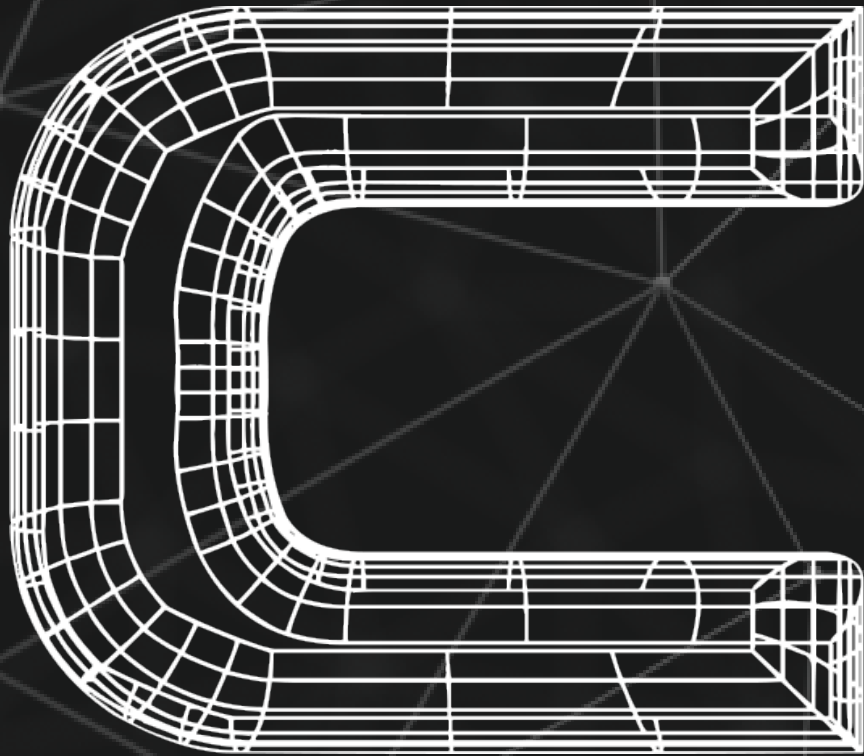


LINGUAGEM



PARA REVERSING

THIAGO PEIXOTO

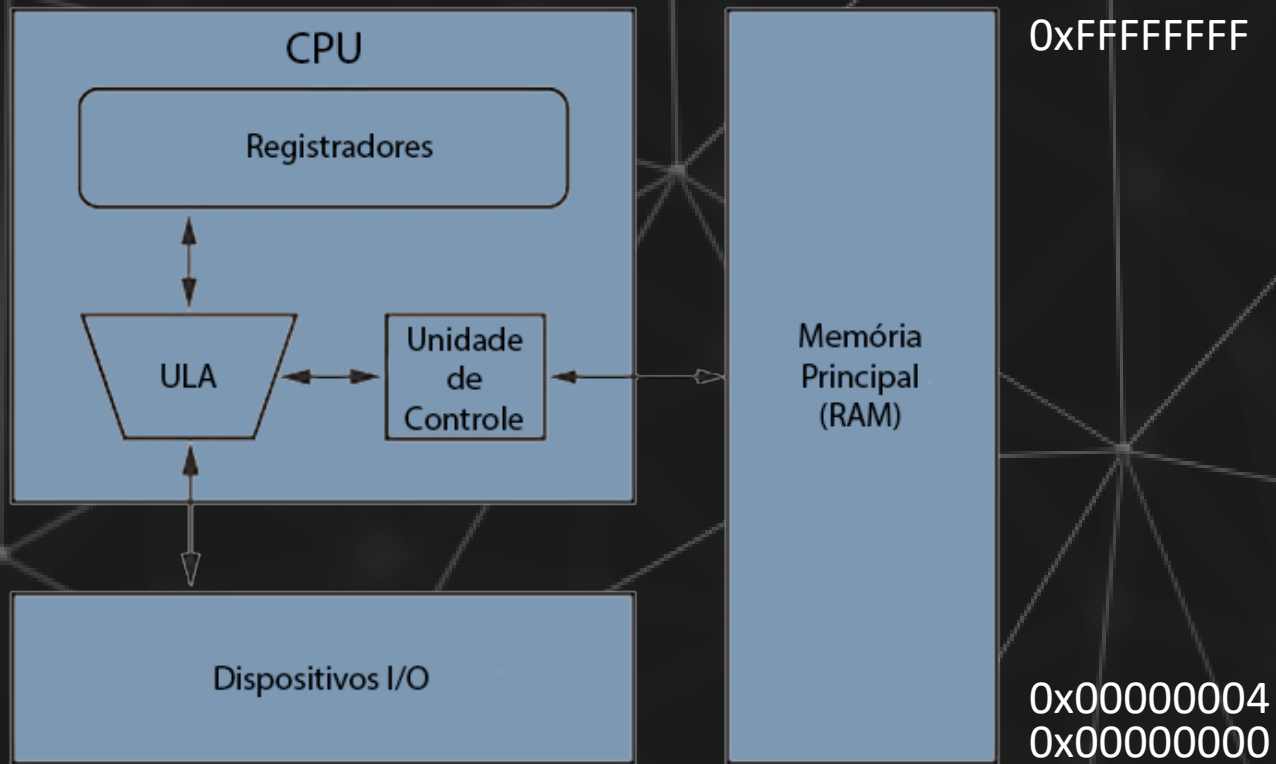
thiago@zero2sec.com.br

<http://www.zero2sec.com.br>

<http://telegram.me/EngenhariaReversa>

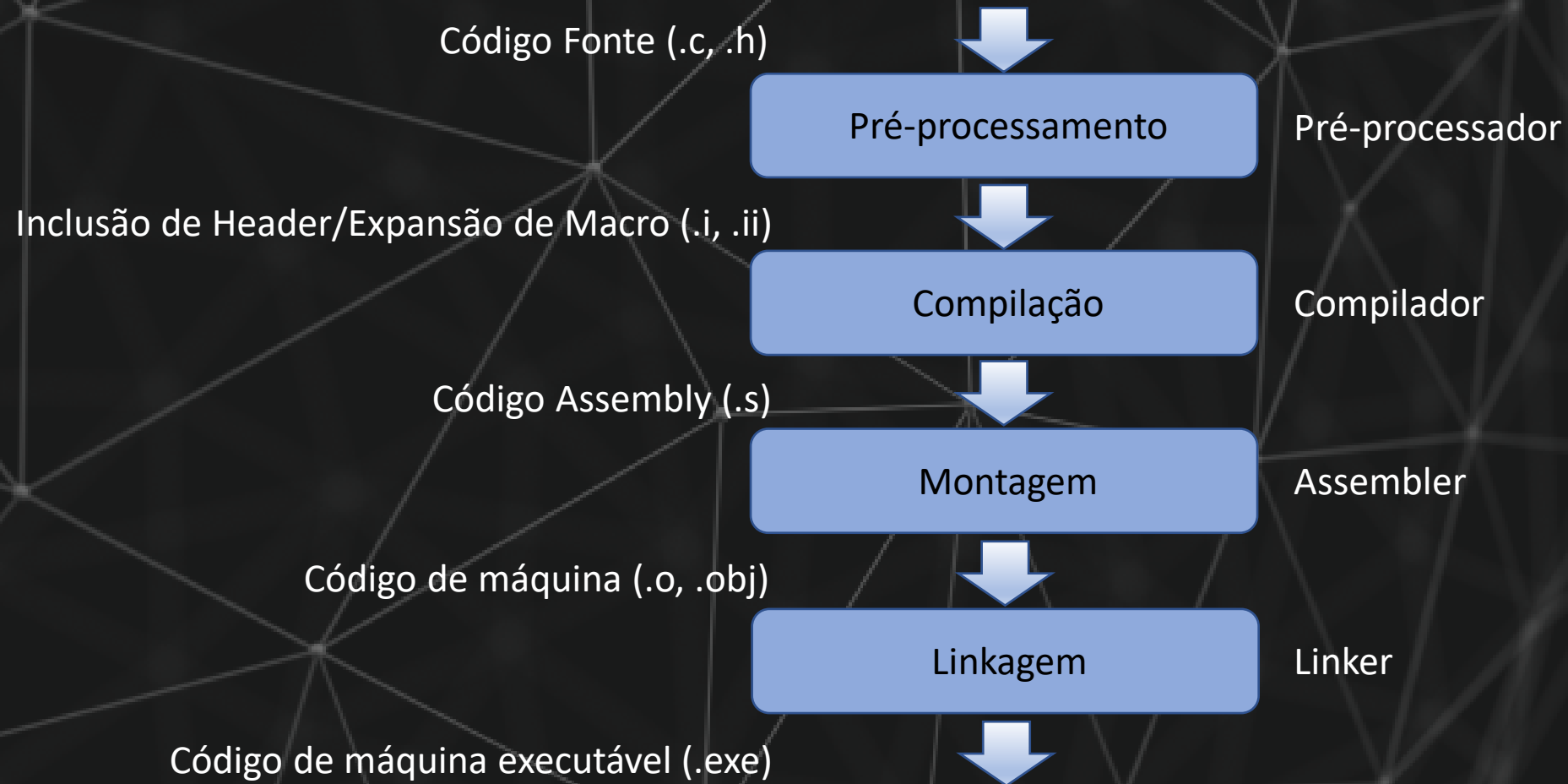
REVISÃO

- **ARQUITETURA DE VON NEUMANN**



REVISÃO

• PROCESSO DE COMPILAÇÃO (GCC)



COMENTÁRIOS

- **ÚNICA LINHA**

```
// Este é um comentário de uma única linha.  
// Somente disponível em > C99.
```

- **COMENTÁRIO DE BLOCO**

```
/*  
* Este é um comentário de bloco.  
* Bem melhor que múltiplos comentários de uma linha. :)  
*/
```

- **COMENTÁRIOS DE BLOCO NÃO PODEM SER ANINHADOS**

CONSTANTES

- **DIRETIVA #define**

```
#define NUM_MAX 256
```

- **enum**

```
enum Dias {Dom, Seg, Ter, Qua, Qui, Sex, Sab} dias_da_semana;  
dias_da_semana = Dom;  
// Dom = 0, Seg = 1, Ter = 2,...
```

- **PODEMOS ATRIBUIR UM VALOR EXPLICITAMENTE PARA CADA CONSTANTE**

```
enum Dias {Dom = 1, Seg, Ter = 10, Qua, Qui, Sex, Sab};  
// Dom = 1, Seg = 2, Ter = 10, Qua = 11,...
```

