

Mestrado em Informática

2007/08

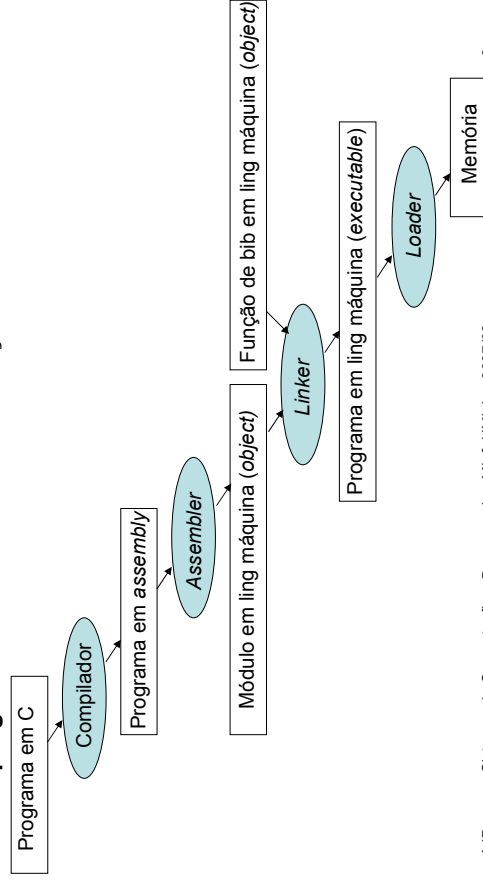
A.J.Proença

Tema

Revisitando os Sistemas de Computação (1)

Execução de programas num computador (2)

De um programa em HLL até à sua execução:



Estrutura do tema RSC

1. Execução de programas num computador
2. Suporte a estruturas de controlo
3. Suporte à invocação/retorno de funções
4. Acesso e manipulação de dados estruturados

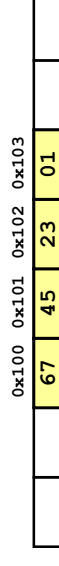
Representação de operandos no IA32

Tamanhos de objectos em C (em bytes)

Declaração em C	Designação Intel	Tamanho IA32
char	byte	1
short	word	2
int	double word	4
long int	double word	4
float	single precision	4
double	double precision	8
long double	extended precision	10/12
char * (ou qq outro apontador)	double word	4

Ordenação dos bytes na memória

- O IA32 é um processador *little endian*
- Exemplo: representação de 0x01234567, cujo endereço dado por *xx* é 0x100



Execução de programas num computador (1)



- **Código C**
 - somar 2 inteiros (c/ sinal)
- **Assembly**
 - somar 2 inteiros de 4-bytes
 - operandos "long" em GCC
 - a mesma instrução, c/ ou s/ sinal
 - operandos:
 - x: em registo %eax
 - y: na memória M[%ebp+8]
 - t: em registo %eax
- **Código object**
 - instrução com 3-bytes
 - na memória em 0x401046

```
int t = x+y;
```

```
addl 8(%ebp), %eax
```

Idêntico à expressão
x += y

```
0x401046: 03 45 08
```

Tipos de instruções básicas no IA32



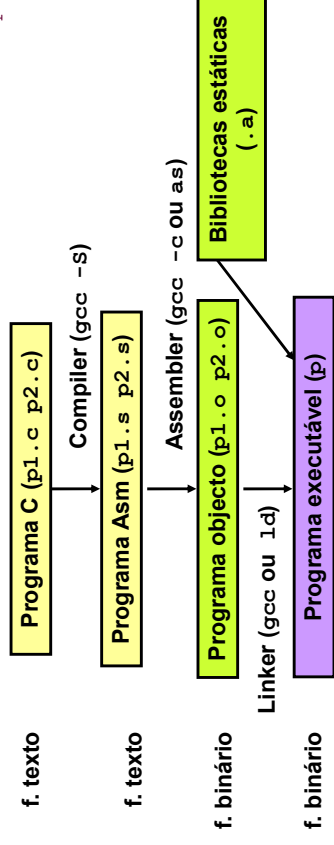
Operações primitivas:

- Operações aritméticas/lógicas com dados em registo ou em memória
 - dados do tipo *integer* de 1, 2 ou 4 bytes
 - dados em formato *fp* de 4, 8 ou 10 bytes
 - apenas dados escalares: *arrays* ou *structures* são vistos apenas como *bytes* continuamente alocados em memória
- Transferir dados entre células de memória e um registo
 - carregar (*load*) em registo dados da memória
 - armazenar (*store*) na memória dados em registo
- Transferir o controlo da execução das instruções
 - saltos incondicionais de/para funções/procedimentos
 - saltos ramificados (*branches*) condicionais

Conversão de um programa em C em código executável (exemplo)



- Código C nos ficheiros **p1.c p2.c**
- Comando para a "compilação": **gcc -O p1.c p2.c -o p**
 - usa optimizações (-O)
 - coloca binário resultante no ficheiro p



A compilação de C para assembly (exemplo)



Código C

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Assembly gerado

```
_sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
```

gcc -O -S p2.c

p2.s

De assembly para binário e executável (exemplo)

