C++ Dynamic Arrays

In this Week

- Motivation: Limitations of C++ Static Arrays
- Creation and Processing of Dynamic Arrays
- Passing Dynamic Arrays to functions
- Returning Dynamic Arrays from functions
- Pointer Arithmetic
- Two dimensional Dynamic Arrays (Matrix)

Motivation

- Consider the problem of reading the marks of **n** students in a class in order to compute some statistics of the students' marks
- The value of **n** is to be determined by asking the user after the program starts running
- We would like to create an array in order to read the marks to the array
- **Can we use a C++ static array?**
- **No!** Because in order to use a C++ static array, we need to know the value of **n** before the program starts running... But we don't have it!

Motivation

- In order to address this shortcoming of static arrays, C++ provides **Dynamic Arrays** whose size can be decided after the program starts running, say by asking the user
- C++ Dynamic Arrays are arrays whose memory space is reserved from the heap memory
- Thus pointers are used for C++ Dynamic Arrays
- We first declare a pointer. For example

```
float *A;
```

1D C++ Dynamic Array

- Next, we ask the user for the size of the array

```
int n;
cout << "Enter the size n";
cin >> n;
```

- Finally, we reserve **n** consecutive float memory locations on the heap and point the pointer **A** to the first element of the array

```
A = new float[n];
```

- **Such an array whose memory space is reserved on the heap that is pointed to by a pointer variable is known as a dynamic array!**

1D C++ Dynamic Array

- Now, the pointer variable **A** is just an array variable whose elements can be accessed by indexing
- The indexing starts at **0** and goes all the way to **n-1**
- Therefore the following code fills the array **A** with random floats in the range [0.0, 1.0]

```
for (int i = 0; i < n; i++)
{
    A[i] = static_cast<float>(rand()) /
RAND_MAX;
}
```

1D C++ Dynamic Array

- Similarly, we can print the elements of the array **A** using a loop as follows:

```
for (int i = 0; i < n; i++)
     cout << "Element at index " << i << " = " << A[i] << endl;
```

- We can also modify the elements of the array in a similar manner. The following code doubles each element of the array

```
for (int i = 0; i < n; i++)
    A[i]  = 2 * A[i];
```

1D Dynamic Array Creation Syntax

- In summary, the creation of a one dimensional dynamic array follows the following syntax

  Syntax

  ```
  data_type *varName = new data_type[n];
  ```

- Alternatively,

  ```
  data_type *varName;
  ⋮
  //read value of n
   ⋮
  varName = new data_type[n];
  ```

The delete operator

- Recall that the new operator reserves memory from the heap memory
- Therefore the memory allocated to a dynamic array is obtained from the heap memory
- Since the heap memory doesn't get cleared even after the pointers pointing to it go out of scope, we need to explicitly free the memory space allocated to dynamic arrays

The delete Operator

- In order to free the consecutive chunk of memory reserved for dynamic arrays, we use the delete operator

  Syntax

  ```
  delete[] arrayVariableName;
  ```

- For example, in the previous example, we created a dynamic array of float of size **n**. Once we don't need the memory space anymore, we free it as follows

  ```
  delete[] A;
  ```

Passing Dynamic Arrays to functions

- Dynamic arrays can also be passed to functions
- Since a dynamic array is effectively a pointer, passing a dynamic array to a function is equivalent to passing a pointer to a function
- In passing a dynamic array to a function, any modification made to an element of the array inside the function will be reflected back to the main program
- Suppose we have a one dimensional dynamic array variable **A** of float of size **size**
- Then we can pass this array to a function that prints the elements of the array as

  printArray(A, size);

- Now the function needs to be defined to take two arguments: a pointer and an integer

Passing Dynamic Arrays to functions

- Also, it is obvious the function does not return anything, hence void. Therefore the function declaration should look like

  void printArray(const float *p, const int s)

- Notice that the size parameter is better made constant for it will not be modified; similarly for the dynamic array as well because the print function does not modify any of the elements of the array

