

## Mestrado em Informática

2010/11

A.J.Proença

### Tema

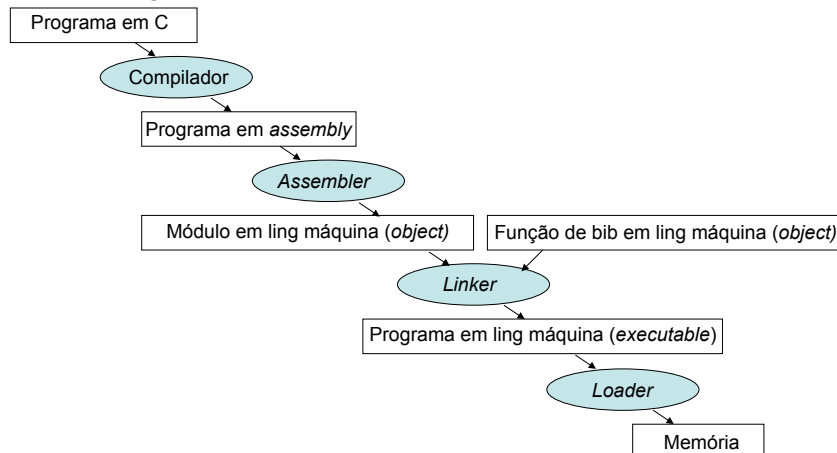
### Revisitando os Sistemas de Computação (1)

## 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

### Execução de programas num computador (2)

#### De um programa em HLL até à sua execução:



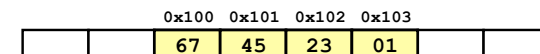
### Representação de operandos no IA32

#### 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 &x é 0x100



`int t = x+y;`

- **Código C**
  - somar 2 inteiros (c/ sinal)

`addl 8(%ebp), %eax`

- **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

Idêntico à expressão `x += y`

`0x401046: 03 45 08`

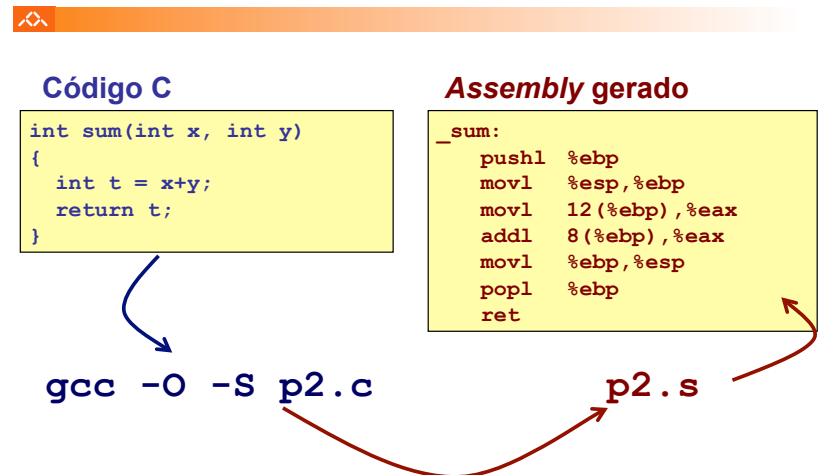
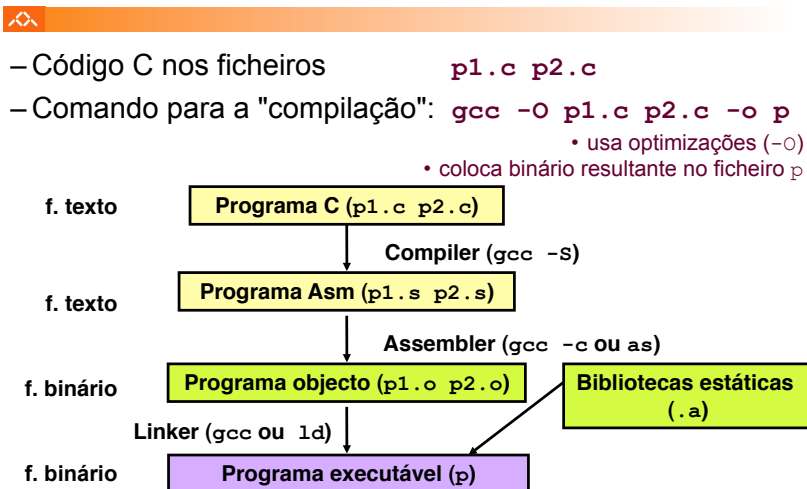
- **Código object**
  - instrução com 3-bytes
  - na memória em `0x401046`

