LẬP TRÌNH HỆ THỐNG

ThS. Đỗ Thị Thu Hiền
(hiendtt@uit.edu.vn)

TRƯỜNG ĐH CÔNG NGHỆ THÔNG TIN - ĐHQG-HCM
KHOA MẠNG MÁY TÍNH & TRUYỀN THÔNG
FACULTY OF COMPUTER NETWORK AND COMMUNICATIONS
Tầng 8 - Tòa nhà E, trường ĐH Công nghệ Thông tin, ĐHQG-HCM
Điện thoại: (08)3 725 1993 (122)

Mảng - Cấu trúc

Nhắc lại về các kiểu dữ liệu cơ bản

Đơn vị: bytes

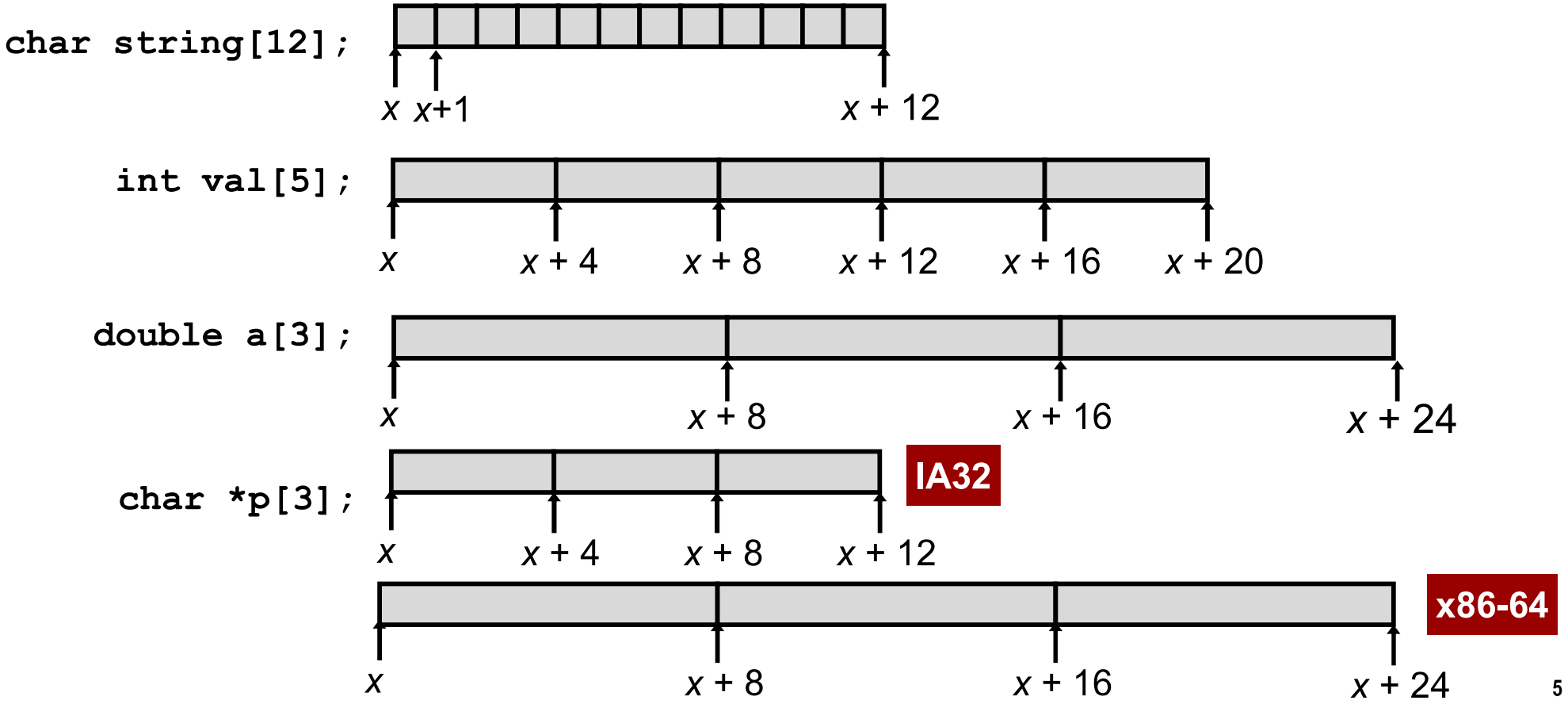
Kiểu dữ liệu	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	–	–	10/16
pointer	**4**	**8**	**8**

