Conceitos fundamentais em C para entender o código do toupper em assembly.

Por: Felipe Ribeiro.

Como strings são feitas.

1. Começando pelo uso de char para armazenamento de caracteres, o char é capaz de apontar para um único caractere. O código a seguir demonstra a impressão de um caractere “A”, armazenado no char letra.

```c
#include <stdio.h>

int main(){

    char letra = 'A';

    printf("sua letra é: %c \n", letra);
}
```

```
[felipe@archlinux ~]$ ./char
sua letra é: A
```

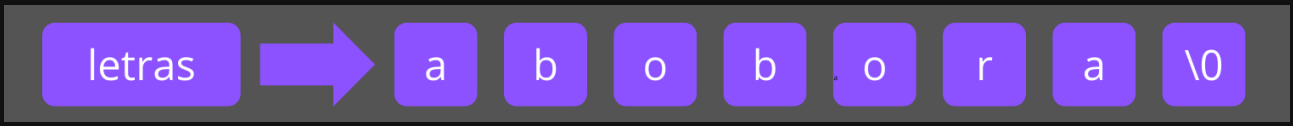
O conteúdo impresso é indicado pelo formato char no momento do printf com o símbolo %c.

1.1 Uso de char[] para indicar uma array, uma array é definida por uma lista de caracteres. O código a seguir exemplifica a impressão da palavra abobora, na array letras[].

```c
#include <stdio.h>

int main(){

    char letras[] = {'a', 'b', 'o', 'b', 'o', 'r', 'a'};

    printf("sua palavra é: %s \n", letras);
}
```

```
[felipe@archlinux ~]$ ./char
sua palavra é: abobora
[felipe@archlinux ~]$
```

O conteúdo impresso, agora é indicado pelo formato %s, pela array letras[] apontando o printf a printar todos os caracteres da array até que chegue no \0.



A figura mostra que letras[] aponta para cada caractere, estando 'a' na posição [0] b na posição [1] … até que chegue no ultimo termo que tem conteúdo \0 declarando o final de uma array.

```c
#include <stdio.h>

int main(){

    char letras[] = {'a', 'b', 'o', 'b', 'o', 'r', 'a'};

    printf("sua letra [0] é: %c \n", letras[0]);
    printf("sua letra [1] é: %c \n", letras[1]);
    printf("sua letra [2] é: %c \n", letras[2]);
    printf("sua letra [3] é: %c \n", letras[3]);
    printf("sua letra [4] é: %c \n", letras[4]);
    printf("sua letra [5] é: %c \n", letras[5]);
    printf("sua letra [6] é: %c \n", letras[6]);
    printf("sua letra [7] {/n} é: %c \n", letras[7]);
    printf("sua letra [8], deve printar lixo, é: %c \n", letras[8])
}
```

```
[felipe@archlinux ~]$ ./char
sua letra [0] é: a
sua letra [1] é: b
sua letra [2] é: o
sua letra [3] é: b
sua letra [4] é: o
sua letra [5] é: r
sua letra [6] é: a
sua letra [7] {/n} é:
sua letra [8], deve printar lixo, é: 
[felipe@archlinux ~]$
```

A imagem acima demonstra o print dos 8 elementos da array letras. O 8nesimo elemento não printa nada pois não é um caractere é o símbolo reservado: {\0} que indica o final de uma array.

A prova de que existe um \0 é demonstrada no próximo elemento(9nesimo), ao tentar printa-lo, o programa exibe lixo, provando que não existe nada propositalmente especificado naquele local.

