Articles » Languages » C / C++ Language » Smart Pointers

Comments & Discussions

C Pointer Tricks

By **Ali BaderEddin**, 2 Jun 2010

4.46 (14 votes)

I've been exploring/reading more about C pointers (mostly from this source) and I've found the following interesting tricks.

Trick 1 – Pointer Arithmetics

Question

What is the result of running the following code?

```
void trick1()
{
    int arr[] = {1, 2, 3};
    int *ptr;

    ptr = arr;

    printf("arr[0] = %d, arr[1] = %d, arr[2] = %d, ptr = %p, *ptr = %d\n",
            arr[0], arr[1], arr[2], ptr, *ptr);
    *ptr++ = -1;
    printf("arr[0] = %d, arr[1] = %d, arr[2] = %d, ptr = %p, *ptr = %d\n",
            arr[0], arr[1], arr[2], ptr, *ptr);
    *++ptr = -2;
    printf("arr[0] = %d, arr[1] = %d, arr[2] = %d, ptr = %p, *ptr = %d\n",
            arr[0], arr[1], arr[2], ptr, *ptr);
    (*ptr)++;
    printf("arr[0] = %d, arr[1] = %d, arr[2] = %d, ptr = %p, *ptr = %d\n",
            arr[0], arr[1], arr[2], ptr, *ptr);
}
```

Solution

- ptr = arr
  - This will make the pointer ptr point to the first element in the array arr.
- *ptr++ = -1
  - There are two operators (* and ++) to be applied on a single operand (ptr). We have to know which one takes precedence. The answer is that the post-increment (++) will be applied first. Once it's applied, it will not affect the value of ptr until the statement completes execution. So as we are evaluating the current statement, ptr is still pointing to the first element of the array (arr[0]). Now, we apply the * operator to the ptr operand. This will dereference the pointer and will let us access the content it points to. Since it's pointing to arr[0], *ptr = -1 will set the value of arr[0] to -1. After the statement completes execution, the pointer ptr is incremented and thus will point to the next element in the array (arr[1]).
- *++ptr = -2
  - As we mentioned previously, the increment operator (++) takes precedence over the dereference

As we mentioned previously, the increment operator (++) takes precedence over the dereference operator (*). Since this is a pre-increment operator, ++ptr will make the pointer point to the next element in the array arr[2]. *ptr = -2 will then set the value of arr[2] to -2.

- (*ptr)++
  - This one should be clear by now. *ptr will point us to arr[2] and thus running (*ptr)++ will increment the value of the integer that ptr is pointing to. So the value of arr[2] will be incremented by 1 once the statement completes execution and the pointer ptr will still point to arr[2].
  - If we try to compile (*ptr)++ = 0 we will get the following compilation error: `“error C2106: ‘=’ : left operand must be l-value”.`

The result would look like this:

```
arr[0] = 1, arr[1] = 2, arr[2] = 3, ptr = 0043FE44, *ptr = 1
arr[0] = -1, arr[1] = 2, arr[2] = 3, ptr = 0043FE48, *ptr = 2
arr[0] = -1, arr[1] = 2, arr[2] = -2, ptr = 0043FE4C, *ptr = -2
arr[0] = -1, arr[1] = 2, arr[2] = -1, ptr = 0043FE4C, *ptr = -1
```

Trick 2 – Array Indexing

Question

What is the result of running the following code?

```
void trick2()
{
    int arr[] = {1, 2, 3};

    *arr = 5;
    printf("arr[0] = %d, arr[1] = %d, arr[2] = %d\n",
           arr[0], arr[1], arr[2]);
    *(arr + 1) = 10;
    printf("arr[0] = %d, arr[1] = %d, arr[2] = %d\n",
           arr[0], arr[1], arr[2]);
    2[arr] = 15;
    printf("0[arr] = %d, 1[arr] = %d, 2[arr] = %d\n",
           0[arr], 1[arr], 2[arr]);
}
```