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)

A compilação de C para assembly (exemplo)



**Assembly**

```

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

**Código binário**

```

0x401040 <sum>:
0x55
0x89
0xe5 • Começa
0x8b no
0x45 endereço
0x0c 0x401040
0x03
0x45 • Total 13
0x08 bytes
0x89
0xec
0x5d • Cada
0xc3 instrução
      1, 2, ou 3
      bytes
    
```

*p2.s* → *p2.o*

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

`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
    
```

Método alternativo de análise do  
código binário executável (exemplo)

Que código  
pode ser desmontado?

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
    
```

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 00 30  push  $0x30001090
3000100a:  68 91 dc 4c 30  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



- Por omissão, 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 . . .

## Suporte a estruturas de controlo do C



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

## `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:
```

**Versão com goto**

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

**Conversão while em do-while**

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

**Conversão para do-while**

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

**Versão do-while com goto**

**switch statement**

**"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

## 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 do código de gestão de uma função

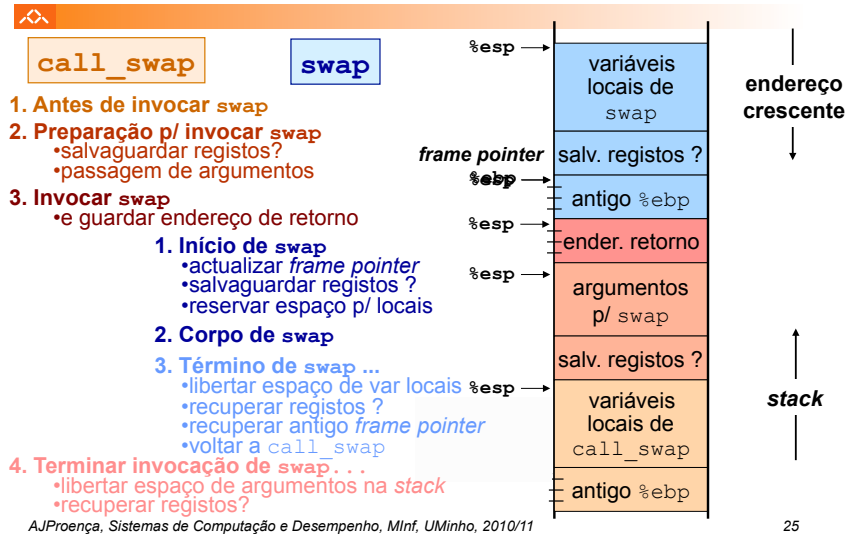
- invocação e retorno
  - instrução de salto, mas com salvaguarda do end. retorno
    - em registo (RISC; aninhamento / recursividade ?)
    - em memória/stack (IA32; aninhamento / recursividade ?)
- invocação e retorno
  - instrução de salto para o endereço de retorno
- salvaguarda & recuperação de 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