Passing Dynamic Arrays to functions

- Similarly, we can also populate the array inside a function and call the function as follows:

  populateArray(A, size);

- The function declaration will be

  void populateArray(float *p, const int s)

- Observe that the function declaration does not have a **constant** for the array pointer parameter because this function will fill the elements of the array with some values, hence it will modify them

Passing Dynamic Arrays to functions

- **Example 3.** Modify the program given in Example 1 so that this time the array is populated by calling a function named **populateArray**, it is printed by calling a function named **printArray**, and finally the minimum and maximum values are computed by calling a function named **computeMinMax**. Design the **computeMinMax** function so that the main program gets both the min and max values from this function

Passing Dynamic Arrays to functions

```
int main()
{
    //Ask user for a positive value array size
    int size;
    do
    {
        cout << "Enter positive number array size ";
        cin >> size;
    }while (size <= 0);
    //Create a dynamic array of the user specified size
    float *A = new float[size];

    //Seed the random number generator
    srand(time(0));

    //Fill the array with random floats in [0.0, 1.0)
    populateArray(A, size);

    //Print the elements of the array
    printArray(A, size);

    //Print the minimum and maximum elements of the array
    float min, max;
    computeMinMax(A, size, min, max);
    cout << "Minimum = " << min << " and maximum = " << max << endl;

    //Delete the dynamically allocated memory
    delete[] A;
    system("Pause");
    return 0;
}
```

Passing Dynamic Arrays to functions

```
void populateArray(float *p, const int s)
{
    for (int i = 0; i < s; i++)
        p[i] = (1.0 * rand()) / RAND_MAX;
    return;
}

void printArray(const float *p, const int s)
{
    for (int i = 0; i < s; i++)
        cout << p[i] << "\t";
    cout << endl;
    return;
}

void computeMinMax(const float *p, const int s, float &min, float &max)
{
    min = p[0];
    max = p[0];
    for (int i = 1; i < s; i++)
    {
        if (p[i] < min)
            min = p[i];
        if (p[i] > max)
            max = p[i];
    }
    return;
}
```

Returning Dynamic Arrays from functions

- **The most important feature of C++ dynamic arrays that makes them different from static arrays is the fact that we can create them inside functions and return them from the functions!!!**
- **This is not possible with static arrays!!!**
- In order to return a dynamic array from a function
  - Declare the function such that the return type is a pointer
  - Create the dynamic array inside the function
  - Return the pointer dynamic array

Returning Dynamic Arrays from functions

- **Example 4.** Modify the program given in Example 3 so that the array size is read in the main program and then the main program calls a function named **createPopulatedArray** that will create the dynamic array, populates the array elements with random floats in the range [0.0, 1.0], and finally returns the dynamic array to the main program

Returning Dynamic Arrays from functions

```
int main()
{
    //Ask user for a positive value array size
    int size;
    do
    {
        cout << "Enter positive number array size ";
        cin >> size;
    }while (size <= 0);

    //Seed the random number generator
    srand(time(0));

    //Create a dynamic array of the user specified size
    //Also pupolate its elements with random floats in the range [0.0, 1.0)
    float *A = createPopulatedArray(size);

    //Print the elements of the array
    printArray(A, size);

    //Print the minimum and maximum elements of the array
    float min, max;
    computeMinMax(A, size, min, max);
    cout << "Minimum = " << min << " and maximum = " << max << endl;

    //Delete the dynamically allocated memory
    delete[] A;
    system("Pause");
    return 0;
}
```

Returning Dynamic Arrays from functions

```
float* createPopulatedArray(const int s)
{
    float *p = new float[s];
    for (int i = 0; i < s; i++)
        p[i] = (1.0 * rand()) / RAND_MAX;
    return p;
}

void printArray(const float *p, const int s)
{
    for (int i = 0; i < s; i++)
        cout << p[i] << "\t";
    cout << endl;
    return;
}

void computeMinMax(const float *p, const int s, float &min, float &max)
{
    min = p[0];
    max = p[0];
    for (int i = 1; i < s; i++)
    {
        if (p[i] < min)
            min = p[i];
        if (p[i] > max)
            max = p[i];
    }
    return;
}
```

