

Estruturas de Iteração

João Marcelo Uchôa de Alencar

17 de abril de 2023

Introdução

Estrutura de Laços com Pré-teste

Estrutura de Laços com Pós-teste

Estruturas de Laços com Iterações Fixas

Laços e Entrada /Saída

Laços Aninhados

Programa Completo

Resumo

Introdução

- ▶ Há várias estruturas de iteração disponíveis em linguagens de alto nível.
- ▶ Existem versões correspondentes em MASM:
 - ▶ Pré-teste.
 - ▶ Pós-teste.
 - ▶ Número fixo de iterações.
- ▶ O uso de cada uma vai depender do contexto.

Estrutura de Laços com Pré-teste

```
i = 1;
while (i < 3) {
    // body of loop
    i++;
}
```

Três partes do laço:

- ▶ Inicialização: $i = 1$;
- ▶ Teste: $i \leq 3$.
- ▶ Mudança: $i++$;

Veja que i é a variável de controle do laço.

Estrutura de Laços com Pré-teste

Em MASM, também temos diretivas para laços:

```
mov i, 1
.while i <= 3
; body loop
inc i
.endw
```

- ▶ Lembrando que no teste, não podemos fazer comparação entre duas posições de memória.
- ▶ Poderíamos usar um registrador no lugar de *i* para melhor desempenho.
- ▶ Também podemos fazer uma versão sem diretivas.

Estrutura de Laços com Pré-teste

Instrução *cmp* e desvios:

```
        mov i, 1
        ; while i <= 3
while01: cmp i, 3
        jg endw01
        ; body of loop
        inc i
        jmp while01
endw01:  nop
```

- ▶ O nome dos rótulos ajuda na compreensão.
- ▶ Um desvio incondicional é a última instrução do corpo, retornando ao início do laço.

Estrutura de Laços com Pré-teste

- ▶ Considere, como exemplo, uma versão do x86 que não tem instruções de multiplicação.
- ▶ A multiplicação poderia ser implementada como um laço de adições.

```
ans = 0;  
i = 1;  
while (i <= y) {  
    ans = ans + x;  
    i++;  
}
```

- ▶ Se $y == 0$, o laço não executa, não há desperdício.
- ▶ Agora se $x == 0$, várias somas com 0 seriam feitas.

Estrutura de Laços com Pré-teste

```
ans = 0;
if (x!=0) {
    i = 1;
    while (i <= y) {
        ans = ans + x;
        i++;
    }
}
```

- ▶ Um teste em x elimina as somas desnecessárias.
- ▶ Podemos implementar a versão acima com diretivas *.while*.

Estrutura de Laços com Pré-teste

```
mov ans, 0           ; initialize ans to 0
.if x != 0
mov i, 1             ; initialize i to 1
mov eax, y           ; load eax with y for while
.while i <= eax
mov eax, ans         ; load eax with ans
add eax, x           ; add eax to ans
mov ans, eax         ; store eax in ans
mov eax, y           ; reload eax with y for while
inc i                ; increment i by 1
.endw
.endif
```

`eax` é usado mais de uma vez, porém seu valor é sempre restaurado.

Estrutura de Laços com Pré-teste

```
mov ans, 0           ; initialize ans to 0
.if x != 0
mov ecx, 1           ; initialize ecx to 1
.while ecx <= y
mov eax, ans         ; load eax with ans
add eax, x           ; add eax to ans
mov ans, eax         ; store eax in ans
inc ecx              ; increment i by 1
.endw
mov i, ecx           ; store ecx in i
.endif
```

ecx pode ser usado no lugar de *i* como controle.

Estrutura de Laços com Pós-teste

- ▶ Laços em que o corpo é executado pelo menos uma vez.
- ▶ Em MASM, temos as diretivas *.repeat* e *.until*:

<code>i = 1;</code>	<code>mov i, 1</code>
<code>do {</code>	<code>.repeat</code>
<code> // body of loop</code>	<code>; body of loop</code>
<code> i++;</code>	<code>inc i</code>
<code>} while (i <= 3);</code>	<code>.until i > 3</code>

- ▶ Percebam que no lugar de $i \leq 3$, usamos $i > 3$.

Estrutura de Laços com Pós-teste

- ▶ Usando instrução *cmp* e desvios:

```
        mov i, 1
repeat01:  nop
           ; body of loop
           inc i
           cmp i, 3
           jle repeat01
endrpt01:  nop
```

- ▶ Como implementar o exemplo anterior, multiplicação através de somas?

