Chapter III
References and Pointers

In this chapter we will cover:

1. Reference variables.
2. Pointers.
3. Allocating and Deallocating Dynamic Arrays.
4. Pointers to structures.

3.1 Reference variables

References create aliases to variables or objects:

```
int x;
int &y=x;       // y is a reference or an alias to the variable x
```

Note that type& var, type &var, and type & var are all equivalent. The following program illustrates the alias nature of a reference variable:

```
#include <conio.h>
#include <iostream.h>

int main()
{
  int x;
  int& y=x;

  x=2;
  y+=2;

  cout << "x=" << x << endl;
  cout << "y=" << y << endl;

  getch();
     return 0;
}
```

The print out of the above program is:

```
x=4;
y=4;
```

Since y is an alias to x then x and y should contain the same value.

In chapter I a brief introduction to functions was introduced. In C++ arguments of a function can be passed by value or by reference. Passing arguments by value allows the function to use the value but any modification to the argument is not reflected in the calling program. That is, the function receives a private copy of the argument and any changes to the private copy will not reflect to the original variable. If, on the other hand, the argument is passed by reference then the function can operate directly on the original variable. The following program illustrates the difference between passing arguments by value and by reference.

```
#include <conio.h>
#include <iostream.h>
void func1(int); //protoype for func1 (argument by value)
void func2(int&); //prototype for func2 (argument by reference)
int main()
{
int x,y;
  x=y=2;
  func1(x);
  func2(y);
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  getch();
       return 0;
}

//Definition of func1
void func1 ( int x)
   {
   x++;
   }

//Definition of func2
void func2 (int & x)
   {
   x++;
   }
```

The print out of the above program is:

```
x=2
y=3
```

Since the argument in func1 is passed by value the changes made to the argument within the function is not reflected in main(). On the other hand the argument in func2 is an alias to the original variable and hence any changes in the argument is reflected to the original variable.

3.2 Pointers to simple variables

A pointer is basically an address in memory identifying a variable. The following syntax is used in declaring a pointer to variable *var1* of *type1*:

```
type1 *var1;
```

For example the following statement declares a pointer to an integer.

```
int * i;    // i is a pointer to an int
```

i is an address in memory identifying an integer.
To store a value at i we use the following structure:

```
*i=2;
```

The asterisk when used on a pointer in the non-declaration part of the program is considered by the compiler as a de-reference to that pointer.

To obtain of a simple variable within the non-declaration part of the program we use the ampersand as follows:

```
int j;      // declaring a variable j
int * i;

i = &j;  // the address of j is assigned to the pointer i
```

Note that an ampersand when used in a declaration statement is interpreted by the compiler as an alias to a previously defined variable. When an ampersand precedes a variable in any other non-declarative statement it is meant for obtaining the address of the variable. Before we can illustrate these new definition in a program we need first to understand two operators used in C++ for memory allocation and deallocation.

3.3 The new and delete operators

In C++ the region of memory that is available at run time is called the free store. The **new** operator allows you to allocate memory for pointers to any type (e.g. int, float, structures, classes, etc.). The **delete** operator deallocates memory, returning them to the free store for subsequent use.

The syntax for the **new** operator

```
pointer = new type;
```

The syntax for the **delete** operator

```
delete pointer;
```

The following program illustrates the usage of the **new** and **delete** operators

```
#include <conio.h>
#include <iostream.h>

int main()
{
  int *x;

  x=new int;

  *x=4;

  cout << "*x = " << *x << endl;

  delete x;

// cout << "*x = " << *x; //incorrect

  getch();
     return 0;
}
```

3.4 Allocating and Deallocating a Dynamic Array

3.4.1 One Dimensional array

The syntax

```
ArrayPointer = new type[ArraySize];

delete [ ] ArrayPointer;
```

The following program illustrates usage.

```
#include <conio.h>
#include <iostream.h>

int main()
{
  float *p;
  int n,i;
```

```
  cout << "Enter size of array --> ";
  cin >> n;

  p = new float[n];

  for(i=0; i<n; i++ )
    *(p+i) = i;  //same as p[i]=i;

  for(i=0; i<n; i++)
   cout << p[i] << " "; //same as cout << p[i] << " ";

  delete [] p;

  getch();
  return 0;
}
```

The $(i+1)^{th}$ location in an array can be obtained using either syntax:

*(p+i) or p[i]

3.4.2 Two-Dimensional Array

A two-dimensional array can described as an array of pointers, each pointer points to a block of memory containing one row of the matrix. An array of pointers can be described as a double pointer **p as shown in Fig.1.

