Announcements

- Graded lab this week
  - Based on Assignments 2 & 3
- Assignment 5
  - Posted October 26; Due November 4
  - String operations in IA-32

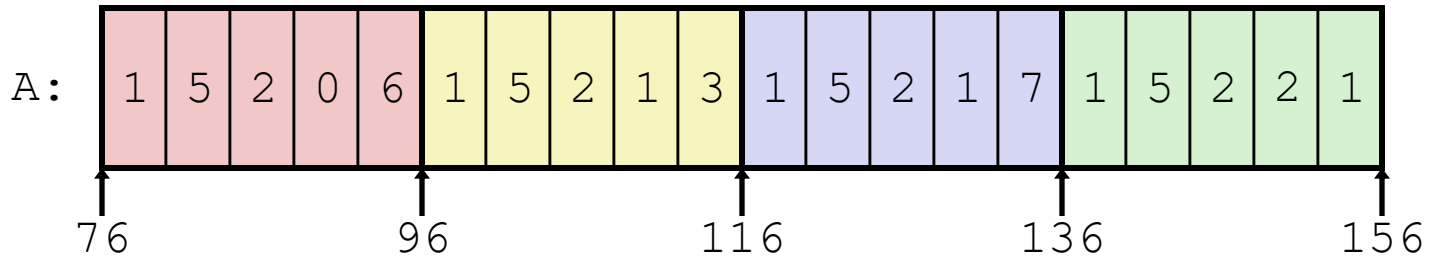
Lecture 21

Multi-Dimensional Arrays

CPSC 275
Introduction to Computer Systems

Multi-Dimensional Arrays

```
A[4][5] = { {1, 5, 2, 0, 6},
            {1, 5, 2, 1, 3},
            {1, 5, 2, 1, 7},
            {1, 5, 2, 2, 1} };
```



- Variable **`A`**: array of 4 elements, allocated contiguously
- Each element is an array of 5 **`int`**'s, allocated contiguously
- *Row-major* ordering of all elements guaranteed

Multi-Dimensional Arrays, cont'd

- Declaration

  T **A**[*R*][*C*];

  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes

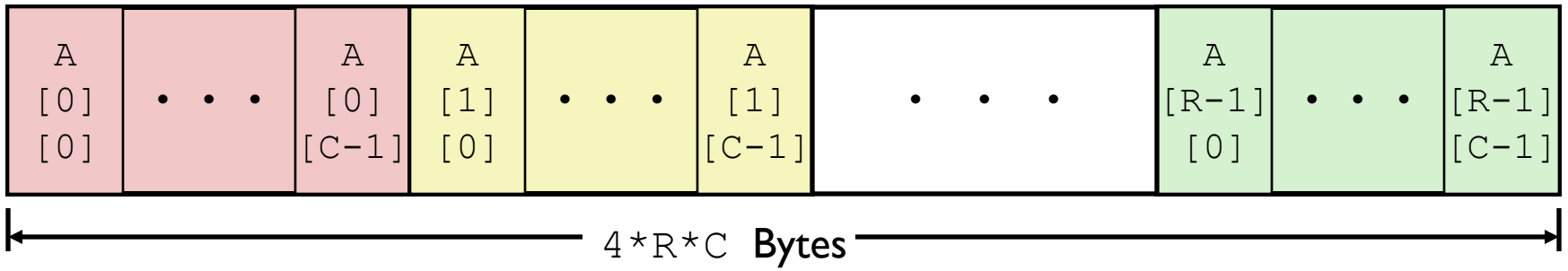
  Q: Array size in terms of # bytes?