Solution

- *arr = 5
  - An array variable can be used as a pointer, but its value can’t be changed. You can think of it as a constant pointer, but they're not quite the same. Throughout the array’s life cycle on the stack, it will always point to the first element it pointed to when it was initialized. If we try to compile arr = &i or arr = arr2 or arr = ptr, we will get a compilation error: `“error C2106: ‘=’ : left operand must be l-value”`.
  - So back to the statement we are trying to evaluate, this will simply set the value of the first element in the array to 5. So now arr[0] = 5.
- *(arr + 1) = 10
  - arr is a pointer to a value of type int. Running arr + 1 would actually result in arr + 1 * `sizeof(int).` Assuming that arr represents address 2500 and `sizeof(int)` is 4, arr + 1 will actually represent address 2504 and not 2501. This is called *pointer arithmetics*, where the type of the object the pointer points to is taken into consideration when incrementing or decrementing the pointer.
  - *(arr + 1) is like saying arr[1] so it will let us access the second element of the array. Thus, the statement will set arr[1] to 10.
- 2[arr] = 15
  - This is the weirdest thing I’ve learned in C, but it does make sense after all. At first, you’d think this will result in a compilation error, but when compiling 2[arr], C compiler will convert it to *(2 + arr), which is

the same as *(arr + 2) since addition is commutative.

- So 2[arr] = 15 is like saying arr[2] = 15 and this the third element in the array will be set to 15.

The result would look like this:

```
arr[0] = 5, arr[1] = 2, arr[2] = 3
arr[0] = 5, arr[1] = 10, arr[2] = 3
0[arr] = 5, 1[arr] = 10, 2[arr] = 15
```

Trick 3 – Pointers to Arrays

Question

What's the difference between str1, str2, str3 and cPtr? What is the difference between sArr, sArr2D, sPtr1, sPtr2 and SPtr3? What will be the result of running the code below?

```
void trick3()
{
    char str1[] = { 'A', 'B', 'C', 'D', 'E' };
    char str2[] = "ABCDE";
    char *str3 = "ABCDE";
    char  *cPtr = str1;

    short sArr[] = {1, 2, 3, 4, 5};
    short sArr2D[][5] = { {1, 2, 3, 4, 5},
                          {6, 7, 8, 9, 10} };

    short *sPtr1 = sArr;
    short (*sPtr2)[5] = sArr2D;
    short *sPtr3[5];

    printf("sizeof(str1) = %u\n", sizeof(str1));
    printf("sizeof(str2) = %u\n", sizeof(str2));
    printf("sizeof(str3) = %u\n", sizeof(str3));
    printf("sizeof(cPtr) = %u\n", sizeof(cPtr));
    printf("\n");

    printf("sizeof(sArr) = %u\n", sizeof(sArr));
    printf("sizeof(sPtr1) = %u\n", sizeof(sPtr1));
    printf("sizeof(sArr2D) = %u\n", sizeof(sArr2D));
    printf("sizeof(sPtr2) = %u\n", sizeof(sPtr2));
    printf("sizeof(*sPtr2) = %u\n", sizeof(*sPtr2));
    printf("sizeof(sPtr3) = %u\n", sizeof(sPtr3));
    printf("\n");

    printf("sArr2D[1][2] = %d\n", sArr2D[1][2]);
    printf("sPtr2[0][0] = %d, sPtr2[1][2] = %d\n", sPtr2[0][0], sPtr2[1][2]);
    printf("*(* (sArr2D + 1) + 2) = %d\n", *(* (sArr2D + 1) + 2));
    printf("*(*(sArr2D) + 1*5 + 2) = %d\n\n", *(*(sArr2D) + 1*5 + 2));
}
```

