Matheus Laidler Vidal Cunha
matheuslaidler@ufrj.br
matheuslaidler@gmail.com

Análise de programa com ponteiro e relações com IP.

É muito comum em diversas atividades contemporâneas a necessidade de referências para os mais variados tipos de coisa. Até mesmo quando falamos de línguas, podemos usar referências da linguagem de base para explicar o sentido por trás da palavra da nossa língua. Um exemplo disso é quando explicamos uma palavra do latim para explicar o sentido por trás da palavra do nosso idioma. Ou até mesmo quando separamos por partes algumas palavras para percebermos o sentido referente por trás da mesma. Portanto, as referências sempre são importantes em diversas áreas do conhecimento, e na programação não será diferente.

Em C e em outras linguagens de programação, usar um ponteiro para dar referência onde algo se encontra dentro da memória acaba sendo bastante útil e interessante. Será isso que analisaremos neste texto.

Para começar a falar sobre ponteiros vamos primeiro dar um resumo e depois vamos explicar melhor como funciona junto com códigos e afins.

O resumo mais breve é dizer que o símbolo '*',além de ser usado para multiplicação, também é usado para ponteiros e referências.

Um ponteiro é uma variável que armazena um endereço de memória. Endereços são números inteiros, então um ponteiro armazena um número inteiro.
Ele aponta para algum lugar na memória, já que um ponteiro é uma variável que armazena um endereço de memória, então faz referência a esse endereço guardado também. O ponteiro que você quer usar vai apontar para algum lugar na memória e nesse lugar pode ter guardado nele qualquer coisa, como um caractere. Nesse caso basta declarar:

```
char* p;
```

Isso significa dizer para o compilador chamar de p um número que armazena um endereço de memória onde haverá um caractere lá.

Vamos entender melhor na prática ao analisar códigos:

Analisando programa:* x.c: *

```
#include <stdio.h>

int main() {
  char c; c = 'a';  /*Declarando C*/
   /*Agora que criamos e indicamos um valor na variável,
   o nome C nada mais representa do que uma posição na memória.
   O compilador vai armazenar uma quantidade de bytes para
   representar o caractere dessa variável.
   Nesse caso será 1 byte. Podemos usar o sizes para saber mais*/

  printf("c = %d = %c\n", c, c);  /*Mostra o valor de c e o inteiro que
                                    o representa*/
  printf("c is at %p\n", &c);  /*Mostre que posição na memória é essa -
          em número hexadecimal como endereço de memória*/
}
```

O operador '&' na última linha do printf é interessante;
Ele nada mais é do que um comando para nos mostrar o endereço em
memória onde uma variável está.
Daria no mesmo se escrevermos o código assim:

```
#define ENDERECO(x) &(x)  -> substitua isso: ENDERECO(x) por isto: &(x)
```

na última linha do printf -> printf("c is at %p\n", ENDERECO(x));

A resposta será dada com endereço de memória em base 16 (podemos ver que
começa com 0x) representando um inteiro, ou seja, as bases são como outras
formas de escrever um número.
Podemos descobrir que número é esse modificando o código assim:

```
printf("c is at %p = %d\n", ENDERECO(x), ENDERECO(x));
```

*errado pq foi dado 'int' esperando ponteiro, o problema aí é que o
número fica negativo.

-Para vermos de forma positiva usamos o unsigned, porém:

```
printf("c is at %p = %d\n", ENDERECO(x), (unsigned long)ENDERECO(x));
```

*errado pq precisaria ser 'long long' e não foi colocado os 'll's referentes no %d
-Ou seja, para uma saída long precisará de uma entrada decimal também em long,
tal qual se for long long, o decimal deverá ser escrito como long long também;
- Para unsigned long usar %ld e para unsigned long long usar o %lld

Porém como vimos que ele quer um não sinalizado, de acordo com o manual printf,
temos que usar nesse caso o %u, um conversor sem sinal.
Ou seja, sempre que usarmos unsigned é interessante usarmos o %u ao invés do %d.
O %d ou %i será lido como int, já o %u como unsigned int.