Some Remarks: Dynamic Array of Size 1

- When we introduced pointers, we have seen that we can declare a pointer and point it to one memory location on the heap as follows:
  float *p = new float;
- Now, we may also think of the pointer variable **p**, as if it was a dynamic array of size **1** defined as **float *p = new float[1];**
- In fact the following program shows we can initialize and access the heap memory using the pointer variable p together with indexing

```
int main()
{
    int *p;
    p = new int;
    p[0] = 5;
    cout << p[0] << endl;

    system("Pause");
    return 0;
}
```

Some Remarks: Dynamic Array and Pointers

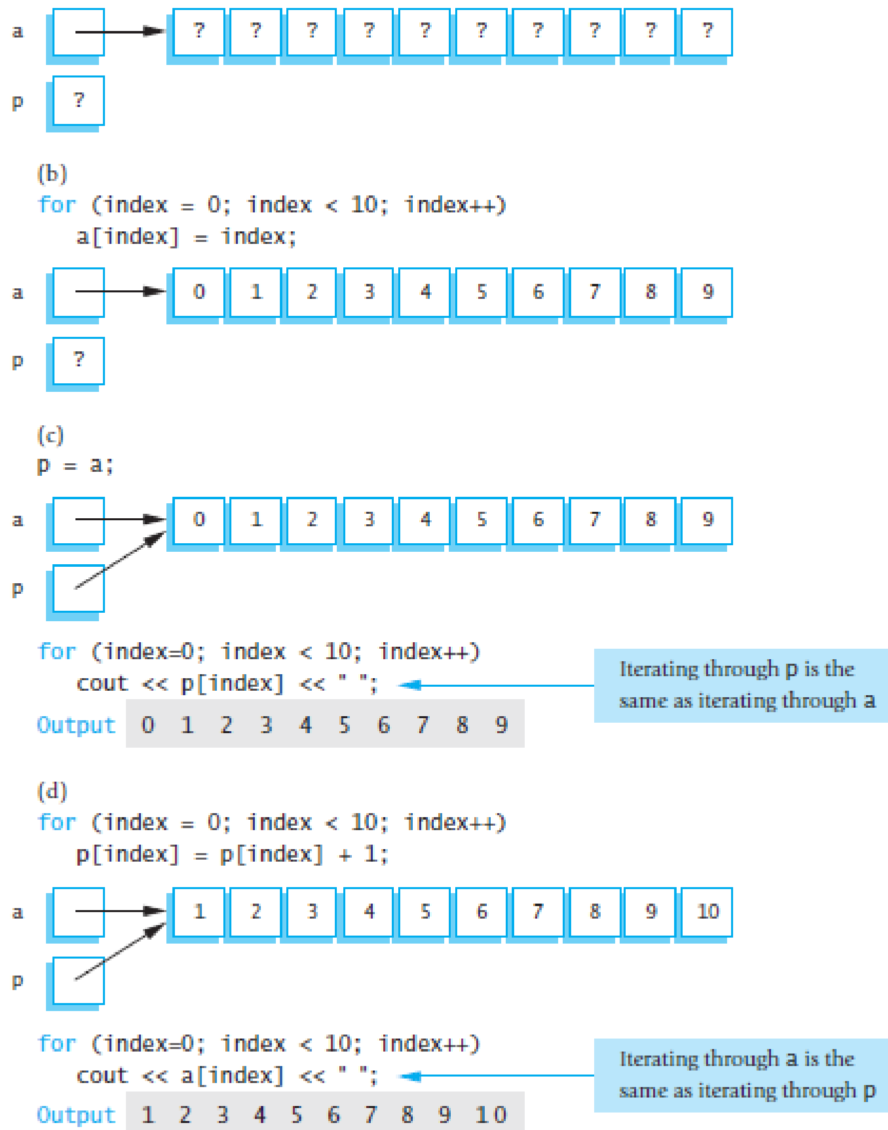
- Consider the dynamic array
  float *A = new float[10];
- Now, **A** is a one dimensional array of size 10
- Next declare a pointer to float
  float *p;
- We can now assign to **p** the value of the variable **A** and then both **p** and **A** will be pointers pointing to the same first element of the 10 consecutive memory locations of floats on the heap
  p = A;
- We can then process the array using the pointer **A** or equivalently using the pointer **p**
  cout << p[2] << endl; is equivalent to cout << A[2] << endl;
- Moreover, deleting **p** deletes the allocated memory on the heap; just like deleting **A** would do the same
- However we should not delete both **A** and **p** together (simultaneously)
  delete [] A; is equivalent to delete [] p;