- **MODIFICADOR const**

```
const int NUM_MAX = 256;
```

CONSTANTES

- **DIRETIVA #define**

```
#define NUM_MAX 256
```

- **enum**

```
enum Dias {Dom, Seg, Ter, Qua, Qui, Sex, Sab} dias_da_semana;
```

```
dias_da_semana = Dom;  mov     DWORD PTR [ebp-4], 0
```

```
// Dom = 0, Seg = 1, Ter = 2,...
```

- **PODEMOS ATRIBUIR UM VALOR EXPLICITAMENTE PARA CADA CONSTANTE**

```
enum Dias {Dom = 1, Seg, Ter = 10, Qua, Qui, Sex, Sab};
```

```
// Dom = 1, Seg = 2, Ter = 10, Qua = 11,...
```

- **MODIFICADOR const**

```
const int NUM_MAX = 256;  mov     DWORD PTR [ebp-4], 256
```

CONSTANTES

```
#define CONST_INT 31
#define CONST_LONG 1000L
#define CONST_UINT 1000U
#define CONST_ULONG 1000UL
#define CONST_FLOAT 256.1234F
#define CONST_DOUBLE 256.1234
#define CONST_INT_HEX 0X1F
#define CONST_INT_OCTAL 037
#define CONST_CHAR 'A'
#define NOVA_LINHA '\n'
#define EXPR_CONST CONST_INT + 1
#define STRING_LITERAL "Temos uma string!"
```

- AS LETRAS DOS SUFIXOS E PREFIXOS TAMBÉM PODEM SER MINÚSCULAS

CONSTANTES

```
#define CONST_INT 31
#define CONST_LONG 1000L
#define CONST_UINT 1000U
#define CONST_ULONG 1000UL
#define CONST_FLOAT 256.1234F
#define CONST_DOUBLE 256.1234
#define CONST_INT_HEX 0X1F
#define CONST_INT_OCTAL 037
#define CONST_CHAR 'A'
#define NOVA_LINHA '\n'
#define EXPR_CONST 0
#define STRING_LITERAL "Temos uma string!"
```

T	e	m	o	s		u	m	a		s	t	r	i	n	g	!	0
---	---	---	---	---	--	---	---	---	--	---	---	---	---	---	---	---	---

- **AS LETRAS DOS SUFIXOS E PREFIXOS TAMBÉM PODEM SER MINÚSCULAS**

VARIÁVEIS

- **UMA VARIÁVEL DE PROGRAMA É UMA ABSTRAÇÃO DE UMA CÉLULA DE MEMÓRIA OU DE UMA COLEÇÃO DE CÉLULAS**
- **CADA VARIÁVEL POSSUI UM NOME, UM ENDEREÇO, UM TIPO E UM VALOR:**
 - **NOME: IDENTIFICADOR DA VARIÁVEL**
 - **ENDEREÇO: ENDEREÇO DE MEMÓRIA DE MÁQUINA**
 - **TIPO: DETERMINA A FAIXA DE VALORES QUE A VARIÁVEL PODE ARMAZENAR E O CONJUNTO DE OPERAÇÕES DEFINIDAS PARA VALORES DO TIPO**
 - **VALOR: É O CONTEÚDO DA(S) CÉLULA(S) DE MEMÓRIA ASSOCIADA(S) A ELA**

VARIÁVEIS

- **NOME**

- **CARACTERES PERMITIDOS:**

- **UNDERSCORE (_)**
 - **LETRAS MAIÚSCULAS (A – Z)**
 - **LETRAS MINÚSCULAS (a – z)**
 - **DÍGITOS (0 – 9)**

- **PRIMEIRO CARACTERE DEVE SER LETRA OU UNDERSCORE**

- **NÃO DEVE SER UMA PALAVRA RESERVADA**

- **DECLARAÇÃO**

- **FORMATO: TIPO NOME_DA_VARIAVEL**

- **DECLARAÇÕES PODEM SER SEPARADAS POR VÍRGULAS E/OU INICIALIZADAS**

```
int x = 10, y, z;
```

PALAVRAS RESERVADAS

- C POSSUI 32 PALAVRAS RESERVADAS PELO PADRÃO ANSI

PALAVRAS RESERVADAS			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

TIPOS

- **TIPOS PRIMITIVOS**

```
// Tipo caractere (1 byte) [-127, 127]
```

```
char c = 'A';
```

```
// Tipos inteiros
```

```
short si = 100; // (2 bytes) [-32767, 32767]
```

```
int i = 32000; // (4 bytes) [-2147483647, +2147483647]
```

```
long l = 32480; // (4 bytes) [-2147483647, +2147483647]
```

```
// (8 bytes) [-9223372036854775807, 9223372036854775807]
```

```
long long ll = 32800012;
```

```
// Tipos decimais
```

```
float f = 1.80; // (4 bytes)
```

```
double d = 3.2903; // (8 bytes)
```

