Pointers

Introduction to pointers

Every byte in the computer's memory has an address. Addresses are numbers just as they are for our homes on a street. When a program is loaded into the memory, depending on its size, it occupies a certain range of these addresses. Suppose the following variables are declared in our program. Each one of them take up some space in the memory as described in the table:

Memory Addresses	Variables (Type & Name)
123000	int num
123001	
123002	char c
123003	int count
123004	
123005	float salary
123006	
123007	
123008	
123009	

The address of the first variable, integer variable num, is 123000. Note that two bytes are allocated to store num. **The top location is what we refer to as the address of num**.

The address of the second variable, character variable c, is 123002. Similarly, we can determine the addresses of other variables.

In C++, you can find the address of a variable by using the "address of" operator, ***&***. Consider the following program segment:

```
1.  int num;
2.  // ....
3.  // ....
4.  cout << &num;
```

The above cout statement prints out 123000, i.e. the address of the variable num. Note that the address of a variable **is NOT the same** as its content. Consider the following statements in a C++ program:

Declaration:

```
1.  int     num;
2.  char    c;
3.  int     count;
4.  float   salary;
```

Assignment:

```
num = 7777;
c = 'z';
count = 999;
salary = 6920.77;
```

By assigning a value to a variable, we change the content in the memory location that is assigned to the variable. The following table shows how the assignments are made.

Memory Addresses	Variables (Type & Name)	Variable Name
123000	7777	num
123001		
123002	z	c
123003	999	count
123004		
123005	6920.77	salary
123006		
123007		
123008		

We have seen variable types that store characters, integers and floating-point numbers. In a very similar fashion, addresses can be are stored. A variable that stores an address value is called a **pointer variable**, or simply a **pointer**.

Let us look at an example.

```
//P12_1.cpp. This program defines pointer variables
#include<iostream>
using namespace std;

int main()
{
    int   num;
    char  c;
    int   count;
    float salary;

    // Declaring bunch of pointers that point to nothing (point to null)
    int    *numptr;     // declare a pointer variable to an integer
    char   *cptr;       // declare a pointer variable to a character
    int    *countptr;   // declare a pointer variable to an integer
    float  *salaryptr;  // declare a pointer variable to a float

    // Making them point somewhere
    numptr = &num;          //numptr is pointing to num
    cptr = &c;              //cptr is pointing to c
    countptr = &count;      //countptr is pointing to count
    salaryptr = &salary;    //salaryptr is pointing to salary
```

```
23.
24.     // Assign values to the locations where the pointers point
25.     *numptr = 2;
26.     *cptr = 'A';
27.     *countptr = 100;
28.     *salaryptr = 3200;
29.
30.     // Display the contents of those memory locations
31.     cout << num << endl;
32.     cout << c << endl;
33.     cout << count << endl;
34.     cout << salary << endl;
35.
36.    return 0;
37. }
```

This program defines four pointer variables. The asterisk (*) before the variable name implies pointer to. The * is called the dereferencing operator. Thus,

```
1.  int *numptr;
```

defines the variable numptr as a pointer variable to an integer value, i.e. numptr can hold addresses of integer variables. *Note that the above pointers are not pointing at anything when they are declared.*

```
1.  int    *numptr;     // declare a pointer variable to an integer
2.  char   *cptr;       // declare a pointer variable to a character
3.  int    *countptr;   // declare a pointer variable to an integer
4.  float  *salaryptr;  // declare a pointer variable to a float
```

It is said that they are **pointing to null**. We have only defined pointers that can point to variables of different types, but they do not point to anything yet. In order to make use of them, they have to point to something of their types.