Solution

- str1 is a char array with 5 elements: 'A', 'B', 'C', 'D', 'E'. str2 is a char array with 6 elements 'A', 'B', 'C', 'D', 'E' and '\0'. str3 is a *pointer to an array* of 6 elements 'A', 'B', 'C', 'D', 'E' and '\0'. str2 and str3 seem to mean the same thing, but actually they are different. str2 is not stored on the stack (from symbol table, it directly refers to the first element in the array) while str3 is stored on the stack and contains the value of the address of the first element of the array. In other words, str3 is storing an extra pointer of size sizeof(char *), which is 4 bytes on a 32-bit system and 8 bytes on a 64-bit system. cPtr and str3 are the same in that they are pointers that point to a char array, though they are not pointing to the same array.
- There is one additional difference between pointers and arrays. If we run sizeof on an array, we'll get the full size of the array. However, if we run sizeof on a pointer, it will return the size of the pointer only and not the content it refers to.
    - So sizeof(str1) is 5 * sizeof(char), which is 5 * 1 = 5. sizeof(str2) is 6.
    - sizeof(str3) is the size of the pointer sizeof(char *), which is 4 bytes. sizeof(cPtr) is also 4 bytes.
- sArr is an array of 5 short numbers. Thus, its size is 5 * sizeof(short) = 10. sPtr is a pointer that points

sArr is an array of 5 short numbers. Thus, its size is 5 * sizeof(short) = 10. sPtr is a pointer that points to sArr[0] and its size sizeof(sPtr) is 4 bytes. sArr2D is a 2×5 2-dimensional array with size = 2 * 5 * sizeof(short) = 20. The statement short (*sPtr2)[5] = sArr2D will declare a pointer to an array of 2 dimensions where dimension 2 has a size of 5 then make the pointer point to the 2-dimensional array sArr2D. It does not do any allocation. So sizeof(sPtr2) is 4 bytes since it's a pointer. The sizeof(*sPtr2) is the 5 * sizeof(short) = 10. The short (*sPtr2)[5] statement creates a new array of 5 short numbers and makes the sPtr2 pointer point to it. So sizeof(sPtr2) is also 4 bytes. short *sPtr3[5] creates an array of 5 pointers to short values. The array contains pointers and not short values. So sizeof(sPtr3) = 5 * sizeof(short *) = 20.

- *Multi-dimensional Array*
    - sArr2D[1][2] simply tries to access the third element of the second array in the 2-dimensional array sArr2D. Think of sArr2D as an array of 2 arrays of 5 int values. So sArr2D[1][2] = 8.
    - sPtr2 is a pointer to the 2-dimensional array sArr2D and thus can be used in the same way as variable sArr2D. So sPtr2[1][2] is the same as sArr2D[1][2].
    - * (sArr2D + 1) is like sArr2D[1] which points us to the first element in the 2nd array sArr2D[1][0]. *(*(sArr2D + 1) + 2) is like *(sArr2D[1] + 2), which is like *(&sArr2D[1][0] + 2), which is like *(&sArr2D[1][1]), which is sArr2D[1][2].
    - *(sArr2D) + 1 * (column count) is the same as sArr2D[1]. So *(*(sArr2D) + 1*5 + 2) is *(&sArr2D[1][0] + 2), which is sArr2D[1][2].

The full result of the above code would look like this:

```
sizeof(str1) = 5
sizeof(str2) = 6
sizeof(str3) = 4
sizeof(cPtr) = 4

sizeof(sArr) = 10
sizeof(sPtr1) = 4
sizeof(sArr2D) = 20
sizeof(sPtr2) = 4
sizeof(*sPtr2) = 10
sizeof(sPtr3) = 20

sArr2D[1][2] = 8
sPtr2[0][0] = 1, sPtr2[1][2] = 8
*(* (sArr2D + 1) + 2) = 8
*(*(sArr2D) + 1*5 + 2) = 8
```

Source Code

The full source code for the above tricks is available here.

Filed under: C Tagged: *ptr++, Array Indexing, Multi-dimensional Array, Pointer Arithmetics, pointers, Pointers to Arrays Comments: 4 Stumble it! Digg it! Add to Reddit!

License

About the Author

Ali BaderEddin
Software Developer Microsoft

United States 🇺🇸
Member

http://mycodelog.com/about/

Comments and Discussions

22 messages have been posted for this article Visit **http://www.codeproject.com/Articles/82880/C-Pointer-Tricks** to post and view comments on this article, or click **here** to get a print view with messages.