Assembly

```

_sum:
  pushl %ebp
  movl  %esp, %ebp
  movl  12(%ebp), %eax
  addl  8(%ebp), %eax
  movl  %ebp, %esp
  popl  %ebp
  ret

```

p2.s

p2.o

Código binário

```

0x401040 <sum>:
  0x55
  0x89
  0xe5 • Começa no
  0x8b endereço
  0x45 0x401040
  0x0c
  0x03
  0x45 • Total 13
  0x08 bytes
  0x89
  0xec
  0x5d • Cada
  0xc3 instrução

```

Papel do linker

- Resolve as referências entre ficheiros
- Junta as *static run-time libraries*
 - E.g., código para `malloc, printf`
- Algumas bibliotecas são *dynamically linked*
 - E.g., junção ocorre no início da execução

Método alternativo de análise do código binário executável (exemplo)

Entrar primeiro no depurador `gdb`: **`gdb p e...`**

- examinar apenas alguns bytes: **`x/13b sum`**

```

0x401040<sum>:  0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45
0x401040<sum+8>: 0x08 0x89 0xec 0x5d 0xc3

```

... OU

• proceder à desmontagem do código: **`disassemble sum`**

```

0x401040 <sum>:  push  %ebp
0x401041 <sum+1>:  mov   %esp, %ebp
0x401043 <sum+3>:  mov   0xc(%ebp), %eax
0x401046 <sum+6>:  add  0x8(%ebp), %eax
0x401049 <sum+9>:  mov  %ebp, %esp
0x40104b <sum+11>: pop  %ebp
0x40104c <sum+12>: ret
0x40104d <sum+13>: lea  0x0(%esi), %esi

```

Desmontagem de código binário executável (exemplo)

`objdump -d p`

Código binário desmontado

```

00401040 <_sum>:
  0: 55          push  %ebp
  1: 89 e5      mov   %esp, %ebp
  3: 8b 45 0c   mov   0xc(%ebp), %eax
  6: 03 45 08   add  0x8(%ebp), %eax
  9: 89 ec      mov  %ebp, %esp
  b: 5d        pop  %ebp
  c: c3        ret
  d: 8d 76 00   lea  0x0(%esi), %esi

```

Que código pode ser desmontado?

Qualquer ficheiro que possa ser interpretado como código executável
 - o disassembler examina os *bytes* e reconstrói a fonte *assembly*

```

% objdump -d WINWORD.EXE
WINWORD.EXE:  file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000: 55          push  %ebp
30001001: 8b ec      mov   %esp, %ebp
30001003: 6a ff      push $0xffffffff
30001005: 68 90 10 30 push $0x30001090
3000100a: 68 91 dc 4c push $0x304cdc91

```

Estrutura do tema RSC

1. Execução de programas num computador
2. Suporte a estruturas de controlo
3. Suporte à invocação/retorno de funções
4. Acesso e manipulação de dados estruturados

Suporte a estruturas de controlo do C

- Estruturas de controlo do C
 - **if-else statement**
 - **do-while statement**
 - **while statement**
 - **for loop**
 - **switch statement**

- Por defeito, as instruções são sempre executadas sequencialmente, i.e., uma após outra (em HLL & em ling. máq.)
- Em HLL o fluxo de instruções poderá ser alterado:
 - na execução de estruturas de controlo (adiante...)
 - na invocação / retorno de funções (mais adiante...)
 - na ocorrência de excepções / interrupções (mais adiante?)
- Em linguagem máquina isso traduz-se na alteração do IP, de modo incondic/condicional, por um valor absoluto/relativo
 - `jump / branch`
 - `call` (com salvaguarda do endereço de retorno) e `ret`
 - em excepções / interrupções ...

if-else statement

Generalização

```
if (expressão_de_teste)
  then_statement
else
  else_statement
```

Forma genérica em C

```
cond = expressão_de_teste
if (cond)
  goto true;
else_statement
goto done;
true:
  then_statement
done:
```

Versão com goto, ou
assembly com sintaxe C

Generalização

```
do
  body_statement
while (expressão_de_teste);
```

Forma genérica em C

```
Loop:
  body_statement
  cond = expressão_de_teste
  if (cond)
    goto loop;
```

Versão com goto, ou assembly com sintaxe C

Generalização

```
while (expressão_de_teste)
  body_statement
```

Forma genérica em C

```
Loop:
  cond = expressão_de_teste
  if (!cond)
    goto done;
  body_statement
  goto loop;
done:
```

Conversão while em do-while

```
if (!expressão_de_teste)
  goto done;
do
  body_statement
while (expressão_de_teste);
done:
```

Versão do-while com goto

```
cond = expressão_de_teste
if (!cond)
  goto done;
Loop:
  body_statement
  cond = expressão_de_teste
  if (cond)
    goto loop;
done:
```

Versão do-while com goto

Generalização

```
for (expr_inic; expr_test; act_expr)
  body_statement
```

Forma genérica em C

```
expr_inic;
while (expr_test) {
  body_statement
  act_expr;
}
```