Estrutura de Laços com Pós-teste

- Precisamos testar y antes, para evitar que o laço execute uma vez se $y == 0$.

```
ans = 0;
if (y != 0) {
    i = 1;
    do {
        ans = ans + x;
        i++;
    } while (i <= y);
}

mov ans, 0
.if y != 0
mov ecx, 1
.repeat
mov eax, ans
add eax, x
mov ans, eax
inc ecx
.until ecx > y
mov i, ecx
.endif
```

- Também poderíamos checar de $x == 0$.

Estruturas de Laços com Iterações Fixas

- ▶ Nas linguagens de alto nível, são os laços *for*.
- ▶ O laço só precisa executar um número fixo de vezes.

```
for (i = 1; i <= 3; i++) {  
    // body of loop  
}
```

- ▶ Costuma ser um pouco mais rápido que os outros laços.
- ▶ No MASM, temos as diretivas *.repeat* e *.untilcxz*.

Estruturas de Laços com Iterações Fixas

- ▶ *.repeat* e *.untilcxz* usam o registrador *ecx* como contador.
- ▶ *.untilcxz* **primeiro** decrementa *ecx* em 1 e **depois** desvia para *.repeat* caso *ecx* não seja igual a 0.

```
mov ecx, 3
.repeat
; body of the loop
.untilcxz
```

- ▶ *.repeat* e *.untilcxz* exigem que o corpo do laço seja executado pelo menos uma vez.
- ▶ É perigoso alterar o valor de *ecx* no corpo do laço.

Estruturas de Laços com Iterações Fixas

- ▶ É possível implementar o laço usando instruções, sem diretivas.
- ▶ Mas no lugar de *cmp*, usamos a instrução *loop*.

```
        mov    ecx, 3
for01:   nop
        ; body of the loop
        loop   for1
endfor01: nop
```

- ▶ *loop* decrementa *ecx* em 1 e desvia para o rótulo indicado se *ecx* não for igual a 0.
- ▶ Problema: se $ecx \leq 0$ antes do corpo do laço, qual a consequência?

Estruturas de Laços com Iterações Fixas

- ▶ A instrução *jecxz* irá desviar para após a instrução *loop* se $ecx == 0$.

	<code>; check for zero</code>	<code>; check for non-positive</code>
	<code>jecxz endfor01</code>	<code>.if ecx > 0</code>
<code>for01:</code>	<code>nop</code>	<code>.repeat</code>
	<code>; body of the loop</code>	<code>; body of the loop</code>
	<code>loop for1</code>	<code>.untilcxz</code>
<code>endfor01:</code>	<code>nop</code>	<code>.endif</code>

- ▶ A versão com diretivas testa ainda o caso de *ecx* negativo, o que não ocorre em *jecxz*.

Estruturas de Laços com Iterações Fixas

- ▶ A diretiva *.repeat* só pode estar até 128 *bytes* antes da diretiva *.untilcxz*.
- ▶ O rótulo referenciado por uma instrução *loop* também só pode estar até 128 *bytes*.
- ▶ Isso limita o tamanho do laço, pois cada instrução pode ocupar entre 1 e 6 *bytes*.
- ▶ Na prática, laços muito grandes são raros.
- ▶ Caso seja necessário, um *.while* pode ser utilizado.

Estruturas de Laços com Iterações Fixas

```
ans = 0;
if (y != 0)
    for (i=1; i <= y; i++)
        ans = ans + x;
```

```
mov ans, 0
.if y != 0
mov ecx, y
.repeat
mov eax, ans
add eax, x
mov ans, eax
.untilcxz
.endif
```

Laços e Entrada /Saída

- ▶ Aceitando uma quantidade fixa de inteiros.
- ▶ Fazendo a soma em seguida.

```
sum = 0;
for (i = 0; i <= 10; i++){
    printf("%s", "Enter an integer: ");
    scanf("%d", &sum);
    sum = sum + sum;
}
printf("\n%s%d\n\n", "The sum is ", sum);
return 0;
```

