Mestrado em Informática

2009/10

A.J.Proença

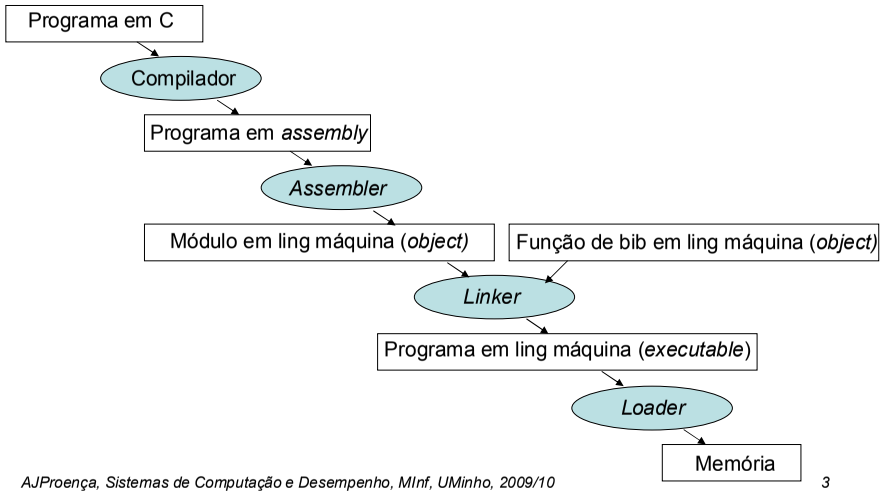
Tema

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

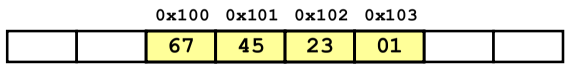
De um programa em HLL até à sua execução:



- 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
```

Idêntico à expressão
`x += y`

- *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

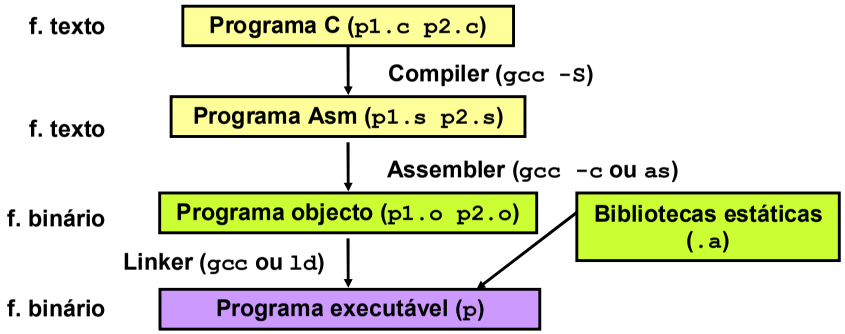
```
0x401046:    03 45 08
```

- 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

- 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`



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
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Começa no endereço 0x401040
- Total 13 bytes
- Cada instrução 1, 2, ou 3 bytes

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

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
```


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
```


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 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 . . .

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

"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;
}
```

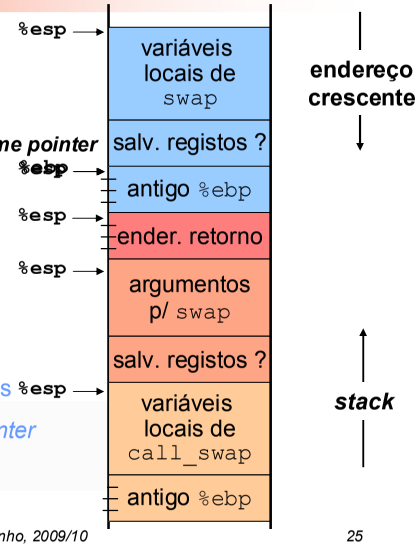
```
void call_swap()
{
int zip1 = 15213;
int zip2 = 91125;
  (…)
  swap(&zip1, &zip2);
  (…)
}
```

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

Evolução dos contextos na stack do IA32

call_swap **swap**

1. Antes de invocar `swap`
2. Preparação p/ invocar `swap`
   - •salvaguardar registos?
   - •passagem de argumentos
3. Invocar `swap`
   - •e guardar endereço de retorno
   1. Início de `swap`
      - •actualizar *frame pointer*
      - •salvaguardar registos ?
      - •reservar espaço p/ locais
   2. Corpo de `swap`
   3. Término de `swap` ...
      - •libertar espaço de var locais
      - •recuperar registos ?
      - •recuperar antigo *frame pointer*
      - •voltar a `call_swap`
4. Terminar invocação de `swap`...
   - •libertar espaço de argumentos na *stack*
   - •recuperar registos?



