LẬP TRÌNH HỆ THỐNG

ThS. Đỗ Thị Hương Lan
(landth@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**

Kích thước các kiểu dữ liệu (x86-64)

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>


int main(){
    char c;
    short s;
    int i;
    long l;
    float f;
    double d;
    long double ld;
    char *p;

    // sizeof
    printf("Size of char is %d\n", sizeof c);
    printf("Size of short is %d\n", sizeof s);
    printf("Size of int is %d\n", sizeof i);
    printf("Size of long is %d\n", sizeof l);
    printf("Size of float is %d\n", sizeof f);
    printf("Size of double is %d\n", sizeof d);
    printf("Size of long double is %d\n", sizeof ld);
    printf("Size of pointer is %d\n", sizeof p);
}
```

```
$gcc -o main *.c

$main

Size of char is 1
Size of short is 2
Size of int is 4
Size of long is 8
Size of float is 4
Size of double is 8
Size of long double is 16
Size of pointer is 8
```

Đơn vị: bytes

Nội dung

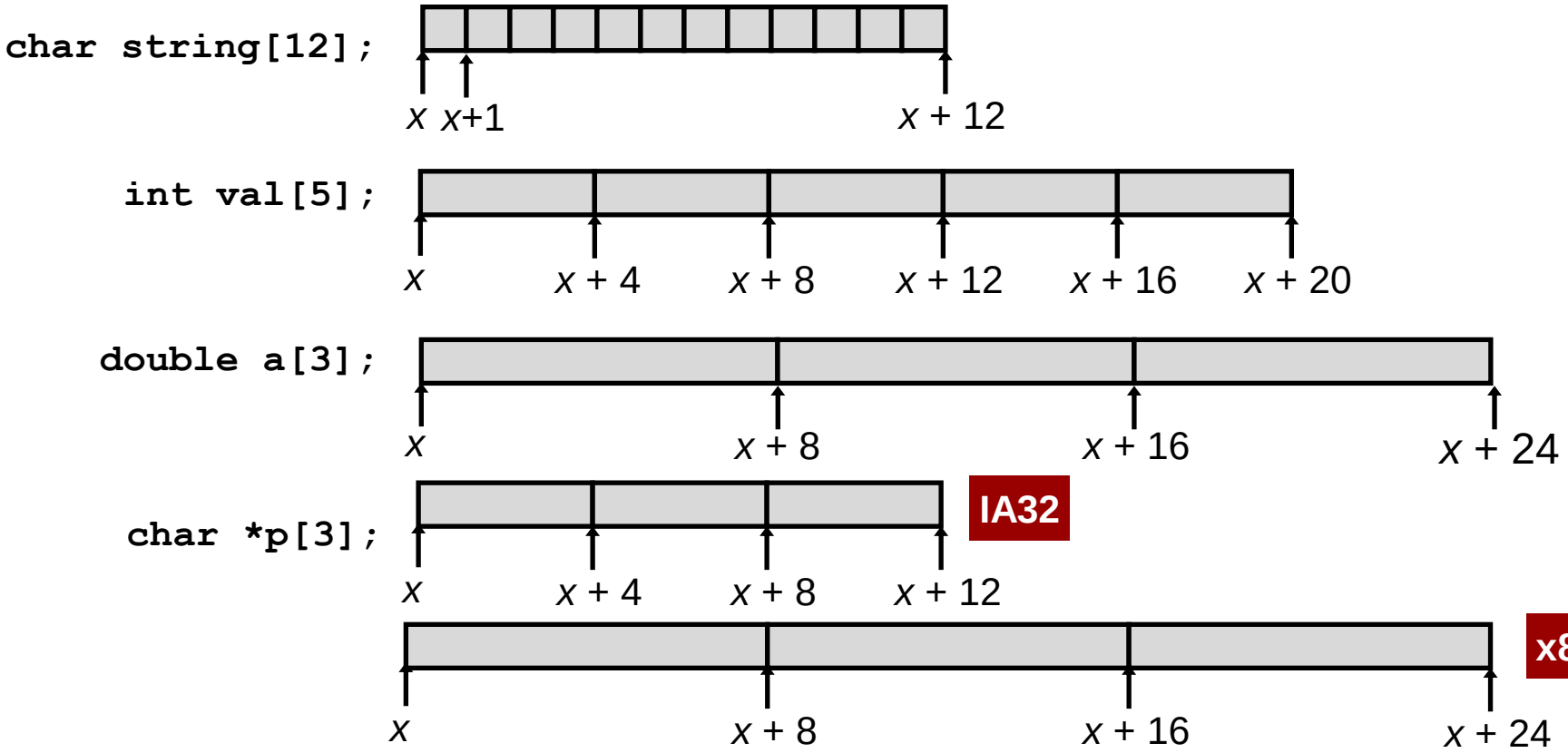
- **Mảng - Array**
  - Mảng 1 chiều
  - Mảng 2 chiều (nested)
  - Mảng nhiều cấp
- **Cấu trúc – Structure**
  - Cấp phát
  - Truy xuất
  - Alignment (căn chỉnh)

Cấp phát mảng

- **Nguyên tắc cơ bản**

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

  - 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



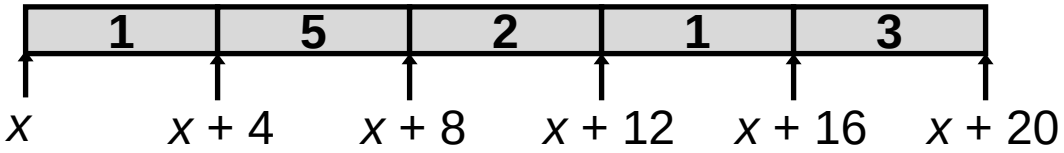
Truy xuất mảng

- **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]
val
val+1
&val[2]
val[5]
*(val+1)
val + 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:

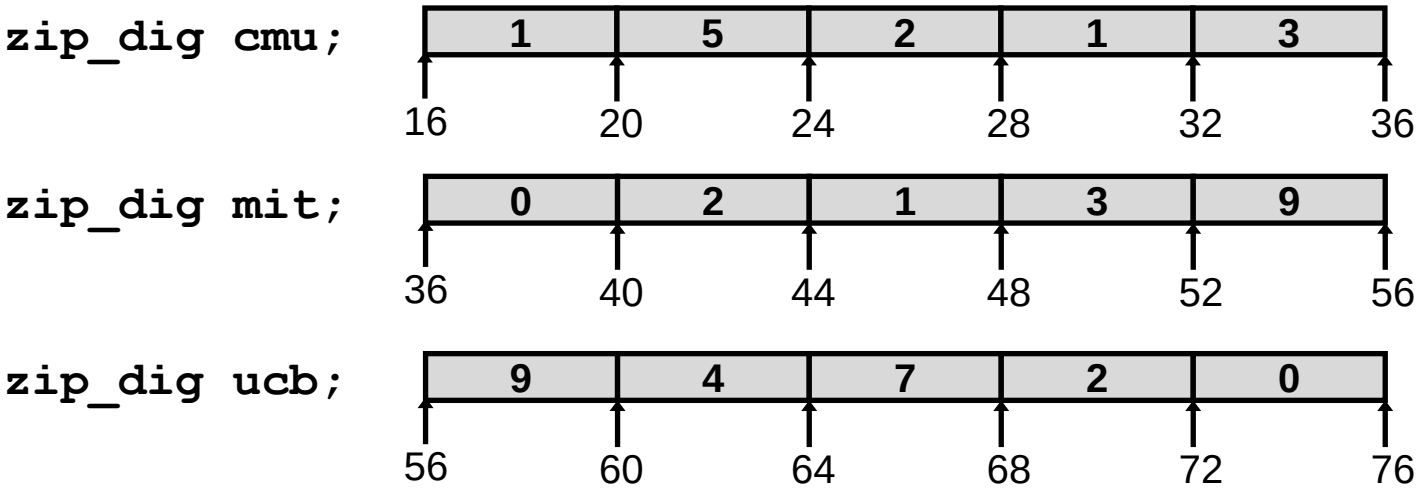
Mảng	Kích thước phần tử (byte)	Tổng kích thước (byte)	Địa chỉ bắt đầu	Địa chỉ phần tử thứ *i*
A			x_A	
B			x_B	
C			x_C	
D			x_D	
E			x_E	

Mảng: Ví dụ

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

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] = ??
cmu[6] = ??
mit[4] = ??
mit[7] = ??

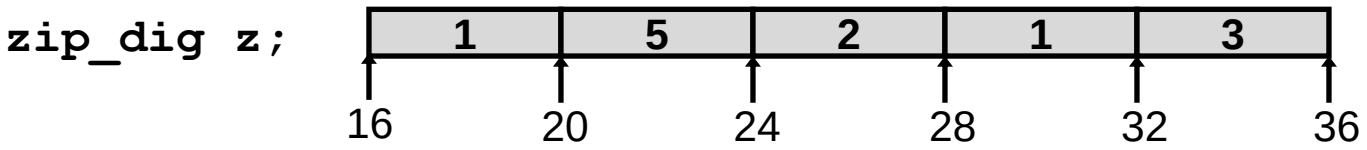


- **Khai báo "`zip_dig cmu`" tương đương với "`int cmu[5]`"**
- **Các mảng ví dụ được cấp phát trong các block 20 bytes liên tiếp nhau**
  - Không phải lúc nào cũng đảm bảo như vậy

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++
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?

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
```