Nội dung

- **Mảng - Array**
  - Mảng 1 chiều
  - Mảng 2 chiều (nested)
  - Mảng nhiều cấp
- **Cấu trúc – Structure**
  - Cấp phát
  - Truy xuất
  - Alignment (căn chỉnh)

Cấp phát mảng

- **Nguyên tắc cơ bản**
  - *T* **A[***L***];** Phần tử đầu tiên có địa chỉ thấp nhất
  - Mảng của kiểu dữ liệu *T* và có độ dài *L*
  - Mảng gồm *L* phần tử có cùng kiểu dữ liệu *T*
  - Vùng nhớ được cấp phát liên tục với ***L* * sizeof**(*T*) **bytes** trong bộ nhớ



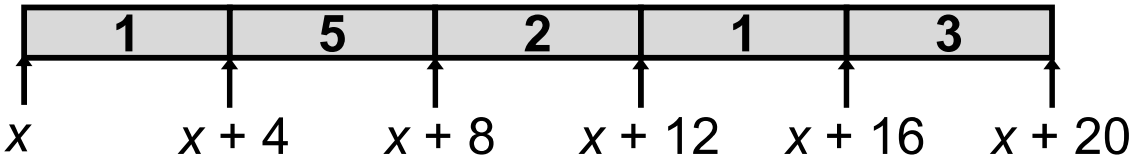
Truy xuất mảng

- **Nguyên tắc cơ bản**

  T **A**[*L*];

  - Mảng của kiểu dữ liệu T và có độ dài L
  - Định danh **A** có thể dùng như là *con trỏ* trỏ đến mảng hoặc thành phần 0 của mảng: Type *T**

```
int val[5];
```



- **Tham chiếu Kiểu dữ liệu Giá trị**

`val[4]`	int - 3
`val`	int * - x (địa chỉ mảng)
`val+1`	int * - x+4 (địa chi phần tử index bằng 1)
`&val[2]`	int * - x+8 (địa chỉ phần tử index bằng 2) = val+2
`val[5]`	int - ??
`*(val+1)`	int - 5 (giá trị tại địa chỉ phần tử index bằng 1) = val[1]
`val + i`	int * - x+4*i (địa chỉ phần tử index bằng i)

Truy xuất mảng: Ví dụ

- Cho các mảng sau trong IA32:

```
char A[12];
char *B[8];
double C[7];
double *D[5];
char **E[5];
```

Điền vào bảng dưới đây các thông tin còn thiếu:

Mảng	Kích thước phần tử (byte)	Tổng kích thước (byte)	Địa chỉ bắt đầu	Địa chỉ phần tử thứ *i*
A	1	12	x_A	xA + i
B	4	32	x_B	xB + 4i
C	8	56	x_C	xC + 8i
D	4	20	x_D	xD + 4i
E	4	20	x_E	xE + 4i