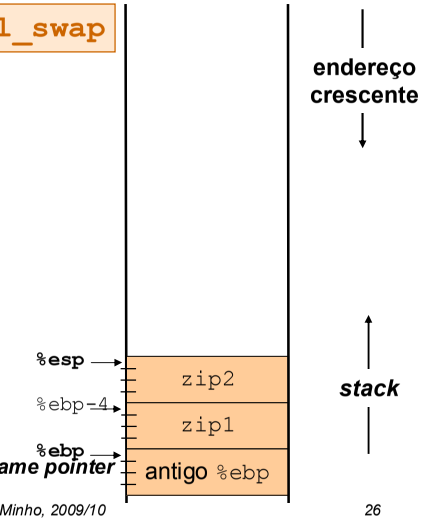
Evolução da stack, no IA32 (1)

call_swap

1. Antes de invocar `swap`

```
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);
  (…)
}
```



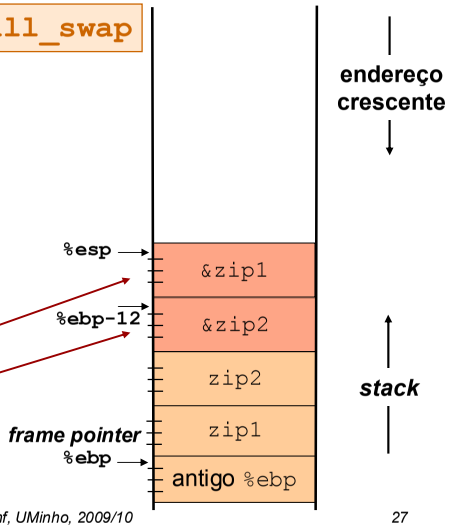
Evolução da stack, no IA32 (2)

call_swap

2. Preparação p/ invocar `swap`
   - • salvaguardar registos?...**não**...
   - • passagem de argumentos

```
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);
  (…)
}
```

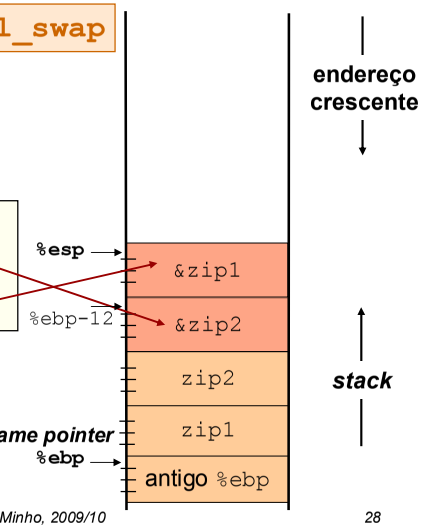


Evolução da stack, no IA32 (3)

call_swap

2. Preparação p/ invocar `swap`
   - • salvaguardar registos?...**não**...
   - • passagem de argumentos

```
leal  -8(%ebp),%eax   Calcula &zip2
pushl %eax            Push &zip2
leal  -4(%ebp),%eax   Calcula &zip1
pushl %eax            Push &zip1
```

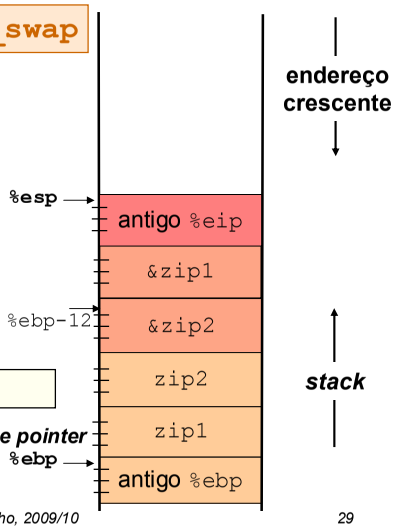


Evolução da stack, no IA32 (4)

3. Invocar `swap`
- e guardar endereço de retorno

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

```
call swap          Invoca função swap
```

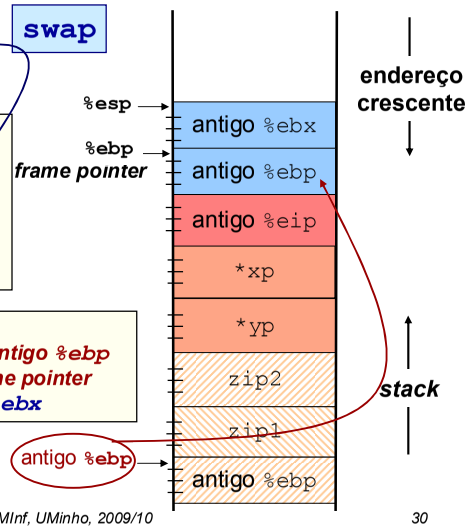


Evolução da stack, no IA32 (5)