Some Remarks: Pictorial Representation

int *a = new int[10];

int *p;

Pictorial representation and manipulation of one dimensional array and pointer variables...



Some Remarks: Function Declarations for Dynamic Arrays

- We have presented the function declarations for our dynamic arrays to use pointer data type as

  void printArray(const float *A, const int size)

- Some people rather like to use the style presented for static arrays

  void printArray(const float A[ ], const int size)

- For all practical purposes, these two are identical!
- Moreover we can use either of these function declarations for static arrays as well as for dynamic arrays!

Pointer Arithmetic

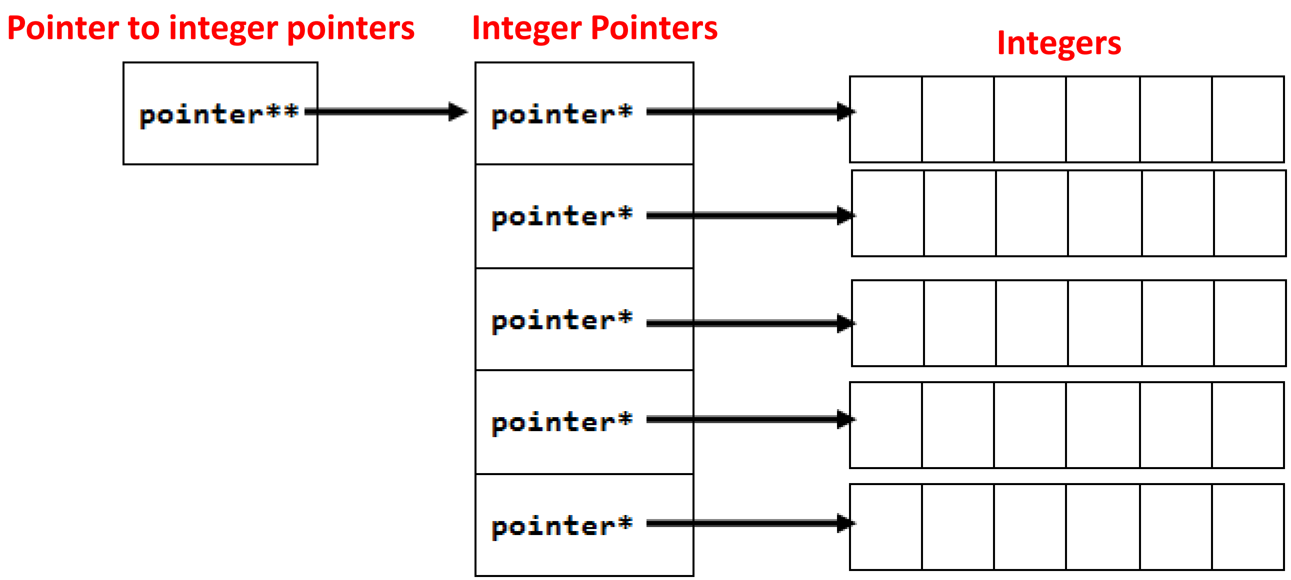
- Consider the dynamic array
  int *A = new int[10];
- Now **A** is nothing but a pointer pointing to the first element of the 10 consecutive int memory locations we just allocated
- As such **A+1** is also a pointer that points to the second element, **A+2** points to the third element, and etc
- Therefore, if need be; we can process the elements of the array with such pointer arithmetic with the help of dereferencing. See example below…