Como colocamos como long long, podemos indicar o ll antes do u.
Ficando:
printf("c is at %p = %llu\n", ENDERECO(x), (unsigned long long)ENDERECO(x));

saída:
c = 97 = 'a'
c is at 0xffffcc3f = 429495047

Agora está saindo corretamente e sem erros.

Nessa saída podemos ver que a base 16 0xffffcc3f significa 429495047 em base 10.
Se quiser rever o conversor, basta ir no site
https://www.binaryhexconverter.com/hex-to-decimal-converter
ou qualquer outro conversor 'hex to decimal' e colocar o que endereço hex do seu programa e comparar com a saída decimal que ele te deu = Mesmo resultado.

Percebe-se, então, que: 'uint' -> inteiro unsigned -> usar todos os inteiros para números positivos.

Sobre o sizes>

Podemos saber melhor como será o armazenamento pelo programa sizes:

```
$ ./sizes
 1 is the size of char
 1 is the size of unsigned char
 4 is the size of int
 4 is the size of unsigned int
 2 is the size of short int
 2 is the size of unsigned short int
 2 is the size of short
 2 is the size of unsigned short
 8 is the size of long int
 8 is the size of unsigned long int
 8 is the size of long
 8 is the size of unsigned long
 8 is the size of long long
 8 is the size of unsigned long long

 8 is the size of char*
 8 is the size of unsigned char*
 8 is the size of int*
 8 is the size of unsigned int*
 8 is the size of void*

100 is the size of array cs
400 is the size of array is
```

1 is the size of int8_t
1 is the size of uint8_t
2 is the size of int16_t
2 is the size of uint16_t
4 is the size of int32_t
4 is the size of uint32_t
8 is the size of int64_t
8 is the size of uint64_t
8 is the size of uintmax_t

- Podemos ver que um inteiro ocupa 4 bytes, a sua forma reduzida short int já ocupa metade: 2 bytes.

Se declararmos um inteiro e o colocasse como um número grande, podemos tentar ver o quanto pesa cada parte desse inteiro com o programa bytes-of-int.c;

Para x = 16909060
x[0] = 04
x[1] = 03
x[2] = 02
x[3] = 01

Se esse número fosse escrito em hexadecimal, seria : 0x01020304
a mesma coisa dos bytes de cada parte da lista que forma o inteiro.
É interessante vermos,também, que meu (e provavelmente o seu) computador escreve na memória de trás para frente, já que o primeiro byte foi o 04.
Esses computadores são chamados de LITTLE-ENDIAN - trata-se da ordem binário de um computador. Ou seja, o número 123 acaba sendo lido como 321;

hexdecimal (base 16)
04 03 02 01
00000100 00000011 00000010 00000001
binario (base 2)
a base 16 acaba sendo mais conveniente para falarmos de memória, não só os endereços como também o conteúdo, isso pq é mais simplificado.

como 1byte = 8 bits, como dito alguma redação passada, então para converter de hex para binário acaba sendo mais simples. Podemos ver que sempre possui 8 caracteres, sendo representação do número em binário preenchido de zeros à esquerda.

Bom,como o número acaba sendo lido de trás pra frente,o que é bastante conveniente, podemos verificar, então:

Primeiro pelas unidades, depois às dezenas e depois às centenas.

123

1*10^2 + 2*10^1 + 3*10^0 -> 1*100 + 2*10 + 3*1 -> 123

3*10^0 + 2*10^1 + 1*10^2 -> 'lido pelo pc' -> 123 (mais curto escrever dessa forma)

os computadores big endian são os que leem na mesma ordem da escrita,como a máquina Sparc da sunmicrosstems

Leitura de byte a byte -> observando programa bytes-of-int.c

Ponteiro são usados como *, e para ler byte a byte precisa usar que aponte para um byte, então:

char* p; p = &x; -> declara o ponteiro que aponta para um byte e que escreva nele o endereço de memória onde está o número cujo nome é x - representa posição na memória onde está.

(forma básica para ler byte a byte) -> o compilador dará erro; mas pode ser executando quando o erro se tornar apenas um aviso, ent basta usarmos o comando gcc -o bytes-of-int bytes-of-int.c que poderemos seguir.

Agora será compilado.

