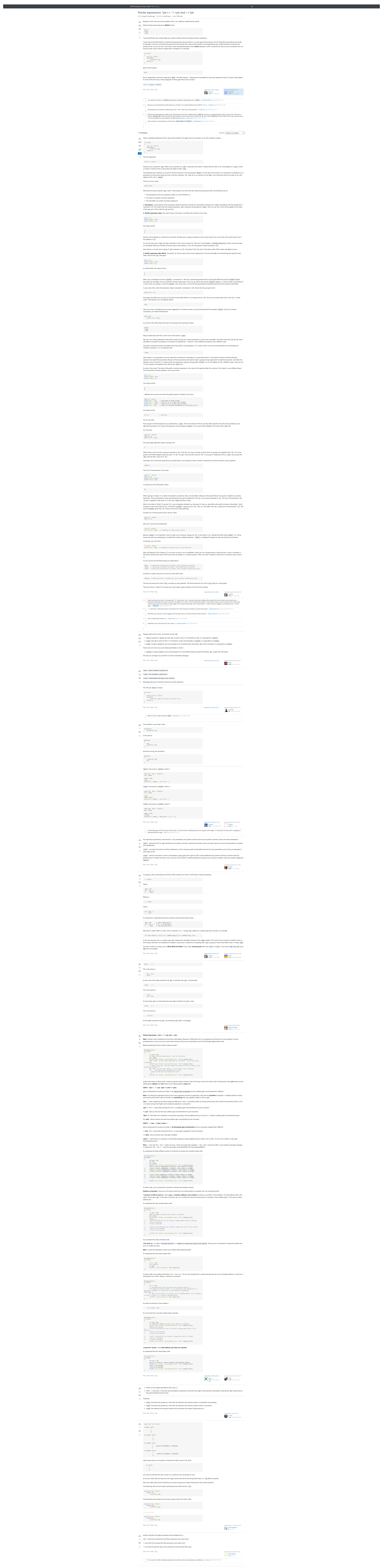

Pointer expressions: *ptr++, *++ptr and ++*ptr

Asked 8 years, 8 months ago Modified 3 months ago Viewed 93k times

145

Recently I have come across this problem which I am unable to understand by myself.

What do these three Expressions **REALLY** mean?

```
*ptr++
*++ptr
++*ptr
```

I have tried Ritchie. But unfortunately was unable to follow what he told about these 3 operations.

I know they are all performed to increment the pointer/the value pointed to. I can also guess there may be a lot of things about precedence and order of evaluation. Like one increments the pointer first then fetches the content of that pointer, one simply fetches the content and then increments the pointer etc etc. As you can see, I don't have a clear understanding about their **actual** operations, which i would like to clear as soon as possible. But I am truly lost when I get a chance to apply them in programs. For example:

```
int main()
{
    char *p = "Hello";
    while(*p++)
         printf("%c",*p);
    return 0;
}
```

gives me this output:

```
ello
```

But my expectation was that it would print `Hello` . One final request -- Please give me examples for how each expression works in a given code snippet. As most of the time only a mere paragraph of theory gets flown over my head.

c++ c pointers increment

Share Edit Follow Flag — edited Mar 4, 2021 at 20:46 verbose 7,569 ●1 ●24 ●39 — asked Aug 28, 2013 at 7:27 allocated 1,673 ●3 ●12 ●15

7 You missed the fourth one: `(*ptr)++` (parentheses needed to disambiguate from `*ptr++`) – user4815162342 Aug 28, 2013 at 7:28

16 Because you incremented the pointer before you printed it. You wanted while(*p) and printf("%c", *p++); – dcaswell Aug 28, 2013 at 7:30

Great questions for interview. Limited practical use. I wish C didn't have those pointers :) – Himanshu Aug 28, 2013 at 7:37

5 @Himanshu If that bakes the noodle of your interviewee try this: Have a *global* pointer, `char* p` , pointing to a valid terminated string of unique chars. Then have a function `fn(char ch)` that prints *both* the `ch` parameter *and* the current char pointed to by `p` . Now invoke `fn(*p++);` Q: Does `fn` print the same character *twice* ? You'd be amazed how many *professors* get that question wrong. – WhozCraig Aug 28, 2013 at 7:52