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

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 kiểu dữ liệu**, N** là số phần tử và đều chưa biết. Biết tổng kích thước của **A** là **20 bytes.**

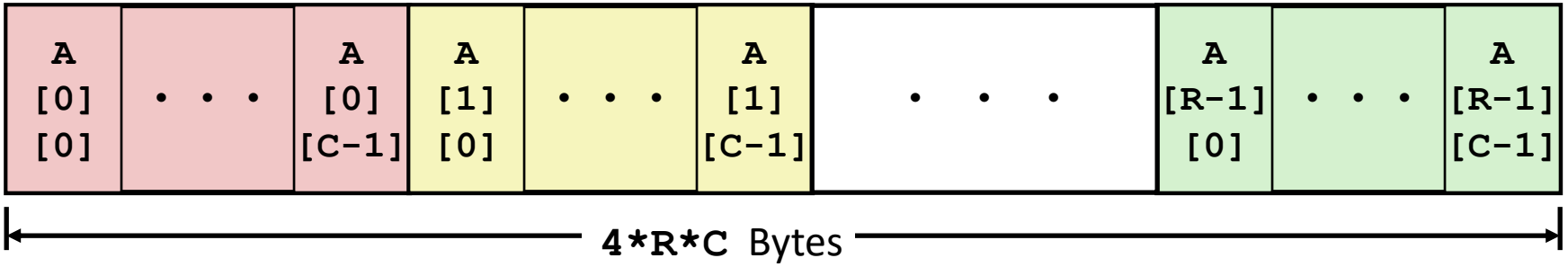
Tìm **T và N?**

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

$$\begin{bmatrix} A[0][0] & \cdots & A[0][C-1] \\ \vdots & & \vdots \\ A[R-1][0] & \cdots & A[R-1][C-1] \end{bmatrix}$$

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



Mảng 2 chiều: Ví dụ

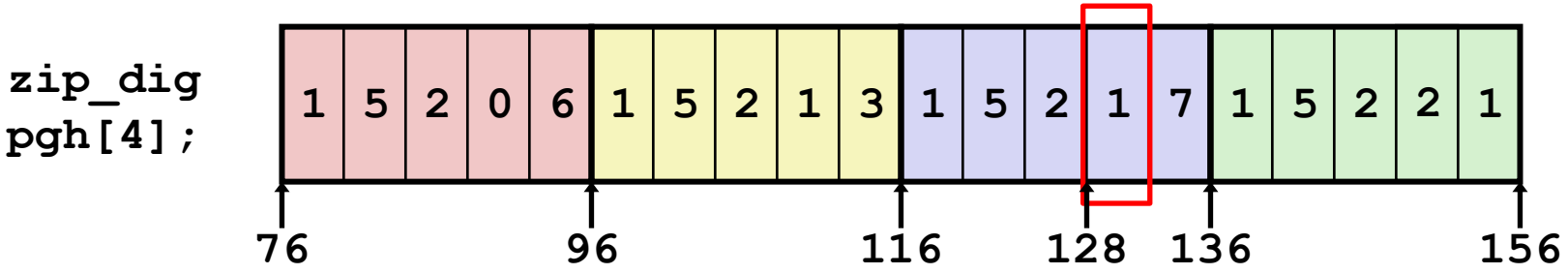
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

Yêu cầu lấy giá trị 1 trên hình?