1. Início de `swap`
- actualizar *frame pointer*
- salvaguardar registos
- reservar espaço p/ locais...**não**...

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

```
swap:
  pushl  %ebp        Salvaguarda antigo %ebp
  movl   %esp,%ebp   Faz %ebp frame pointer
  pushl  %ebx        Salvaguarda %ebx
```

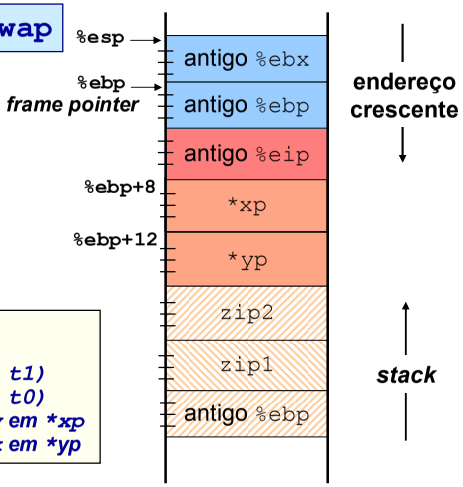


Evolução da stack, no IA32 (6)

2. Corpo de `swap`

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

```
movl  12(%ebp),%ecx  Get yp
movl  8(%ebp),%edx   Get xp
movl  (%ecx),%eax    Get y (em t1)
movl  (%edx),%ebx    Get x (em t0)
movl  %eax,(%edx)    Armazena y em *xp
movl  %ebx,(%ecx)    Armazena x em *yp
```

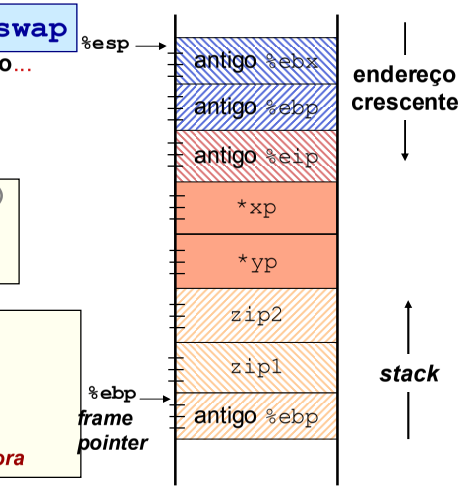


Evolução da stack, no IA32 (7)

3. Término de `swap` ...
- libertar espaço de var locais...**não**...
- recuperar registos
- recuperar antigo *frame pointer*
- voltar a `call_swap`

```
void swap(int *xp, int *yp)
{
  (…)
}
```

```
popl  %ebx          Recupera %ebx
movl  %ebp,%esp     Recupera %esp
popl  %ebp          Recupera %ebp
          ou
leave  Recupera %esp, %ebp
ret                 Volta à f. chamadora
```

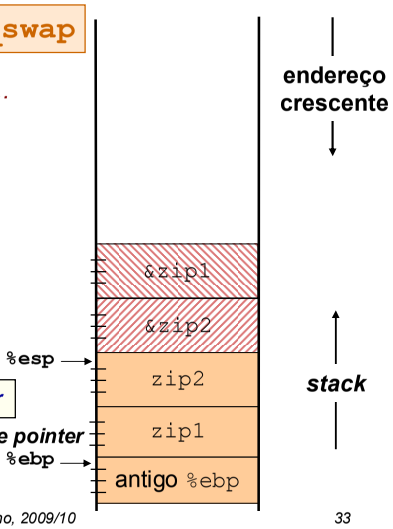


Evolução da stack, no IA32 (8)

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)
```
Actualiza stack pointer



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_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_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

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 (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          Actualiza frame pointer

   subl   $12, %esp           Reserva espaço na stack para 3 int's
   movl   %ebx, -8(%ebp)      Salvaguarda 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)

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       Coloca o argumento n em %esi
   movl   $1, %eax            Coloca já o valor de retorno em %eax
   cmpl   $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)

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
   ...
```


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     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)

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
   popl                    %ebp            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