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

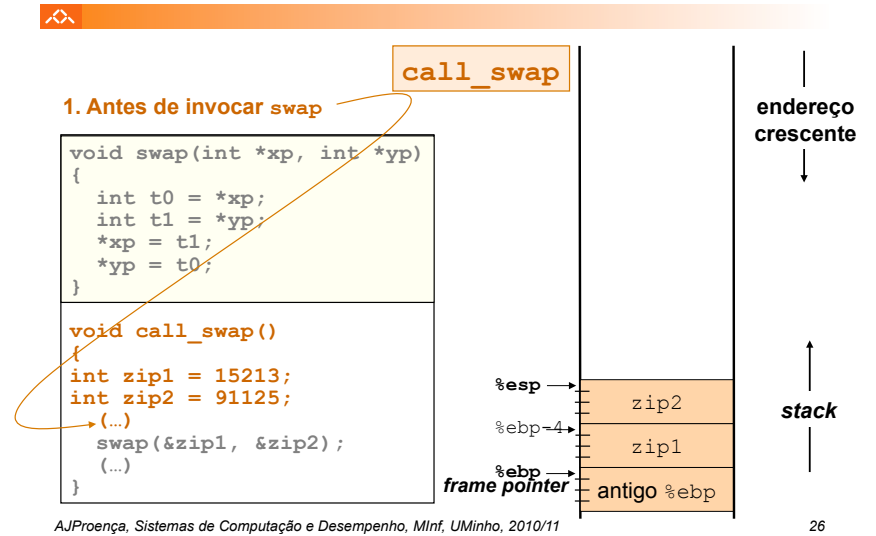
void call_swap()
{
  int zip1 = 15213;
  int zip2 = 91125;
  (...)
  swap(&zip1, &zip2);
  (...)
}
```

- em call\_swap
- na invocação de swap
- na execução de swap
- de volta a call\_swap

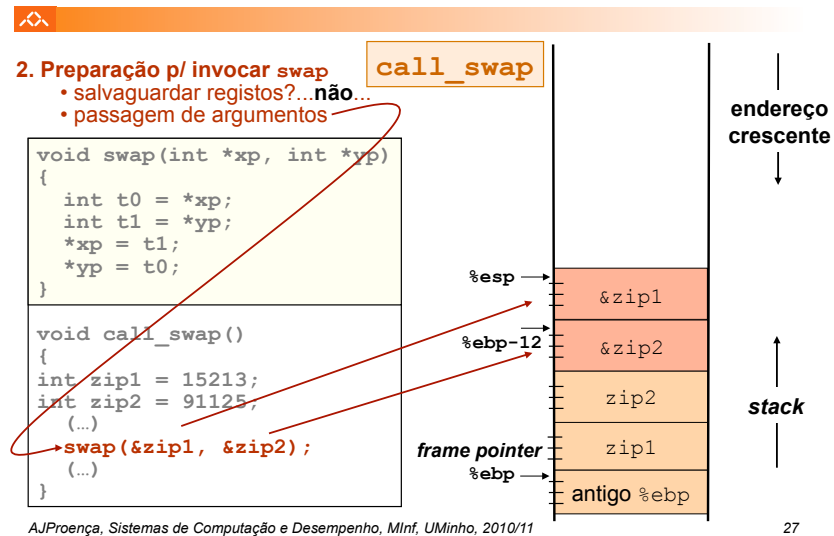