Cú pháp trong C?

Xác định địa chỉ?

Địa chỉ của pgh[i][j]?

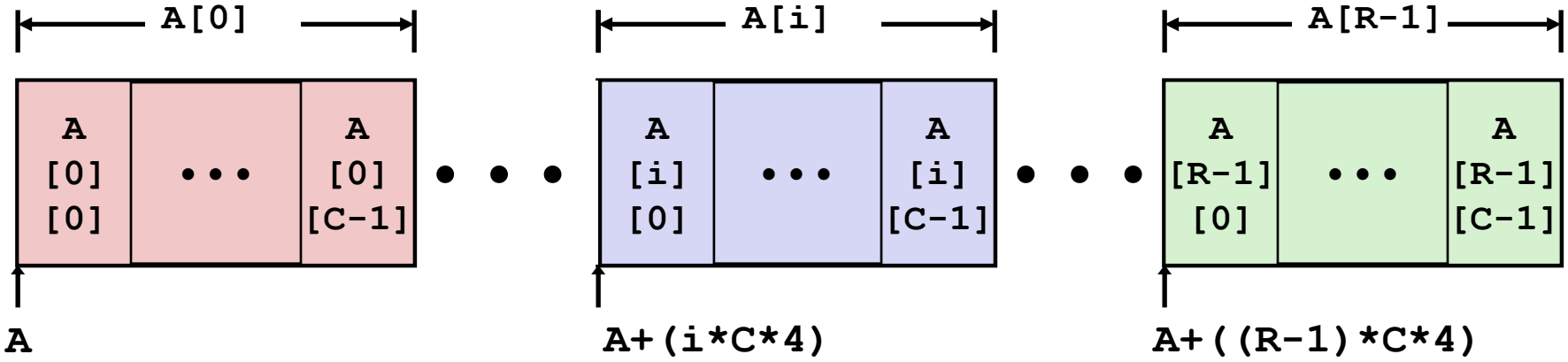


- **"`zip_dig pgh[4]`" tương đương với "`int pgh[4][5]`"**
  - Biến **`pgh`**: Mảng 4 phần tử, được cấp phát liên tục
  - Mỗi phần tử là một mảng 5 số **`int`**, được cấp phát liên tục
- **Sắp xếp tất cả các phần tử đảm bảo theo "Row-Major"**

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];
```

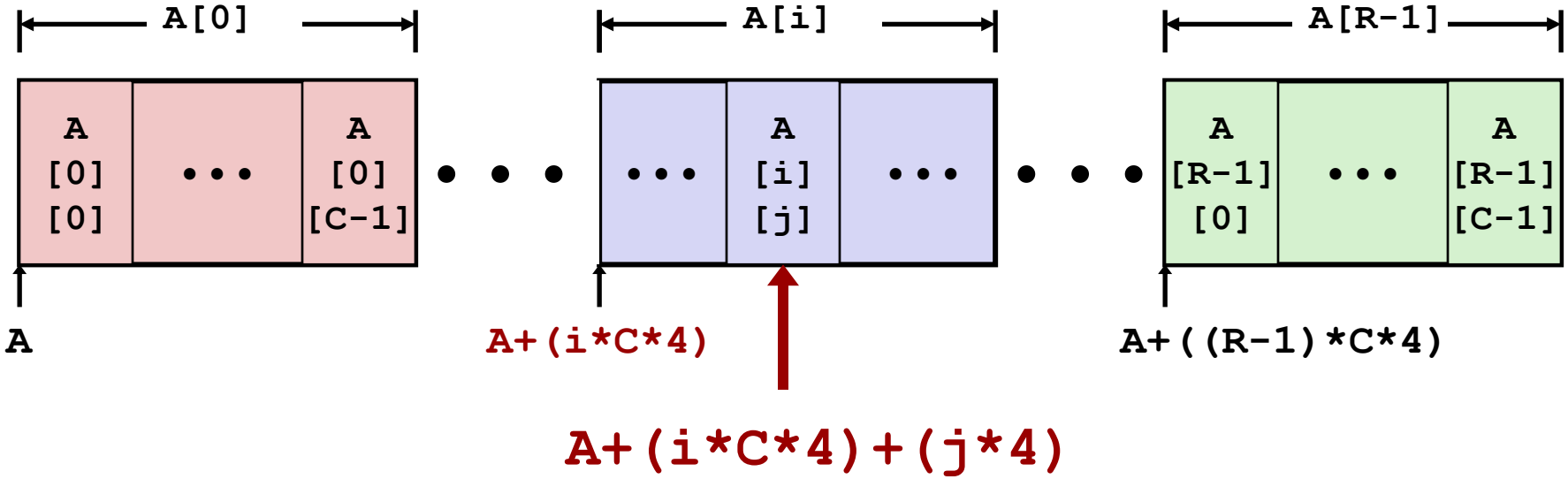


Truy xuất phần tử trong mảng 2 chiều

- **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`

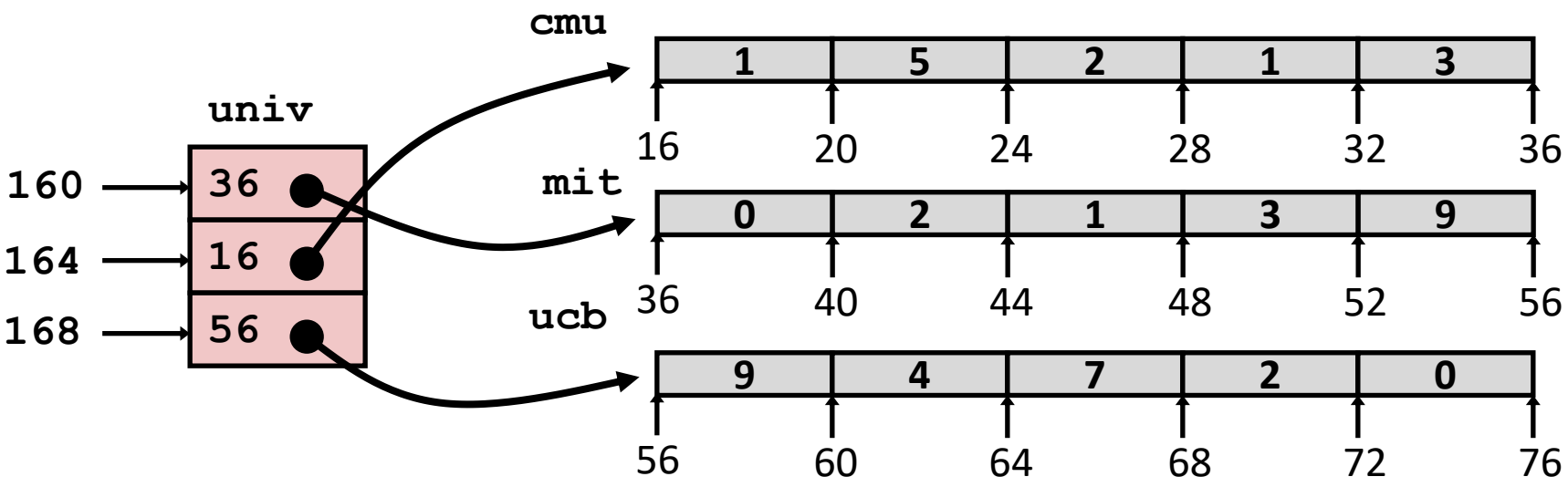
    = `pgh + 4*(5*index + dig)`
- **IA32 Code**
  - Tính toán địa chỉ `pgh + 4*((index+4*index)+dig)`

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

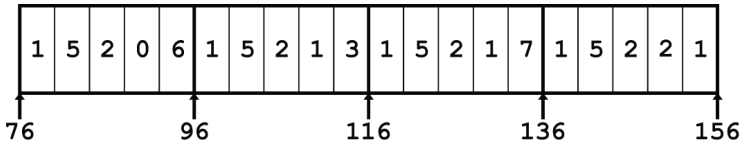
Truy xuất phần tử mảng

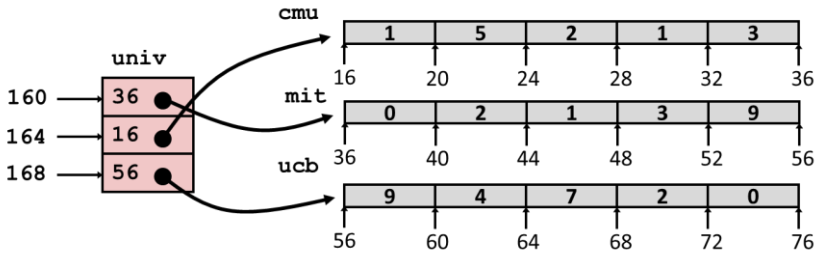
Nested array