1 since p points to a string literal you should write `const char* p = "Hello";` – hetepeperfan Aug 28, 2013 at 8:21

11 Answers

Sorted by: Highest score (default)

309 +500

Here's a detailed explanation which I hope will be helpful. Let's begin with your program, as it's the simplest to explain.

```
int main()
{
    char *p = "Hello";
    while(*p++)
        printf("%c",*p);
    return 0;
}
```

The first statement:

```
char* p = "Hello";
```

declares `p` as a pointer to `char`. When we say "pointer to a `char`", what does that mean? It means that the value of `p` is the address of a `char`; `p` tells us where in memory there is some space set aside to hold a `char`.

The statement also initializes `p` to point to the first character in the string literal `"Hello"`. For the sake of this exercise, it's important to understand `p` as pointing not to the entire string, but only to the first character, `'H'`. After all, `p` is a pointer to one `char`, not to the entire string. The value of `p` is the address of the `'H'` in `"Hello"`.

Then you set up a loop:

```
while (*p++)
```

What does the loop condition `*p++` mean? Three things are at work here that make this puzzling (at least until familiarity sets in):

1. The precedence of the two operators, postfix `++` and indirection `*`
2. The value of a postfix increment expression
3. The side effect of a postfix increment expression

1. Precedence. A quick glance at the precedence table for operators will tell you that postfix increment has a higher precedence (16) than dereference / indirection (15). This means that the complex expression `*p++` is going to be grouped as: `*(p++)`. That is to say, the `*` part will be applied to the value of the `p++` part. So let's take the `p++` part first.

2. Postfix expression value. The value of `p++` is the value of `p` *before the increment*. If you have:

```
int i = 7;
printf ("%d\n", i++);
printf ("%d\n", i);
```

the output will be:

```
7
8
```

because `i++` evaluates to `i` before the increment. Similarly `p++` is going to evaluate to the current value of `p`. As we know, the current value of `p` is the address of `'H'`.

So now the `p++` part of `*p++` has been evaluated; it's the current value of `p`. Then the `*` part happens. `*(current value of p)` means: access the value at the address held by `p`. We know that the value at that address is `'H'`. So the expression `*p++` evaluates to `'H'`.

Now hold on a minute, you're saying. If `*p++` evaluates to `'H'`, why doesn't that `'H'` print in the above code? That's where *side effects* come in.

3. Postfix expression side effects. The postfix `++` has the *value* of the current operand, but it has the *side effect* of incrementing that operand. Huh? Take a look at that `int` code again:

```
int i = 7;
printf ("%d\n", i++);
printf ("%d\n", i);
```

As noted earlier, the output will be:

```
7
8
```

When `i++` is evaluated in the first `printf()`, it evaluates to 7. But the C standard guarantees that at some point before the second `printf()` begins executing, the *side effect* of the `++` operator will have taken place. That is to say, before the second `printf()` happens, `i` will have been incremented as a result of the `++` operator in the first `printf()`. This, by the way, is one of the few guarantees the standard gives about the timing of side effects.

In your code, then, when the expression `*p++` is evaluated, it evaluates to `'H'`. But by the time you get to this:

```
printf ("%c", *p)
```

that pesky side-effect has occurred. `p` has been incremented. Whoa! It no longer points to `'H'`, but to one character past `'H'`: to the `'e'`, in other words. That explains your cockneyfied output:

```
ello
```

Hence the chorus of helpful (and accurate) suggestions in the other answers: to print the Received Pronunciation `"Hello"` and not its cockney counterpart, you need something like

```
while (*p)
    printf ("%c", *p++);
```

So much for that. What about the rest? You ask about the meanings of these:

```
*ptr++
*++ptr
++*ptr
```

We just talked about the first, so let's look at the second: `*++ptr`.

We saw in our earlier explanation that postfix increment `p++` has a certain *precedence*, a *value*, and a *side effect*. The prefix increment `++p` has the same *side effect* as its postfix counterpart: it increments its operand by 1. However, it has a different *precedence* and a different *value*.