### Evolução dos contextos na stack do IA32



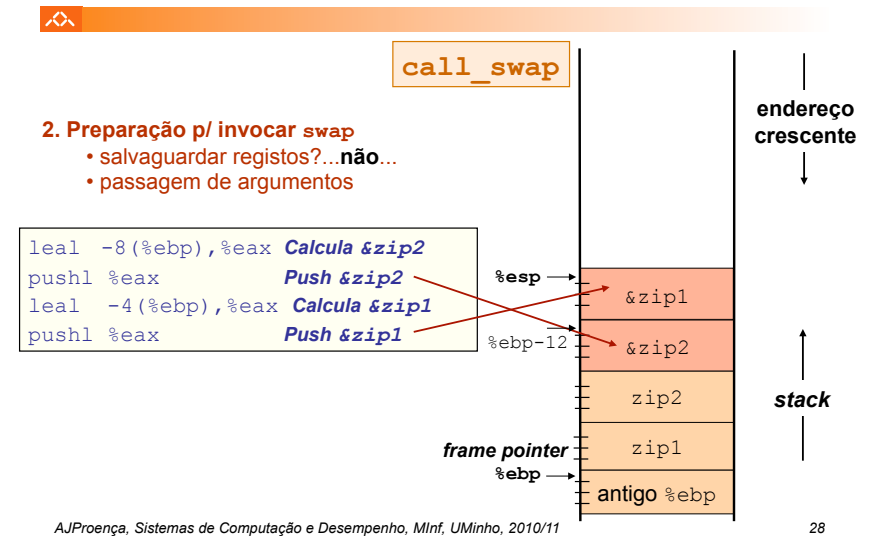
### Evolução da stack, no IA32 (1)



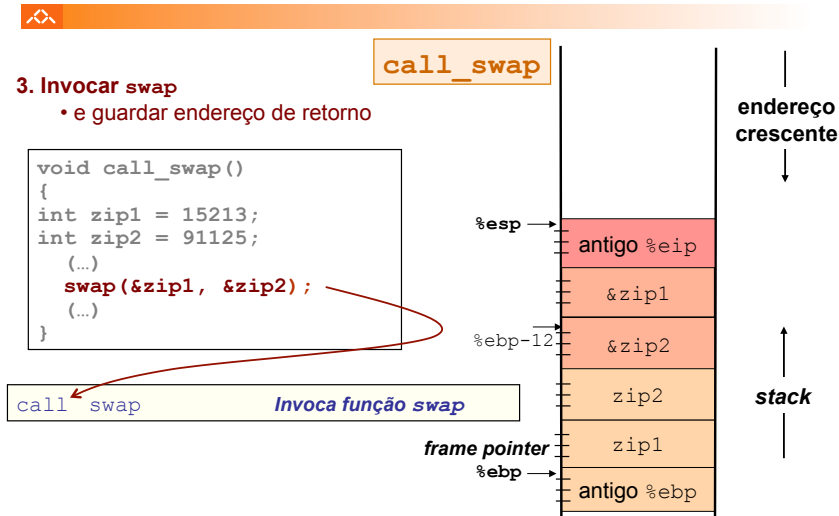
### Evolução da stack, no IA32 (2)



### Evolução da stack, no IA32 (3)

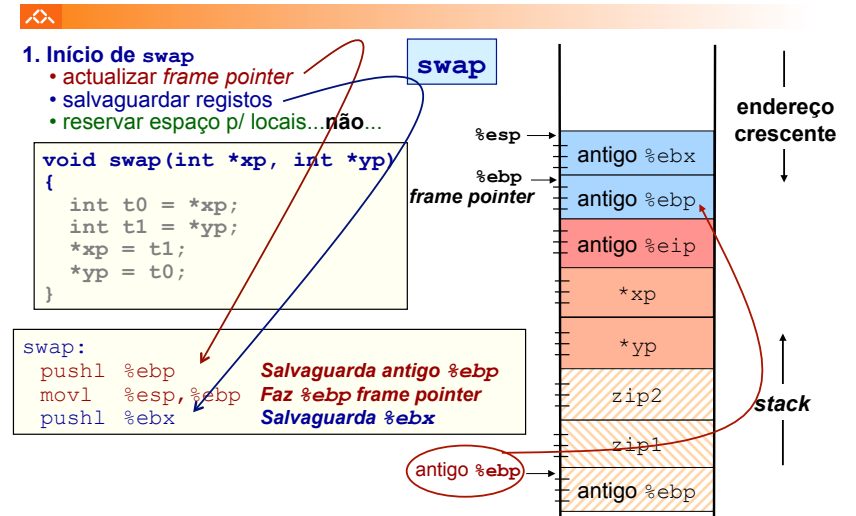


### Evolução da stack, no IA32 (4)



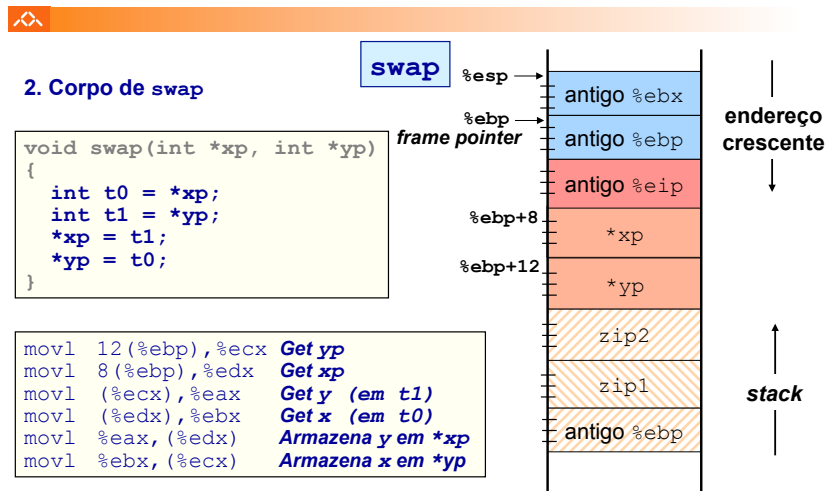