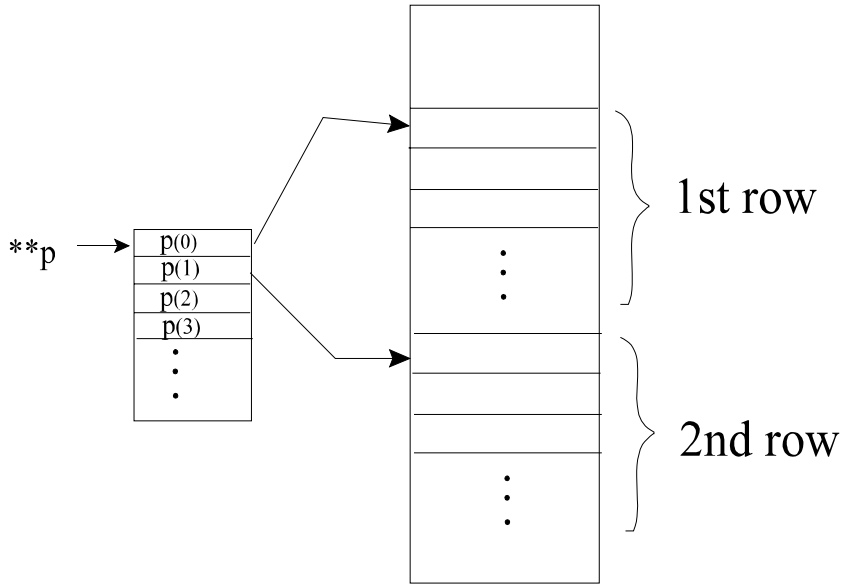

Figure 3.1 Describing a two-dimensional array in pointers.

To delete the two-dimensional array from memory you first have to delete the array of pointers and the pointer to the array of pointers. To clarify matters examine the following program:

```
#include <iostream.h>
#include <conio.h>

int main()
{
float **matrix;   //double pointer
int NR, NC;    //Number of rows, number of columns
int i,j;

cout << "Enter # of rows & columns --> ";
cin >> NR >> NC;

// Dynamic memory allocation

matrix = new float *[NR]; //setup the row pointers

for( j=0; j<NR; j++)
  matrix[j] = new float[NC]; //setup the number columns per row

// assign a value to each element of the matrix
for(i=0; i<NR; i++)
  {
  cout << endl;
  for( j=0; j<NC; j++)
   {
   matrix[i][j]=float(j+NC*i);
   cout << " " << matrix[i][j];
   }
  }

//Deallocate memory

for(i=0; i<NR; i++)
  delete[] matrix[i]; //free the array of pointers

delete [] matrix; //free the pointer to the array of pointers

getch();

      return 0;
```

```
}
```

Note the syntax used to allocate and deallocate memory for a double pointer. To assign values to a matrix you can also use:

```
*(*(matrix+i)+j)  ≡ matrix[i][j];
```

3.5 Pointers to Structures

An instance of a structure can be either a regular variable of type structure or a pointer. In the previous chapter members of a structure were accessed using the **dot** operator when a variable of type structure was used as an instance of the structure. However, if an instance of a structure is declared as a pointer then the members of the structure can be accessed using the **->** operator. As with pointers of any instance the **new** operator has to be used to dynamically allocate memory to the instance of the structure prior to usage. The following program clarifies these points:

```
#include <iostream.h>
#include <conio.h>

struct Complex
  {
  double Real;
  double Imag;
  };

int main()
{
Complex *p;
p = new Complex;

p->Real = double(4.5);
p->Imag = double(6.7);

cout << p->Real << ", i" << p->Imag << endl;

delete p;

getch();
return 0;
}
```

An array of pointers to a structure is the same as declaring a double pointer. Dynamic memory allocation in this case is similar to what we used in matrices. The following program

illustrates this point:

```
#include <iostream.h>
#include <conio.h>

struct Complex
  {
  double Real;
  double Imag;
  };

int main()
{
Complex **p;
int n=3, i;

p = new Complex *[n];

for(i=0; i<n; i++)
 {
 p[i]=new Complex;
 p[i]->Real = double(i);
 p[i]->Imag = double(i);
 }

for(i=0; i<n; i++)
 cout << '(' << p[i]->Real << ", i" << p[i]->Imag
              << ')' << endl;

for(i=0; i < n; i++)
 delete [ ] p[i];

delete [ ] p;

getch();
return 0;
}
```

Problems

1. Complete the following program:

```
#include <iostream.h>
```

```
void main()
  {
float *p;
int n;

// Dynamically allocate memory for p to represent an array of n float elements

...

// Assign n random values to the linear array

...
// Determine and print out the maximum and minimum values in the array
...

// Deallocate the memory assigned to p
...
}
```

2. Repeat problem 1 but this time for a double pointer representing a matrix.

3. Develop a program for multiplying two matrices A and B and assign the result to matrix C. A is of size N×M and B is of size M×L. The elements of C can be obtained using the equation:

$$c_{ij} = \sum_{k=0}^{M} a_{ik} b_{kj}$$

Use dynamic memory allocation for all three matrices.

4. Test the above program on the following two matrices:

$$A = \begin{bmatrix} 4 & -2 & 1 \\ 1 & -3 & -4 \\ 2 & 5 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} -2 & 3 & 1 & 5 \\ 2 & 0 & -4 & -1 \\ 1 & 6 & -3 & 2 \end{bmatrix}$$

Check your answer by hand calculation.