[Computer Architecture]

Programming in C

Structures and unions

Agenda

- Structure
  - Syntax
  - Applications
- Union
  - Syntax

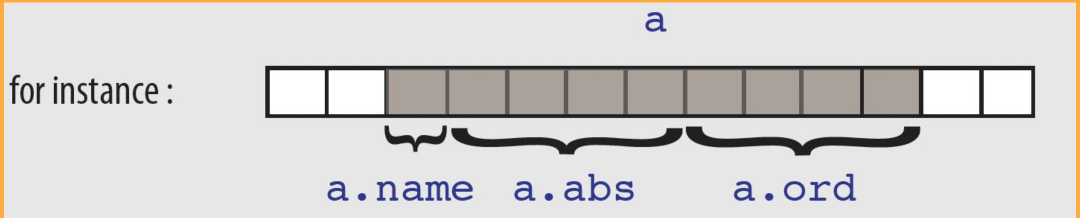
Structure type declaration

```c
//declare a structure to represent
//the coordinates of the plane
//labelled by chars.
struct coord {
    char name; //the name of the point
    int abs;   //abscissa, i.e., x coordinate;
    int ord;   //ordinate, i.e., y coordinate;
};

int main(){
    struct coord a; // variable a of type struct coord
    // storing values in the fields of a
    a.name = 'a';
    a.abs = 10;
    a.ord = 10;
    //read and print the values of the fields of a
    printf("%c %d %d\n", a.name, a.abs, a.ord);
}
```

- `struct` (short for structure) is a user-defined data type that allows grouping of different type of variables under a single name.
- Each element in a structure is called a field (or member)
  - e.g., `name`, `abs`, `ord` are fields of `coord`.
- To access the fields, one can use the dot operator (`.`)
  - e.g., `a.name`, `a.abs`, `a.ord`
- The field can also be a **pointer**.

The structure variable

- `struct coord a;` in the function `main()` ⇒ allocate in memory a region of memory sufficiently large to store a `char` and two `int`s.



- `a.name, a.abs, a.ord` ⇒ gives access to the field of the variable `a`. (i.e., `var_name.field_name`)

Immediate and partial initiation

- `struct coord a;` ⇒ the fields of `a` are allocated but not initialized (undefined values)
- `struct coord a = {'a', 38, 50};` ⇒ the fields of `a` are all initialized by the field in the order the field were declared: `name` then `abs` then `ord`
- `struct coord a = {'a'};` ⇒ the first field of `a` is initialized, the following fields take the value 0 of their type.

Assignment of structure variables

```c
int main(){
    struct coord a = {'a', 38, 17} , b;
    b = a;  //copy the values of the fields of a
            //into the fields of b.
    b.name = 'b';
    b.abs = b.abs + 10;
    b.ord = b.ord + 10;

    printf("%c %d %d\n", a.name, a.abs, a.ord);
    printf("%c %d %d\n", b.name, b.abs, b.ord);
    //a 38 17
    //b 48 27
    return 0;
}
```

- `b=a;` will make a copy of `a` to `b`.

Structures as function parameters

```c
struct coord translate(struct coord c, int dx, int dy){
    c.abs = c.abs + dx;
    c.ord = c.ord + dy;
    return c;
}

int main(){
    struct coord a = {'a', 38, 17}, b;
    printf("%c %d %d\n", a.name, a.abs, a.ord);
    //a 38 17
    b = translate(a, 10, 10);
    b.name = 'b';
    printf("%c %d %d\n", b.name, b.abs, b.ord);
    //b 48 27
    return 0;
}
```

Structures can be used as function parameters and as return value.

- The function `translate` takes a `struct coord` as one of the parameters and returns a `struct coord` type value.

Note: In this example, it uses the call-by-value method for passing parameters.

What will happen when the translate is called?

```c
24 struct coord translate(struct coord c, int dx, int dy){
25     c.abs = c.abs + dx;
26     c.ord = c.ord + dy;
27     return c;
28 }
29 
30 int main(){
31     struct coord a = {'a', 38, 17}, b;
32     printf("%c %d %d\n", a.name, a.abs, a.ord);
33     //a 38 17
34     b = translate(a, 10, 10);
35     b.name = 'b';
36     printf("%c %d %d\n", b.name, b.abs, b.ord);
37     //b 48 27
38     return 0;
39 }
```

It uses the *call-by-value* method for passing parameters:

1. In line 34, `translate` is called; so, it will **copy** `a` to the local variable `c` in the `translate`.
2. Inside the `translate`, the fields `abs` and `ord` will be updated.
3. In line 27, `c` is returned, which means, it will be **copied** to the variable `b` in `main`.
4. When the call ends, the local variable `c` will vanish.

