Programming Fundamentals

Muhammad Ateeq

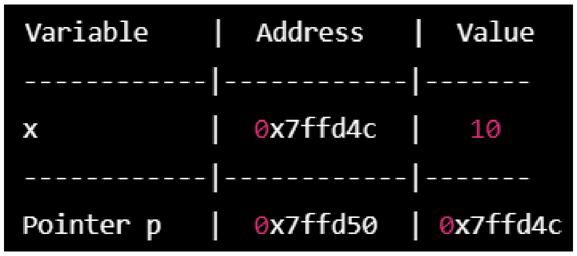
[Updated: 07-04-2023]

Pointers

What are pointers about?
Variables are stored in memory (RAM). In C++, a pointer is a variable that stores the memory address of another variable. It is declared using the asterisk symbol (*) before the variable name.

Normal integer variable:
```
int x = 10;
```
Pointer to an integer:
```
int *p = &x;
```

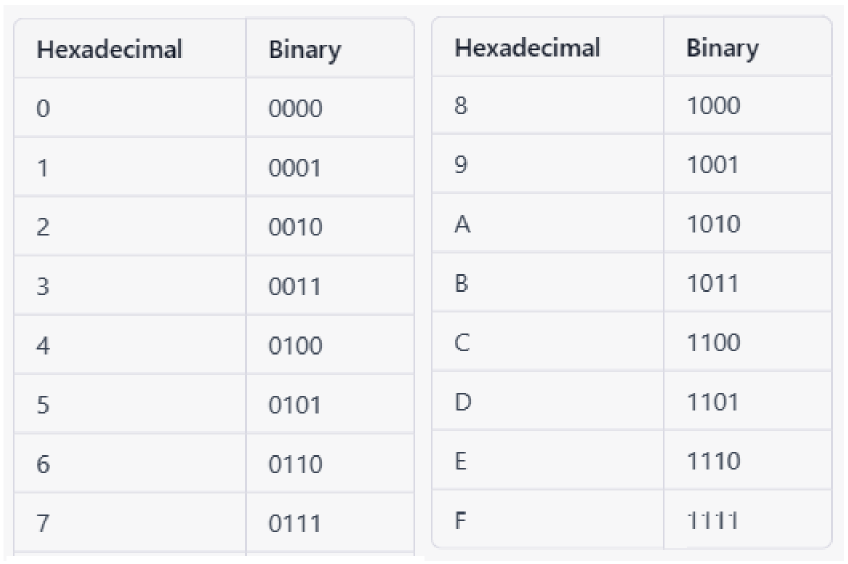
Here **&** used before the variable x is regarded as the **reference operator** or the **address operator**. Here's an illustration of a pointer addressing a variable in memory in C++:


```
Variable     |  Address     |  Value
-------------|-------------|-------
x            |  0x7ffd4c   |   10
-------------|-------------|-------
Pointer p    |  0x7ffd50   | 0x7ffd4c
```

In this example, x is a variable of type int with a value of 10, and p is a pointer to int that points to the address of x. The address of x in memory is 0x7ffd4c, and its value is 10. The address of p in memory is 0x7ffd50, which is the address of the pointer variable itself. The value of p is 0x7ffd4c, which is the address of x.
To access the value of x using the pointer p, we can use the dereference operator *. For example, *p would return the value 10.

The * is used for declaring the pointer and accessing the value, both. Isn't it confusing?
No actually the context matters. This simply means that *** serves two purposes**: one, it is used to **declare a pointer**; two, it is used as a **dereference** operator to access the value of a variable through the pointer to that variable.