Conversão for em while

```
expr_inic ;
cond = expr_test ;
if (!cond)
  goto done;
Loop:
  body_statement
  act_expr ;
  cond = expr_test ;
  if (cond)
    goto loop;
done:
```

Conversão para do-while

```
expr_inic ;
if (!expr_test)
  goto done;
do {
  body_statement
  act_expr ;
} while (expr_test);
done:
```

Versão do-while com goto

for loop

"Salto" com escolha múltipla; alternativas de implementação:

- Sequência de if-else statements
- Com saltos "indirectos": endereços especificados numa tabela de salto (jump table)



Estrutura do tema RSC

1. Execução de programas num computador
2. Suporte a estruturas de controlo
3. Suporte à invocação/retorno de funções
4. Acesso e manipulação de dados estruturados

Suporte a funções e procedimentos no IA32 (2)



Análise do código de gestão de uma função

- **invocação e retorno**
 - instrução de salto, mas com salvaguarda do end. retorno
 - em registo (RISC; aninhamento / recursividade ?)
 - em memória/stack (IA32; aninhamento / recursividade ?)
- **invocação e retorno**
 - instrução de salto para o endereço de retorno
- **salvaguarda & recuperação de registos (na stack)**
 - função chamadora ? (nenhum/ alguns/ todos ? RISC/IA32 ?)
 - função chamada? (nenhum/ alguns/ todos ? RISC/IA32 ?)
- **gestão do contexto (em stack)**
 - actualização/recuperação do *frame pointer* (IA32...)
 - reserva/libertação de espaço para variáveis locais



Estrutura de uma função (/ procedimento)

– parte visível ao programador em HLL

- código do corpo da função
- alcance das variáveis: locais ou globais
- passagem de argumentos para a função ...
... e valor de retorno da função

– parte menos visível em HLL:

a gestão do contexto da função

- variáveis globais (localização e acesso)
- argumentos e valor de retorno (propriedades)
- variáveis locais (propriedades)
- gestão do contexto (controlo & dados)

Análise dos contextos em swap, no IA32

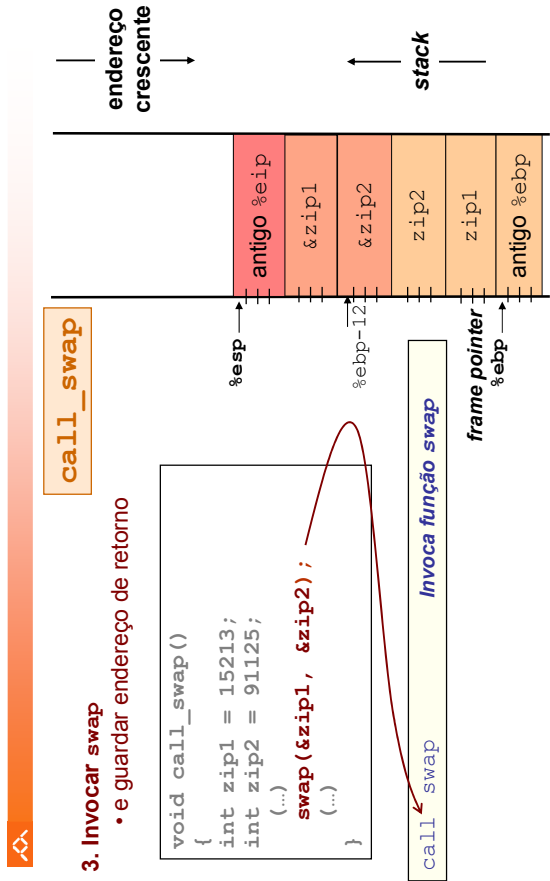


```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

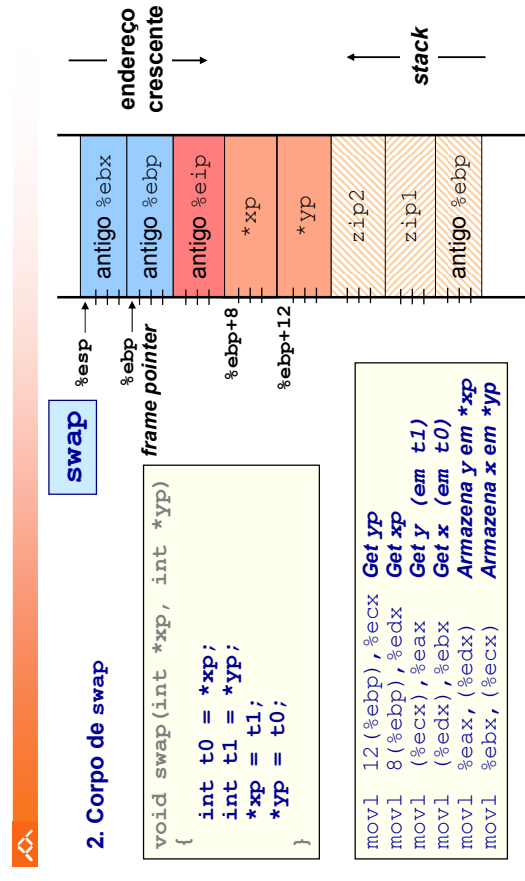
void call_swap()
{
  int zip1 = 15213;
  int zip2 = 91125;
  (...)
  swap(&zip1, &zip2);
  (...)
}
```

