LECTURE 8 – POINTERS

Contents

7.1 Introduction

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

the following which will print the address of the variables defined.

```
#include <iostream>

using namespace std;
int main () {
  int  var1;
  char var2[10];

  cout << "Address of var1 variable: ";
  cout << &var1 << endl;

  cout << "Address of var2 variable: ";
  cout << &var2 << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6

7.2 What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication.

However, in this statement the asterisk is being used to designate a variable as a pointer.

Following are the valid pointer declaration −

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

7.3 Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently.

a. We define a pointer variable.
b. Assign the address of a variable to a pointer.
c. Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Following example makes use of these operations −

```
#include <iostream>

using namespace std;

int main () {
  int  var = 20;   // actual variable declaration.
  int  *ip;        // pointer variable

  ip = &var;       // store address of var in pointer variable

  cout << "Value of var variable: ";
  cout << var << endl;

  // print the address stored in ip pointer variable
  cout << "Address stored in ip variable: ";
  cout << ip << endl;

  // access the value at the address available in pointer
  cout << "Value of *ip variable: ";
  cout << *ip << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

7.4 Null pointers

It is always a good practice to assign the pointer NULL to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream. Consider the following program −

```
#include <iostream>

using namespace std;
int main () {
```

```
   int  *ptr = NULL;
   cout << "The value of ptr is " << ptr ;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows −

```
if(ptr)     // succeeds if p is not null
if(!ptr)    // succeeds if p is null
```

Thus, if all unused pointers are given the null value and you avoid the use of a null pointer, you can avoid the accidental misuse of an uninitialized pointer. Many times, uninitialized variables hold some junk values and it becomes difficult to debug the program.

7.5 Pointer Arithmetic

As you understood pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −

```
ptr++
```

the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer. This operation will move the pointer to next memory location without impacting actual value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

7.5.1 Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer.

The following program increments the variable pointer to access each succeeding element of the array −

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
  int  var[MAX] = {10, 100, 200};
  int  *ptr;

  // let us have array address in pointer.
  ptr = var;

  for (int i = 0; i < MAX; i++) {
    cout << "Address of var[" << i << "] = ";
    cout << ptr << endl;

    cout << "Value of var[" << i << "] = ";
    cout << *ptr << endl;

    // point to the next location
    ptr++;
  }

  return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

7.5.2 Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

```
#include <iostream>

using namespace std;
const int MAX = 3;
```

```
int main () {
  int  var[MAX] = {10, 100, 200};
  int  *ptr;

  // let us have address of the last element in pointer.
  ptr = &var[MAX-1];

  for (int i = MAX; i > 0; i--) {
    cout << "Address of var[" << i << "] = ";
    cout << ptr << endl;

    cout << "Value of var[" << i << "] = ";
    cout << *ptr << endl;

    // point to the previous location
    ptr--;
  }

  return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Address of var[3] = 0xbfdb70f8
Value of var[3] = 200
Address of var[2] = 0xbfdb70f4
Value of var[2] = 100
Address of var[1] = 0xbfdb70f0
Value of var[1] = 10
```

7.5.3 Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] −

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
```

```
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // let us have address of the first element in pointer.
   ptr = var;
   int i = 0;

   while ( ptr <= &var[MAX - 1] ) {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the previous location
      ptr++;
      i++;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Address of var[0] = 0xbfce42d0
Value of var[0] = 10
Address of var[1] = 0xbfce42d4
Value of var[1] = 100
Address of var[2] = 0xbfce42d8
Value of var[2] = 200
```

7.6 Pointers vs Arrays

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Consider the following program −

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int  *ptr;
```

```
   // let us have array address in pointer.
   ptr = var;

   for (int i = 0; i < MAX; i++) {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the next location
      ptr++;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

However, pointers and arrays are not completely interchangeable. For example, consider the following program −

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++) {
      *var = i;    // This is a correct syntax
      var++;       // This is incorrect.
   }

   return 0;
}
```

It is perfectly acceptable to apply the pointer operator * to var but it is illegal to modify var value. The reason for this is that var is a constant that points to the beginning of an array and can not be used as l-value.

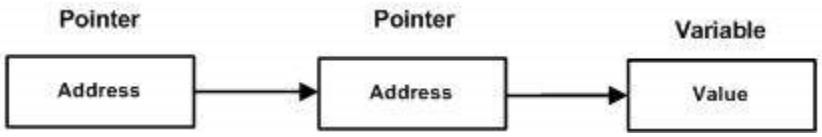
Because an array name generates a pointer constant, it can still be used in pointer-style expressions, as long as it is not modified. For example, the following is a valid statement that assigns var[2] the value 500 −

```
*(var + 2) = 500;
```

Above statement is valid and will compile successfully because var is not changed

7.7 Pointer to pointer

A pointer to a pointer is a form of multiple indirections or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int −

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example −

```
#include <iostream>

using namespace std;

int main () {
  int  var;
  int  *ptr;
  int  **pptr;

  var = 3000;

  // take the address of var
  ptr = &var;
```

```
   // take the address of ptr using address of operator &
   pptr = &ptr;

   // take the value using pptr
   cout << "Value of var :" << var << endl;
   cout << "Value available at *ptr :" << *ptr << endl;
   cout << "Value available at **pptr :" << **pptr << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of var :3000
Value available at *ptr :3000
Value available at **pptr :3000
```

7.8 Passing Pointers to Functions in C++

C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function −

```
#include <iostream>
#include <ctime>

using namespace std;
void getSeconds(unsigned long *par);

int main () {
   unsigned long sec;
   getSeconds( &sec );

   // print the actual value
   cout << "Number of seconds :" << sec << endl;

   return 0;
}

void getSeconds(unsigned long *par) {
   // get the current number of seconds
   *par = time( NULL );

   return;
}
```

When the above code is compiled and executed, it produces the following result −

```
Number of seconds :1294450468
```

The function which can accept a pointer, can also accept an array as shown in the following example −

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

// function declaration:
double getAverage(int *arr, int size);

int main () {
  // an int array with 5 elements.
  int balance[5] = {1000, 2, 3, 17, 50};
  double avg;

  // pass pointer to the array as an argument.
  avg = getAverage( balance, 5 ) ;

  // output the returned value
  cout << "Average value is: " << avg << endl;

  return 0;
}

double getAverage(int *arr, int size) {
  int i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }
  avg = double(sum) / size;

  return avg;
}
```

When the above code is compiled together and executed, it produces the following result −

```
Average value is: 214.4
```

7.8 Return Pointer from Functions in C++

As we have seen in last chapter how C++ allows to return an array from a function, similar way C++ allows you to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example −

```
int * myFunction() {
   .
   .
   .
}
```

Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function, so you would have to define the local variable as **static** variable.

Now, consider the following function, which will generate 10 random numbers and return them using an array name which represents a pointer i.e., address of first array element.

```
#include <iostream>
#include <ctime>

using namespace std;

// function to generate and retrun random numbers.
int * getRandom( ) {
   static int  r[10];

   // set the seed
   srand( (unsigned)time( NULL ) );

   for (int i = 0; i < 10; ++i) {
      r[i] = rand();
      cout << r[i] << endl;
   }

   return r;
}

// main function to call above defined function.
int main () {
   // a pointer to an int.
   int *p;

   p = getRandom();
   for ( int i = 0; i < 10; i++ ) {
      cout << "*(p + " << i << ") : ";
```

```
      cout << *(p + i) << endl;
   }

   return 0;
}
```

When the above code is compiled together and executed, it produces result something as follows

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```