Mảng: Ví dụ

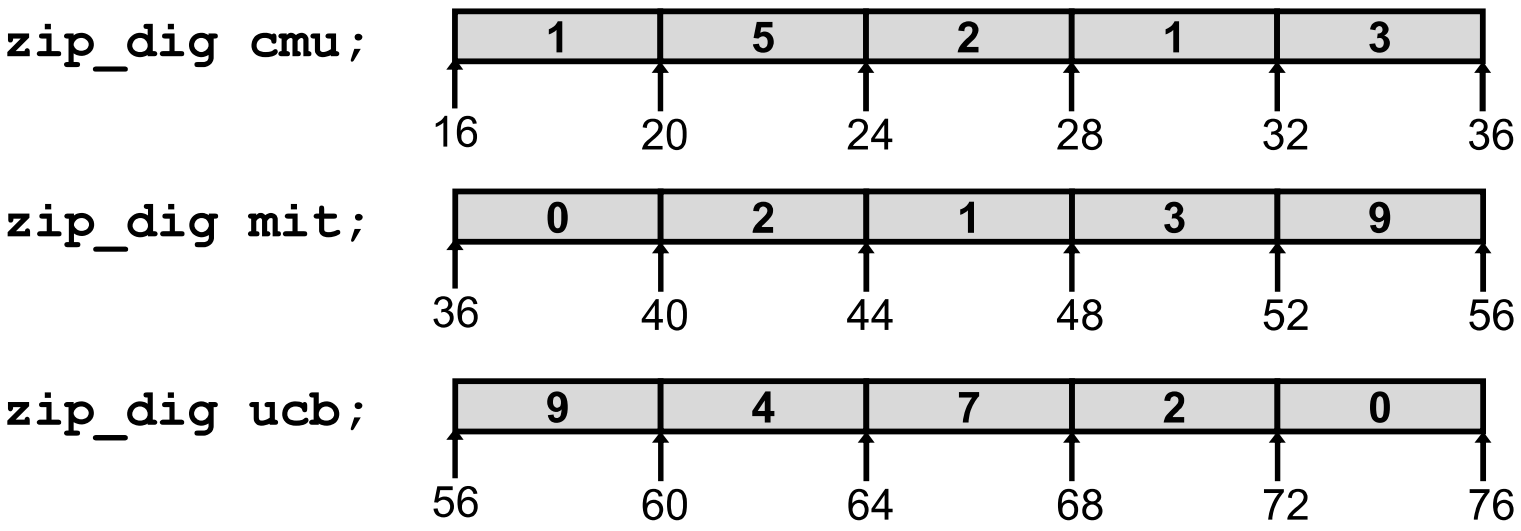
```
#define ZLEN 5
typedef int zip_dig[ZLEN];
  //zip_dig = int
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

cmu[3] = ?? 1

cmu[6] = ?? 2 (mit[2]) vì 3 mảng liên tiếp nhau

mit[4] = ?? 9

mit[7] = ?? 7

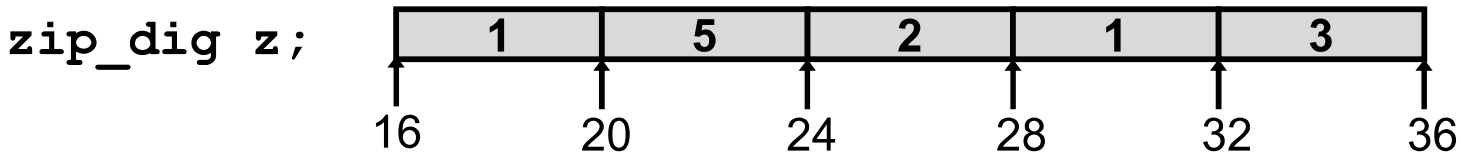


- **Khai báo "`zip_dig cmu`" tương đương với "`int cmu[5]`"**
- **Các mảng ví dụ được cấp phát trong các block 20 bytes liên tiếp nhau**
  - Không phải lúc nào cũng đảm bảo như vậy

Truy xuất mảng: Ví dụ

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



IA32

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

Vị trí (z + 4*i)

- Thanh ghi `%edx` chứa địa chỉ bắt đầu của mảng
- Thanh ghi `%eax` chứa chỉ số index
- Vị trí của giá trị muốn lấy là `4*%eax + %edx`
- Sử dụng tham chiếu ô nhớ `(%edx,%eax,4)`

Vòng lặp trong mảng: Ví dụ (IA32)

```
#define ZLEN 5
typedef int zip_dig[ZLEN];
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

- Dùng chỉ số i để tính toán địa chỉ của phần tử z[i].
- Chỉ số i tăng dần qua từng vòng lặp

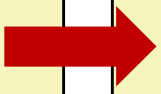
```
1.     # edx = z                          # Địa chỉ của mảng z
2.     movl   $0, %eax                   # %eax = i
3.     jmp    .L3                        # goto middle
4.  .L4:                                 # loop:
5.     addl   $1, (%edx,%eax,4) # z[i]++
6.     addl   $1, %eax    #i++           # i++
7.  .L3:
8.     cmpl   $5, %eax                   # i:5
9.     jne    .L4                        # if !=, goto loop
```

? Nếu sửa thành `short zip_dig[5]`, đoạn code assembly trên sẽ khác gì?