29

### Evolução da stack, no IA32 (5)



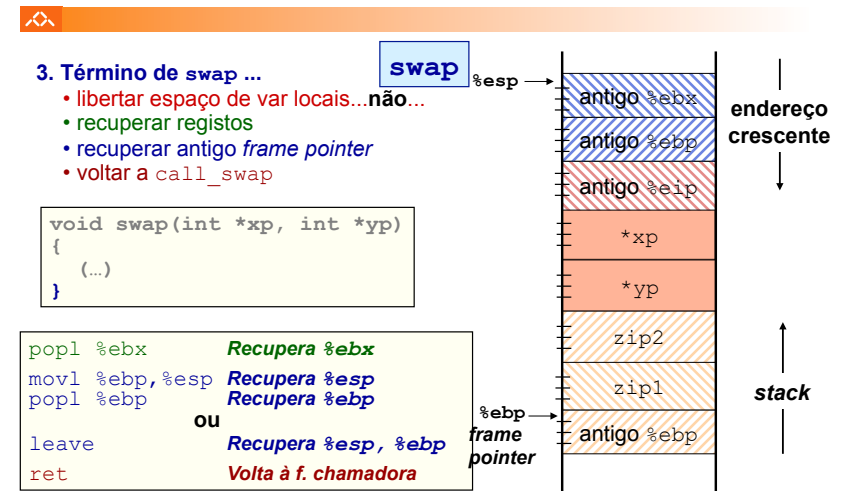
30

### Evolução da stack, no IA32 (6)



31

### Evolução da stack, no IA32 (7)



32



### 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`

endereço crescente ↓  
↑ 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;
}
```