  A: *R* * *C* * *K* bytes

- Arrangement - row-major ordering

  int A[R][C];

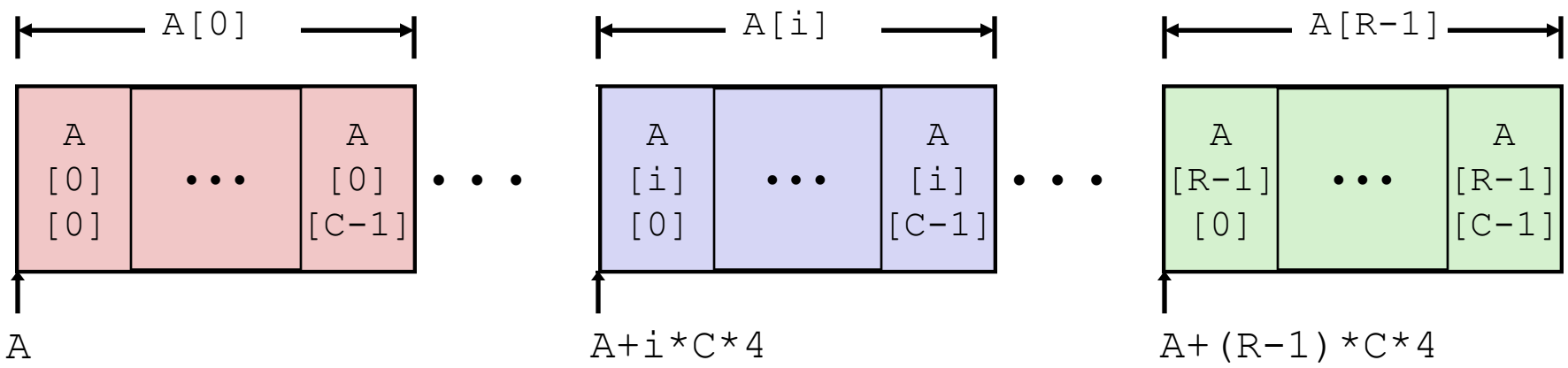
```
[ A[0][0]    • • •    A[0][C-1]
      •                    •
      •                    •
      •                    •
  A[R-1][0]  • • •  A[R-1][C-1] ]
```



Multi-Dimensional Array Row Access

- Row vectors
  - *Q*: How many elements does **`A[i]`** have?
  - *A*: *C* elements
    - Each element of type *T* requires *K* bytes
  - *Q*: What is the starting address **`A[i]`** ?
  - *A*: **`A + i`** * (*C* * *K*)

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



Row Access Code

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

```
int *get_row(int row) {
    return A[row];
}
```

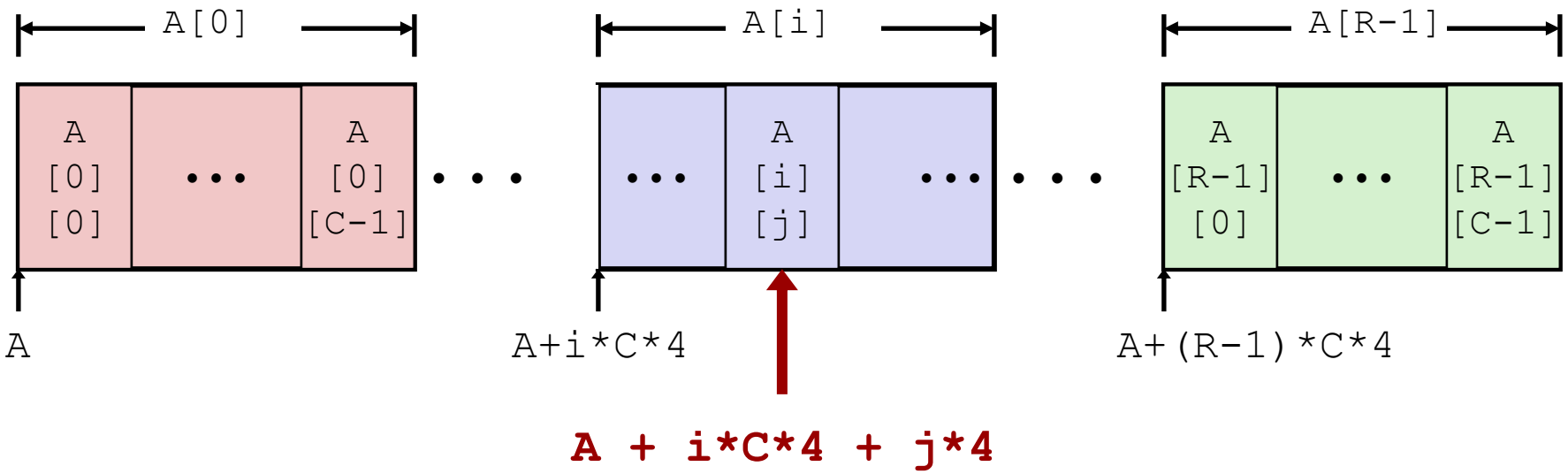
- **`A[row]`** is an array of 5 **`int`**'s
- Its starting address is