+ Sửa 4 thành 2: kiểu dữ liệu ảnh hưởng kích thước, địa chỉ
addl $1, (%edx, %eax, 2)
+ Sử hậu tố: l thành w (2 bytes)
addw $1, (%edx, %eax, 2)

Vòng lặp với con trỏ: Ví dụ (IA32)

```
void zincr_p(zip_dig z) {
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```



```
void zincr_v(zip_dig z) {
  void *vz = z;
  int dt = 0;
  do {
    (*((int *) (vz+dt)))++;
    dt += ISIZE;
  } while (dt != ISIZE*ZLEN);
}
```

- Dùng khoảng cách từ phần tử z[i] đến z[0]
- Khoảng cách tăng dần qua từng vòng lặp

```
1.        # edx = z = vz                  # Địa chỉ của z
2.        movl   $0, %eax                 # dt = 0
3. .L8:                                   # loop:
4.        addl   $1, (%edx,%eax)          # Increment vz+dt
5.        addl   $4, %eax                 # dt +=  4
6.        cmpl   $20, %eax                # Compare dt:20
7.        jne    .L8                      # if !=, goto loop
```

cmpl $16, %eax
jle .L8

Khoảng cách từ z[ii] đến phần tử đầu tiên khác nhau
dt0 = &z[0] - &z = 0
dt[1] = &z[1] - &z = 4
dt[2] = &z[2] - &z = 8

=> z[i] = &z + dti

? Nếu sửa thành `short zip_dig[5]`, đoạn code assembly trên sẽ khác gì?