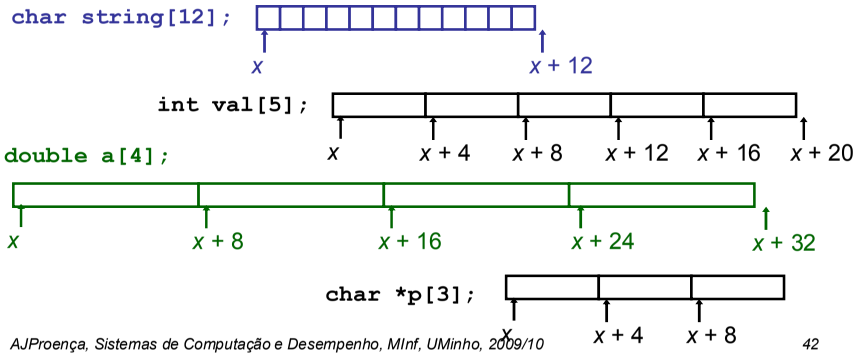
Tipos de dados estruturados mais comuns em C

- ***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];
```

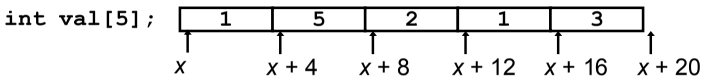
Alocação em memória de uma região com `length * sizeof(data_type)` *bytes*



Declaração em C:

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

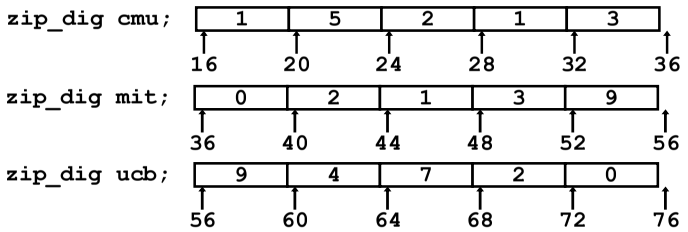
O identificador **`Array_name`** pode ser usado como apontador para o elemento 0



Referência	Tipo	Valor
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*


```
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*

```
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]
```


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;
}
```


Análise do código compilado

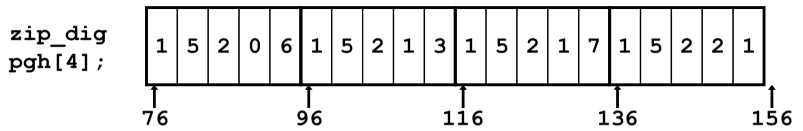
- Registos
  - `%ecx` `z`
  - `%eax` `zi`
  - `%ebx` `zend`
- Cálculos
  - `10*zi + *z` **`=>`** `*z + 2*(zi+4*zi)`
  - `z++` incrementa 4

```
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
```



```
#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*"

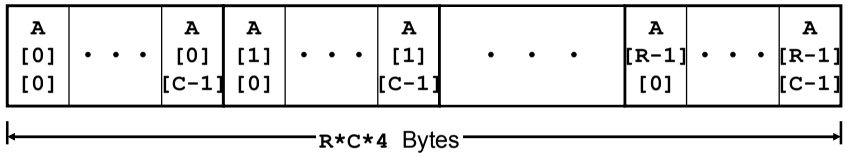
Declaração em C:

data_type Array_name[***R***][**C**];

- Alocação em memória de uma região com
  R * ***c*** * sizeof(*data_type*) *bytes*
- Ordenação
  Row-Major

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

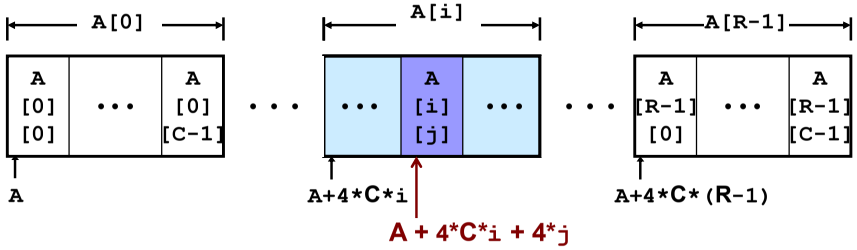
```
int A[R][C];
```



Elementos de um *array* R*C

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

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



- **Localização em memória de**
  `pgh[index][dig]`:
  `pgh + 20*index + 4*dig`

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

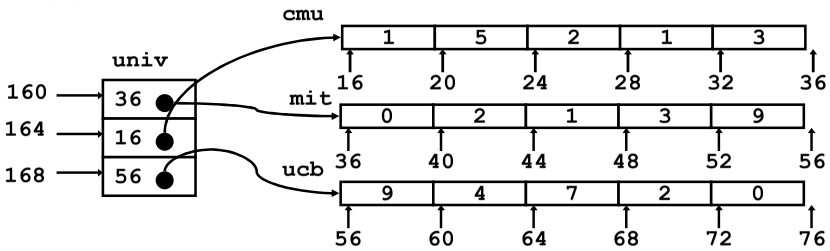
- **Código em *assembly*:**
  - cálculo do endereço
    `pgh + 4*dig + 4*(index+4*index)`
  - acesso ao elemento: com `movl`

```
 # %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)
```