O aviso se baseia em ver que seu inteiro x está sendo usado para a verificação byte-byte,

Ou seja, está pegando um ponteiro que aponta 1 byte para apontar a um endereço de memória que tem 4 bytes. Entretanto, não é um engano, é exatamente o que queremos fazer.

Para que o compilador no modo restrito não bloqueie o uso, devemos então trocar a sintaxe.

Podemos então deixar a linha assim:

char *p; p = (char *) &x;

Quando colocamos o (char *) antes do &x estamos falando que queremos olhar a região de memória de x e verificar byte a byte (caractere a caractere). Assim você explica melhor pro compilador que é exatamente isso mesmo que você quer fazer, sem precisar de aviso.

Nos prints estamos falando para imprimir byte a byte usando hexadecimal e para imprimir em dois dígitos. Caso não explicitasse dois dígitos, ao invés de 04 sairia 4, por exemplo.

Quando tem o 0 você consegue lembrar que tem quatro bits = 0.

Ponteiros e Arrays são muito próximos então usar como um array nesse caso do print funcionará

da mesma maneira. O compilador já transcreve o p[0] como notação de ponteiro *(p + 0)

Ficando uma aritmética de ponteiros. Nessa notação podemos perceber que estamos falando pro programa que quando tem o * e um endereço de memória associado, queremos que o compilador vá até esta posição de memória e leia o valor que está lá.

(LOGO: '*' atribui valor de referência, ponteiros e multiplicação)

Podemos ver melhor no exemplo de 'address-of.c'

```
#include <stdio.h>

#define ADDRESS_OF *

int main() {
  char c; c = 'a'; /*Atribui caractere c e atribui o byte 97 lá*/
   char ADDRESS_OF p;    /*P é um endereço de memória que estamos apontando, e nele  quero encontrar um caractere (char)*/
 /*Podemos tirar o define address_of e deixar nessa linha apenas o char * p */
  p = &c;
 /*Escreva em P a posição da memória em que está o caractere c encontrado*/
  printf("c = '%c'\n", c); /*Imprimindo o caractere C*/
   printf("p = %p\n", p); /*Imprimindo o valor do ponteiro P - podíamos ter colocado %d (decimal) ou %x(hexa), mas usamods o %p que é o  modificador p de ponteiro, que já formata em hexadecimal o valor de um ponteiro.*/
  printf("No endereço referenciado por p, encontra-se o caractere '%c'\n", *p); /*No  endereço apontado/referenciado por p, encontramos que caractere tal. Usa-se a '*' com o p para dar a referência pedida */
  }
```

Testando os programas com IP's
Após as explicações sobre a base, podemos então seguir com mais alguns testes interessantes.

Supondo que um computador na Internet possua o endereço IP externo 65.21.52.178, que ao usarmos no programa 'antartida.c' podemos ter um resultado interessante. Mas primeiro, que número inteiro é esse? Esse será o IP do serviço, porém sem ordem inversa. Isto é, como meu computador é little-endian e, portanto, lerá de trás para frente, devemos colocar: 178.52.21.65 no array do programa;
E se colocarmos na ordem normal: 65.21.52.178
O resultado será: 2989757761.
Porque veio diferente? Na verdade, não foi nada diferente do previsto, se entrarmos nesse serviço pela web será percebido que este endereço será lido como 178.52.21.65(mas não retornará nenhum serviço, já que é só um exemplo errado).

Portanto, se quiséssemos acessar o endereço 65.21.52.178, então teríamos que ter colocado no programa de trás para frente para que o programa trabalhe na ordem correta.
Lembre-se que este endereço é um número inteiro, mas que está sendo lido byte a byte, por isso essa divisão em quatro partes.

saída do programa de ordem certa:
$ ./antartida
antartida.xyz has address http://1091908786
Colando o endereço, o navegador lerá 1091908786 como 65.21.52.178 que redireciona ao site antartida.xyz
http://65.21.52.178 -> http://antartida.xyz - serviço certo