2. Análise do código e início aos loops em C.

```c
#include <stdio.h>

int main(){

    char letras[] = {'a', 'b', 'o', 'b', 'o', 'r', 'a'};

    printf("sua letra [0] é: %c \n", letras[0]);
    printf("sua letra [1] é: %c \n", letras[1]);
    printf("sua letra [2] é: %c \n", letras[2]);
    printf("sua letra [3] é: %c \n", letras[3]);
    printf("sua letra [4] é: %c \n", letras[4]);
    printf("sua letra [5] é: %c \n", letras[5]);
    printf("sua letra [6] é: %c \n", letras[6]);
    printf("sua letra [7] {/n} é: %c \n", letras[7]);
    printf("sua letra [8], deve printar lixo, é: %c \n", letras[8])
}
```

O código acima demonstrado, exibe tremendo mal manuseio e otimização do código. A seguir será apresentado o conceito de loops utilizando primeiramente o for.

```c
#include <stdio.h>

int main(){

    char letras[] = {'a', 'b', 'o', 'b', 'o', 'r', 'a'};

    int tamanho = 9;

    for(int n = 0; n < tamanho; n = n + 1){

        printf("elemento [%d] conteúdo: {%c}\n", n, letras[n]);

        }
}
```

Para meios didáticos é interessante ler o código for como:

“para um numero n que eu mesmo defini ser igual a zero, enquanto esse numero n for menor que o tamanho(que foi outro numero que criei (= 9 neste caso)), incremente 1 no meu numero n”

Demonstrando o ciclo do loop;

primeiro loop n = 0:

```
for(int n = 0; 0 < 9; n = 0 + 1) {
printf(“elemento [0], conteúdo: [‘a’]”, n, letras[0]);
}
```

segundo loop n = 1:

```
//n é igual a 1 agora, por conta do (n = 0 + 1).

for(int n = 0; 1 < 9; n = 1 + 1) {
printf(“elemento [1], conteúdo: [‘b’]”, n, letras[1]);
}
```

```
[felipe@archlinux ~]$ ./char
elemento [0] conteúdo: {a}
elemento [1] conteúdo: {b}
elemento [2] conteúdo: {o}
elemento [3] conteúdo: {b}
elemento [4] conteúdo: {o}
elemento [5] conteúdo: {r}
elemento [6] conteúdo: {a}
elemento [7] conteúdo: {}
elemento [8] conteúdo: {}
[felipe@archlinux ~]$ ./char
elemento [0] conteúdo: {a}
elemento [1] conteúdo: {b}
elemento [2] conteúdo: {o}
elemento [3] conteúdo: {b}
elemento [4] conteúdo: {o}
elemento [5] conteúdo: {r}
elemento [6] conteúdo: {a}
elemento [7] conteúdo: {}
elemento [8] conteúdo: {N}
[felipe@archlinux ~]$
```

Execução do código.
É possivel ver a “aleatoriedade” do 9nesimo termo letras[8], ao na primeira execução aparecer caractere estranho “?” e na segunda execução aparecer “N”.

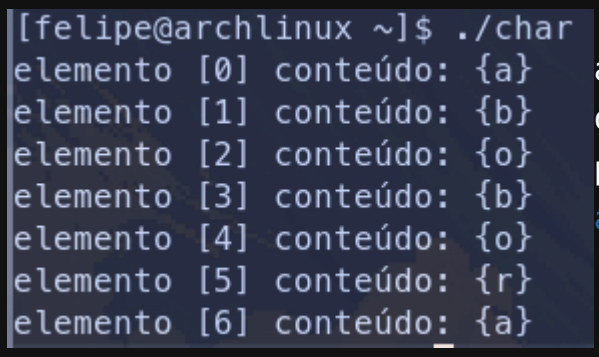
É possível ver também o 8nesimo termo fixo como um espaço vazio, representando o `\0` que indica final da `array`.

Mesmo loop, utilizando `while` para realizá-lo.

```c
#include <stdio.h>

int main(){

    char letras[] = {'a', 'b', 'o', 'b', 'o', 'r', 'a'};

    int n = 0;

    while (letras[n] != '\0') {

        printf("elemento [%d] conteúdo: {%c}\n", n, letras[n]);
        n++; // n++ é o mesmo que n = n + 1.

        }
}
```

Para fins didáticos é interessante ler o código como:

“enquanto o elemento letras[n] da array for diferente(!=) de ‘\0’, incremente 1 no meu numero n e continue caçando esse \0 que indica o fim da minha array”.