- **OS TAMANHOS PODEM VARIAR EM DIFERENTES ARQUITETURAS**

TIPOS

• TIPOS PRIMITIVOS

// Tipo caractere (1 byte) [-127, 127]

char c = 'A';


 mov BYTE PTR [ebp-9], 65

// Tipos inteiros


short si = 100;

 mov WORD PTR [ebp-12], 100

int i = 32000;

 mov DWORD PTR [ebp-16], 32000

long l = 32480;

 mov DWORD PTR [ebp-20], 32480

// (8 bytes) [-922337203



 mov DWORD PTR [ebp-32], 32800012

long long ll = 32800012;

 mov DWORD PTR [ebp-28], 0


// Tipos decimais

float f = 1.80; //

 fld DWORD PTR .LC0
 fstp DWORD PTR [ebp-36]

double d = 3.2903;

.LC0:
.long 1072064102

 fld QWORD PTR .LC1
 fstp QWORD PTR [ebp-48]

.LC1:
.long 3463461627
.long 1074418312

• OS TAMANHOS PODEM VARIAR EM DIFERENTES A

TIPOS

- **MODIFICADOR unsigned/signed**

```
unsigned char uc = 'A'; // [0, 255]
unsigned short usi = 100; // [0, 65535]
unsigned int ui = 32000; // [0, 4294967295]
unsigned long ul = 32480; // [0, 4294967295]
// [0, 18446744073709551615]
unsigned long long ull = 32800012;
```

- **0 USO DO signed É OPCIONAL**

TIPOS

- **E A STRING??**

- **UMA STRING É REPRESENTADA POR UMA SEQUÊNCIA CONTÍNUA DE CARACTERES, SEGUIDOS PELO CARACTERE NULO ('\\0'), QUE DEMARCA O FIM DE UMA STRING**

```
char *string1 = "Hello, String1!";
```

```
char string2[] = "Hello, String2!";
```

- **CUIDADO AO TENTAR UM ELEMENTO DA STRING DO PRIMEIRO CASO. UMA TENTATIVA DE MUDANÇA REPRESENTA UM UNDEFINED BEHAVIOUR**
- **EXISTE UMA DIFERENÇA SUTIL ENTRE AS DUAS DECLARAÇÕES :)**

TIPOS

- **E A STRING??**

- **UMA STRING É REPRESENTADA POR CARACTERES, SEGUIDOS PELO FIM DE UMA STRING**

```
char *string1 = "Hello, String1!";  
char string2[] = "Hello, String2!";
```

- **CUIDADO AO TENTAR UM ELEMENTO DE UMA STRING, UMA TENTATIVA DE MUDANÇA REPRESENTA UM ERRO**
- **EXISTE UMA DIFERENÇA SUTIL ENTRE UMA STRING E UM CARACTER**

```
mov     DWORD PTR [ebp-4], OFFSET FLAT:.LC0  
.Ltext0:  
        .section      .rodata  
.LC0:  
        .string "Hello, String1!"
```

```
mov     DWORD PTR [ebp-0x18],0x6c6c6548  
mov     DWORD PTR [ebp-0x14],0x53202c6f  
mov     DWORD PTR [ebp-0x10],0x6e697274  
mov     DWORD PTR [ebp-0xc],0x213267
```


ESCOPO

- **ESCOPO DE UM NOME É A PARTE DO PROGRAMA EM QUE UM NOME PODE SER USADO**
- **O ESCOPO DE VARIÁVEIS PODE SER DIVIDIDO EM:**
 - **VARIÁVEIS LOCAIS:**
 - **DECLARADAS DENTRO DO BLOCO DE UMA FUNÇÃO ({...})**
 - **NÃO PODEM SER ACESSADAS POR OUTRAS FUNÇÕES**
 - **EXISTEM ENQUANTO A FUNÇÃO ONDE ELA FOR DECLARADA ESTIVER SENDO EXECUTADA**
 - **PARÂMETROS FORMAIS:**
 - **TAMBÉM SÃO VARIÁVEIS LOCAIS DA FUNÇÃO**
 - **VARIÁVEIS GLOBAIS:**
 - **ACESSÍVEIS EM QUALQUER PARTE DO PROGRAMA (POR TODAS AS OUTRAS FUNÇÕES)**
 - **EXISTEM DURANTE TODA A EXECUÇÃO DO PROGRAMA**
 - **DECLARADAS FORA DE TODOS OS BLOCOS DE FUNÇÕES**

ESCOPO

```
int z = 10, w;
```

```
int add(int x, int y)
{
    int z = x + y;
    return z;
}
```

```
int main(void)
{
    z = 20;
    return add(w, z);
}
```

- **VARIÁVEIS LOCAIS**
- **PARÂMETROS FORMAIS**
- **VARIÁVEIS GLOBAIS**

FUNÇÃO	VISIBILIDADE
add	z, x, y, w
main	z, w

- **SE VARIÁVEIS LOCAIS E GLOBAIS POSSUÍREM O MESMO NOME, ELAS SE COMPORTARÃO COMO VARIÁVEIS DIFERENTES**

ESCOPO

```
int z = 10, w;
```

```
int {  
    i  
    r  
}  
  
    .globl z  
    .data  
    .align 4  
    .type z, @object  
    .size z, 4  
z:  
    .long 10
```

```
    .globl w  
    .bss  
    .align 4  
    .type w, @object  
    .size w, 4  
w:  
    .zero 4
```

```
w, z);
```



add:

```
push    ebp  
mov     ebp, esp  
sub     esp, 16  
mov     edx, DWORD PTR [ebp+8]  
mov     eax, DWORD PTR [ebp+12]  
add     eax, edx  
mov     DWORD PTR [ebp-4], eax  
mov     eax, DWORD PTR [ebp-4]  
leave  
ret
```

main:


```
push    ebp  
mov     ebp, esp  
mov     DWORD PTR z, 20  
mov     edx, DWORD PTR z  
mov     eax, DWORD PTR w  
push    edx  
push    eax  
call    add  
add     esp, 8  
leave  
ret
```