Laços e Entrada /Saída

```
    .data
    ...
msg1    byte    "Enter an integer: ", 0
msg2    byte    "The sum is ", 0
    ...
    .code
    mov sum, 0
    mov ecx, 10
    .repeat
    mov temp, ecx
    INVOKE printf, ADDR msg1fmt, ADDR msg1
    INVOKE scanf, ADDR in1fmt, ADDR num
    mov eax, sum
    add eax, num
    mov sum, eax
    mov ecx, temp
    .untilcxz
    INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Laços e Entrada /Saída

- ▶ Veja que `ecx` é copiado para uma variável *temp*.
- ▶ *INVOKE* apaga os registradores *eax*, *ecx* e *edx*.
- ▶ *.repeat* e *.untilcxz* são controladas por *ecx*.
- ▶ Portanto precisamos salvar e restaurar o valor.
- ▶ Como fazer para somar mais de 10 números?

Laços e Entrada /Saída

```
msg0      .data
byte      "Enter the number of integers to input: ", 0
msg1      byte      "Enter an integer: ", 0
msg2      byte      "The sum is ", 0
...
.code
mov sum, 0
INVOKE printf, ADDR msg1fmt, ADDR msg0
INVOKE scanf, ADDR in1fmt, ADDR count
mov ecx, 1
.while ecx <= count
mov temp, ecx
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR num
mov eax, sum
add eax, num
mov sum, eax
mov ecx, temp
inc ecx
.endw
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Laços e Entrada /Saída

- ▶ Trocamos *.repeat*, *.untilcxz* por *.while*, *.endw*.
- ▶ Antes do laço, indagamos a quantidade de inteiros a ser lida.
- ▶ Como fazer para não ter que exigir a quantidade de inteiros?
- ▶ Usamos uma *flag* para interromper o laço.

Laços e Entrada /Saída

```
sum = 0;
printf("%s", "Enter an integer or a negative integer to stop: ");
scanf("%d", &num);
while (num > 0) {
    sum = sum + num;
    printf("%s", "Enter an integer or a negative integer to stop: ");
    scanf("%d", &num);
}
printf("\n%s%d\n\n", "The sum is ", sum);
```

Laços e Entrada /Saída

```
.data
...
msg1    byte    "Enter an integer or a negative integer to stop: ", 0
msg2    byte    "The sum is ", 0
...
.code
mov sum, 0
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR count
mov ecx, 1
.while num >= 0
mov eax, sum
add eax, num
mov sum, eax
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR num
.endw
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Laços e Entrada /Saída

- ▶ É possível fazer um laço controlado usando apenas um *scanf*, mas não é recomendado.
- ▶ O estilo seria comparar a entrada com a variável sentinela, desviando para fora no meio do laço.
- ▶ Nas linguagens de alto nível, podemos usar o *break*.
- ▶ Em MASM, usaremos as instruções de desvio.

Laços e Entrada /Saída

```
sum = 0;
while (1) {
    printf("%s", "Enter an integer or a negative integer to stop: ");
    scanf("%d", &num);
    if (num < 0)
        break;
    sum = sum + num;
}
printf("\n%s%d\n\n", "The sum is ", sum);
```

Laços e Entrada /Saída

```
.data
...
msg1fmt byte    "%s",0
msg2fmt byte    0Ah,"%s%d",0Ah,0Ah,0
...
.code
mov sum, 0
.while 1
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR num
cmp num, 0
jl endl
mov eax, sum
add eax, num
mov sum, eax
.endw
endl: nop
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Laços Aninhados

- ▶ Laços aninhados são importantes para acessar *arrays* de várias dimensões e algoritmos de ordenação.
- ▶ Temos que lembrar de utilizar variáveis de controle diferentes.

```
i = 1;
while (i <= 2) {
    j = 1;
    while (j <= 3) {
        // body of nested loop
        j++;
    }
    i++;
}
```

```
mov i, 1
    .while i <= 2
    mov j, 1
        .while j <= 3
            ; body of nested loop
        inc j
    .endw
    inc i
    .endw
```

Laços Aninhados

- ▶ Sem diretivas, é preciso tomar cuidado.
- ▶ Os desvios condicionais devem ser revertidos e os desvios incondicionais precisam estar na posição correta.

```
                mov i, 2
while02:        cmp i, 2
                jg endwhile02
                mov j, 1
while03:        cmp j, 3
                jg endwhile03
                ; body of nested loop
                inc j
                jmp while 03
endwhile03:     nop
                inc i
                jmp while02
endwhile02:     nop
```

Laços Aninhados

- ▶ Muitas vezes é melhor usar um laço com iterações fixas:

```
for (i = 1; i <= 2; i++)  
    for (j = 1; j <= 3; j++) {  
        // body of nested loop  
    }
```