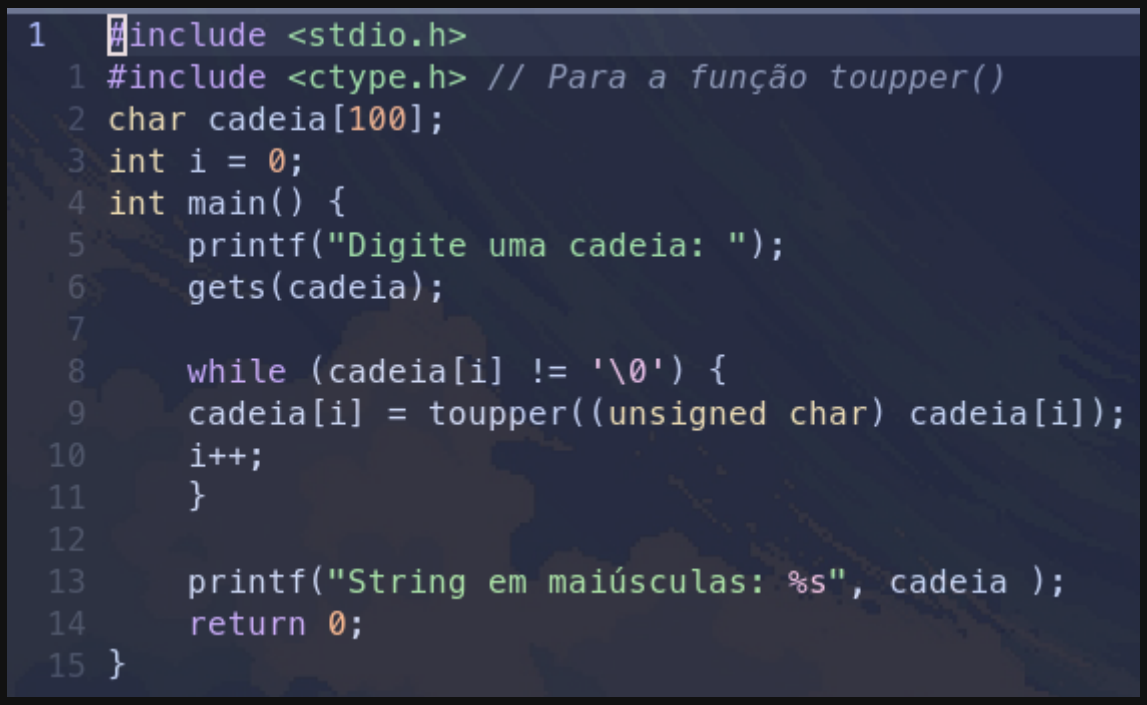


O output mostra apenas os elementos anteriores ao \0 pois no momento que o loop encontra o \0 ele para de procurar entre os caracteres da array.

//O gets() funciona como um scanf que armazenará o conteúdo digitado dentro de um buffer pré definido.

Leitura do código toupper em C.

Por: Felipe Ribeiro.



```c
#include <stdio.h>
#include <ctype.h> // Para a função toupper()
char cadeia[100];
int i = 0;
int main() {
    printf("Digite uma cadeia: ");
    gets(cadeia);

    while (cadeia[i] != '\0') {
    cadeia[i] = toupper((unsigned char) cadeia[i]);
    i++;
    }

    printf("String em maiúsculas: %s", cadeia );
    return 0;
}
```

Agora vamos dar início a leitura e aplicação dos conceitos anteriormente explicados.