Pointer Arithmetic

```
int main()
{
    //Create three dynamic arrays of same size
    int *A = new int[10];
    int *B = new int[10];
    int *C = new int[10];

    //Fill the firt two arrays using indexing or de-referencing
    for (int i = 0; i < 10; i++)
    {
        *(A+i) = i;      //using de-referencing
        B[i] = 10 - i;   //using indexing
    }

    //Fill the third array using de-referencing
    for (int i = 0; i < 10; i++)
        *(C+i) = A[i] + *(B+i);

    //Print the arrays using indexing or de-referencing
    for (int i = 0; i < 10; i++)
        cout << *(A+i) << "\t" << B[i] << "\t" << *(C+i) << endl;

    //Delete the heap memory
    delete[] A;
    delete[] B;
    delete[] C;

    system("Pause");
    return 0;
}
```

Multidimensional Arrays

- Multidimensional dynamic arrays are created using pointer of pointers
- In this course, we restrict our attention to two dimensional dynamic arrays
- A 2D dynamic array is simply a pointer to a pointer
- A 2D array is a rectangular object with some rows and some columns
- Thus a 2D array is a matrix



2D Dynamic Array

- **Syntax**

```
data_type **M;
```

- In order to create a 2D array (matrix) of **R** rows and **C** columns, we proceed as follows:

```
M = new data_type*[R];
for (int i = 0; i < R; i++)
   M[i] = new data_type[C];
```

- The following example demonstrates creation of a 4x5 dynamic array (matrix) of floats

2D Dynamic Array

```
float **M;
M = new float*[4];
for (int i = 0; i < 4; i++)
   M[i] = new float[5];
```

- Alternatively

```
float **M = new float* [4];
for (int i = 0; i < 4; i++)
   M[i] = new float[5];
```

Processing 2D Dynamic Array

- Elements are accessed with two indices
- The first index denotes the row index and the second index denotes the column index
- Thus

  `M[3][2]`

  is the element of the matrix at the 4th row and 3rd column! (Remember indexing starts at 0)

Freeing 2D Dynamic Arrays

- Just like in the 1D dynamic arrays, 2D dynamic arrays also need to be deleted manually by the programmer when they are not needed any more
- To do so, we first delete each row (inner pointers) and then delete the 2D array pointer (outer pointer) as follows

```
for (int i = 0; i < rowSize; i++)
   delete[] M[i];
delete[] M;
```

2D Dynamic Array Complete Example

- In order to demonstrate the creation, processing, and deletion of two dimensional arrays, we look at the following example:

- **Example 5.** Write a complete C++ program that asks the user for row size and column size, creates two matrices **A** and **B** of integers of the user defined sizes, populates the matrices with random integers in the range [1, 6], prints the original matrices, computes the sum of the matrices **A** and **B** into a matrix **C**, prints the sum matrix **C**, and finally deletes the dynamically allocated memory of the matrices **A**, **B** and **C**.

2D Dynamic Array Complete Example

Let’s Do It

2D Dynamic Array with functions

- Similar to 1D dynamic arrays, we can also pass 2D dynamic arrays to functions
- As an example, in order to populate a 2D dynamic array inside a function, the following function can be deployed

```
void populateMatrix(int **p, const int R, const int C)
{
	for (int i = 0; i < R; i++)
	{
		for (int j = 0; j < C; j++)
			p[i][j] = rand() % 6 + 1;
	}
	return;
}
```

- The function call to populate a matrix **M** with **R** rows and **C** columns would then be
  populateMatrix(M, R, C);

2D Dynamic Array with functions

- Similarly a 2D array of floats can be created inside a function and returned as follows:

```
float** createMatrix(const int R, const int C)
{
    float **p = new float*[R];
    for (int i = 0; i < R; i++)
          p[i] = new float[C];
    return p;
}
```

- The function call in order to create a new matrix would then be

```
float **M = createMatrix(rowSize, colSize);
```