That what we did in the following four lines:

```
1.  numptr = &num;          //numptr is pointing to num
2.  cptr = &c;              //cptr is pointing to c
3.  countptr = &count;      //countptr is pointing to count
4.  salaryptr = &salary;    //salaryptr is pointing to salary
```

These pointers are pointing to some memory locations, we can used these pointers to assign values to those locations:

```
1.  *numptr = 2;
2.  *cptr = 'A';
3.  *countptr = 100;
4.  *salaryptr = 3200;
```

At last, we displayed the contents of those locations:

```
1.  cout << num << endl;
2.  cout << c << endl;
3.  cout << count << endl;
4.  cout << salary << endl;
```

As you may have noticed, we did not directly assign anything to num, c, count, and salary. Instead, we used the pointers to assign values to where those pointers were pointing.

Note that:

`int *numptr;`	`int* numptr;`

are equivalent. It is the same to the compiler whether you define it one way or the other. However, you must be careful when you make multiple definitions in one line.

```
int* ptr1, ptr2;
```

The above example defines ptr1 as a pointer variable that can hold addresses of integer variables, but ptr2 is simply an integer and **NOT** a pointer variable. i.e.

`int* ptr1, ptr2;`	`int *ptr1, *ptr2;`

The above examples are NOT equivalent. The first statement defines ptr1 as a pointer variable but ptr2 as an integer variable, while the second statement defines both ptr1 and ptr2 as pointer variables to integers. Thus, if you define more than one pointer variable of the same type on one line, you need only insert the type pointed to once, but you need to place the * before EACH variable name.

A pointer can be used to refer to a variable. Your program can manipulate variables even if the variables have no identifiers to name them. These nameless variables are referred to by pointers.

Let us look at P12_1a.cpp to see how this is done.

```
1.  //P12_1a.cpp . This program illustrates dynamic variables
2.  #include <iostream>
3.  using namespace std;
4.
5.  int main ()
6.  {
7.     int *p1;
8.
9.     p1 = new int; // Variables created using the new operator are called dynamic variabl
    es
10.
11.    cout << "Enter an integer \n";
12.    cin >> *p1;
13.    *p1 = *p1 + 7;
14.    cout << << "Your input + 7 = " << *p1 << endl;
15.
16.    delete p1; // Delete the dynamic variable p1 and return the memory occupied by p1 to
     the freestore to be reused.
17.
18.    return 0;
19. }
```

The new operator produces a new nameless variable and returns a pointer that points to this new variable. This type of variable is called a **dynamic variable**.

Dynamic variables are stored in a special area of memory called the freestore or the heap. Any new dynamic variable created by a program consumes some of the memory in the freestore. If your program creates too many dynamic variables, it will consume all of the memory in the freestore. If this happens, any additional calls to the new will fail.

If your program no longer needs a dynamic variable, the memory used by that dynamic variable can be returned to the freestore to be reused to create other dynamic variables. The **delete operator** eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore, so the memory can be reused.

Dynamic Arrays

There is a close association between pointers and arrays (how array elements are accessed). In C++, an array variable is actually a pointer variable that points to the first indexed variable of the array. Array elements can be accessed using pointer notation as well as array notation. For example:

```
1.  // P10_1.cpp - A program that uses an array of integers
2.  #include <iostream>
3.  using namespace std;
4.
5.  int main()
6.  {
7.     int numlist[8];
8.     int i;
9.
10.    // Read 8 integers from the keyboard
11.    for (i = 0; i < 8; i++)
12.    {
13.       cout << "Enter value #" << i + 1 << ": ";
14.       cin >> numlist[i];
15.    }
16.    // Display the numbers in a reverse order
17.    for (i = 8; i > 0; i--)
18.    {
19.       cout << "Value #" << i << ": ";
20.       cout << numlist[i-1] << endl; //Pay attention to i-1!
21.    }
22.
23.    return 0;
24. }
```