Para fins didáticos é interessante ler o código como:

“para um char cadeia com 100 bytes reservados para caracteres, printe para o usuário digitar o que ele quer, após isso, armazene essa informação dentro do char cadeia com o gets, após isso, faça um loop while(revizar fundamento while) para cada caractere até encontrar o \0, a cada loop ele vai pegar o caractere [i] e transformar esse caractere na versão maiúscula dele e armazenar no meesmo local (cadeia[i]), incremente i(i++) e continue o loop, após isso printe o char cadeia todo loopado(maiúsculo)”.

Exemplo prático cogitando que o usuário digitou a palavra abobora.

Primeiro loop while i = 0:

```
while (cadeia[0] != '\0') {
    cadeia[0] = toupper((unsigned char) cadeia[0]);
    i++;
     }// ele vai pegar o termo na posição 0 neste caso: ‘a’ e armazenar ‘A’, assim o char cadeia[100] ficaria algo como: {‘A’,’b’,’o’,’b’,’o’,’r’,’a’,’’,’’,’’,’’...[100]};
```

Segundo loop while i = 1:

```
while (cadeia[1] != '\0') {
    cadeia[1] = toupper((unsigned char) cadeia[1]);
    i++;
     }// ele vai pegar o termo na posição 1 neste caso: ‘b’ e armazenar ‘B’, assim o char cadeia[100] ficaria algo como: {‘A’,’B’,’o’,’b’,’o’,’r’,’a’,’’,’’,’’,’’...[100]};
```

Terceiro loop while i = 2:

```
while (cadeia[2] != '\0') {
    cadeia[2] = toupper((unsigned char) cadeia[2]);
    i++;
     }// ele vai pegar o termo na posição 2 neste caso: ‘o’ e armazenar ‘O’, assim o char cadeia[100] ficaria algo como: {‘A’,’B’,’O’,’b’,’o’,’r’,’a’,’’,’’,’’,’’...[100]}
```

e assim sucessivamente.

Tradução do código para Assembly na prática.

Início do código, ele exporta algumas funções externas:

```
extern stdin
extern printf
extern scanf
extern toupper
extern gets
```

@curiosidade: não é necessário dar extern antes do section .text, podendo ser utilizadas no própio section.text

```
4 section .data
3 i dd 0
2 str0 db "Digite uma string: ",0
1 str1 db "String em maiúsculas: %s",0
```

Na seção .data do código, definimos i = 0 (que iremos utilizar no loop while), reservando 4 bytes para armazenar o número com o formato dd(double word) ficando:
i dd 0 //lê-se como: defino agora que i tem espaço de uma dd(double word) com valor de 0.

também definimos a str0 db “Digite uma string: ”,0 e str1 db “String em maiúsculas: %s”,0 que serão as mensagens que enviaremos ao usuário futuramente no código, o 0 após a definição da str0 e str1 são crúciais, eles atuam igualmente como um \0, indicando o final da string(cadeia[]).

@comentário: caso fossemos utilizar o scanf para pegar informações do usuário, deveriamos declarar um fmt db “%s”,0 para utilizar futuramente ao tentar capturar informação do usuário e armazenar no char[].

```
1 section .bss
2 cadeia resb 100
```

Definimos o char cadeia[] reservando 100 bytes de memória para armazenamento da informação do usuário em C => char cadeia[100].

cadeia resb 100 //lê-se como: reserve 100 bytes para a variável char cadeia;

Criando uma representação gráfica do meu char cadeia resb 100:

em que tenho 100 bytes (cada caixinha = 1 byte) podendo armazenar um caractere ascii em cada caixinha.

Na seção text do código, definimos uma global main para servir de ponto de link para as variáveis externas préviamente importadas.