Structures vs. address as parameters

```c
void translate(struct coord *ps, int dx, int dy){
    (*ps).abs += dx;
    (*ps).ord += dy;
}

int main(){
    struct coord a = {'a', 38, 17}, b;
    printf("%c %d %d\n", a.name, a.abs, a.ord);
    //a 38 17
    translate(&a, 10, 10);
    printf("%c %d %d\n", a.name, a.abs, a.ord);
    //a 48 27
    return 0;
}
```

It is in general more efficient to give a pointer as parameter than a whole structure.

If the task is to shift the `a`, then, passing the pointer is faster than passing the whole structure because no need to return (i.e., copy) the local variable.

In the example, the fields of `a` are modified "in-place"

A useful notation

```c
void translate(struct coord *ps, int dx, int dy){
    (*ps).abs += dx;
    (*ps).ord += dy;
}

void translate(struct coord *ps, int dx, int dy){
    ps->abs += dx;
    ps->ord += dy;
}
```

`(*ps).field_name` is equal to `ps->field_name`

- We have a simpler way to access the fields of a pointer of structure:

`(*ps).name` can be alternatively written `ps -> name`

`(*ps).abs` can be alternatively written `ps -> abs`

`(*ps).ord` can be alternatively written `ps -> ord`

Comparing two structures

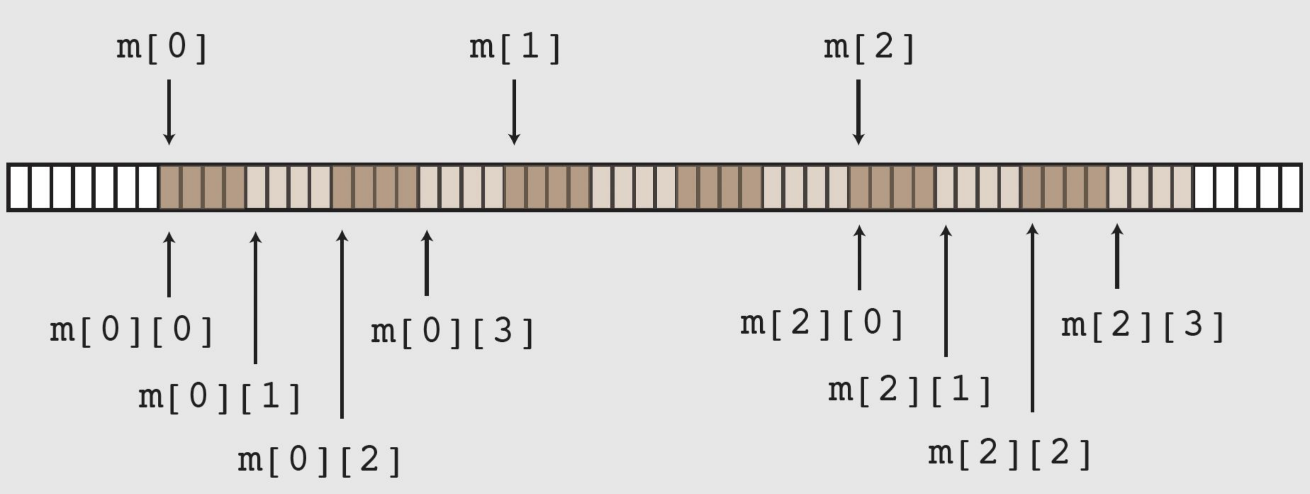
Structures can be copied, sent to a function as parameter, returned to a function as returned value.

However, structures **cannot** be directly compared by == or !=

So, in order to determine whether two structures are equal, one needs to compare their fields:

```
if (a.name == b.name &&
    a.abs  == b.abs  &&
    a.ord  == b.ord) { ... }
```

Typical applications of structures



- a 2D array allocated in the memory: elements can be accessed by (row, column) pairs, but essentially, they are stored in a group of consecutive memory cells.