```
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 array[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 = array[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.**

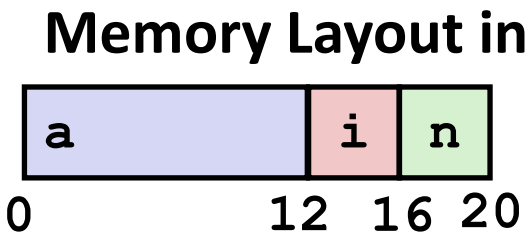
Tìm **T** và **N**?

Nội dung

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

Cấp phát Structure

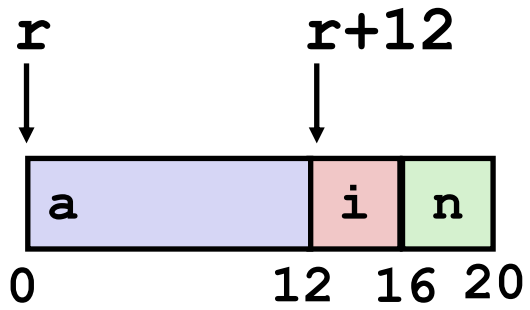
```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```



- **Ý tưởng**
  - Là một vùng nhớ được cấp phát liên tục
  - Tham chiếu đến các thành phần trong structure bằng tên
  - Các thành phần có thể khác kiểu dữ liệu
- **Các trường được sắp xếp dựa trên định nghĩa**
  - Ngay cả khi cách sắp xếp khác có thể biểu diễn gọn hơn
- **Compiler quyết định kích thước tổng + vị trí các trường**
  - Các chương trình mức máy tính không biết về cấu trúc trong source code

Truy xuất structure

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



- **Truy xuất các thành phần trong structure dựa trên:**
  - Con trỏ xác định vị trí ***bắt đầu*** của structure
  - ***Offset*** hay ***khoảng cách*** từ vị trí bắt đầu structure đến vị trí của từng thành phần

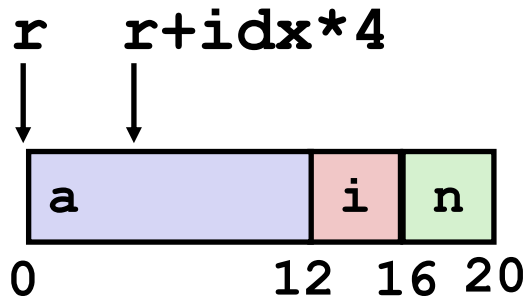
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32 Assembly

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```

Con trỏ đến thành phần Structure - 1

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



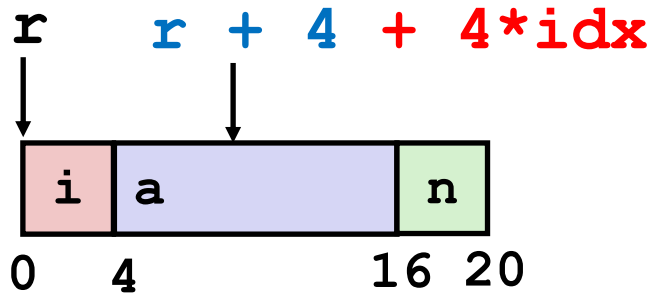
- **Tạo con trỏ đến thành phần trong structure**
  - Offset của mỗi thành phần structure được xác định lúc biên dịch
  - Ví dụ: con trỏ đến `a[idx]` trong structure: `r + 4*idx`

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