**for**

```
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;
}
```

**while**

```
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);
}
```

**função recursiva**

### A série de Fibonacci no IA32 (2)

```
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);
}
```

**função recursiva**

```
_fib_rec:
  pushl %ebp
  movl  %esp, %ebp      Atualiza frame pointer
  subl  $12, %esp       Reserva espaço na stack para 3 int's
  movl  %ebx, -8(%ebp)  Salv guarda os 2 reg's que vão ser usados;
  movl  %esi, -4(%ebp)  de notar a forma de usar a stack...
  movl  8(%ebp), %esi
```

### A série de Fibonacci no IA32 (3)

```
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);
}
```

**função recursiva**

```
...
movl  %esi, -4(%ebp)
movl  8(%ebp), %esi      Coloca o argumento n em %esi
movl  $1, %eax          Coloca já o valor de retorno em %eax
cml  $2, %esi           n <= 2 ?
jle   L1                Se sim, salta para o fim
leal  -2(%esi), %eax    Se não, ...
...
L1:
movl  -8(%ebp), %ebx
```

## A série de Fibonacci no IA32 (4)

## 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);
}
```

←

```
...
jle L1          Se sim, salta para o fim
leal -2(%esi), %eax  Se não, ... calcula n-2, e...
movl %eax, (%esp)  ... coloca-o no topo da stack (argumento)
call _fib_rec      Invoca a função fib_rec e ...
movl %eax, %ebx    ... guarda o valor de prev_val em %ebx
leal -1(%esi), %eax
...
```

**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  Calcula n-1, e...
movl %eax, (%esp)  ... coloca-o no topo da stack (argumento)
call _fib_rec      Chama de novo a função fib_rec
leal (%eax,%ebx), %eax
...
```

## A série de Fibonacci no IA32 (6)

## Revisitando os Sistemas de Computação

**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  Calcula e coloca
em %eax o valor de retorno
L1:
movl -8(%ebp), %ebx
movl -4(%ebp), %esi  Recupera o valor dos 2
reg's usados
movl %ebp, %esp  Actualiza o valor do stack
pointer
pointer
pointer  Recupera o anterior valor do frame
ret
```

**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

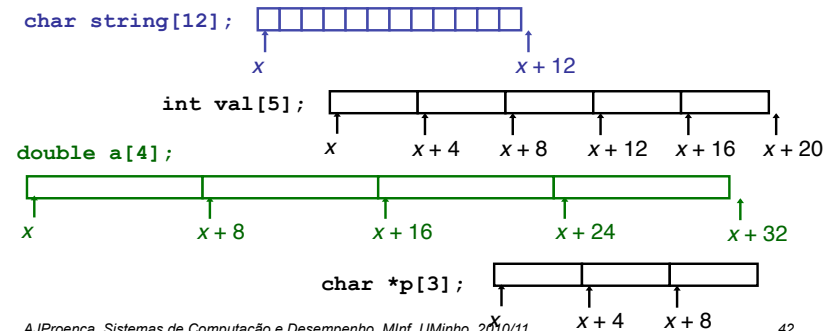
### Tipos de dados estruturados mais comuns em C

- **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];`

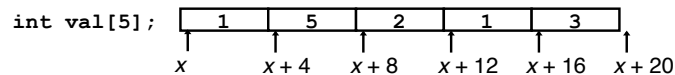
Alocação em memória de uma região com  $length * sizeof(data\_type)$  bytes



### Declaração em C:

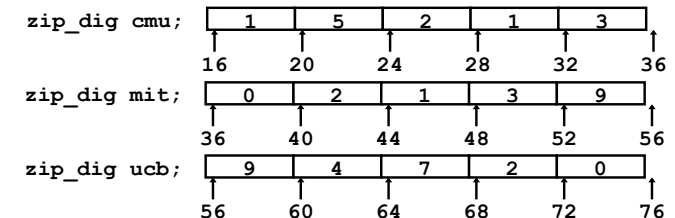
`data_type Array_name[length];`

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



Referência	Tipo	Valor
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	<code>x</code>
<code>val+1</code>	<code>int *</code>	<code>x+4</code>
<code>&amp;val[2]</code>	<code>int *</code>	<code>x+8</code>
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	<code>x + 4 i</code>

```
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)

**Código original**  
com referências a arrays  
dentro de ciclos

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

**Transformação pelo GCC**  
– eliminou a variável i  
– converteu índices em  
apontadores  
– reduziu à forma do-while

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

**Arrays:**  
apontadores em vez de índices (2)

Análise do código compilado

• Registos

%ecx z  
%eax zi  
%ebx zend

• Cálculos

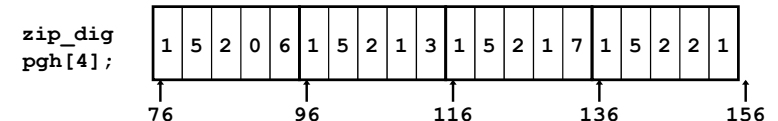
- $10 * zi + *z \Rightarrow *z + 2 * (zi + 4 * zi)$
- z++ incrementa 4