Can you explain the addresses shown in the above example? For example, how to interpret 0x7ffd4c?
The addresses you see are represented using hexadecimal notation. You understand that we can use base 2, base 8, base 10, and base 16 number systems and convert a number from one base to the other. The default number system that we humans are familiar with is decimal (base 10). Computers on the other hand only understand binary number system (base 2). So at the backend addresses are also in binary notation. However, on modern machines address can be as long as 64 bits and it becomes hard to read and understand. Therefore, hexadecimal is a standard way to represent addresses. The reasons for adoption of hexadecimal notation include: compactness (large address can be represented in short space with less characters), ease of conversion (it is very easy to inter-convert hexadecimal and binary numbers. In fact each hexadecimal character ranging from 0-F can be represented using 4 binary digits). The conversion is given below:



Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Can we get a working example here?
Here is a working example of how pointers work.

```
#include <iostream>
using namespace std;
int main() {
    int x = 10; // Declare and initialize an integer variable x
    int *p; // Declare a pointer to an integer

    p = &x; // Assign the address of x to the pointer p

    cout << "The value of x is: " << x << endl;
    cout << "The address of x is: " << &x << endl;
    cout << "The value of p is: " << p << endl;
    cout << "The value that p points to is: " << *p << endl;

    *p = 20; // Assign a new value to the memory location pointed to by p

    cout << "The new value of x is: " << x << endl;

    return 0;
}
```

In this example, we declare an integer variable x and a pointer to an integer p. We assign the address of x to the pointer p using the address-of operator &. We then print out the value of x, the address of x, the value of p, and the value that p points to using the dereference operator *. Next, we assign a new value of 20 to the memory location pointed to by p using the dereference operator *. This updates the value of x to 20. Finally, we print out the new value of x to confirm that it has been updated.

Do arrays have anything to do with pointer?
Yes, an array (when used without subscript) is actually a pointer to the start location of memory where it is stored. If you dereference an array using * operator, it only gives you the first element in array. Here is an example demonstrating all this.

```
#include <iostream>
```

```
using namespace std;

int main() {
    int arr[3] = {1, 2, 3};
    cout << “The address of array is: ” << arr << endl;
    cout << “The first element in array is: ” << *arr << endl;

    int* ptr = arr; // initialize pointer to first element of array

    // Accessing elements using pointer
    cout << "Using pointer: ";
    for (int i = 0; i < 3; i++) {
        cout << *(ptr+i) << " "; // dereference pointer and add offset
    }
    cout << endl;

    // Accessing elements using subscript
    cout << "Using subscript: ";
    for (int i = 0; i < 3; i++) {
        cout << arr[i] << " "; // use subscript to access element
    }
    cout << endl;

    return 0;
}
```

In this example, we define an integer array arr of size 3 with values 1, 2, and 3. We first print the address of location where array is stored in memory and then the value of first element using the dereference (*) operator. We then define a pointer ptr and initialize it to the address of the first element of the array (&arr[0]).

To access the elements of the array using the pointer, we use the dereference operator * to get the value at the memory address pointed to by ptr, and we add an offset to the pointer to access subsequent elements in the array. In this case, we use the expression *(ptr+i) to access the i-th element of the array.

To access the elements of the array using subscripts, we simply use the subscript operator [] with the index of the element we want to access. In this case, we use the expression arr[i] to access the i-th element of the array.

The output of the program will be:

```
The address of array is: 0x5579d5
The first element in array is: 1
Using pointer: 1 2 3
Using subscript: 1 2 3
```

which shows that both methods of accessing the elements produce the same output.

Fair enough. But why do we need pointer at all?
Some question! Why would we make this effort at all? Let first discuss a bit about character arrays and then turn our attention to this pertinent question.

What is a character?

In C++, a char is a built-in data type that represents a single character, such as a letter, number, or symbol. It is a one-byte integer type that can store values between -128 and 127 (or 0 and 255, if unsigned).

Characters are typically represented in C++ using ASCII or Unicode encoding, where each character is assigned a unique numerical code that corresponds to its binary representation. For example, the character 'A' is typically represented using the ASCII code 65 (or 01000001 in binary).