```
movl   12(%ebp), %eax   # Get idx
sall   $2, %eax         # idx*4
addl   8(%ebp), %eax    # r+idx*4
```

Con trỏ đến thành phần Structure - 2

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



- **Tạo con trỏ đến thành phần trong structure**
  - Offset của mỗi thành phần structure được xác định lúc biên dịch
  - Ví dụ: con trỏ đến `a[idx]` trong structure: `r + 4 + 4*idx`

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

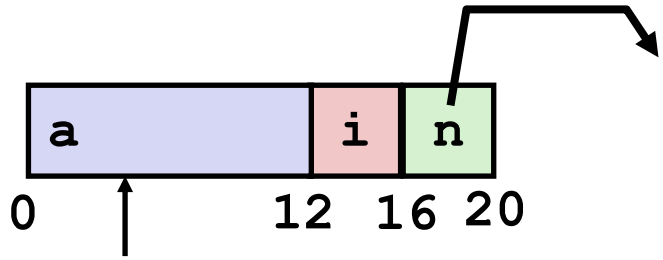
```
 movl   8(%ebp),%ebx      # Get r
 leal   4(%ebx),%esi      # r+4
 movl   12(%ebp),%edi     # Get idx
 sall   $2,%edi           # 4*idx
 leal   (%edi,%esi),%eax  # r+4+4*idx
```

Ví dụ: Danh sách liên kết

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

- C Code

```
void set_val(struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->n;
  }
}
```



Phần tử thứ **i**

Register	Value
%edx	r
%ecx	val

```
.L17:                           # loop:
   movl   12(%edx), %eax        # r->i
   movl   %ecx, (%edx,%eax,4)   # r->a[i] = val
   movl   16(%edx), %edx        # r = r->n
   testl  %edx, %edx            # Test r
   jne    .L17                  # If != 0 goto loop
```

Ví dụ: Truy xuất structure

- C Code

```
struct example{
        int b;
        double p;
        short a[4];
        char* c;
};
```

- Xác định offset của các trường trong IA32?

Trường	Offset
b	
p	
c	
a[i]	

Dự đoán tổng kích thước struct?

- C Code