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

```
# %ecx = z
xorl %eax,%eax # zi = 0
leal 16(%ecx),%ebx # zend = z+4
.L59:
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 .L59 # 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}};
```



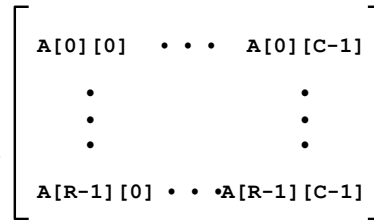
- Declaração “zip\_dig pgh[4]” equivalente a “int pgh[4][5]”
  - variável pgh é um array de 4 elementos
    - alocados em memória em blocos contíguos
  - cada elemento é um array de 5 int’s
    - alocados em memória em células contíguas
- Ordenação dos elementos (garantido em C): “Row-Major”

## Array de arrays: alocação em memória

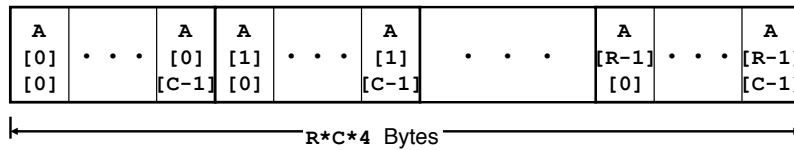
### Declaração em C:

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

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



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

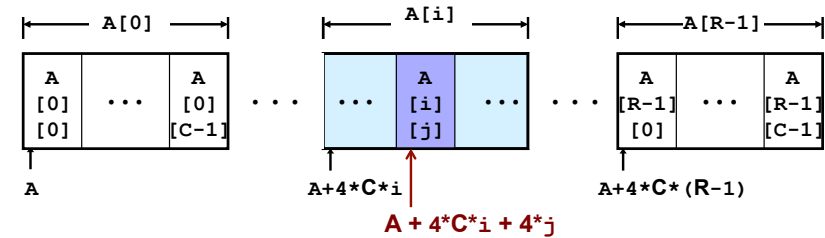
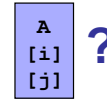


## Array de arrays: acesso a um elemento

### Elementos de um array R\*C

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

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



## Array de arrays: código para acesso a um elemento

- Localização em memória de  $[\text{index}][\text{dig}]$ :

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

- Código em assembly:

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

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

```
# %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 pontadores para arrays: uma visão alternativa

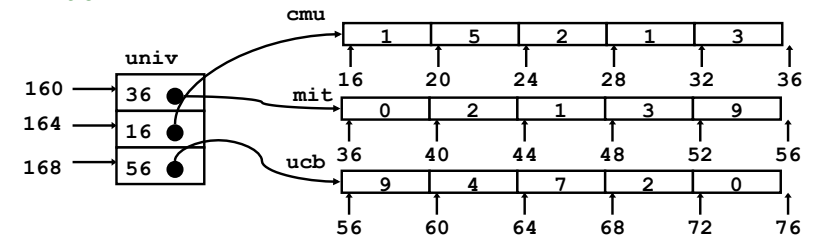
- Variável `univ` é um array de 3 elementos

```
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

### Cálculo da localização

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

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

```
# %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]
```

## Array de arrays versus array de apontadores para arrays

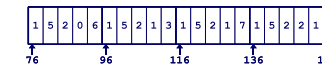
### Modos distintos de cálculo da localização dos elementos:

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

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

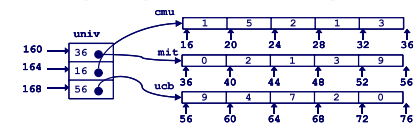
#### Array de arrays

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



#### Array de apontadores para arrays

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



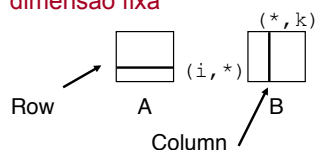
## Arrays multi-dimensionais de tamanho fixo: a eficiência do compilador (1)

### 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;
}
```

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

### Optimizações automáticas do compilador:

–antes...