  `A + 20*row`

```
                                # %eax = row
leal (%eax,%eax,4),%eax         # 5 * row
leal A(,%eax,4),%eax            # A + (20 * row)
```

Array Element Access

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



- **`A[i][j]`** is element of type *T*, which requires *K* bytes
- Its address?

$$\mathtt{A} + \mathtt{i}*C*K + \mathtt{j}*K = \mathtt{A} + (\mathtt{i}*C + \mathtt{j})*K$$

Nested Array Element Access Code

```
int get_element (int row, int col) {
  return A[row][col];
}
```

- **`A[row][col]` is `int`**
- Its address?

  `A + 20*row + 4*col = A + 4*(5*row + col)`

```
                                    # row at 8(%ebp), col at 12(%ebp)
movl    8(%ebp), %eax               # %eax = row
leal    (%eax,%eax,4), %eax         # 5*row
addl    12(%ebp), %eax              # 5*row + col
movl    A(,%eax,4), %eax            # 4*(5*row+col)
```

Structures and Unions in C

CPSC 275
Introduction to Computer Systems

Structure Variables

- The properties of a ***structure*** are different from those of an array.
  - The elements of a structure (its ***members***) aren't required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.

Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items.
- A declaration of two structure variables that store information about parts in a warehouse:

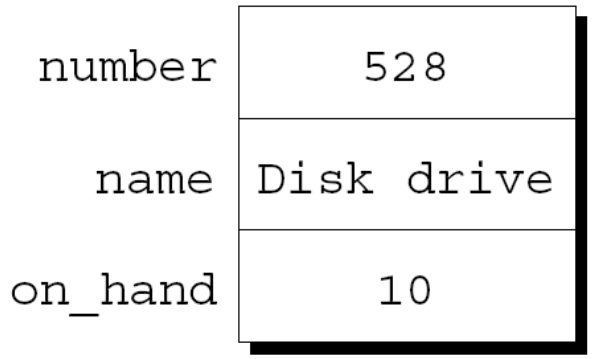
```
#define NAME_LEN 30

struct {
  int number; // part number
  char name[NAME_LEN+1]; // part name
  int on_hand; // # in stock
} part1, part2;
```

Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization:



Operations on Structures

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the member.
- Statements that display the values of `part1`'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operations on Structures

- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
  /* changes part1's part number */

part1.on_hand++;
  /* increments part1's quantity on hand */
```

Operations on Structures

- The other major structure operation is assignment:

```
part2 = part1;
```

- The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Declaring a Structure Tag

- A ***structure tag*** is a name used to identify a particular kind of structure.
- The declaration of a structure tag named `part`:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Note that a semicolon must follow the right brace.

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
- A definition of a type named `Part`:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` can be used in the same way as the built-in types:

```
Part part1, part2;
```

Structures as Arguments

- Functions may have structures as arguments.
- A function with a structure argument:

```
void print_part(struct part p) {
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`:

```
print_part(part1);
```

Structures as Return Values

- A function that returns a `part` structure:

```
struct part build_part(int number,
                       const char *name,
                       int on_hand) {
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- A call of `build_part`:

```
part1 = build_part(528, "Disk drive", 10);
```

Passing a Pointer to Structure

- Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure.
- To avoid this overhead, it's sometimes advisable to pass a pointer to a structure or return a pointer to a structure.