- em `call_swap`
- na invocação de `swap`
- na execução de `swap`
- de volta a `call_swap`

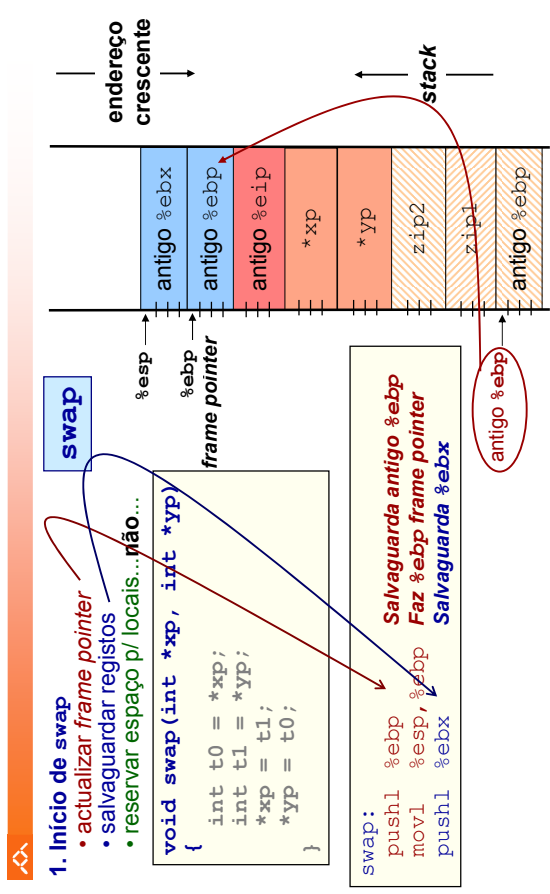
Evolução da stack, no IA32 (4)



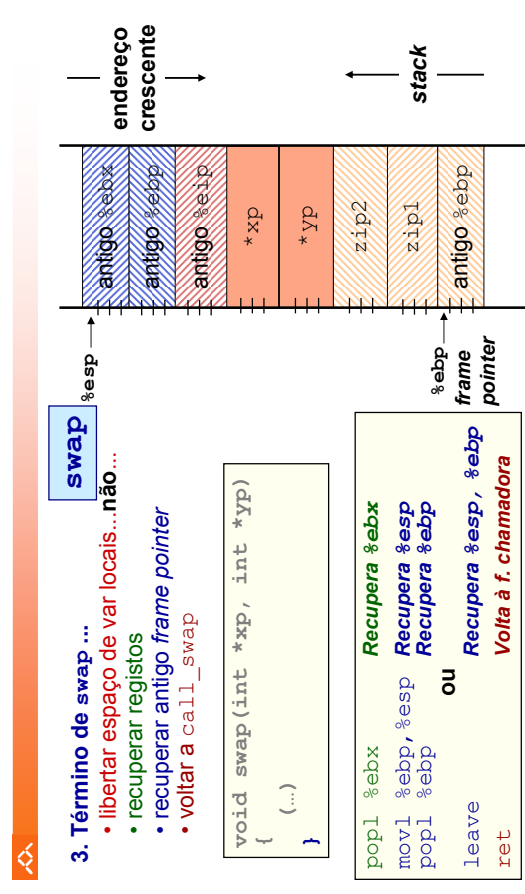
Evolução da stack, no IA32 (6)



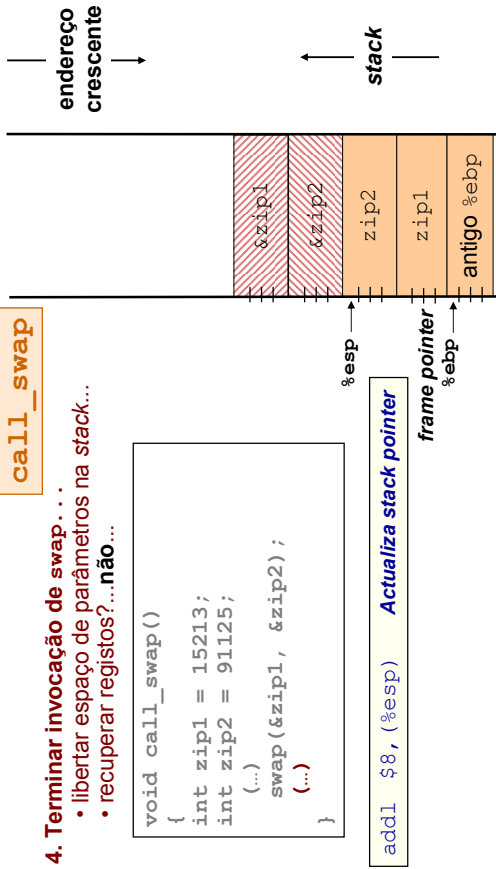
Evolução da stack, no IA32 (5)