ESCOPO

```
int main(void)
{
    int n = 0;
    if (n == 0) {
        int n = 10;
        n *= 3; // 30
    }
    n *= 3; // 0
    return 0;
}
```

ESCOPO

```
int main(void)
{
    int n = 0;
    if (n == 0) {
        int n = 10;
        n *= 3; // 30
    }
    n *= 3; // 0
    return 0;
}
```

```
 mov     DWORD PTR [ebp-4], 0

 cmp     DWORD PTR [ebp-4], 0
jne     .L2
mov     DWORD PTR [ebp-8], 10
mov     edx, DWORD PTR [ebp-8]
mov     eax, edx
add     eax, eax
add     eax, edx
mov     DWORD PTR [ebp-8], eax

.L2:
mov     edx, DWORD PTR [ebp-4]
mov     eax, edx
add     eax, eax
add     eax, edx
mov     DWORD PTR [ebp-4], eax
mov     eax, 0
leave
ret
```

OPERADORES

- ARITMÉTICOS

OPERADOR		DESCRIÇÃO
+	BINÁRIO	ADIÇÃO
-	BINÁRIO	SUBTRAÇÃO
*	BINÁRIO	MULTIPLICAÇÃO
/	BINÁRIO	DIVISÃO
%	BINÁRIO	RESTO
++	UNÁRIO	INCREMENTO DE 1
--	UNÁRIO	DECREMENTO DE 1

OPERADORES

- **ARITMÉTICOS**

```
int x = 10, y = 20;
```

```
int add = x + y;
```

```
int sub = y - x;
```

```
int mult = y * x;
```

```
int div = y / x;
```

```
int mod = y % x;
```

OPERADORES

• ARITMÉTICOS

```
int x = 10, y = 20;
```

```
int add = x + y;
```

```
int sub = y - x;
```

```
int mult = y * x;
```

```
int div = y / x;
```

```
int mod = y % x;
```

	mov	DWORD PTR [ebp-4], 10
	mov	DWORD PTR [ebp-8], 20
	mov	edx, DWORD PTR [ebp-4]
	mov	eax, DWORD PTR [ebp-8]
	add	eax, edx
	mov	DWORD PTR [ebp-12], eax
	mov	eax, DWORD PTR [ebp-8]
	sub	eax, DWORD PTR [ebp-4]
	mov	DWORD PTR [ebp-16], eax
	mov	eax, DWORD PTR [ebp-8]
	imul	eax, DWORD PTR [ebp-4]
	mov	DWORD PTR [ebp-20], eax
	mov	eax, DWORD PTR [ebp-8]
	cdq	
	idiv	DWORD PTR [ebp-4]
	mov	DWORD PTR [ebp-24], eax
	mov	eax, DWORD PTR [ebp-8]
	cdq	
	idiv	DWORD PTR [ebp-4]
	mov	DWORD PTR [ebp-28], edx

OPERADORES

- **ARITMÉTICOS**

```
int x = 10, y = 20;  
// inc1 = 10, x = 11  
int inc1 = x++;  
// inc2 = 12, x = 12  
int inc2 = ++x;  
// dec1 = 20, y = 19  
int dec1 = y--;  
// dec2 = 18, y = 18  
int dec2 = --y;
```

OPERADORES

• ARITMÉTICOS

```
int x = 10, y = 20;  
// inc1 = 10, x = 11  
int inc1 = x++;  
// inc2 = 12, x = 12  
int inc2 = ++x;  
// dec1 = 20, y = 19  
int dec1 = y--;  
// dec2 = 18, y = 18  
int dec2 = --y;
```



Assembly code listing showing the implementation of the arithmetic operations from the C code. Each instruction is preceded by a red circular icon containing a black x86-64 instruction symbol.

```
mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
mov     eax, DWORD PTR [ebp-4]  
lea     edx, [eax+1]  
mov     DWORD PTR [ebp-4], edx  
mov     DWORD PTR [ebp-12], eax  
  
add     DWORD PTR [ebp-4], 1  
mov     eax, DWORD PTR [ebp-4]  
mov     DWORD PTR [ebp-16], eax  
  
mov     eax, DWORD PTR [ebp-8]  
lea     edx, [eax-1]  
mov     DWORD PTR [ebp-8], edx  
mov     DWORD PTR [ebp-20], eax  
  
sub     DWORD PTR [ebp-8], 1  
mov     eax, DWORD PTR [ebp-8]  
mov     DWORD PTR [ebp-24], eax
```

OPERADORES

- **RELACIONAIS**

OPERADOR		DESCRIÇÃO
>	BINÁRIO	MAIOR QUE
<	BINÁRIO	MENOR QUE
==	BINÁRIO	IGUAL
!=	BINÁRIO	DIFERENTE
>=	BINÁRIO	MAIOR OU IGUAL
<=	UNÁRIO	MENOR OU IGUAL