To declare a char variable in C++, we use the char keyword followed by the variable name, like this:

```
char myChar = 'A';
```

In this example, myChar is a char variable that has been initialized with the character 'A'.

char variables can be used in many ways in C++. For example, we can use them to represent individual characters in a string, or to store single characters of input from the user. We can also perform operations on char variables, such as comparing them for equality or converting them to different types.

Ok, now I seem to sense what a character array would be! What are character arrays then?
In C++, a character array is a sequence of characters stored in contiguous memory locations. It is essentially a one-dimensional array of characters terminated by a **null character '\0'**. Character arrays are commonly used to represent strings in C++. For example:

```
char myString[] = "Hello, world!";
```

In this example, myString is a character array containing the string "Hello, world!". The size of the array is determined automatically based on the length of the string, including the null character. Individual characters in a character array can be accessed using array indexing, just like with other types of arrays. For example:

```
char myString[] = "Hello, world!";
char firstChar = myString[0]; // firstChar now contains 'H'
```

We can also use pointers to access character arrays. In fact, when we use the name of a character array without an index, it is automatically converted to a pointer to the first element of the array. For example:

```
char myString[] = "Hello, world!";
char* ptr = myString; // ptr now points to the first character of myString
```

This allows us to use pointer arithmetic to access individual characters in the array, just like with other types of arrays. For example:

```
char myString[] = "Hello, world!";
char* ptr = myString;

for (int i = 0; i < 5; i++) {
    cout << *ptr << endl; // output each character of myString
    ptr++; // move pointer to the next character
}
```

This would output the first 5 characters of the string "Hello, world!" one at a time: "H", "e", "l", "l", and "o".

What does incrementing a pointer mean? Is it the same as incrementing a normal variables? Does it vary from data type to data type?
Incrementing a pointer in C++ means moving the pointer to point to the next element in an array, where the size of the element depends on the data type of the array.

For example, let's say we have an integer array myArray with three elements, and a pointer p that points to the first element of the array:

```
int myArray[] = {1, 2, 3};
int* p = myArray;
```

If we increment the pointer p, it will point to the next integer element in the array (by moving 4 bytes head because integer takes 4 bytes), like this:

```
p++; // p now points to the second element of myArray
```

However, if we have a character array myString instead, and a pointer q that points to the first character of the array:

```
char myString[] = "Hello";
char* q = myString;
```

If we increment the pointer q, it will point to the next character in the array (by moving just 1 byte forward), since a char is one byte in size:

```
q++; // q now points to the second character of myString
```

So when we increment a pointer, the size of the data type being pointed to determines how much the pointer will be incremented. This is important to keep in mind when working with pointers to different data types.

What if we need to store multiple character arrays? Do we have 2D character arrays?
C++ supports 2D character arrays. A 2D character array is a 2D array where each element in the array is a character array (also known as a C-style string). We can think of it as a table or grid where each cell contains a character array. Here's an example of a 2D character array:

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

int main() {
    char names[3][6] = { "Alan", "Bob", "Carol" };

    // Accessing individual elements of the array
    cout << names[0]; // Output: Alan
    cout << names[2]; // Output: Carol

    // Iterating over the elements of the array
    for (int i = 0; i < 3; i++) {
        cout << names[i] << endl;
    }

    return 0;
}
```

In this example, names is a 2D character array with 3 rows and 6 columns. Each row represents a string of characters, and each column can hold a single character in the string (plus a null terminator).

We can access individual elements of the array using the row index, like this:

```
cout << names[0]; // Output: Alan
cout << names[2]; // Output: Carol
```

We can also use loops to iterate over the elements of the array, like this:

```
for (int i = 0; i < 3; i++) {
    cout << names[i] << endl;
}
```

This will output:

```
Alan
Bob
Carol
```

But, we have a problem with this type of 2D character array declaration, i.e., memory wastage. Lets see how?
Imagine the following declaration:

```
char names[3][10] = { "Alaxender", "Bob", "Carol" };
```

This is how the array will be represented in memory:

```
  0   1   2   3   4   5   6   7   8   9
+---+---+---+---+---+---+---+---+---+---+
| A | l | e | x | a | n | d | e | r | \0|
+---+---+---+---+---+---+---+---+---+---+
| B | o | b | \0|   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
| C | a | r | o | l | \0|   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
```

Each row of the array is allocated 10 characters of space, regardless of the actual length of the string. This means that in the first row, 9 characters are used for "Alexander", but 1 characters of space are wasted (although the '\0' oe null character is mandatory in this case). In the second row, only 3 characters are used for "Bob", but 7 characters of space are wasted. In the third row, only 5 characters are used for "Carol", but 5 characters of space are wasted.

This type of memory allocation can be wasteful if the strings in the array are of varying lengths. It may be more efficient to use dynamic memory allocation to allocate only the necessary amount of space for each string, rather than allocating a fixed amount of space for each row of the array.

Ok, so how can we make it better?
Consider the following two declarations:

```
char names[3][10] = { "Alexander", "Bob", "Carol" };
```
and
```
char *names[3] = { "Alexander", "Bob", "Carol" };
```

The first declaration, char names[3][10] = { "Alexander", "Bob", "Carol" };, creates a 2D character array where each row can store a string of up to 10 characters. The array is allocated with a fixed size, which means that each element takes up a fixed amount of memory regardless of whether it is used or not. In this case, the longest name is "Alexander", which takes up 9 characters, so each row has some unused memory. This leads to memory wastage, especially when dealing with large arrays.

On the other hand, the second declaration, char *names[3] = { "Alexander", "Bob", "Carol" };, creates an array of pointers to character arrays. Each element of the array is a pointer that points to a character array (i.e., a string). This allows for flexibility in terms of the size of each string because each string can be allocated separately and can be of different sizes. This means that there is no memory wastage because each element only takes up as much memory as needed to store the corresponding string.

In summary, the first declaration uses a fixed-size 2D character array, which can lead to memory wastage when the elements have varying sizes. The second declaration uses an array of pointers to character arrays, which allows for flexibility in the size of each element and avoids memory wastage.

```
char names[3][10] = { "Alaxender", "Bob", "Carol" };
```

Memory usage:

```
   0   1   2   3   4   5   6   7   8   9
 +---+---+---+---+---+---+---+---+---+---+
 | A | l | e | x | a | n | d | e | r | \0|
 +---+---+---+---+---+---+---+---+---+---+
 | B | o | b | \0|   |   |   |   |   |   |
 +---+---+---+---+---+---+---+---+---+---+
 | C | a | r | o | l | \0|   |   |   |   |
 +---+---+---+---+---+---+---+---+---+---+
```

Total memory used: 30 bytes (3 rows of 10 characters each) for data and the compiler uses pointers (8 bytes for each array) behind the scene to access the array elements. This makes it 30+24 = 54 bytes

```
char *names[3] = { "Alaxender", "Bob", "Carol" };
```

Memory usage:

```
                      0   1   2   3   4   5   6   7   8   9
                    +---+---+---+---+---+---+---+---+---+---+
| 0x7ffeedcbcd80 -> | A | l | e | x | a | n | d | e | r | \0|
                    +---+---+---+---+---+---+---+---+---+---+
| 0x7ffeedcbcd88 -> | B | o | b | \0|   **Free Space**
                    +---+---+---+---+---+---+---+---+---+---+
| 0x7ffeedcbcd90 -> | C | a | r | o | l | \0| **Free Space**
                    +---+---+---+---+---+---+---+---+---+---+
```

Total memory used: 20 bytes to store the data and 24 bytes (3 pointers, each pointing to a string literal). This makes it 20+24 = 44 bytes.

Simply, bigger arrays can cause far more wastage, therefore, pointers help.