The prefix increment has lower precedence than the postfix; it has precedence 15. In other words, it has the same precedence as the dereference / indirection operator `*`. In an expression like

```
*++ptr
```

what matters is not precedence: the two operators are identical in precedence. So *associativity* kicks in. The prefix increment and the indirection operator have right-left associativity. Because of that associativity, the operand `ptr` is going to be grouped with the rightmost operator `++` before the operator more to the left, `*`. In other words, the expression is going to be grouped `*(++ptr)`. So, as with `*ptr++` but for a different reason, here too the `*` part is going to be applied to the value of the `++ptr` part.

So what is that value? The value of the prefix increment expression is the value of the operand *after the increment*. This makes it a very different beast from the postfix increment operator. Let's say you have:

```
int i = 7;
printf ("%d\n", ++i);
printf ("%d\n", i);
```

The output will be:

```
8
8
```

... different from what we saw with the postfix operator. Similarly, if you have:

```
char* p = "Hello";
printf ("%c ", *p);    // note space in format string
printf ("%c ", *++p);  // value of ++p is p after the increment
printf ("%c ", *p++);  // value of p++ is p before the increment
printf ("%c ", *p);    // value of p has been incremented as a side effect of p++
```

the output will be:

```
H e e l                // good dog
```

Do you see why?

Now we get to the third expression you asked about, `++*ptr`. That's the trickiest of the lot, actually. Both operators have the same precedence, and right-left associativity. This means the expression will be grouped `++(*ptr)`. The `++` part will be applied to the value of the `*ptr` part.

So if we have:

```
char q[] = "Hello";
char* p = q;
printf ("%c", ++*p);
```

the surprisingly egotistical output is going to be:

```
I
```

What?! Okay, so the `*p` part is going to evaluate to `'H'`. Then the `++` comes into play, at which point, it's going to be applied to the `'H'`, not to the pointer at all! What happens when you add 1 to `'H'`? You get 1 plus the ASCII value of `'H'`, 72; you get 73. Represent that as a `char`, and you get the `char` with the ASCII value of 73: `'I'`.

That takes care of the three expressions you asked about in your question. Here is another, mentioned in the first comment to your question:

```
(*ptr)++
```

That one is interesting too. If you have:

```
char q[] = "Hello";
char* p = q;
printf ("%c", (*p)++);
printf ("%c\n", *p);
```

it will give you this enthusiastic output:

```
HI
```

What's going on? Again, it's a matter of *precedence*, *expression value*, and *side effects*. Because of the parentheses, the `*p` part is treated as a primary expression. Primary expressions trump everything else; they get evaluated first. And `*p`, as you know, evaluates to `'H'`. The rest of the expression, the `++` part, is applied to that value. So, in this case, `(*p)++` becomes `'H'++`.

What is the value of `'H'++`? If you said `'I'`, you've forgotten (already!) our discussion of value vs. side effect with postfix increment. Remember, `'H'++` evaluates to the *current value of* `'H'`. So that first `printf()` is going to print `'H'`. Then, as a *side effect*, that `'H'` is going to be incremented to `'I'`. The second `printf()` prints that `'I'`. And you have your cheery greeting.

All right, but in those last two cases, why do I need

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

Why can't I just have something like

```
char* p = "Hello";
printf ("%c", ++*p);   // attempting to change string literal!
```

Because `"Hello"` is a string literal. If you try `++*p`, you're trying to change the `'H'` in the string to `'I'`, making the whole string `"Iello"`. In C, string literals are read-only; attempting to modify them invokes undefined behavior. `"Iello"` is undefined in English as well, but that's just coincidence.

Conversely, you can't have

```
char p[] = "Hello";
printf ("%c", *++p);  // attempting to modify value of array identifier!
```

Why not? Because in this instance, `p` is an array. An array is not a modifiable l-value; you can't change where `p` points by pre- or post- increment or decrement, because the name of the array works as though it's a constant pointer. (That's not what it actually is; that's just a convenient way to look at it.)