Đổi 4 thành 2, 20 thành 10 (hoặc 16 thành 8)

Vòng lặp trong mảng: Ví dụ (x86_64)

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

Câu hỏi: Mảng 1 chiều

Trong hệ thống 32bit, cho mảng 1 chiều **T A[N]** với **T** là 1 kiểu dữ liệu cơ bản**, N** là số phần tử và đều chưa biết. Biết tổng kích thước của **A** là **20 bytes.** = N * sizeof(T)

Tìm **T và N?**

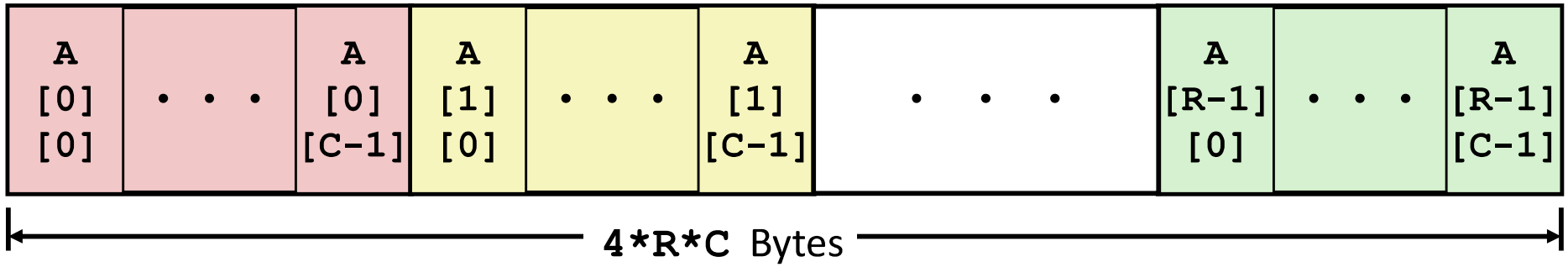
Ta có: N * sizeof(T) = 20
Điều kiện:
+ N nguyên dương (>0)
+ sizeof(T) = {1, 2, 4, 8}
Với:
+ sizeof(T) = 1 => N = 20 (nhận) => char A[20]
+ sizeof(T) = 2 => N = 10 (nhận) => short A[10]
+ sizeof(T) = 4 => N = 5 (nhận) => int A[5], double A[5], long A[5], char* A[5],...
+ sizeof(T) = 8 => N = 2,5 (loại)

Mảng 2 chiều (Nested array)

- **Định nghĩa**
  - *T* **A**[*R*][*C*];
    - Mảng 2 chiều của kiểu dữ liệu *T*
    - *R* dòng, *C* cột
    - Phần tử kiểu *T* cần *K* bytes
- **Kích thước mảng**
  - *R* * *C* * *K* bytes
- **Sắp xếp**
  - Thứ tự Row-Major

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

`int A[R][C];`



Mảng 2 chiều: Ví dụ

Với mảng 2 chiều A[R][C]
&A[i][j] = &A + i*(sizeof(A))*C + j*(sizeof(A))

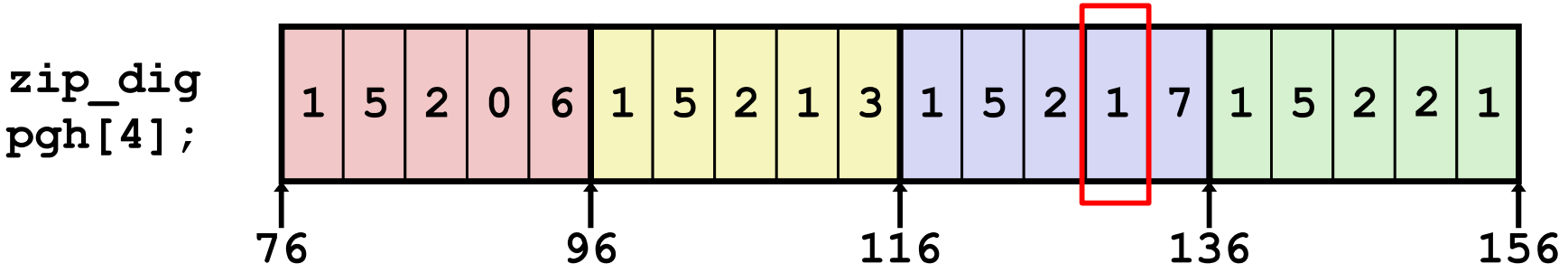
