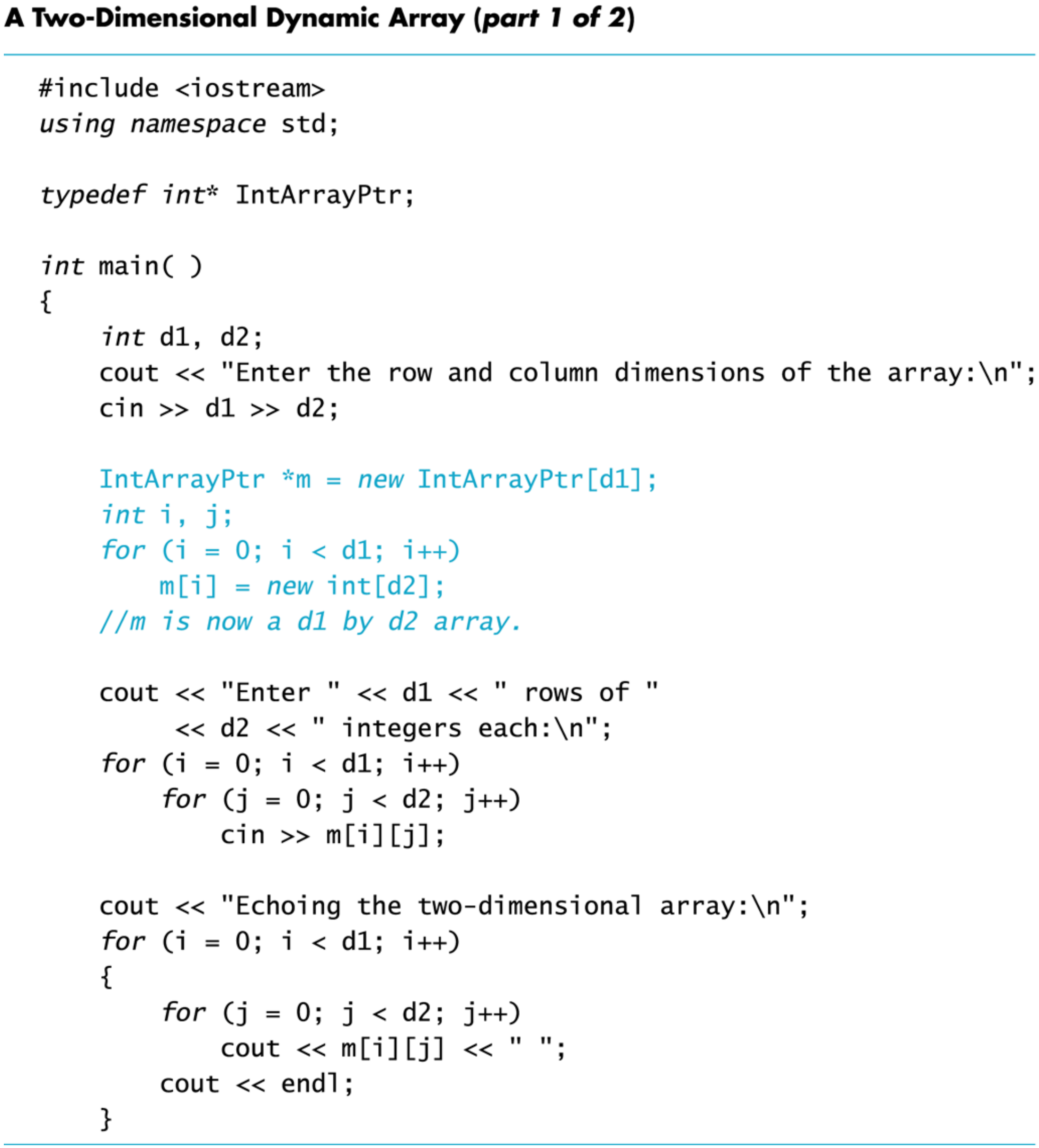
- The dynamic array deleted on the previous slide could be visualized like this:



```
for ( i = 0; i < 3; i++)
      delete [ ] m[i];
delete [ ] m;
```

Display 9.7 (1/2)