- **AVALIAM DUAS EXPRESSÕES E RETORNAM 1 (VERDADEIRO) OU 0 (FALSO)**
- **NÃO CONFUNDIR “==” COM “=” :)**

OPERADORES



- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y > x) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y > x) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
 mov     eax, DWORD PTR [ebp-8]  
cmp     eax, DWORD PTR [ebp-4]  
jle     .L2  
mov     eax, 1  
jmp     .L3  
.L2:  
mov     eax, 0  
.L3:  
leave  
ret
```

OPERADORES



- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y < x) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y < x) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
 mov     eax, DWORD PTR [ebp-8]  
cmp     eax, DWORD PTR [ebp-4]  
jge     .L2  
mov     eax, 1  
jmp     .L3  
.L2:  
mov     eax, 0  
.L3:  
leave  
ret
```

OPERADORES


- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y == x) {  
    return 1;  
} else {  
    return 0;  
}
```



OPERADORES

- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y == x) {  
    return 1;  
} else {  
    return 0;  
}
```



```
mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20
```



```
mov     eax, DWORD PTR [ebp-8]  
cmp     eax, DWORD PTR [ebp-4]  
jne     .L2  
mov     eax, 1  
jmp     .L3
```

```
.L2:  
mov     eax, 0
```

```
.L3:  
leave  
ret
```

OPERADORES


- **RELACIONAIS**


```
int x = 10, y = 20;  
if (y != x) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y != x) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20
```

```
 mov     eax, DWORD PTR [ebp-8]  
cmp     eax, DWORD PTR [ebp-4]  
je      .L2  
mov     eax, 1  
jmp     .L3
```

```
.L2:  
mov     eax, 0
```

```
.L3:  
leave  
ret
```

OPERADORES



- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y >= x) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y >= x) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
 mov     eax, DWORD PTR [ebp-8]  
cmp     eax, DWORD PTR [ebp-4]  
jl      .L2  
mov     eax, 1  
jmp     .L3  
.L2:  
mov     eax, 0  
.L3:  
leave  
ret
```

OPERADORES



- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y <= x) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **RELACIONAIS**

```
int x = 10, y = 20;  
if (y <= x) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
 mov     eax, DWORD PTR [ebp-8]  
cmp     eax, DWORD PTR [ebp-4]  
jg      .L2  
mov     eax, 1  
jmp     .L3  
.L2:  
mov     eax, 0  
.L3:  
leave  
ret
```

OPERADORES

- **LÓGICOS**

OPERADOR		DESCRIÇÃO
&&	BINÁRIO	(E LÓGICO) VERDADEIRO SE AMBOS SÃO VERDADEIROS E FALSO NOS DEMAIS CASOS
 	BINÁRIO	(OU LÓGICO) VERDADEIRO SE UM OU AMBOS OS OPERANDOS FOREM VERDADEIROS E FALSO SE AMBOS SÃO FALSOS
!	UNÁRIO	(NÃO LÓGICO) VERDADEIRO SE FALSO E VICE-VERSA

OPERADORES



- **LÓGICOS**

```
int x = 10, y = 20;  
if (y > 0 && x > 0) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **LÓGICOS**

```
int x = 10, y = 20;  
if (y > 0 && x > 0) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
 cmp     DWORD PTR [ebp-8], 0  
jle     .L2  
cmp     DWORD PTR [ebp-4], 0  
jle     .L2  
mov     eax, 1  
jmp     .L3  
.L2:  
mov     eax, 0  
.L3:  
leave  
ret
```

OPERADORES


- **LÓGICOS**

```
int x = 10, y = 20;  
if (y > 0 || x > 0) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **LÓGICOS**

```
int x = 10, y = 20;  
if (y > 0 || x > 0) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
 cmp     DWORD PTR [ebp-8], 0  
jg      .L2  
cmp     DWORD PTR [ebp-4], 0  
jle     .L3  
.L2:  
mov     eax, 1  
jmp     .L4  
.L3:  
mov     eax, 0  
.L4:  
leave  
ret
```

OPERADORES



- **LÓGICOS**

```
int x = 10, y = 20;  
if (!(y > 0 || x > 0)) {  
    return 1;  
} else {  
    return 0;  
}
```

OPERADORES

- **LÓGICOS**

```
int x = 10, y = 20;  
if (!(y > 0 || x > 0)) {  
    return 1;  
} else {  
    return 0;  
}
```

```
 mov     DWORD PTR [ebp-4], 10  
mov     DWORD PTR [ebp-8], 20  
  
 cmp     DWORD PTR [ebp-8], 0  
jg      .L2  
cmp     DWORD PTR [ebp-4], 0  
jg      .L2  
mov     eax, 1  
jmp     .L3  
.L2:  
mov     eax, 0  
.L3:  
leave  
ret
```

CONTROLE DE FLUXO

- **IF-ELSE**

```
if (expressão)
    comando ou bloco
else
    comando ou bloco
```

- **SE A EXPRESSÃO FOR SATISFEITA (VERDADEIRA OU DIFERENTE DE ZERO), O PRIMEIRO COMANDO/BLOCO DE IF É EXECUTADO. CASO CONTRÁRIO, O COMANDO/BLOCO APÓS O ELSE É EXECUTADO.**
- **O ELSE É OPCIONAL**
- **COMO C SIMPLEMENTE TESTA O VALOR NUMÉRICO DE UMA EXPRESSÃO, ALGUNS ATALHOS PODEM SER POSSÍVEIS, COMO:**