```
struct example{
        int b;
        float d;
        double p;
        short a[4];
        char c;
};
```

Kích thước struct example = ?

- Kích thước từng trường?

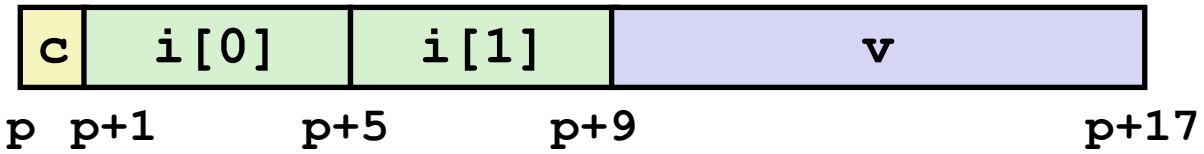
Trường	Offset	Kích thước
b		
d		
p		
c		
a		

Structure & Alignment (căn chỉnh)

- **Dữ liệu được căn chỉnh**
  - Kiểu dữ liệu yêu cầu **K** bytes
  - Địa chỉ phải là bội số của **K**
  - Bắt buộc ở một số hệ thống; được khuyến cáo ở x86_64
- **Vì sao?**
  - Truy xuất bộ nhớ hiệu quả hơn khi alignment
  - Bộ nhớ được truy xuất bằng các khối 4 hoặc 8 byte (tuỳ hệ thống)
    - Load hoặc lưu các dữ liệu lớn hơn 8 bytes không hiệu quả
    - Bộ nhớ ảo phức tạp hơn khi dữ liệu lớn hơn 2 pages
- **Compiler**
  - Thêm những **khoảng trống vào structure** để đảm bảo căn chỉnh đúng cho các trường dữ liệu

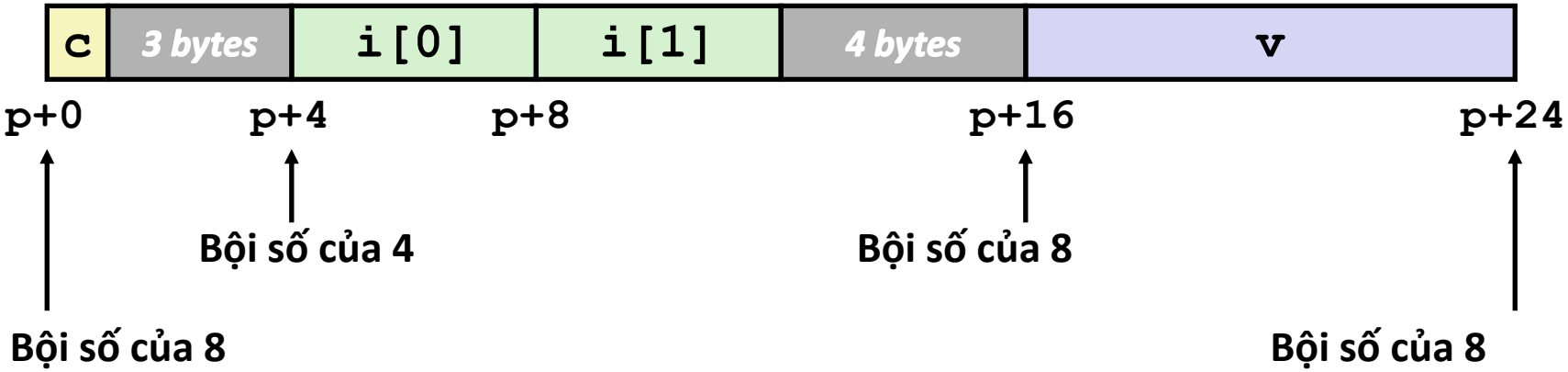
Structure & Alignment (căn chỉnh)

- **Dữ liệu không căn chỉnh**




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

- **Dữ liệu được căn chỉnh**
  - Kiểu dữ liệu yêu cầu **K** bytes
  - Địa chỉ phải là bội số của **K**



Yêu cầu căn chỉnh (IA32)

- **1 byte: `char`, …**
  - Không có ràng buộc về địa chỉ
- **2 bytes: `short`, …**
  - Bit thấp nhất của địa chỉ phải bằng 0_2
- **4 bytes: `int, float, char *` …**
  - 2 bit thấp nhất của địa chỉ phải bằng 00_2
- **8 bytes: `double`,** …
  - Windows (và hầu hết các OS và instruction set khác):
    - 3 bits thấp nhất của địa chỉ phải bằng 000_2
  - Linux:
    - 2 bit thấp nhất của địa chỉ phải bằng 00_2 (xem như xử lý kiểu dữ liệu 4 byte)
- **12 bytes: `long double`**
  - Windows, Linux:
    - 2 bit thấp nhất của địa chỉ phải bằng 00_2 (xem như xử lý kiểu dữ liệu 4 byte)

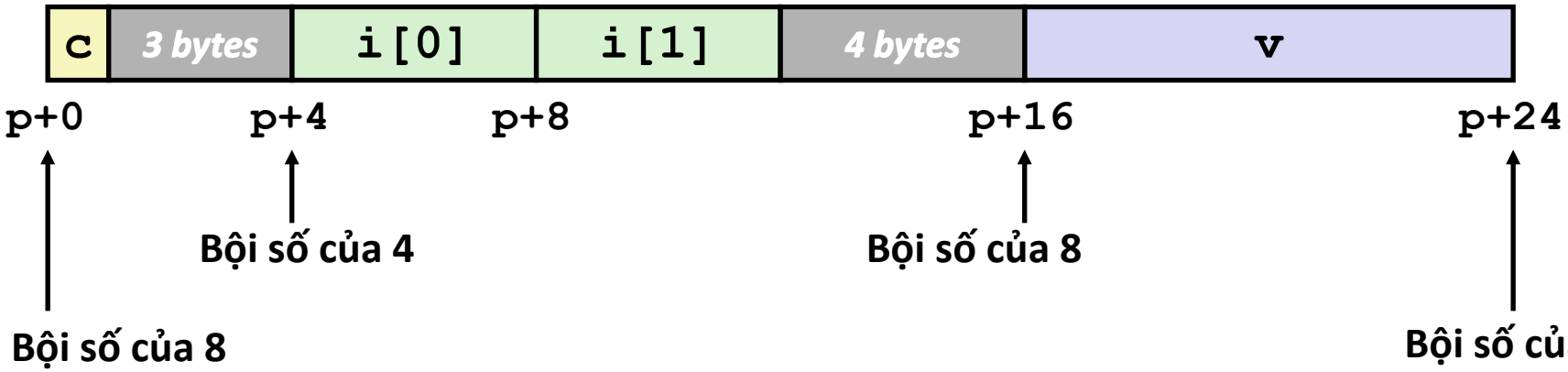