To sum up, here are the three things you asked about:

```
*ptr++   // effectively dereferences the pointer, then increments the pointer
*++ptr   // effectively increments the pointer, then dereferences the pointer
++*ptr   // effectively dereferences the pointer, then increments dereferenced value
```

And here's a fourth, every bit as much fun as the other three:

```
(*ptr)++ // effectively forces a dereference, then increments dereferenced value
```

The first and second will crash if `ptr` is actually an array identifier. The third and fourth will crash if `ptr` points to a string literal.

There you have it. I hope it's all crystal now. You've been a great audience, and I'll be here all week.

Share Edit Follow Flag — edited Mar 4, 2021 at 20:46 — answered Aug 28, 2013 at 10:05 verbose 7,569 ●1 ●24 ●39

30 Before coming to this forum, I have searched 3 "C" books which I own. I have also tried some notable online tutorials. But none of them comes close to your explanation (especially the way you have put it all together). You have not only answered the question I have asked, but also you have discussed a lot more things from the grass-root level. Actually you have taught me a lot basic things today, which I lacked before. I couldn't help but toggling my accepted answer. :) Thanks again. – allocated Aug 28, 2013 at 17:07

30 +1 I think this is a best long answer I have read on SO. I think everyone could learn a lot from this answer. – Shafik Yaghmour Aug 29, 2013 at 2:25

1 @verbose your welcome! I like to suggest avoid important texts as comment and horizontal scroll bars. – Grijesh Chauhan Aug 29, 2013 at 5:44

11 You sir, should write a book on C. – Dillon Burton Sep 17, 2013 at 4:41

9 @verbose you sir, have lived up to your name.. :) – sleeping_dragon Jan 9, 2014 at 8:10

46

Suppose `ptr` points to the i-th element of array `arr`.

1. `*ptr++` evaluates to `arr[i]` and sets `ptr` to point to the (i+1)-th element of `arr`. It is equivalent to `*(ptr++)`.
2. `*++ptr` sets `ptr` to point to the (i+1)-th element of `arr` and evaluates to `arr[i+1]`. It is equivalent to `*(++ptr)`.
3. `++*ptr` increases `arr[i]` by one and evaluates to its increased value; the pointer `ptr` is left untouched. It is equivalent to `++(*ptr)`.

There's also one more, but you'd need parentheses to write it:

1. `(*ptr)++` increases `arr[i]` by one and evaluates to its value before being increased; the pointer `ptr` is again left untouched.

The rest you can figure out yourself; it was also answered by @Jaguar.

Share Edit Follow Flag — edited Aug 28, 2013 at 7:44 — answered Aug 28, 2013 at 7:38 nickie 5,388 ●2 ●21 ●36

14

`*ptr++ : post increment a pointer ptr`

`*++ptr : Pre Increment a pointer ptr`

`++*ptr : preincrement the value at ptr location`

Read here about pre increment and post increment operators

This will give `Hello` as output

```
int main()
{
    const char *p = "Hello";
    while(*p)
         printf("%c",*p++);//Increment the pointer here
    return 0;
}
```

Share Edit Follow Flag — edited Oct 10, 2018 at 3:49 — answered Aug 28, 2013 at 7:32 Jainendra 24k ●30 ●119 ●167

@Nik-Lz Yes, the output would be `Hello` – Jainendra Oct 10, 2018 at 3:50

7

The condition in your loop is bad:

```
while(*p++)
    printf("%c",*p);
```

Is the same as

```
while(*p)
{
    p++;
    printf("%c",*p);
}
```

And that's wrong, this should be:

```
while(*p)
{
    printf("%c",*p);
    p++;
}
```

`*ptr++` is the same as `*(ptr++)`, which is:

```
const char  *ptr = "example";
char  value;

value = *ptr;
++ptr;
printf("%c", value); // will print 'e'
```

`*++ptr` is the same as `*(++ptr)`, which is:

```
const char  *ptr = "example";
char  value;

++ptr;
value = *ptr;
printf("%c", value); // will print 'x'
```