```
if (expressão)
    comando ou bloco
else
    comando ou bloco
```

AO INVÉS DE `if (expressão != 0)`

CONTROLE DE FLUXO

- IF-ELSE

```
int x = 0, y = 20;  
if (y > 10)  
    x = 10;  
if (x == 10) {  
    y = y << 1;  
} else {  
    x += y;  
    y += x;  
}  
return 0;
```


CONTROLE DE FLUXO

• IF-ELSE

```
int x = 0, y = 20;
if (y > 10)
    x = 10;
if (x == 10) {
    y = y << 1;
} else {
    x += y;
    y += x;
}
return 0;
```

```
mov     DWORD PTR [ebp-4], 0
mov     DWORD PTR [ebp-8], 20
```

```
cmp     DWORD PTR [rbp-8], 10
jle     .L2
mov     DWORD PTR [rbp-4], 10
```

```
.L2:
```

```
cmp     DWORD PTR [rbp-4], 10
jne     .L3
sal     DWORD PTR [rbp-8]
jmp     .L4
```

```
.L3:
mov     eax, DWORD PTR [rbp-8]
add     DWORD PTR [rbp-4], eax
mov     eax, DWORD PTR [rbp-4]
add     DWORD PTR [rbp-8], eax
```

```
.L4:
mov     eax, 0
pop     rbp
ret
```

CONTROLE DE FLUXO

- **IF-ELSE**

```
if (expressão)
    if (expressão)
        comando ou bloco
else
    comando ou bloco
```

- **EXISTE UMA AMBIGUIDADE QUANDO OCORREM IFS ANINHADOS. NESSES CASOS, O ELSE, SE HOUVER, SERÁ ASSOCIADO AO IF EM ABERTO MAIS PRÓXIMO**

CONTROLE DE FLUXO

- **ELSE-IF**

```
if (expressão)
    comando ou bloco
else if (expressão)
    comando ou bloco
else if (expressão)
    comando ou bloco
else
    comando ou bloco
```

- **O ELSE PODE SER OMITIDO OU USADO PARA CHECAGEM DE ERROS, NO CASO DA VERIFICAÇÃO DE UMA SITUAÇÃO IMPOSSÍVEL**

CONTROLE DE FLUXO



- **ELSE-IF**

```
unsigned int x = 30;  
if ( x >= 1 || x <= 10)  
    x += 10;  
else if (x >= 11 || x <= 20)  
    x += 20;  
else  
    x += 30;  
return 0;
```

CONTROLE DE FLUXO

- **ELSE-IF**

```
unsigned int x = 30;  
if ( x >= 1 || x <= 10)  
    x += 10;  
else if (x >= 11 || x <= 20)  
    x += 20;  
else  
    x += 30;  
return 0;
```

```
 mov     DWORD PTR [ebp-4], 30  
 cmp     DWORD PTR [ebp-4], 0  
jne     .L2  
cmp     DWORD PTR [ebp-4], 10  
ja      .L3  
.L2:  
add     DWORD PTR [ebp-4], 10  
jmp     .L4  
.L3:  
cmp     DWORD PTR [ebp-4], 10  
ja      .L5  
cmp     DWORD PTR [ebp-4], 20  
ja      .L6  
.L5:  
add     DWORD PTR [ebp-4], 20  
jmp     .L4  
.L6:  
add     DWORD PTR [ebp-4], 30  
.L4:  
mov     eax, 0  
leave  
ret
```

CONTROLE DE FLUXO

- **WHILE**

`while (expressão)`
`comando ou bloco`

- **SE A EXPRESSÃO FOR SATISFEITA (VERDADEIRA OU DIFERENTE DE ZERO), O COMANDO/BLOCO É EXECUTADO E A EXPRESSÃO É REAVALIADA. O CICLO CONTINUA ENQUANTO A EXPRESSÃO FOR SATISFEITA**

```
unsigned int soma = 0, i = 0;  
while (i <= 10) {  
    soma += i;  
    i++;  
}
```

CONTROLE DE FLUXO

- **WHILE**

while (expressão)
comando ou bloco

- **SE A EXPRESSÃO FOR SATISFEITA (VERDADEIRA), O COMANDO/BLOCO É EXECUTADO E O CICLO CONTINUA ENQUANTO A EXPRESSÃO FOR SATISFEITA (VERDADEIRA),**

```
unsigned int soma = 0, i = 0;  
while (i <= 10) {  
    soma += i;  
    i++;  
}
```

```
mov     DWORD PTR [ebp-4], 0  
mov     DWORD PTR [ebp-8], 0  
  
.L3:  
cmp     DWORD PTR [ebp-8], 10  
ja      .L2  
mov     eax, DWORD PTR [ebp-8]  
add     DWORD PTR [ebp-4], eax  
add     DWORD PTR [ebp-8], 1  
jmp     .L3  
.L2:  
mov     eax, 0  
leave  
ret
```

CONTROLE DE FLUXO

- **FOR**

```
for (expressão1; expressão2; expressão3)  
    comando ou bloco
```

- **É EQUIVALENTE A**