- ▶ Mas é fácil cometer um erro com o uso de diretivas:

```
; *** Caution: Incorrectly implemented code ***  
mov ecx, 2  
.repeat  
mov ecx, 3  
.repeat  
; body of nested loop  
.untilcxz  
.untilcxz
```

Laços Aninhados

- ▶ *.repeat* e *.untilcxz* só podem trabalhar com um registrador.
- ▶ Uma solução é guardar o valor em memória temporária.

```
; *** Note: Correctly implemented code ***  
mov ecx, 2  
.repeat  
mov tempecx, ecx  
mov ecx, 3  
.repeat  
; body of nested loop  
.untilcxz  
mov ecx, tempecx  
.untilcxz
```

Programa Completo

x^n = Se $x < 0$ ou $n < 0$, então mensagem negativa.

Senão se $x = 0$ e $n = 0$, então mensagem não definida.

Senão se $n = 0$, então 1.

Caso contrário $1 * x * x * x * \dots * x$ (n vezes).

Programa Completo

```
#include <stdio.h>
int main() {
    int x, n, i, ans;
    printf("%s", "Enter x: ");
    scanf("%d", &x);
    printf("%s", "Enter n: ");
    scanf("%d", &n);
    if (x < 0 || n < 0)
        printf("\n%s\n\n", "Error: Negative x and/or y");
    else if (x == 0 && n == 0)
        printf("\n%s\n\n", "Error: Undefined answer");
    else {
        i = 1;
        ans = 1;
        while (i <= n) {
            ans = ans * x;
            i++;
        }
        printf("\n%s%d\n\n", "The answer is: ", ans);
    }
    return 0;
}
```

Programa Completo

```
.686
.model flat, c
.stack 100h

printf PROTO arg1:Ptr Byte, printlist:VARARG
scanf PROTO arg2:Ptr Byte, inputlist:VARARG

.data

in1fmt byte "%d",0
msg1fmt byte "%s",0
msg3fmt byte "%s%d",0Ah,0Ah,0
errfmt byte "%s",0Ah,0Ah,0
errmsg1 byte "Error: Negative x and/or y",0
errmsg2 byte "Error: Undefined answer",0
msg1 byte "Enter x: ",0
msg2 byte "Enter n: ",0
msg3 byte 0Ah, "The answer is: ",0
x sdword ?
n sdword ?
ans sdword ?
i sdword ?
```

Programa Completo

```
main      .code
proc
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR x
INVOKE printf, ADDR msg1fmt, ADDR msg2
INVOKE scanf, ADDR in1fmt, ADDR n
.if x < 0 || n < 0
INVOKE printf, ADDR errfmt, ADDR errmsg1
.else
.if x == 0 && n == 0
INVOKE printf, ADDR errfmt, ADDR errmsg2
.else
mov ecx,1
mov ans,1
.while ecx <= n
mov eax,ans
imul x
mov ans,eax
inc ecx
.endw
mov i,ecx
INVOKE printf, ADDR msg3fmt, ADDR msg3, ans
.endif
.endif
ret
main      endp
end
```

Resumo

- ▶ As diretivas *.while* - *.endw* implementam um laço pré-teste.
- ▶ As diretivas *.repeat* - *.until* e *.repeat* - *.untilcxz* são ambas de pós-teste.
- ▶ A diretiva *loop* é a base das diretivas *.repeat* - *.untilcxz*.
- ▶ Assim como a diretiva *.if*, as diretivas *.while* - *.endw* e *.repeat* - *.until* não podem comparar dois valores em memória.

Resumo

- ▶ Tenha cuidado de inicializar *ecx* com um número positivo ao usar a instrução *loop* ou as diretivas *.repeat* - *.untilcxz*. A diretiva *.if* e a instrução *jecxz* podem ajudar.
- ▶ Quando usando a instrução *loop* ou a diretiva *.repeat* - *.untilcxz* é uma boa ideia não alterar o conteúdo do registrador *ecx*.
- ▶ Quando aninhando diretivas *.repeat* - *.untilcxz* ou instruções *loop*, tenha cuidado de preservar o registrador *ecx* antes e depois do laço interno.
- ▶ O começo de diretivas *.repeat* - *.until* ou instruções *loop* não podem estar a mais de 128 *bytes* de distância.