

Forum:[Hack Forums](#)

Thread: [Pointer Arithmetic](#)

Consulting Detective - October 4, 2017, 4:07 am

Hi guys, I'm currently practicing my pointer arithmetic due to I have noticed that I am not utilizing them much as I should.

First one that I did is to print out each character of a string using simple address iteration. What you guys think?

PHP Code

```
<?php
#include<stdio.h>
#include<string.h>

int main(int argc, char *argv[]){
    char *s = "Hello World";
    while(s <= s+(strlen(s)-1)){
        printf("%c", *s);
        s++;
    }

    return 0;
}
```

Can any of you give me some good exercises or problems to solve to practice?
Thanks.

Dostoevsky - October 4, 2017, 4:26 am

You can just learn by using pointer arithmetic syntax as opposed to using the `[]` array syntax sugar.

Firstly, you have to understand that arrays start at 0 to satisfy the basic formula:

Address of `T[x]` in `T[]`:

Code `T[x] = &T[0] + (x * sizeof(T))` So, the base address (starting address) of the array plus the index multiplied by the size of the array type in bytes. Thus, if you want to access the first element (which is at the base address - $(x = 0)$), the right hand side of the multiplication will cancel to 0 nicely.

You just have to understand that the underlying pointer types don't really matter until you try to dereference them (as the type information about the size and alignment of types is all the compiler cares about). This is actually why `sizeof(char)` is defined to be 1 by the standard despite the fact GCC actually allows you to do pointer arithmetic with `void*` (other compilers like clang simply support this as a GCC extension/plugin).

So, to understand what I mean:

PHP Code

```
<?php
puts("memelord" + 2);
```

will print melord since string literals will yield `const char*` and the `sizeof(char) == 1`, so you're adding (2 1) to that string literal's address.

Similarly:

PHP Code

```
<?php
```

```
int array[] = { 2, 3, 4 };
```

```
printf("%d\n", *(array + 2));
```

Again, the compiler knows that the underlying type is an int (sizeof(int) == 4) so it's abstracted this from you and lets you just write *(array + 2) rather than array + (2 * sizeof(int)) which would be the actual memory offset. So this code prints array[2], which as you know, is 4.

Consulting Detective - October 4, 2017, 5:48 am

(Oct 4, 2017 04:26 AM)Dostoevsky Wrote: [\[CITE\]](#) You can just learn by using pointer arithmetic syntax as opposed to using the [] array syntax sugar.

Firstly, you have to understand that arrays start at 0 to satisfy the basic formula:

Address of T[x] in T[]:

Code T[x] = &T[0] + (x * sizeof(T)) So, the base address (starting address) of the array plus the index multiplied by the size of the array type in bytes. Thus, if you want to access the first element (which is at the base address - (x = 0)), the right hand side of the multiplication will cancel to 0 nicely.

You just have to understand that the underlying pointer types don't really matter until you try to dereference them (as the type information about the size and alignment of types is all the compiler cares about). This is actually why sizeof(char) is defined to be 1 by the standard despite the fact GCC actually allows you to do pointer arithmetic with void* (other compilers like clang simply support this as a GCC extension/plugin).

So, to understand what I mean:

PHP Code

```
<?php
```

```
puts("memelord" + 2);
```

will print melord since string literals will yield const char* and the sizeof(char) == 1, so you're adding (2 - 1) to that string literal's address.

Similarly:

PHP Code

```
<?php
```

```
int array[] = { 2, 3, 4 };
```

```
printf("%d\n", *(array + 2));
```

Again, the compiler knows that the underlying type is an int (sizeof(int) == 4) so it's abstracted this from you and lets you just write *(array + 2) rather than array + (2 * sizeof(int)) which would be the actual memory offset. So this code prints array[2], which as you know, is 4.

Pretty sure I got most of that. It's just that I don't utilize them very often.

But arithmetic on a void pointer? Only way that I can think of to make it work is through typecasting.

0xDEAD10CC - October 4, 2017, 10:17 am

The fact that GCC allows pointer arithmetic on void* is just stupid even if it is an extension. It's one of the reasons why I don't like GCC, they do provide a lot of extensions that can lead to undefined behavior. Some are valid in C and undefined behavior in C++, but most of them are non-standard to the languages, and a sizeof(void) doesn't really make a whole lot of sense to begin with.

Btw, you could have just done:

```
Code const char *p = "Hello World";
while(*p) putchar(*p++);
```

Knowing what `strlen()` actually does, and also, NEVER declare string literals as `char*` because they are pointers to read-only memory and the compiler can't warn you if you don't make it `const` like I did above. You'll get a runtime error if you modify that string, but if you declared it as `char[]` it would be a pointer to a copy of that data which you CAN modify.

As for what Dostoevsky points out.

`a[10]` is the same as `*(a + 10)`
`a[0]` is the same as `*(a + 0)` which is the same as `*a`

It's syntactic sugar, which is why you can also do this:
`0[a]`, because it is the same as `*(0 + a)` but it's not as common to write that way so people don't do it.

The type of the pointer with the mathematical expression operating on the pointer address is what determines the byte offset for each index, and this is dealt with for you.

Consulting Detective - October 4, 2017, 1:31 pm

(Oct 4, 2017 10:17 AM)0xDEAD10CC Wrote: [\[CITE\]](#) Code const char *p = "Hello World";
while(*p) putchar(*p++);

Knowing what `strlen()` actually does, and also, NEVER declare string literals as `char*` because they are pointers to read-only memory and the compiler can't warn you if you don't make it `const` like I did above. You'll get a runtime error if you modify that string, but if you declared it as `char[]` it would be a pointer to a copy of that data which you CAN modify.

As for what Dostoevsky points out.

`a[10]` is the same as `(a + 10)`
`a[0]` is the same as `*(a + 0)` which is the same as `*a`

It's syntactic sugar, which is why you can also do this:
`0[a]`, because it is the same as `*(0 + a)` but it's not as common to write that way so people don't do it.

The type of the pointer with the mathematical expression operating on the pointer address is what determines the byte offset for each index, and this is dealt with for you.

Let's say that it's guaranteed that the program will not modify the string literal declared as `char*`, is there gonna be difference between that and a string declared as `const char*` performance wise?

Also, would it be ok if I use `while(*p != 0)` or `while(*p != '\0')` instead of `while(*p)`?

0xDEAD10CC - October 4, 2017, 4:12 pm

(Oct 4, 2017 01:31 PM)Consulting Detective Wrote: [\[CITE\]](#) Let's say that it's guaranteed that the program will not modify the string literal declared as `char*`, is there gonna be difference between that and a string declared as `const char*` performance wise?

Also, would it be ok if I use `while(*p != 0)` or `while(*p != '\0')` instead of `while(*p)`?

1) From the standpoint of optimizations being disabled, no, it's only used to prevent you from modifying something that isn't supposed to change. If you take into account compiler optimizations, then yes in the general sense of constants. There's still no reason to declare a string literal `char` as it will save you tons of future problems if you helped yourself avoid successfully compiling such code

that would modify a char at compile time by just adding the const qualifier.

2) It's pretty much the same thing. `0 = '\0'`, and `0 != 0` is the same as `1` or the boolean representation -- `TRUE`.