Evolução da stack, no IA32 (7)



Evolução da stack, no IA32 (8)



call_swap

4. Terminar invocação de swap...

- libertar espaço de parâmetros na stack...
- recuperar registos?...**não**...

```
void call_swap()
{
    int zip1 = 15213;
    int zip2 = 91125;
    (...)
    swap(&zip1, &zip2);
    (...)
}
```

```
addl $8, (%esp) Atualiza stack pointer
```

```
frame pointer
%ebp →
```

A série de Fibonacci no IA32 (2)

```
função recursiva
int fib_rec (int n)
{
    int prev_val, val;
    if (n<=2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```

```
__fib_rec:
    pushl %ebp
    movl %esp, %ebp
    subl $12, %esp
    movl %ebx, -8(%ebp)
    movl %esi, -4(%ebp)
    movl 8(%ebp), %esi
```

Atualiza frame pointer

Reserva espaço na stack para 3 int's
Salvaguarda os 2 reg's que vão ser usados;
de notar a forma de usar a stack...

A série de Fibonacci no IA32 (1)

```
int fib_dw(int n)
{
    int i = 0;
    int val = 0;
    int nval = 1;
    do {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    } while (i<n);
    return val;
}
```

do-while

```
int fib_f(int n)
{
    int i;
    int val = 1;
    int nval = 1;
    for (i=1; i<n; i++) {
        int t = val + nval;
        val = nval;
        nval = t;
    }
    return val;
}
```

while

```
int fib_w(int n)
{
    int i = 1;
    int val = 1;
    int nval = 1;
    while (i<n) {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    }
    return val;
}
```

função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n<=2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```

A série de Fibonacci no IA32 (3)

```
função recursiva
int fib_rec (int n)
{
    int prev_val, val;
    if (n<=2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```

```
...
movl %esi, -4(%ebp)
movl 8(%ebp), %esi
movl $1, %eax
cmpl $2, %esi
jle l1
leal -2(%esi), %eax
...
l1:
movl -8(%ebp), %ebx
```

Coloca o argumento `n` em `%esi`
Coloca já o valor de retorno em `%eax`
`n<=2`?
Se sim, salta para o fim
Se não, ...

A série de Fibonacci no IA32 (4)

função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n<=2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```

```
...
jle I1
leal -2(%esi), %eax
movl %eax, (%esp)
call fib_rec
movl %eax, %ebx
leal -1(%esi), %eax
...
```

Se sim, salta para o fim
Se não, ... calcula n-2, e...
... coloca-o no topo da stack (argumento)
Invoca a função fib_rec e ...
... guarda o valor de prev_val em %ebx

A série de Fibonacci no IA32 (5)

função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n<=2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```

```
...
movl %eax, %ebx
leal -1(%esi), %eax
movl %eax, (%esp)
call fib_rec
leal (%eax,%ebx), %eax
...
```

Calcula n-1, e...
... coloca-o no topo da stack (argumento)
Chama de novo a função fib_rec

A série de Fibonacci no IA32 (6)

função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n<=2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```

```
...
call fib_rec
leal (%eax,%ebx), %eax
I1:
movl -8(%ebp), %ebx
movl -4(%ebp), %esi
movl %ebp, %esp
popl %ebp
ret
```

Calcula e coloca em %eax o valor de retorno
Recupera o valor dos 2 reg's usados
Atualiza o valor do stack pointer
Recupera o anterior valor do frame pointer

Revisitando os Sistemas de Computação

Estrutura do tema RSC

1. Execução de programas num computador
2. Suporte a estruturas de controlo
3. Suporte à invocação/retorno de funções
4. Acesso e manipulação de dados estruturados

Propriedades dos dados estruturados em C

- agregam quantidades escalares do mesmo tipo ou de tipos diferentes
- sempre alocadas a posições contíguas da memória
- a estrutura definida pode ser referenciada pelo apontador para a 1ª posição de memória

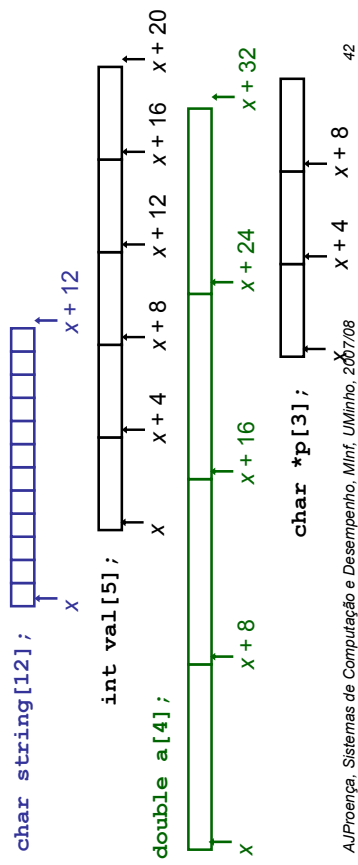
Tipos de dados estruturados mais comuns em C