`++*ptr` is the same as `++(*ptr)`, which is:

```
const char  *ptr = "example";
char  value;

value = *ptr;
++value;
printf("%c", value); // will print 'f' ('e' + 1)
```

Share Edit Follow Flag — edited Nov 10, 2017 at 13:17 Jarod42 190k ●13 ●164 ●271 — answered Aug 28, 2013 at 7:41 nouney 4,303 ●17 ●31

I'll absolutely agree with the first part of the answer. In the second part, initializing pointers (to integers!) with integers is confusing for someone who's struggling to understand pointer usage. – nickie Aug 28, 2013 at 7:47

4

You right about precedence, note that the `*` has precedence over prefix increment, but not over postfix increment. Here's how these breakdown:

`*ptr++` - going from left-to-right, dereference the pointer, and then increment the pointer value (not what it points to, due to the precedence of postfix over dereference)

`*++ptr` - increment the pointer and then dereference it, this is because prefix and dereference have the same precedence and so they are evaluated in order right-to-left

`++*ptr` - similar to the above in terms of precedence, again going from right-to-left in order dereference the pointer and then increment what the pointer points to. Please note that in your case this one will lead to undefined behaviour because you're trying to modify a read-only variable (`char* p = "Hello";`).

Share Edit Follow Flag — answered Aug 28, 2013 at 7:38 Nobilis 6,880 ●1 ●30 ●62

3

I'm going to add my take because while the other answers are correct I think they're missing something.

```
 v = *ptr++
```

means

```
 temp = ptr;
 ptr  = ptr + 1
 v    = *temp;
```

Where as

```
 v = *++ptr
```

means

```
 ptr = ptr + 1
 v   = *ptr
```

It's important to understand that post increment (and post decrement) mean

```
 temp = ptr       // Temp created here!!!
 ptr  = ptr + 1   // or - 1 if decrement)
 v    = *temp     // Temp destroyed here!!!
```

Why does it matter? Well in C that's not so important. In C++ though `ptr` might be a complex type like an iterator. For example

```
 for (std::set<int>::iterator it = someSet.begin(); it != someSet.end(); it++)
```

In this case, because `it` is a complex type `it++` maybe have side effects because of the `temp` creation. Of course if you're lucky the compiler will try to throw away code that's not needed but if iterator's constructor or destructor do anything then `it++` is going to show those effects when it creates `temp`.

The short of what I'm trying to say is **Write What You Mean**. If you mean **increment ptr** then write `++ptr` not `ptr++`. If you mean `temp = ptr, ptr += 1, temp` then write `ptr++`

Share Edit Follow Flag — edited Feb 26, 2019 at 5:47 verbose 7,569 ●1 ●24 ●39 — answered Aug 28, 2013 at 13:08 gman 91.7k ●29 ●230 ●345

0

```
*ptr++    // 1
```

This is the same as:

```
    tmp = *ptr;
    ptr++;
```

So the value of the object pointed to by `ptr` is retrieved, then `ptr` is incremented.

```
*++ptr    // 2
```

This is the same as:

```
    ++ptr;
    tmp = *ptr;
```

So the pointer `ptr` is incremented, then the object pointed to by `ptr` is read.

```
++*ptr    // 3
```

This is the same as:

```
    ++(*ptr);
```

So the object pointed to by `ptr` is incremented; `ptr` itself is unchanged.

Share Edit Follow Flag — answered Sep 4, 2013 at 0:05 David R Tribble 11.3k ●4 ●41 ●50

0

Pointer Expressions : *ptr++, *++ptr and ++*ptr :

Note : pointers must initialized and must have valid address. Because in RAM apart from our program(a.out) there are lot more program running simultaneously i.e if you are try to access some memory which was not reserved for you OS will through Segmentation fault.

Before explaining this lets consider simple example ?

```
#include<stdio.h>
int main()
{
        int num = 300;
        int *ptr;//uninitialized pointer.. must be initialized
        ptr = &num;
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
        *ptr = *ptr + 1;//*ptr means value/data on the address.. so here value gets
incremented
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
        /** observe here that "num" got changed but manually we didn't change, it got
modified by pointer **/
        ptr = ptr + 1;//ptr means address.. so here address got incremented
        /**     char pointer gets incremented by 1 bytes
          Integer pointer gets incremented by 4 bytes
         **/
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
}
```

analyze the output of above code, I hope you got the output of above code. One thing is clear from above code is that pointer name (**ptr**) means we are talking about **address** and ***ptr** means we are talking abbout **value**/data.