```
#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;
}
```

–depois...

```
/* 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 básicos

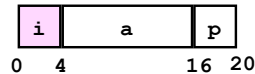


### Propriedades

- 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;
}
```

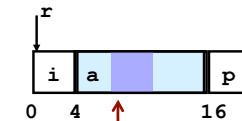
```
# %eax = val
# %edx = r
movl %eax, (%edx) # Mem[r] = val
```

## Structures: apontadores para membros (1)



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

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



$r + 4 + 4 * idx$

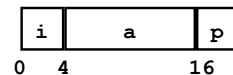
Valor calculado na compilação

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

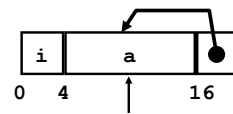
## Structures: apontadores para membros (2)



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



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



Elemento i

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

## 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 passem esses limites
  - ainda mais crítico na gestão da memória virtual (limite da página!)

### • Compilador

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

### Alinhamento de dados na memória: os dados primitivos/escalares

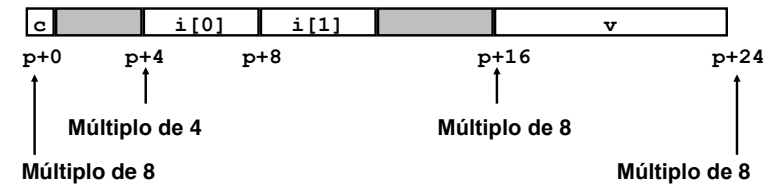
- **1 byte** (e.g., char)
  - sem restrições no endereço
- **2 bytes** (e.g., short)
  - o bit menos significativo do endereço deve ser  $0_2$
- **4 bytes** (e.g., int, float, char \*, etc.)
  - os 2 bits menos significativo do endereço devem ser  $00_2$
- **8 bytes** (e.g., double)
  - Windows (e a maioria dos SO's & instruction sets):
    - os 3 bits menos significativo do endereço devem ser  $000_2$
  - Linux:
    - os 2 bits menos significativo do endereço devem ser  $00_2$
    - 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  $00_2$
    - i.e., mesmo tratamento que um dado escalar de 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  $K$

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

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

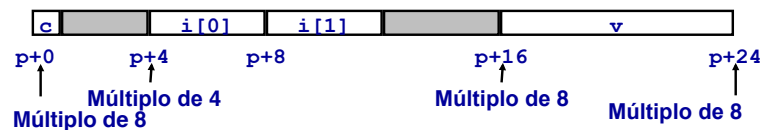


### Alinhamento de dados na memória: Windows versus Linux

- **Windows (incluindo Cygwin):**

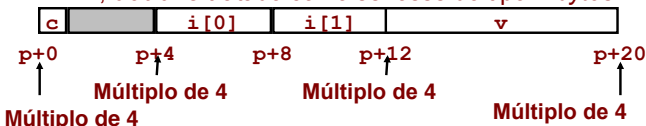
- $K = 8$ , devido ao elemento double

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```



- **Linux:**

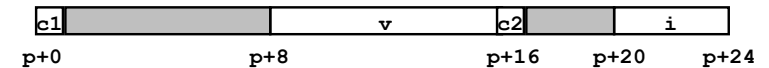
- $K = 4$ ; double tratado como se fosse do tipo 4-bytes



### Alinhamento de dados na memória: ordenação dos membros

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

10 bytes espaço desperdiçado no Windows



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

apenas 2 bytes de espaço desperdiçado