- **array**: agregado de dados escalares do mesmo tipo
 - *string*: array de caracteres terminado com *null*
 - arrays de arrays: arrays multi-dimensionais
- **structure**: agregado de dados de tipos diferentes
 - structures de structures, structures de arrays, ...
- **union**: mesmo objecto mas com visibilidade distinta

Declaração em C:

```
data_type Array_name[length];
```

Alocação em memória de uma região com $length * sizeof(data_type)$ bytes

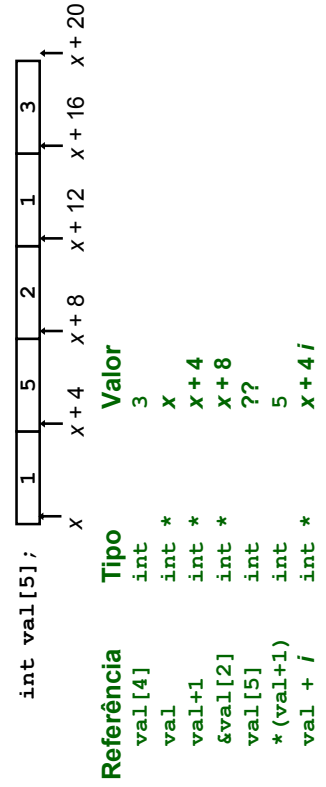


Arrays: acesso aos elementos

Declaração em C:

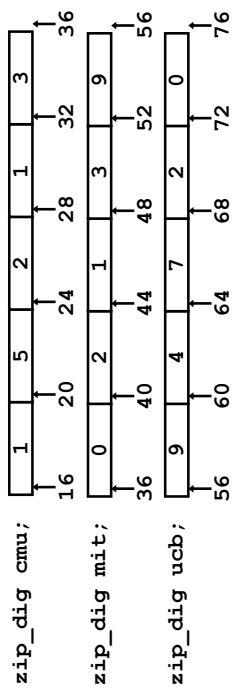
```
data_type Array_name[length];
```

O identificador **Array_name** pode ser usado como apontador para o elemento 0



Arrays: análise de um exemplo

```
typedef int zip_dig[5];
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Notas

- declaração "zip_dig cmu" equivalente a "int cmu[5]"
- os arrays deste exemplo ocupam blocos sucessivos de 20 bytes

Arrays: exemplo de acesso a um elemento

```
int get_digit(zip_dig z, int dig)
{
    return z[dig];
}
```

Argumentos:

- tipo `int` (4 bytes)
- início do `array z` (colocado em `%edx`)
- índice `dig` do `array z` (colocado em `%eax`)

Localização do elemento `z [dig]`:

- `Mem [(%edx) + 4 * (%eax)]`
- IA32/Linux: `(%edx, %eax, 4)`

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z [dig]
```

Arrays: apontadores em vez de índices (1)

```
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

Código original

com referências a `arrays` dentro de ciclos

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```

Transformação pelo GCC

- eliminou a variável `i`
- converteu índices em apontadores
- reduziu à forma `do-while`

Arrays: apontadores em vez de índices (2)

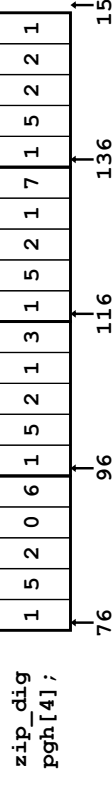
```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```

- **Registos**
 - `%ecx` `z`
 - `%eax` `zi`
 - `%ebx` `zend`
- **Cálculos**
 - `10*zi + *z ==>`
 - `z++` incrementa 4

```
# %ecx = z
xorl %eax,%eax # zi = 0
leal 16(%ecx),%ebx # zend = z+4
.I59:
leal (%eax,%eax,4),%edx # 5*zi
movl (%ecx),%eax # *z
addl $4,%ecx # z++
leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
cmpl %ebx,%ecx # z : zend
jle .I59 # if <= goto loop
```

Array de arrays: análise de um exemplo

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3},
 {1, 5, 2, 1, 7},
 {1, 5, 2, 2, 1}};
```



```
zip_dig
pgh[4];
```

- Declaração “`zip_dig pgh[4]`” equivalente a “`int pgh[4][5]`”
 - variável `pgh` é um `array` de 4 elementos
 - alocados em memória em blocos contíguos
 - cada elemento é um `array` de 5 `int`'s
 - alocados em memória em células contíguas
- Ordenação dos elementos (garantido em C): “**Row-Major**”

Array de arrays: alocação em memória

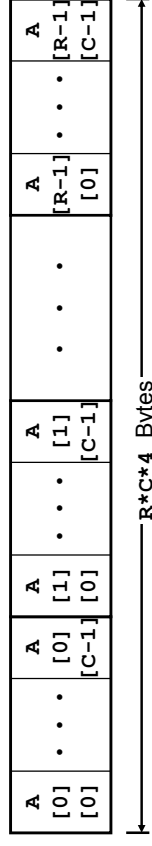
Declaração em C:

```
data_type Array_name [R] [C];
```

- Alocação em memória de uma região com $R * c * \text{sizeof}(\text{data_type})$ bytes
- Ordenação Row-Major