```
void print_part(struct part *p) {
  printf("Part number: %d\n", p->number);
  printf("Part name: %s\n", p->name);
  printf("Quantity on hand: %d\n", p->on_hand);
}
```

- A call of `print_part`:

```
print_part(&part1);
```

Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
- Arrays may have structures as their elements, and structures may contain arrays and structures as members.
- Suppose that `person_name` is the following structure:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

Nested Structures

- We can use `person_name` as part of a larger structure:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing `student1`'s first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of `part` structures capable of storing information about 100 parts:

```
struct part inventory[100];
```

Arrays of Structures

- Accessing a part in the array is done by using subscripting:

```
print_part(inventory[i]);
```

- Accessing a member within a `part` structure requires a combination of subscripting and member selection:

```
inventory[i].number = 883;
```

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:

```
inventory[i].name[0] = '\0';
```

Unions

- A ***union,*** like a structure, consists of one or more members, possibly of different types.
- The compiler allocates *only enough space* for the *largest* of the members, which overlay each other within this space.
- Assigning a new value to one member alters the values of the other members as well.

Unions

- An example of a union variable:

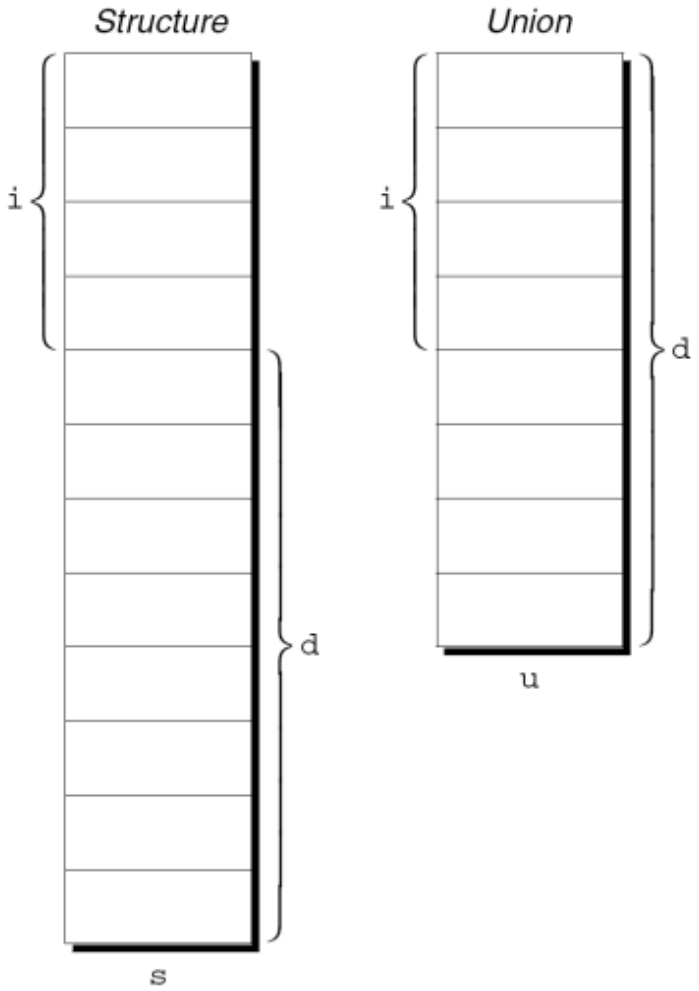
```
union {
  int i;
  double d;
} u;
```

- The declaration of a union closely resembles a structure declaration:

```
struct {
  int i;
  double d;
} s;
```

Unions

- The structure `s` and the union `u` differ in just one way.
- The members of `s` are stored at different addresses in memory.
- The members of `u` are stored at the same address.



Unions

- Members of a union are accessed in the same way as members of a structure:

```
u.i = 82;
u.d = 74.8;
```

- Changing one member of a union alters any value previously stored in any of the other members.
  - Storing a value in `u.d` causes any value previously stored in `u.i` to be lost.
  - Changing `u.i` corrupts `u.d`.

Unions

- The properties of unions are almost identical to the properties of structures.
- We can declare union tags and union types in the same way we declare structure tags and types.
- Like structures, unions can be copied using the = operator, passed to functions, and returned by functions.