O registrador rbp é um registrador de propósito específico que guarda o endereço da base da pilha do programa em execução na memória e o registrador rsp guarda o endereço do topo desta pilha. É comum dizermos que estes registradores apontam para a base e para o topo da pilha respectivamente.

```
22 section .text
21 global main
20 main:  push rbp
19        mov rbp,rsp
18        mov rdi, str0
17        mov rax,0
16        call printf
15
14        mov rdi, cadeia
13        call gets
12
```

criamos o princípio com o main:
push rbp // indica que queremos salvar o valor da base da pilha na memória.

mov rbp,rsp // fixa o ponteiro da base (rbp) no topo da pilha do início da função.

```
22 section .text
21 global main
20 main:  push rbp
19        mov rbp,rsp
18        mov rdi, str0
17        mov rax,0
16        call printf
15
14        mov rdi, cadeia
13        call gets
12
```

mov rdi, str0 //move o registrador rdi (64 bits) para o conteúdo de str0.

mov rax,0 //zera o rax, isso é necessário pois dependendo do que é colocado na string desejada, ao utilizar uma variável externa como o printf ele pode printar lixo.

call printf // printa o que rdi registrou “Digite uma string: ”.

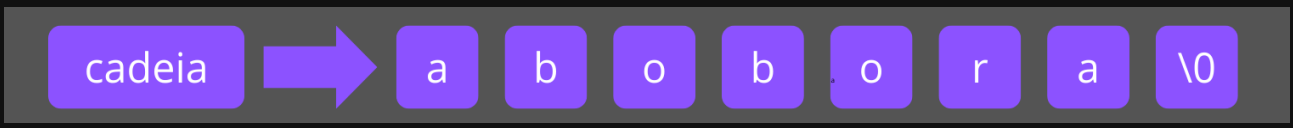
Agora utilizaremos a função externa gets, para colher informação do usuário.

Movemos o registrador rdi para apontar para o char cadeia[].

mov rdi, cadeia

assim, o rdi está preparado para receber a informação do usuário e armazená-la no nosso char cadeia[].

Supondo que o usuário digite abobora o char cadeia[] ficaria algo como:



Primeira parte do while:

```
6  inicio_while:
1          mov rbx,cadeia
2          xor rax,rax
3          mov eax,[i]
4          add rbx,rax
5          mov dh, 0
6          mov dl, [rbx]
7          cmp dl,0
8          je  fim_while
9          xor rdi,rdi
10         mov di,dx
11         call toupper
```

mov rbx, cadeia //move o registrador rbx para apontar ao endereço do início da -> cadeia[].

xor rax, rax //zera o rax

mov eax, [i] // move o registrador eax para registrar o valor([]) de i, neste caso definimos ser 0 em: i dd 0

add rbx,rax // adiciona o valor de rax, neste caso = 0 no ponteiro, ou seja, faz o ponteiro rbx que está apontando para a cadeia que contém {'a','b','o','b','o','r','a'} apontar para cadeia[0] = 'a'.

Para entendermos como funciona o código seguinte é importante estar em mente que:

```
RDX  (64 bits) = [ ........ EDX ........ ]
EDX  (32 bits) = [ ...... DX ...... ]
DX   (16 bits) = [ DH (8) | DL (8) ]
```

DX é um registrador de 16 bits em que DH (D high) contem a parte alta dos 8 bits de DX e DL(D low) contém a parte baixa dos 8 bits.

```
1        mov dh, 0
2        mov dl, [rbx]
3        cmp dl,0
4        je  fim_while
5        xor rdi,rdi
6        mov di,dx
7        call toupper
```

mov dh, 0 //zera a parte alta do DX @importante para usarmos DX sem carregar lixo no futuro.

mov dl, [rbx] // carrega o DL com o byte de valor da letra armazenada em rbx[0] (1 byte = 8 bits) carregando todos os 8 bits para a parte low de DX (DL) com o binário que define o caractere (‘a’ nesse caso = 01100001).