Se verificarmos, o número dado é um decimal que é convertido para IP. O valor 1091908786 = 65.21.52.178;
Podemos verificar que esta é a mesma função feita pelo nosso browser e pela ferramenta: Decimal to IP Converter;
https://codebeautify.org/decimal-to-ip-converter
Essa conversão é feita basicamente e resumidamente seguindo essa fórmula: 16777216*x + 65536*y + 256*z + 1*j
x = primeiro byte; y = segundo; z = terceiro; j = quarto.
ficando: 16777216*65 + 65536*21 + 256*52 + 1*178

saída do programa de ordem errada:
$ ./antartida
antartida.xyz has address http://2989757761

Colando o número ele 'redireciona' a um site com ip http://178.52.21.65/ - serviço errado

Podemos perceber, então, que o programa antartida.c trabalha da seguinte maneira;
Ele armazena em uma lista cada byte do inteiro por partes e indica esse array como um ponteiro para um inteiro positivo p. Depois pede para ir no endereço de memória, pegar o valor e printá-lo.

```
int main() {
  unsigned char bs[] = {178, 52, 21, 65}; /* little-endian */
  /*criando array com caracteres de ordem invertida*/
   unsigned int *p; p = (unsigned int *) bs; /*veja esse array como um ponteiro para um inteiro positivo*/
    printf("antartida.xyz has address http://%u\n", *p); /*p -> vai no endereço de memória e pega o valor decimal */
}
```

Então, quando usamos *p, estamos dizendo ao compilador para pegar o número que está armazenado em p, que é um endereço, vá até ele e

ao chegar lá ler o que está escrito. E nesse caso, coloque este
resultado como unidade no local indicado no print.

Trocando o tipo do ponteiro p para 'unsigned long *' tivemos uma
mudança em sua saída.(*)

Primeiro vendo a saída de sizes podemos ver o tamanho dos tipos de ponteiro:
$ ./sizes | grep -i "of long"
8 is the size of long int
8 is the size of long
8 is the size of long long

$ ./sizes | grep -i "of int"
4 is the size of int
8 is the size of int*
1 is the size of int8_t
2 is the size of int16_t
4 is the size of int32_t
8 is the size of int64_t

- Agora alterando o código antártida com long:

```
#include <stdio.h>

int main() {
  unsigned char bs[] = {178, 52, 21, 65}; /* little-endian */
  unsigned long *p; p = (unsigned long *) bs;
  printf("antartida.xyz has address http://%lu\n", *p);
}
```

Temos, assim, uma nova saída:
antartida.xyz has address http://18097365458982614194
-> nesse novo endereço não temos nada.
E nem chega a ser lido/reconhecido.
Isso ocorre porque está como long

-Engraçado que a cada vez que o novo antartida é executado,
o resultado do endereço de saída muda>
Uma delas foi: http://2747369496625099954
E outra foi: http://6023332505746486450
... Mas todos eles sempre são convertidos no mesmo valor:
o IP 65.21.52.0, que não é o IP certo que colocamos,já
que no último byte analisado está como zero e não 178.
Ou seja, vemos que o long não é capaz de armazenar em
decimal todas as caracteres de um ip 4 bytes.
Se pegarmos o decimal 6023332505746486000 e mudarmos os
três últimos números como 999 ainda ficaria o mesmo
resultado, com o 0 no nosso último byte (pela nossa visão).

-O tipo long sempre ficará com 20 char em decimal-

Resumindo; Este programa pega uma lista de entrada, nela temos um endereço IP adicionada de trás para frente, pois nosso tipo de computador faz essa leitura de forma inversa. Usamos um ponteiro para podermos acessar as informações armazenadas no endereço da memória indicada e pegar as informações destes caracteres contidas no array apontado. Cada byte deste array apontado foi acessado e, para obtivemos as informações totais da lista lida, temos que pegar uma referência de onde está o p, pegarmos a informação contida nele e adicionarmos em nosso print com o tipo de saída que quisermos, sendo como inteiro, unidade ou decimal(usar %u - conversor sem sinal).Desse valor, o navegador converte para IP e consegue achar o serviço web relacionado ao IP que foi colocado dentro do código de forma inversa.

Lembrando>
O * é usado para declarar um ponteiro, para fazer tarefas de multiplicação e é usado como operador de referência