- Variável `univ` é um *array* de 3 elementos
- Cada elemento:
  - um apontador de 4 *bytes*
  - aponta para um *array* de `int`'s

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

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



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*

```
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]
```


Modos distintos de cálculo da localização dos elementos:

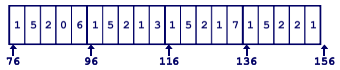
```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

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

Array de *arrays*

- elemento em

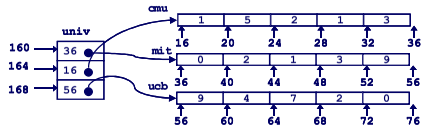
`Mem[pgh+20*index+4*dig]`



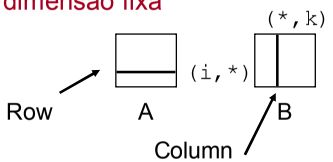
Array de apontadores para *arrays*

- elemento em

`Mem[Mem[univ+4*index]+4*dig]`



- **Oportunidades para optimizar**
  - 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;
}
```


- **Optimizações 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;
}
```

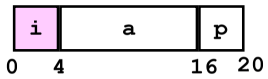
```
/* Compute element i,k … */
int fix_prod_ele (...)
{
  int *Aptr = &A[i][0];
  int *Bptr =&B[0][k];
  int cnt = N-1;
  int result = 0;
  do {
    result += (*Aptr)*(*Bptr);
    Aptr += 1;
    Bptr += N;
    cnt--;
  }while (cnt>=0);
  return result;
}
```

Propriedades

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

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

Organização na memória



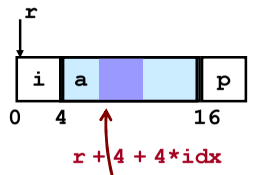
Acesso a um membro da *structure*

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

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



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



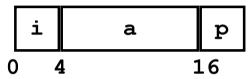

```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

Valor calculado na compilação

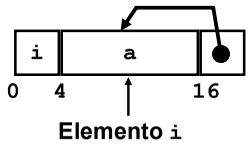
```
 # %ecx = idx
 # %edx = r
leal 0(,%ecx,4),%eax     # 4*idx
leal 4(%eax,%edx),%eax   # r+4*idx+4
```



```
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
```


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

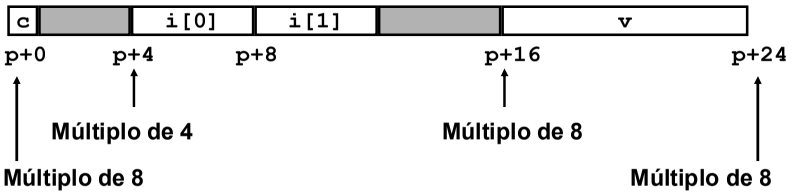
Alinhamento de dados na memória: os dados primitivos/escalares

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

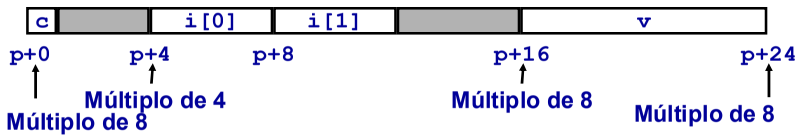
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



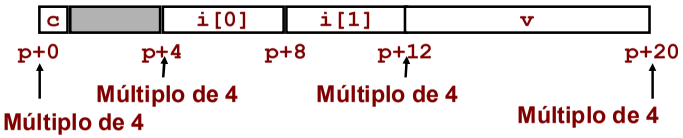
Alinhamento de dados na memória: Windows versus Linux

- **Windows** (incluindo Cygwin):
  - K = 8, devido ao elemento `double`

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



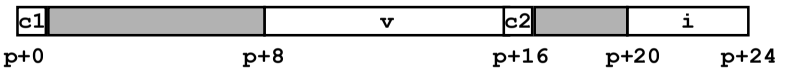
- **Linux**:
  - K = 4; `double` tratado como se fosse do tipo 4-*bytes*



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