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

Yêu cầu lấy giá trị 1 trên hình?

Ký hiệu trong C? pgh[2][3]

Xác định địa chỉ? &pgh + 2*4*5 + 4*3

Địa chỉ của pgh[i][j]?

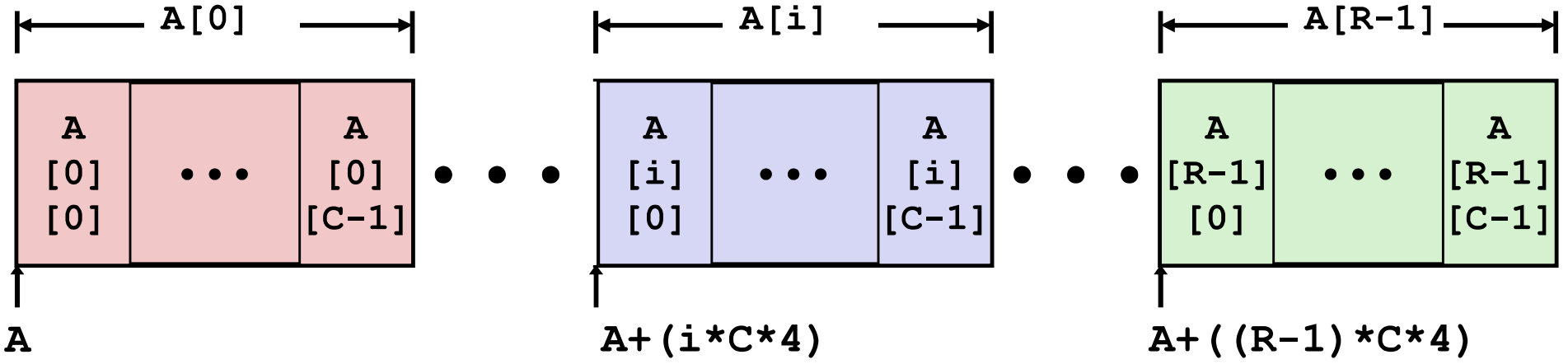


- **"`zip_dig pgh[4]`" tương đương với "`int pgh[4][5]`"**
  - Biến **`pgh`**: Mảng 4 phần tử, được cấp phát liên tục
  - Mỗi phần tử là một mảng 5 số **`int`**, được cấp phát liên tục
- **Sắp xếp tất cả các phần tử đảm bảo theo "Row-Major"**

Truy xuất 1 dòng trong mảng 2 chiều

- **Các dòng trong mảng 2 chiều**
  - `A[i]` là 1 mảng gồm *C* phần tử kiểu T
  - Mỗi phần tử kiểu *T* chiếm *K* bytes.
  - Kích thước của 1 mảng nhỏ `A[i]`: *C* * *K*
  - Địa chỉ bắt đầu **A +** *i* * (*C* * *K*)

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

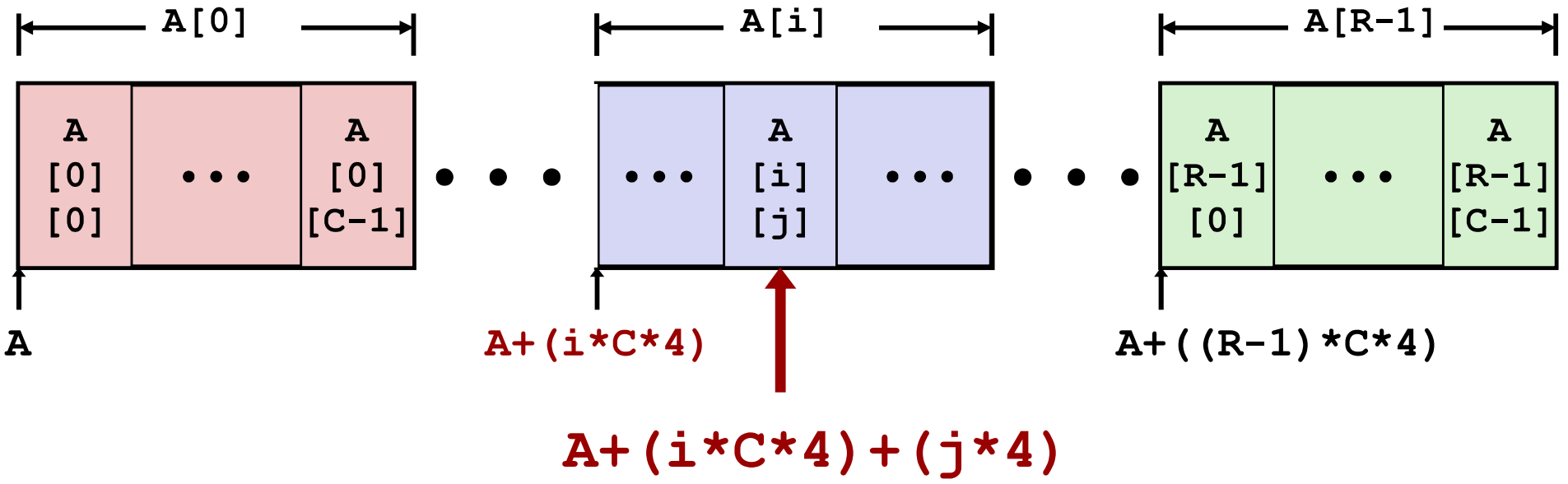


Truy xuất phần tử trong mảng 2 chiều

- **Các phần tử của mảng**
  - `A[i][j]` là phần tử có kiểu *T*, cần *K* bytes
  - Địa chỉ: $A + i * (C * K) + j * K = A + (i * C + j) * K$

địa chỉ của A[i]

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



Truy xuất mảng 2 chiều: Ví dụ

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