```
A[0][0] ... A[0][C-1]
.
.
.
A[R-1][0] ... A[R-1][C-1]
```

```
int A[R][C];
```

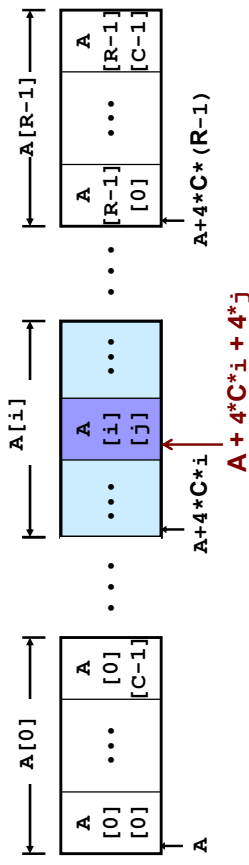


Array de arrays: acesso a um elemento

Elementos de um array R*C

- $A[i][j]$ é um elemento do tipo T (data_type) com dimensão $K = \text{sizeof}(T)$
- sua localização: $A + K * C * i + K * j$

```
int A[R][C];
```



Array de arrays: alocação em memória

Array de arrays: código para acesso a um elemento

• Localização em memória de

```
pgh[index][dig]:
{
    return pgh[index][dig];
}
```

$pgh + 20 * \text{index} + 4 * \text{dig}$

• Código em assembly:

– cálculo do endereço
 $pgh + 4 * \text{dig} + 4 * (\text{index} + 4 * \text{index})$
 – acesso ao elemento: com movl

```
# %ecx = dig
# %eax = index
leal 0(, %ecx, 4), %edx # 4*dig
leal (%eax, %eax, 4), %eax # 5*index
movl pgh(%edx, %eax, 4), %eax # *(pgh + 4*dig + 20*index)
```

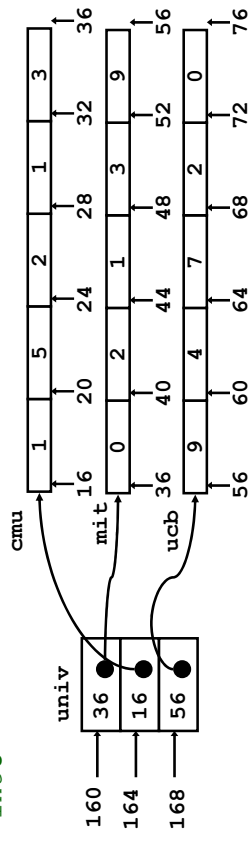
Array de apontadores para arrays: uma visão alternativa

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

• Cada elemento:

– um apontador de 4 bytes
 – aponta para um array de int's

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```



Array de apontadores para arrays: acesso a um elemento

```
int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}
```

Cálculo da localização

- para acesso a um elemento
Mem[Mem[univ+4*index]+4*dig]
- requer 2 acessos à memória
 - para buscar apontador para row array
 - para aceder a elemento do row array

```
# %ecx = index
# %eax = dig
leal 0(, %ecx, 4), %edx # 4*index
movl univ(%edx), %edx # Mem[univ+4*index]
movl (%edx, %eax, 4), %eax # Mem[...+4*dig]
```

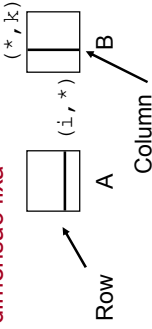
Arrays multi-dimensionais de tamanho fixo: a eficiência do compilador (1)

• Oportunidades para otimizar

- o array *a* está em localizações contíguas, começando em *a[i][0]*: usar apontador!
- o array *b* está em localizações espaçadas de $4*N$ células, começando em *b[0][j]*: usar também apontador!

• Limitações

- apenas funciona com arrays de dimensão fixa



```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

Array de arrays versus array de apontadores para arrays

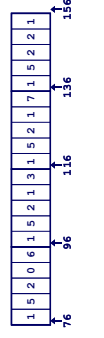
Modos distintos de cálculo da localização dos elementos:

```
int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}
```

```
int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}
```

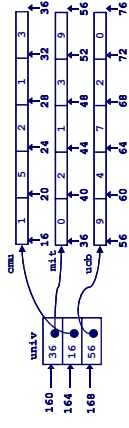
Array de arrays

- elemento em
Mem[pgh+20*index+4*dig]



Array de apontadores para arrays

- elemento em
Mem[Mem[univ+4*index]+4*dig]



Arrays multi-dimensionais de tamanho fixo: a eficiência do compilador (2)

• Otimizações automáticas do compilador:

- antes...
- depois...