Yêu cầu căn chỉnh (x86_64)

- **1 byte: `char`, …**
  - Không có ràng buộc về địa chỉ
- **2 bytes: `short`, …**
  - Bit thấp nhất của địa chỉ phải bằng 0_2
- **4 bytes: `int`, `float`, …**
  - 2 bit thấp nhất của địa chỉ phải bằng 00_2
- **8 bytes: `double`, `long`, `char *`, …**
  - 3 bit thấp nhất của địa chỉ phải bằng 000_2
- **16 bytes: `long double`** (GCC on Linux)
  - 4 bit thấp nhất của địa chỉ phải bằng 0000_2

Đảm bảo căn chỉnh với Structure

- **Trong structure**
  - Phải đảm bảo yêu cầu căn chỉnh của mỗi thành phần
- **Vị trí chung của structure**
  - Mỗi structure có yêu cầu căn chỉnh **K**
    - **K** = Yêu cầu căn chỉnh lớn nhất của các thành phần
  - Địa chỉ bắt đầu & kích thước structure phải là bội số của **K**
- **Ví dụ trong hệ thống 64 bit:**
  - **K** = 8, do thành phần có **K** lớn nhất là kiểu **`double`**

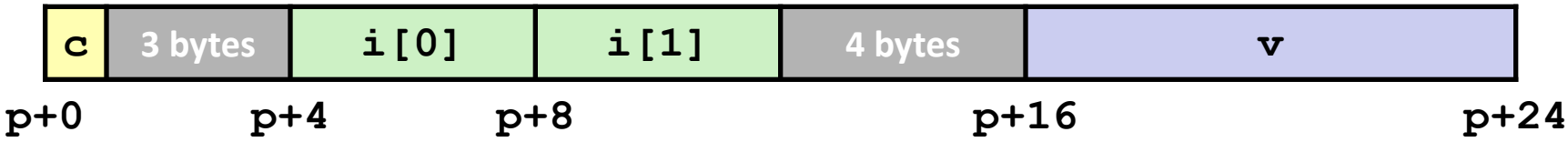
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



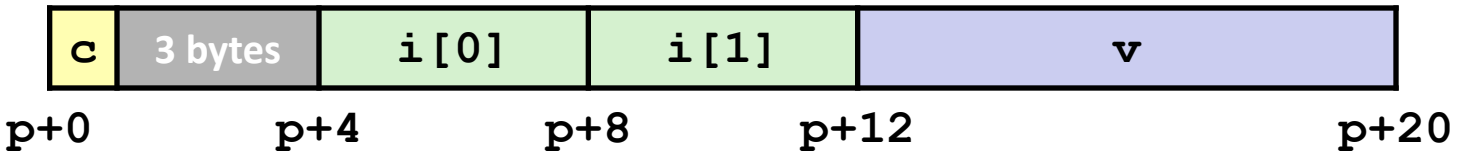
Ví dụ: Các quy ước căn chỉnh khác nhau

- **x86-64 hoặc IA32 Windows:**
  - K = 8, do thành phần lớn nhất có kiểu **`double`**

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



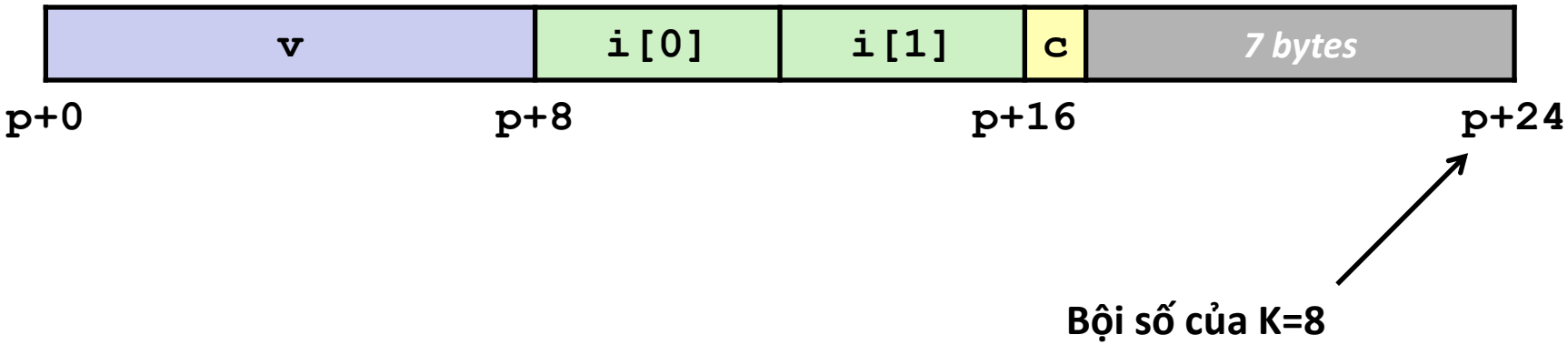
- **IA32 Linux**
  - K = 4; kiểu **`double`** vẫn được xử lý như kiểu dữ liệu 4-byte



Đảm bảo yêu cầu căn chỉnh chung

- Với yêu cầu căn chỉnh lớn nhất **K**
- Structure phải có kích thước là bội số của **K**

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



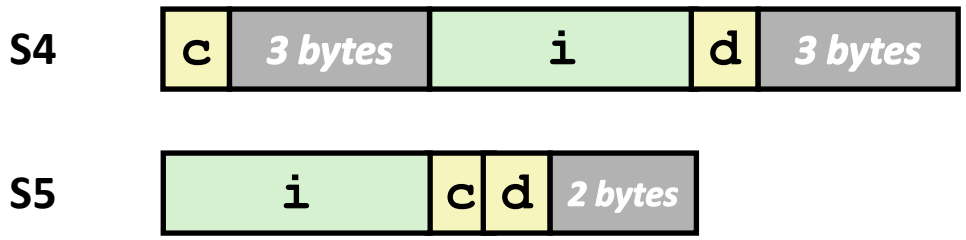
Tiết kiệm không gian lưu trữ

- Khai báo các kiểu dữ liệu lớn trước

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

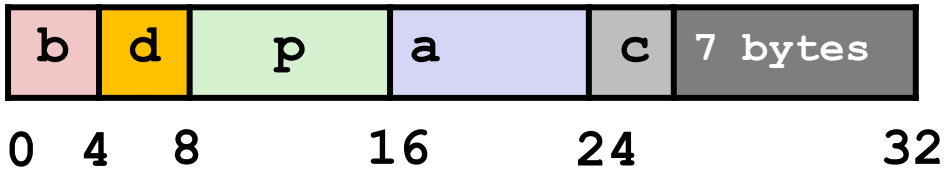
- Tác dụng (K=4)



Tổng kích thước struct (có căn chỉnh)

- C Code