CASE 1 : ***ptr++ , *++ptr, *(ptr++) and *(++ptr) :**

above mentioned all 4 syntax are similar, in all `address gets incremented` but how address gets incremented that's different.

Note : for solving any expression find out how many operators are there in expression, then find out **priorities** of operator. I multiple operators having same priority then check order of evolution or **associativity** that may right(R) to left(L) ot left to right.

***ptr++** : Here 2 operators are there namely de-reference(*) and ++(increment). Both are having same priority then check the associativity which is R to L. So starts solving from Right to Left, whatever operators is coming first.

***ptr++** : first ++ came while solving from R to L, so address gets incremented but its post increment.

***++ptr** : Same as first one here also address gets incremented but its pre increment.

***(ptr++)** : Here there are 3 operators, among them grouping () having highest priority, So first ptr++ solved i.e address gets incremented but post.

***(++ptr)** : Same as above case here also address gets incremented but pre increment.

CASE 2 : **++*ptr, ++(*ptr), (*ptr)++ :**

above mentioned all 4 syntax are similar, in **all value/data gets incremented** but how value gets changed that's different.

++*ptr : first * came while solving from R to L, so value gets changed but its pre increment.

++(*ptr) : Same as above case, value gets modified.

(*ptr)++ : Here there are 3 operators, among them grouping () having highest priority, Inside () *ptr is there , So first *ptr is solved i.e value gets incremented but post.

Note : ++*ptr and *ptr = *ptr + 1 both are same, in both case value gets changed. ++*ptr : only 1 instruction (INC) is used, directly value gets changed in single shot. *ptr = *ptr + 1 : here first value gets incremented(INC) and then assigned(MOV).

To understand all above different syntax of increment on pointer lets consider simple code :

```
#include<stdio.h>
int main()
{
        int num = 300;
        int *ptr;
        ptr = &num;
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
        *ptr++;//address changed(post increment), value remains un-changed
//      *++ptr;//address changed(post increment), value remains un-changed
//      *(ptr)++;//address changed(post increment), value remains un-changed
//      *(++ptr);//address changed(post increment), value remains un-changed

//      ++*ptr;//value changed(pre increment), address remains un-changed
//      (*ptr)++;//value changed(pre increment), address remains un-changed
//      ++(*ptr);//value changed(post increment), address remains un-changed

        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
}
```

In above code, try to comment/un-comment comments and analyze outputs.

Pointers as Constant : there are no of ways by which you can make pointers as constant, few I am mentioning here.

1)**const int *p OR int const *p** : Here `value` is **constant**, **address is not constant** i.e where p is pointing ? Some address ? On that address what is the value ? Some value right ? That value is constant, you can't modify that value but where pointer is pointing ? Some address right ? It can point to other address also.

To understand this lets consider below code :

```
#include<stdio.h>
int main()
{
        int num = 300;
        const int *ptr;//constant value, address is modifible
        ptr = &num;
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
        *ptr++;//
//      *++ptr;//possible bcz you are trying to change address which is possible
//      *(ptr)++;//possible
//      *(++ptr);//possible

//      ++*ptr;//not possible bcz you trying to change value which is not allowed
//      (*ptr)++;//not possible
//      ++(*ptr);//not possible

        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
}
```

Try to analyze the output of above code

2)**int const *p** : it's called '`**constant pointe**r`' i.e `address is constant but value is not constant`. Here you are not allowed to change the address but you can modify the value.

Note : constant pointer(above case) must initialize while declariung itself.