Back Next



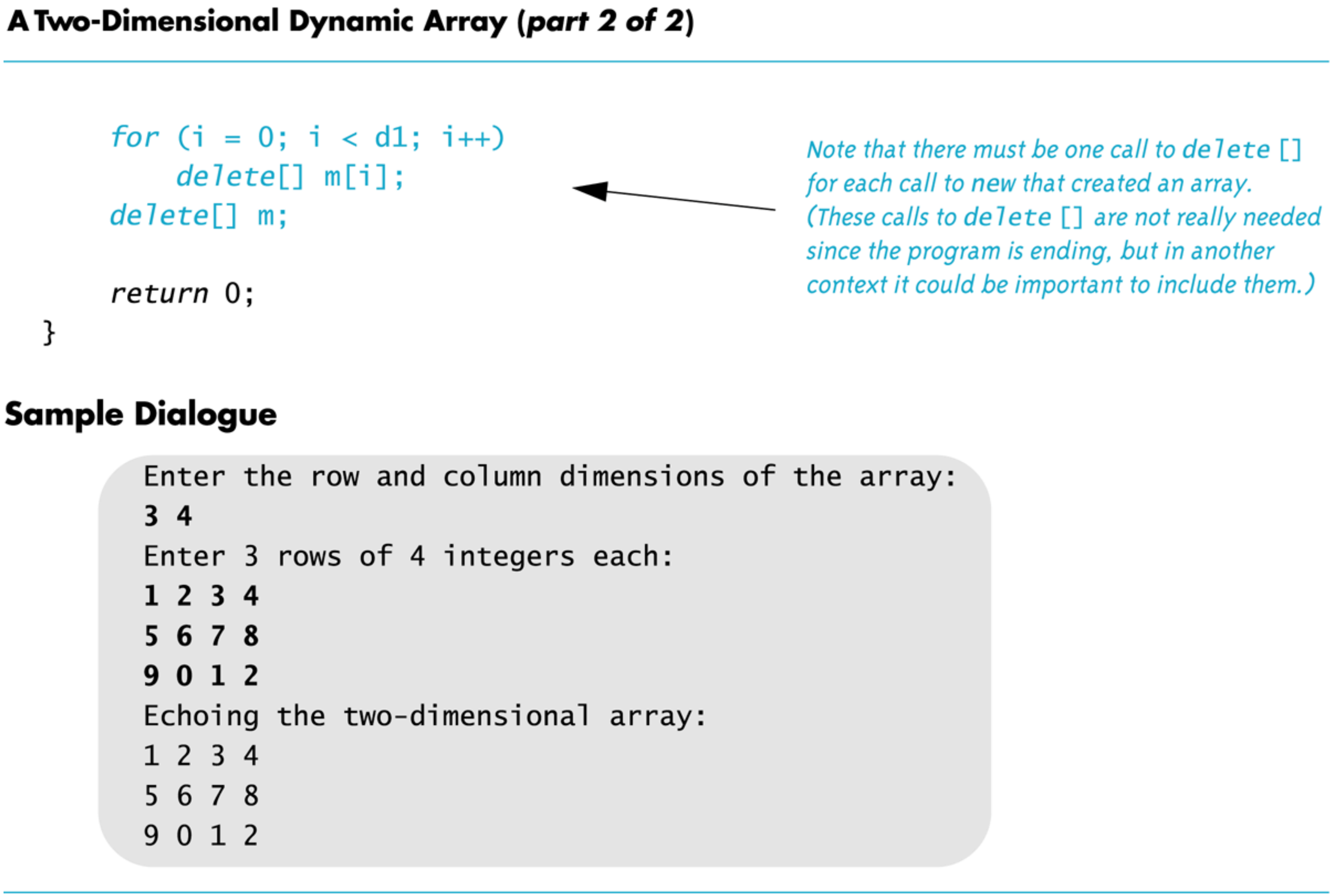
A Two-Dimensional Dynamic Array (*part 1 of 2*)

```
#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;
    }
```

Display 9.7 (2/2)

Back

Next



A Two-Dimensional Dynamic Array (*part 2 of 2*)

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

    return 0;
}
```

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.)*

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
```