```
expressão1;  
while (expressão2) {  
    comando ou bloco;  
    expressão3;  
}
```

- **NORMALMENTE, expressão1 E expressão3 SÃO ATRIBUIÇÕES OU CHAMADAS DE FUNÇÃO E expressão2 É UMA EXPRESSÃO RELACIONAL**
- **QUALQUER UMA DESSAS PARTES PODE SER OMITIDA (OS ; DEVEM PERMANECER)**

CONTROLE DE FLUXO

- **FOR**

```
unsigned int soma, i;  
soma = 0;  
for (i = 0; i <= 10; i++) {  
    soma += i;  
}  
for ( ; ; );
```

- **EXPRESSÕES PODEM SER SEPARADAS POR VÍRGULAS**

```
for (i = 0, j = strlen(s) - 1; i < j; ++i, j--) {  
    ...  
}
```



CONTROLE DE FLUXO

- **FOR**

```
unsigned int soma, i;  
soma = 0;  
for (i = 0; i <= 10; i++) {  
    soma += i;  
}  
for ( ; ; );
```

- **EXPRESSÕES PODEM SER SEPARADAS POR VÍRGULAS**

```
for (i = 0, j = strlen(s) - 1; i < j; ++i, j--) {  
    ...  
}
```

```
 mov     DWORD PTR [ebp-4], 0  
 mov     DWORD PTR [ebp-8], 0  
.L3:  
    cmp     DWORD PTR [ebp-8], 10  
    ja      .L2  
    mov     eax, DWORD PTR [ebp-8]  
    add     DWORD PTR [ebp-4], eax  
    add     DWORD PTR [ebp-8], 1  
    jmp     .L3  
.L2:  
    jmp     .L2
```

CONTROLE DE FLUXO

- **BREAK**

```
int soma, i;  
soma = 0;  
for (i = 0; i <= 10; i++) {  
    if (i == 7) break;  
    soma += i;  
}
```










- **USADO PARA FORÇAR A SAÍDA DE UM LAÇO DE REPETIÇÃO**

CONTROLE DE FLUXO

- **BREAK**

```
int soma, i;  
soma = 0;  
for (i = 0; i <= 10; i++) {  
    if (i == 7) break;  
    soma += i;  
}
```

- **USADO PARA FORÇAR A SAÍDA DE UM**

```
 mov     DWORD PTR [ebp-4], 0  
 mov     DWORD PTR [ebp-8], 0  
  
 .L4:  
 cmp     DWORD PTR [ebp-8], 10  
 jg      .L2  
 cmp     DWORD PTR [ebp-8], 7  
 je      .L6  
 mov     eax, DWORD PTR [ebp-8]  
 add     DWORD PTR [ebp-4], eax  
 add     DWORD PTR [ebp-8], 1  
 jmp     .L4  
  
.L6:  
 nop  
  
.L2:  
 mov     eax, 0  
 leave  
 ret
```

CONTROLE DE FLUXO

- **CONTINUE**

```
int soma, i;  
soma = 0;  
for (i = 0; i <= 10; i++) {  
    if (i % 2) continue;  
    soma += i;  
}
```

- **PULA PARA A PRÓXIMA ITERAÇÃO DO LAÇO, IGNORANDO OS COMANDOS SUBSEQUENTES DO BLOCO**

CONTROLE DE FLUXO

- **CONTINUE**

```
int soma, i;  
soma = 0;  
for (i = 0; i <= 10; i++) {  
    if (i % 2) continue;  
    soma += i;  
}
```

- **PULA PARA A PRÓXIMA ITERAÇÃO DO SUBSEQUENTES DO BLOCO**

```
mov     DWORD PTR [ebp-4], 0  
mov     DWORD PTR [ebp-8], 0  
  
.L5:  
cmp     DWORD PTR [ebp-8], 10  
jg      .L2  
mov     eax, DWORD PTR [ebp-8]  
and     eax, 1  
test    eax, eax  
jne     .L7  
mov     eax, DWORD PTR [ebp-8]  
add     DWORD PTR [ebp-4], eax  
jmp     .L4  
  
.L7:  
nop  
  
.L4:  
add     DWORD PTR [ebp-8], 1  
jmp     .L5  
  
.L2:  
mov     eax, 0  
leave  
ret
```

EXERCÍCIOS

<https://github.com/thiagopeixoto/curso-c-reversing>

Mais exercícios:

<https://www.thehuxley.com>

DICAS

- **UM TÍPICO PROGRAMA EM C**

```
#include <stdio.h>
```

```
int main(void)  
{  
    return 0;  
}
```

- **EXIBINDO UMA MENSAGEM NA TELA**

```
printf("Sua mensagem aqui\n");
```

- **EXIBINDO UMA MENSAGEM + PARÂMETROS INTEIROS**

```
printf("Sua mensagem aqui %d %d\n", var1, var2);
```


DICAS

- EXIBINDO UMA MENSAGEM + PARÂMETROS FLOAT COM 2 CASAS DECIMAIS

```
printf("Sua mensagem aqui %.2f\n", var1);
```

- LEITURA DE UM NÚMERO INTEIRO DO TECLADO

```
scanf("%d", &var1);
```

- LEITURA DE UM NÚMERO FLOAT DO TECLADO

```
scanf("%f", &var1);
```



“VAMBORA, CUMPADI! BIIIRRL!!”