```
struct example{
        int b;
        float d;
        double p;
        short a[4];
        char c;
};
```



Structure phải có kích thước là bội số của K(double)

Kích thước struct example = 32

Tổng kích thước struct

```c
#include <stdio.h>

struct example{
    int b;
    float d;
    double p;
    short a[4];
    char c;
};


int main()
{
    struct example ex1;
    printf("Total size of ex1 is %d\n",sizeof(ex1));
    return 0;
}
```

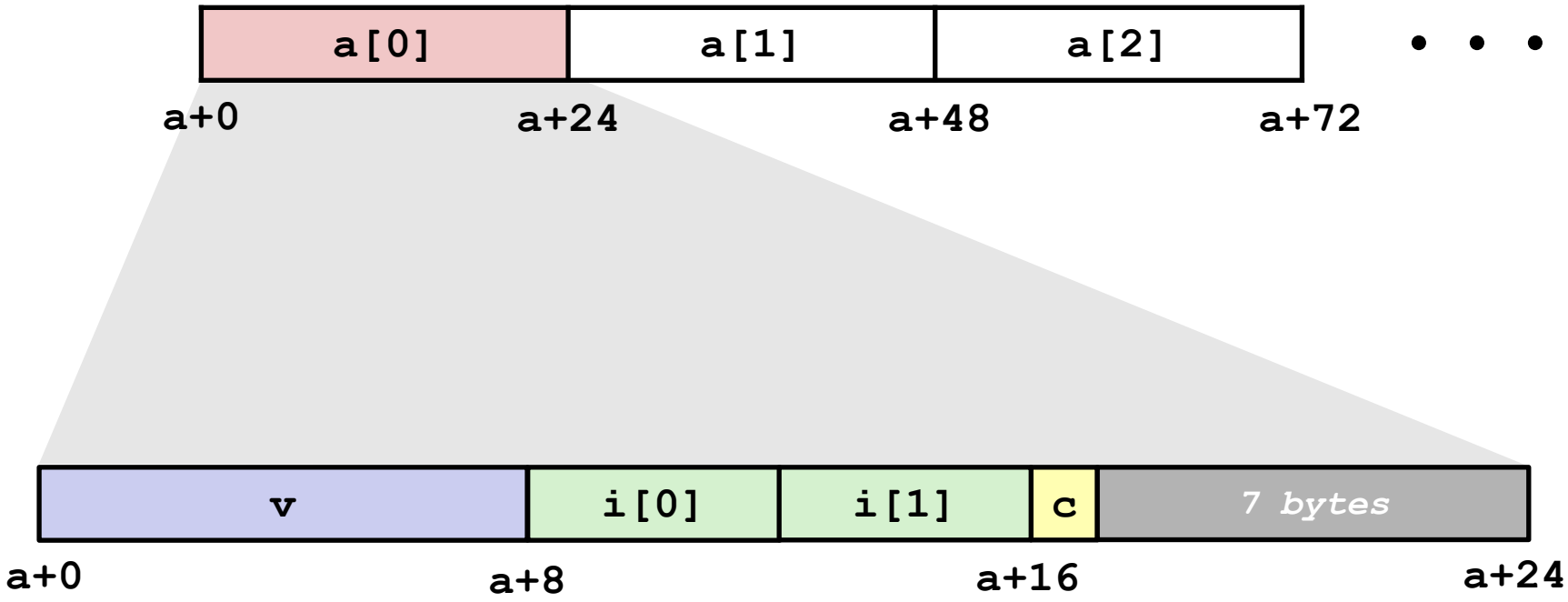
```
$gcc -o main *.c
$main
Total size of ex1 is 32
```

Thêm: Căn chỉnh trong mảng Structures

- Kích thước structure là bội số của K
- Đảm bảo yêu cầu căn chỉnh cho tất cả thành phần

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

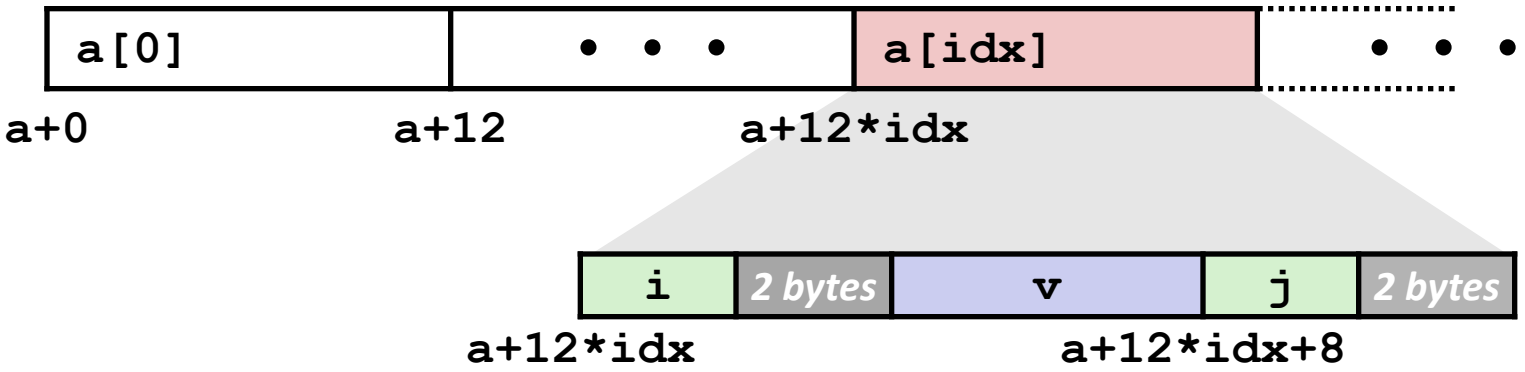
- **Structure S2 trong x86_64**: K = 8



Thêm: Truy xuất phần tử mảng structure

- **Offset để truy xuất phần tử mảng `a`: `12*idx`**
  - `sizeof(S3)=12` gồm cả khoảng trống để căn chỉnh
- **Thành phần `j` nằm ở offset 8 trong structure**
- **Assembler cung cấp sẵn offset `a+8`**
  - Qua quá trình linking

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```




```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

Structure & Alignment: Bài tập 1 (*)

Cho định nghĩa cấu trúc như sau được biên dịch trên 1 máy Windows 32-bit và có yêu cầu alignment.

```
struct {
   char  *a;
   short  b;
   double c;
   char   d;
   float  e;
   void  *f;
} foo;
```

a. Xác định vị trí (offset) của từng trường trong structure này? Vẽ hình minh họa việc cấp phát structure?

b. Tổng kích thước của structure?

c. Sắp xếp lại vị trí các trường để hạn chế tối thiểu không gian trống? Offset của các trường và tổng kích thước sau khi sắp xếp lại?

Nội dung

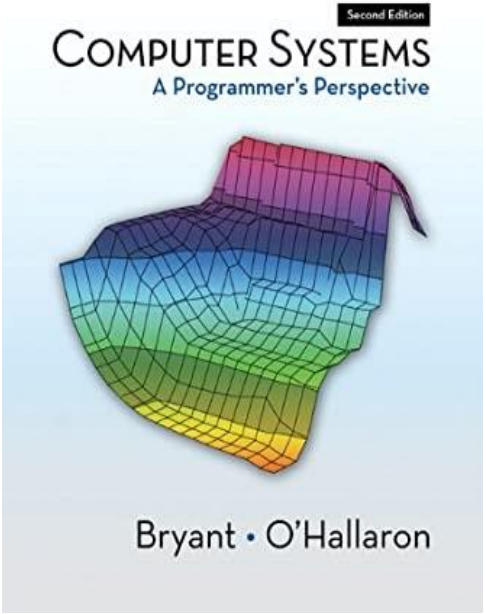
- **Các chủ đề chính:**
  1) Biểu diễn các kiểu dữ liệu và các phép tính toán bit
  2) Ngôn ngữ assembly
  3) Điều khiển luồng trong C với assembly
  4) Các thủ tục/hàm (procedure) trong C ở mức assembly
  5) Biểu diễn mảng, cấu trúc dữ liệu trong C
  6) Một số topic ATTT: reverse engineering, bufferoverflow
  7) Phân cấp bộ nhớ, cache
  8) Linking trong biên dịch file thực thi
- **Lab liên quan**
  - Lab 1: Nội dung **1**
  - Lab 2: Nội dung 1, **2**, **3**
  - Lab 3: Nội dung 1, **2, 3, 4, 5, 6**
  - Lab 4: Nội dung 1, **2,** 3**,** **4**, 5, **6**
  - Lab 5: Nội dung 1, **2,** 3**,** **4**, 5, **6**
  - Lab 6: Nội dung **1, 2, 3**, **4, 5, 6**

Giáo trình

- **Giáo trình chính**

Computer Systems: A Programmer's Perspective

  - Second Edition (CS:APP2e), Pearson, 2010
  - Randal E. Bryant, David R. O'Hallaron
  - http://csapp.cs.cmu.edu



- **Tài liệu khác**
  - *The C Programming Language*, Second Edition, Prentice Hall, 1988
    - Brian Kernighan and Dennis Ritchie
  - *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 1st Edition, 2008
    - Chris Eagle
  - *Reversing: Secrets of Reverse Engineering*, 1st Edition, 2011
    - Eldad Eilam


KEEP
CALM
AND
ENJOY YOUR
SEMESTER :)