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

```
/* Compute element i,k ... */
int fix_prod_ele (...)
{
    int *Aptr = &a[i][0];
    int *Bptr = &b[0][k];
    int cnt = N-1;
    int result = 0;
    do {
        result += (*Aptr)*(*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    }while (cnt>=0);
    return result;
}
```

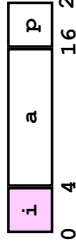
Structures: noções básicas

Propriedades

- em regiões contíguas da memória
- membros podem ser de tipos diferentes
- membros acessados por nomes

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

Organização na memória



Acesso a um membro da structure

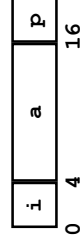
```
void
set_i(struct rec *r,
      int val)
{
    r->i = val;
}
```

```
# %eax = val
# %edx = r
movl %eax, (%edx) # Mem[r] = val
```

Structures: apontadores para membros (2)

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

```
void set_p(struct rec *r)
{
    r->p = &r->a[r->i];
}
```

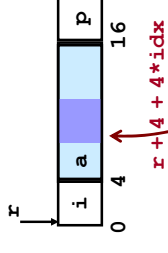


```
# %edx = r
movl (%edx), %ecx # r->i
leal 0(, %ecx, 4), %eax # 4*(r->i)
leal 4(%edx, %eax), %eax # r+4+4*(r->i)
movl %eax, 16(%edx) # Update r->p
```

Structures: apontadores para membros (1)

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

```
int *
find_a
(struct rec *r, int idx)
{
    return &r->a[idx];
}
```



Valor calculado
na compilação

```
# %ecx = idx
# %edx = r
leal 0(, %ecx, 4), %eax # 4*idx
leal 4(%eax, %edx), %eax # r+4*idx+4
```

Alinhamento de dados na memória

Dados alinhados

- Tipos de dados primitivos (escalares) requerem K bytes
- Endereço deve ser múltiplo de K
- Requisito nalgumas máquinas; aconselhado no IA32
 - tratado de modo diferente, consoante Linux ou Windows!

Motivação para alinhar dados

- Memória acessada por *double* ou *quad-words* (alinhada)
 - ineficiente lidar com dados que passam esses limites
 - ainda mais crítico na gestão da memória virtual (limite da página!)

Compilador

- Inserir bolhas na *structure* para garantir o correcto alinhamento dos campos

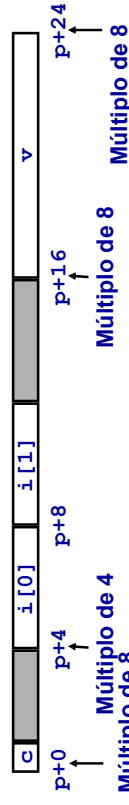
Alinhamento de dados na memória: os dados primitivos/escalares

- 1 byte (e.g., char)
- sem restrições no endereço
- 2 bytes (e.g., short)
 - o bit menos significativo do endereço deve ser 0₂
- 4 bytes (e.g., int, float, char *, etc.)
 - os 2 bits menos significativo do endereço devem ser 00₂
- 8 bytes (e.g., double)
 - Windows (e a maioria dos SO's & instruction sets):
 - os 3 bits menos significativo do endereço devem ser 000₂
 - Linux:
 - os 2 bits menos significativo do endereço devem ser 00₂
 - i.e., mesmo tratamento que um dado escalar de 4 bytes
- 12 bytes (long double)
 - Linux:
 - os 2 bits menos significativo do endereço devem ser 00₂
 - i.e., mesmo tratamento que um dado escalar de 4 bytes

Alinhamento de dados na memória: Windows versus Linux

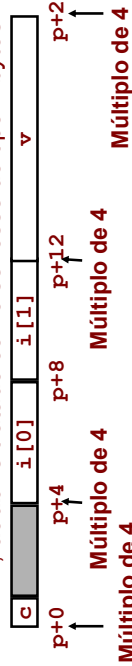
- **Windows** (incluindo Cygwin):

- K = 8, devido ao elemento double



- **Linux:**

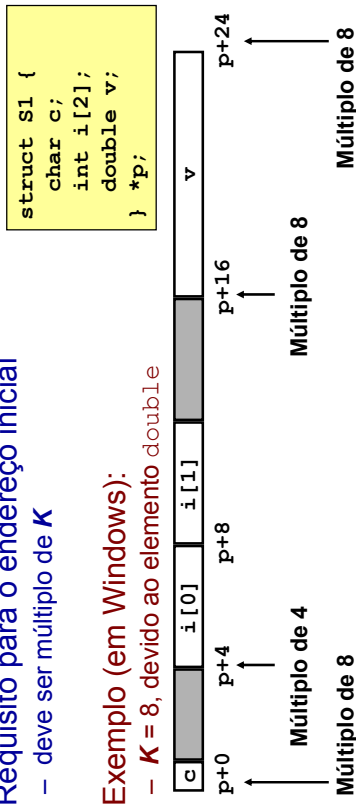
- K = 4; double tratado como se fosse do tipo 4-bytes



Alinhamento de dados na memória: nas structures

- Deslocamentos dentro da *structure*
 - deve satisfazer os requisitos de alinhamento dos elementos (i.e., do seu maior elemento, K)
- **Requisito para o endereço inicial**
 - deve ser múltiplo de K

- **Exemplo (em Windows):**
 - K = 8, devido ao elemento double



Alinhamento de dados na memória: ordenação dos membros

- **10 bytes espaço desperdiçado no Windows**

```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
    *p;
}
```



```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
    *p;
}
```

- **apenas 2 bytes de espaço desperdiçado**