To understand this lets check simple code.

```
#include<stdio.h>
int main()
{
        int x = 300;
        int* const p;
        p = &x;
        printf("x = %d p =%p and *p = %d\n",num,p,*p);
}
```

In above code, if you observe that there is no ++*p or *p++ So you may thought this is simple case because we are not changing address or value but it will produce error. Why ? Reason I mention in comments.

```
#include<stdio.h>
int main()
{
        int x = 300;
        /** constant pointer must initialize while decaring itself **/
        int* const p;//constant pointer i.e its pointing to some address(here its
pointing to garbage), it should point to same address(i.e garbage ad
dress only
        p = &x;// but here what we are doing ? we are changing address. we are making p
to point to address of x instead of garbage address.
        printf("x = %d p =%p and *p = %d\n",num,p,*p);
}
```

So whats the Solution of this problem ?

```
     int* const p = &x;
```

for more about this case lets consider below example.

```
#include<stdio.h>
int main()
{
        int num = 300;
        int *const ptr = &num;//constant value, address is modifible
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
        *ptr++;//not possible
//      *++ptr;//not possible bcz you are trying to change address which is not
possible
//      *(ptr)++;//not possible
//      *(++ptr);//not possible

//      ++*ptr;// possible bcz you trying to change value which is allowed
//      (*ptr)++;// possible
//      ++(*ptr);// possible
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
}
```

3)**const int* const p** : Here **both address and value are constant**.

To understand this lets check below code

```
#include<stdio.h>
int main()
{
        int num = 300;
        const int* const ptr = &num;//constant value,constant address
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
        *ptr++;//not possible
        ++*ptr;//not possible
        printf(" num = %d ptr = %p and data on ptr : %d \n",num,ptr,*ptr);
}
```

Share Edit Follow Flag — edited Mar 4, 2018 at 13:24 marc_s 703k ●168 ●1301 ●1424 — answered Nov 12, 2017 at 19:17 Achal 11.6k ●2 ●15 ●34

0

- Postfix `++` has higher precedence than unary `*`.
- Prefix ++ and unary `*` have the same precedence, expressions with both have right-to-left *operator associativity*, meaning the right one(s) binds to the operand before the left one(s).

Therefore:

- `*ptr++` Increment the pointer by 1 item then de-reference the memory location it had before incrementing.
- `*++ptr` Increment the pointer by 1 item then de-reference the memory location where it now points.
- `++*ptr` De-reference the memory location then increment the *contents* (value) there by 1.

Share Edit Follow Flag — answered Jan 19 at 13:40 Lundin 174k ●38 ●234 ●367

-1

```
const char *p = "Hello";

*p means "Hello"
          ^
          |
          p

*p++ means "Hello"
             ^
             |
             p

*++p means "Hello"
            ^
            |     (WHILE THE STATEMENT IS EXECUTED)
            p

*++p means "Hello"
             ^
             |     (AFTER THE STATEMENT IS EXECUTED)
             p
```

`++*p` means that you are trying to increment the ASCII value of `*p` which

```
   is "Hello"
       ^
       |
       p
```

you cannot increment the value 'cause it's a constant so you would get an error

as for your while loop the loop runs until `*p++` reaches the end of the string where there is a `'\0'` (NULL) character.

Now since `*p++` skips the first character you would only get your output starting from the second character.

The following code will not output anything because while loop has `'\0'`

```
const char *p = "Hello";
    while('\0')
         printf("%c",*p);
```

The following code will give you the same output as the next code i.e ello .

```
const char *p = "Hello";
    while(*++p)
         printf("%c",*p);
```

…………………………………

```
const char *p = "Hello";
    while(*p++)
         printf("%c",*p);
```

Share Edit Follow Flag — answered Nov 11, 2017 at 17:23 iammosespaulr 9 ●1 ●4

-1

postfix and prefix has higher precedence than dereference so

*ptr++ here post increment ptr and then pointing to new value of ptr

*++ptr here Pre Increment fist then pointing to new value of ptr

++*ptr here first get the value of ptr pointing to and increment that vlaue

Share Edit Follow Flag — answered Feb 10, 2016 at 5:36 Kiran Padwal 115 ●1 ●3

1 This is incorrect. Postfix has higher precedence, but prefix has the same precedence as dereference. – verbose Feb 26, 2019 at 5:51