```
movl   8(%ebp), %eax          # index
leal   (%eax,%eax,4), %eax    # 5*index
addl   12(%ebp), %eax         # 5*index+dig  # pgh[index]
movl   pgh(,%eax,4), %eax     # offset 4*(5*index+dig)
```

- **Các phần tử của mảng**
  - `pgh[index][dig]` có kiểu dữ liệu `int`
  - Địa chỉ: `pgh + 20*index + 4*dig`

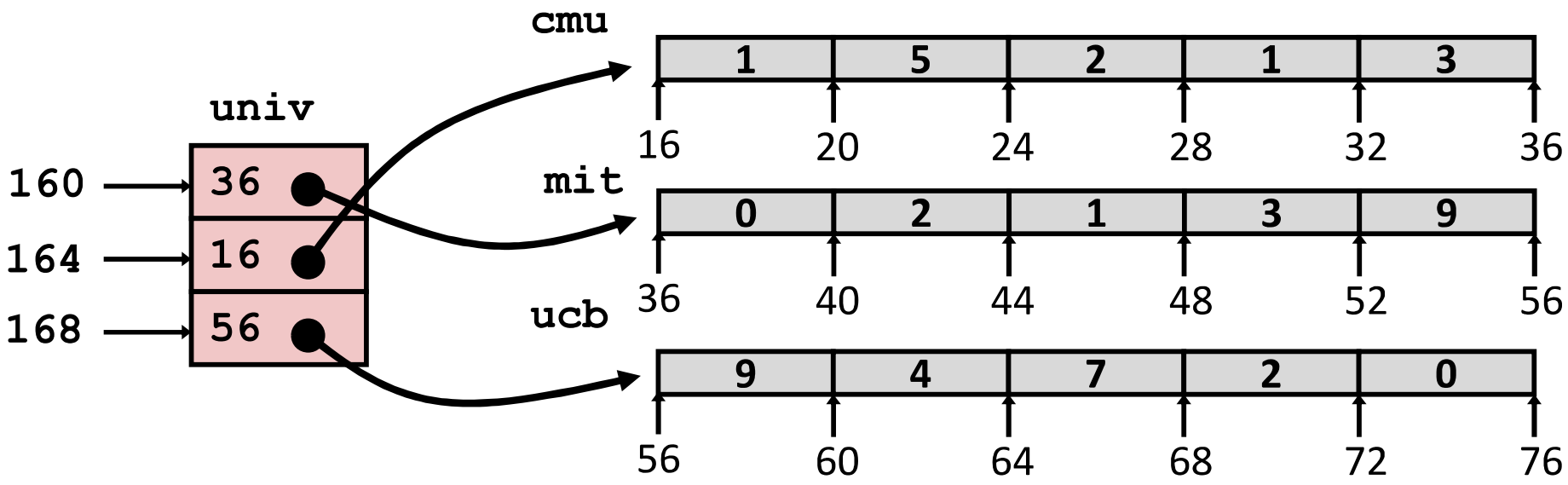
    `= pgh + 4*(5*index + dig)`
- **IA32 Code**
  - Tính toán địa chỉ `pgh + 4*((index+4*index)+dig)`

Mảng nhiều cấp (Multi-Level array)