Now let us re-write the program and access the array *numlist* using pointer notation.

```
1.  // P12_2.cpp - A program that uses an array of integers. Array elements accessed using
    pointer notation
2.  #include <iostream>
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int numlist[8];
8.
9.      // Read 8 integers from the keyboard
10.     for (int i = 0; i < 8; i++)
11.     {
12.         cout << "Enter value #" << i + 1 << ": ";
13.         cin >> numlist[i];
14.     }
15.     // Display the numbers in a reverse order
16.     for (int i = 8; i > 0; i--)
17.     {
18.         cout << "Value #" << i << ": ";
19.         cout << *(numlist + (i-1)) << endl; //Pay attention to i-1!
20.     }
```

```
21.
22.     return 0;
23. }
```

In the above program, note how the array elements are accessed. The variable *numlist* is used as a pointer that points to the first element of the array.

Pointer variable	Memory Address	Contents of memory (Array elements)
numlist[0]	333000	*numlist[0] = 9
numlist[1]	333002	*numlist[1] = 7
numlist[2]	333004	*numlist[2] = 8
numlist[3]	333006	*numlist[3] = 7
numlist[4]	333008	*numlist[4] = 7
numlist[5]	333010	*numlist[5] = 9
numlist[6]	333012	*numlist[6] = 1
numlist[7]	333014	*numlist[7] = 6

One problem with the kinds of arrays we have used until now is that we must specify the size of the array when we write the program. This may cause two different problems:

1. We may create an array much larger than needed.
2. We may create one that is smaller than what is needed.

In general, this problem is created because we do not know the size of the array until the program is running. This is where dynamic arrays are used. The new expression can be used to allocate an array on the freestore. Since array variables are pointer variables you can use the new operator to create dynamic variables that are arrays and treat these dynamic array variables as if they were ordinary arrays. Recall this program:

```
1.  // P10_1b.cpp - A program that uses a flexible size array of integers
2.  #include <iostream>
3.  using namespace std;
4.
5.  const int SIZE = 8; // Set the maximum size for the array
6.
7.  int main()
8.  {
9.     int numlist[SIZE];
10.
11.    // Read SIZE integers from the keyboard
12.    for (int i = 0; i < SIZE; i++)
13.    {
14.       cout << "Enter value #" << i + 1 << ": ";
15.       cin >> numlist[i];
16.    }
17.    // Display the numbers in a reverse order
18.    for (int i = SIZE; i > 0; i--)
19.    {
20.       cout << "Value #" << i << ": ";
21.       cout << numlist[i-1] << endl; //Pay attention to i-1!
22.    }
```

```
23.
24.     return 0;
25. }
```

Note that *SIZE* was defined as a const int. Now let us modify this program and re-write it creating the array dynamically.

```
1.  // P12_2a.cpp. Illustrates dynamic arrays
2.  #include <iostream>
3.  using namespace std;
4.
5.  int main()
6.  {
7.     int SIZE;
8.     cout << "Enter the size of the array" << endl;
9.     cin >> SIZE;
10.
11.    int *numlist = new int[SIZE];
12.
13.    // Read SIZE integers from the keyboard
14.    for (int i = 0; i<SIZE; i++)
15.    {
16.       cout << "Enter value #" << i + 1 << ": ";
17.       cin >> numlist[i];
18.    }
19.    // Display the numbers in a reverse order
20.    for (int i = SIZE; i > 0; i--)
21.    {
22.       cout << "Value #" << i << ": ";
23.       cout << numlist[i-1] << endl; //Pay attention to i-1!
24.    }
25.
26.    delete[] numlist;
27.    return 0;
28. }
```

Note how the array *numlist* has been declared dynamically using the new operator to meet the required size on the freestore. As new reserves memory on the freestore it is not automatically freed once the memory is no longer needed.

In an efficient program, we may take as much memory as we need and free them when once we are done with them. We need to use the delete operator to free the memory that is occupied from the freestore. Notice the empty square brackets along with the pointer variable *numlist*. This statement deletes the space on the occupied freestore by *numlist* and recycles it.

A dynamic array can have a base type which is a class. You can use the base type class just as you would use a predefined data type integer or float.