Representação do DX após isso:

```
DX  (16 bits) = [ DH (00000000) | DL (01100001) ]
```

cmp dl,0 // agora ele compara DL com 0 para ver se chegou no \0 (neste caso ainda não, pois está carregado com ‘a’ = 01100001).

je fim_while // je “jump if equal”(pule se for igual) para fim_while (neste caso não foi igual a 0 então ele não pulará, continuando o loop normalmente).

```
RDI  (64 bits) = [.................EDI.................]
      EDI  (32 bits) = [............DI............]
       DI   (16 bits) = [     DIL (8 bits)     ]
```

xor rdi,rdi // zera o rdi para usarmos o di.

mov di,dx // move a informação de DX = [0000000001100001] para o DI.

Agora DI é = [0000000001100001].

call toupper // armazena o ‘A’ no lugar de ‘a’ @é interessante que quando essa função é chamada, o resultado dela é armazenado em AL que é a parte baixa de RAX, pois funções externas utilizam RAX para colocar as informações e como a informação tem 8 bits, vai para AL.

```
1          mov rbx,cadeia
2          xor rcx,rcx
3          mov ecx,[i]
4          add rbx,rcx
5          mov [rbx],al
6          mov eax,[i]
7          inc eax
8          mov [i], eax
9          jmp inicio_while
```

mov rbx,cadeia // aponta para o endereço de início da cadeia (nesse caso [0]).

```
 RCX  (64 bits) = [.....................ECX.....................]
  ECX  (32 bits) = [.................CX.........................]
       CX   (16 bits) = [      CH      |      CL      ]
            CL   (8 bits)  = [ CL (byte baixo do CX) ]
             CH   (8 bits)  = [ CH (byte alto do CX) ]
```

xor rcx,rcx //zera o rcx para usarmos esse registrador, de forma simplificada (rcx é um registrador que conta o estágio do loop) neste caso 0.

mov ecx, [i] // utiliza o ecx para armazenar o conteúdo de i (neste estágio do loop = 0).

add rbx,rcx // adiciona o valor do ponteiro rcx em rbx (nesse caso [0], focando em rbx[0],

mov [rbx],al // pega o valor de AL = 'A', e coloca dentro do local apontado (rbx[0]) que é cadeia[0].

A próxima parte apresentada utiliza o eax para armazenar o conteudo de i, incrementa o valor de eax (que agora vale i+1) e depois coloca esse valor de volta no i.

mov eax,[i] // carrega o eax com valor de i(vale 0).

inc eax //faz operação de adicionar 1 a eax ficando: eax = eax+1.

mov [i],eax // coloca o conteudo de eax em i, agora i tem valor 1

jmp inicio_while // pula incondicionalmente para o início do while, agora é importante perceber que como i tem valor 1, ele vai fazer tudo exatamente igual, agora utilizando a posição cadeia[1] ('b'), substituindo por 'B' e incrementando i, e assim sucessivamente até que em algum momento o je fim_while vai aceitar a condição de cmp dl,0 e terminará o programa.

Por fim:

```
fim_while:
        mov rdi, str1
        mov rsi, cadeia
        mov rax,0
        call printf

         leave
         mov rax,0
         ret
```

mov rdi, str1 // carrega o rdi com conteúdo de str1 ("String em maiúsculas: %s",0).

mov rsi, cadeia // carrega rsi com conteúdo de cadeia.

mov rax,0 //zera o rax.

call printf // printa o anteriormente passado, @ele identifica %s como a cadeia, por ser um ponteiro.

leave // desfaz o push rbp mov rbp,rsp.

mov rax,0 // zera o rax

ret // retorna 0

```
assemblyUpper.asm  t//~//1/u/b/bash
[felipe@archlinux ~]$ ./maiusculas
Digite uma string: Muito obrigado!
String em maiúsculas: MUITO OBRIGADO![felipe@archlinux ~]$
```

Programa em Assembly funcionando.

#Muito obrigado a todos que leram até aqui!!! Espero ter ajudado na compreensão do código!

Muito obrigado! :)