```
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};
```

- Biến `univ` là mảng có 3 phần tử
- Mỗi phần tử là 1 con trỏ
  - 4 (hoặc 8) bytes
- Mỗi con trỏ trỏ đến một mảng số `int`



Truy xuất phần tử trong Multi-Level Array

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

```
   movl   8(%ebp), %eax              # index
   movl   univ(,%eax,4), %edx        # p = univ[index]
   movl   12(%ebp), %eax             # digit
   movl   (%edx,%eax,4), %eax        # p[digit]
```

- **Tính toán**
  - Truy xuất phần tử: **`Mem[Mem[univ+4*index]+4*digit]`**
  - Cần phải đọc bộ nhớ 2 lần
    - Lần 1 để lấy con trỏ trỏ đến mảng chứa phần tử
    - Lần 2 truy xuất phần tử trong mảng

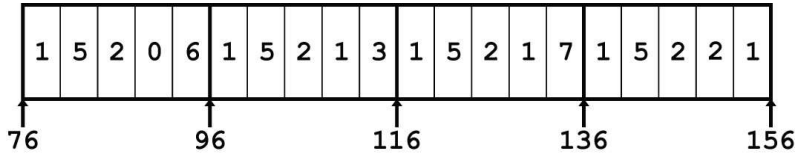
Truy xuất phần tử mảng

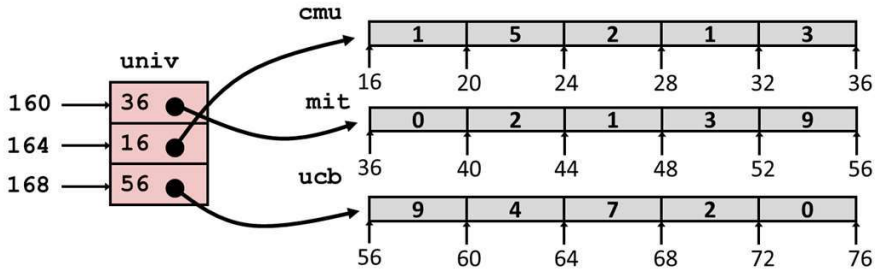
Nested array

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

Multi-level array

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





Truy xuất giống nhau trong C, nhưng cách tính toán địa chỉ khác nhau. Ví dụ trong IA32:

`Mem[pgh+20*index+4*digit]` `Mem[Mem[univ+4*index]+4*digit]`

Ma trận NxN

- **Số chiều cố định**
  - Số chiều N đã biết khi biên dịch

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
  return a[i][j];
}
```

- **Số chiều biến đổi, đánh chỉ số tường minh**
  - Cách truyền thống để hiện thực mảng động

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
  return a[IDX(n,i,j)];
}
```

- **Số chiều biến đổi, đánh chỉ số ngầm**
  - Hiện được hỗ trợ trong gcc

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

Ví dụ: Truy xuất ma trận 16 X 16

- **Các phần tử của mảng**
  - Địa chỉ **A** + *i* * (*C* * *K*) + *j* * *K*
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
```

```
  movl   12(%ebp), %edx     # i
  sall   $6, %edx           # i*64
  movl   16(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   8(%ebp), %eax      # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*64)
```

Truy xuất ma trận N x N

- **Các phần tử của mảng**
  - Địa chỉ **A** + *i* * (*C* * *K*) + *j* * *K*
  - C = n, K = 4

```
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```

Mảng: Bài tập 1

Cho 1 mảng 2 chiều **int array[M][N]** với **M** và **N** chưa biết.

Đoạn mã assembly bên dưới thực hiện **truy xuất phần tử A[i][j]**.

```
1.   // %ecx = i, %edx = j
2.   movl    $array, %eax
3.   sall    $2, %edx
4.   leal    (%edx, %eax), %eax
5.   movl    (%eax, %ecx, 16), %eax          # eax = A[i][j]
```

Chọn nhận định **đúng**?

A. M = 16 **B.** N = 16 **C.** N = 4 **D.** M = 4

Mảng: Bài tập 2

Cho 2 mảng 2 chiều `mat1`, `mat2` với các hằng số `M`, `N` và đoạn mã assembly bên dưới của hàm `sum_element`.

Thử phân tích mã assembly và tìm 2 giá trị cụ thể của M, N?

```
int mat1[M][N]
int mat2[N][M]

int sum_element(int i, int j) {
  return mat1[i][j] + mat2[j][i];
}
```

```
1.   movl    8(%ebp), %ecx
2.   movl    12(%ebp), %edx
3.   leal    0(,%ecx,8), %eax
4.   subl    %ecx, %eax
5.   addl    %edx, %eax
6.   leal    (%edx,%edx,4), %edx
7.   addl    %ecx, %edx
8.   movl    mat1(,%eax,4), %eax
9.   addl    mat2(,%edx,4), %eax
```

Mảng: Bài tập 3

Trong hệ thống 32 bit, cho một ma trận **T A[N][N]** với **T** là kiểu dữ liệu và **N** là hằng số chưa biết.

Cho địa chỉ của **A** là **0x1000**, địa chỉ lưu của phần tử **A[2][2]** là **0x101C.**

Tìm **T** và **N**?

Nội dung

- **Mảng - Array**
  - Mảng 1 chiều
  - Mảng 2 chiều (nested)
  - Nhiều chiều
- **Cấu trúc – Structure**
  - Cấp phát
  - Truy xuất
  - Alignment (căn chỉnh)