Accessing elements in a 2D array

```c
void print_matrix(int *content, int height, int width){
    int i, j;

    for (i=0; i<height; i++){
        for (j=0; j<width; j++){
            printf("%3d", content[(width*i)+j]);
        }
        printf("\n");
    }
}

int main(){
    int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    print_matrix(array, 3, 4);
    return 0;
}
```

Instead of using an array of arrays, a matrix of integers of size `height * width` can be represented as an array of integers with the convention that:

`m(i,j)` is stored in `array[i * width + j]`

In short, given a dataset stored in an 1D array, we can treat it as a matrix of arbitrary dimensions. (e.g., a matrix of 3 x 4 or 2 x 6)

Representing 2D array using structure

```c
struct matrix{
    int height;
    int width;
    int *content;
};

int element(struct matrix *pm, int i, int j){
    return pm->content[(pm->width*i)+j];
};

int main(){
    int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    int row=3, col=4;
    struct matrix m={row, col, array};
    int i, j;

    for(i=0; i<row; i++){
        for(j=0; j<col; j++){
            printf("%3d", element(&m, i, j));
        }
        printf("\n");
    }
}
```

printed:

```
0   1   2   3
4   5   6   7
8   9  10  11
```

We define a structure `matrix` that represents the shape and elements of the matrix.

The function `element` returns the value at the position `(i, j)` according to the `matrix` we defined.

- Changing the row to 2 and col to 6, the array will be treated as a matrix of 2 x 6.

Rewriting the print_matrix

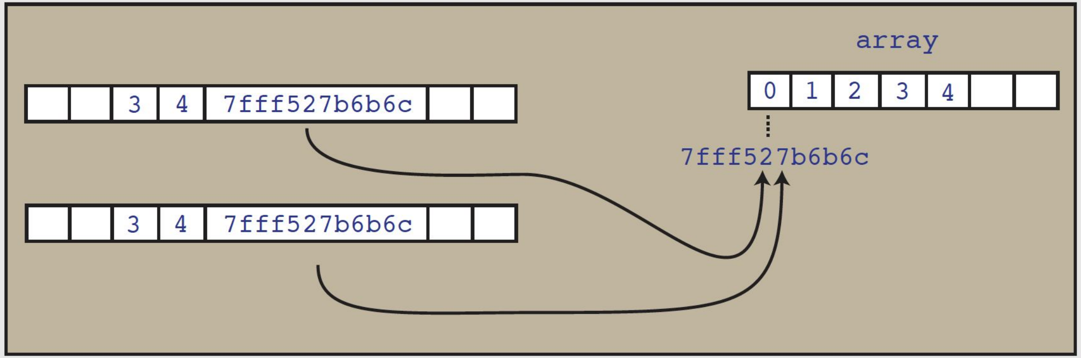
```c
struct matrix{
    int height;
    int width;
    int *content;
};

int element(struct matrix *pm, int i, int j){
    return pm->content[(pm->width*i)+j];
};

void print_matrix(struct matrix *pm){
    int i, j;
    for (i=0; i<pm->height; i++){
        for (j=0; j<pm->width; j++){
            printf("%3d", element(pm, i, j));
        }
        printf("\n");
    }
}

int main(){
    int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    int row=3, col=4;
    struct matrix m={row, col, array};
    print_matrix(&m);
}
```

The `main` wraps the functions; the logic of the program is clearly demonstrated.

Remark

Copying these structures by assignment does not duplicate the associated array, only its address: (like the shallow copy in Python)

```
int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
  struct m = {.height = 3, .width = 4, .content = array};
  struct n = m;
```



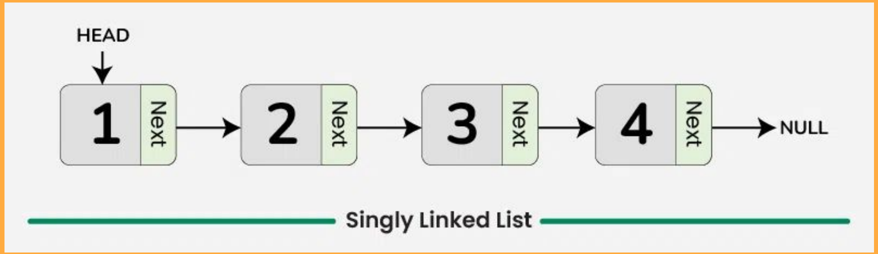
Here, `m.content` and `n.content` contain the same address: the address of `array` hence `m` and `n` share the same content.

Application: construct a linked list

A linked list is a sequence of nodes where each node contains two parts:

- **Data**: The value stored in the node.
- **Pointer**: A reference to the next node in the sequence.

Unlike arrays, linked lists **do not** store elements in **contiguous** memory locations. Instead, each node points to the next, forming a chain-like structure and to access any element (node), we need to first sequentially traverse all the nodes before it.

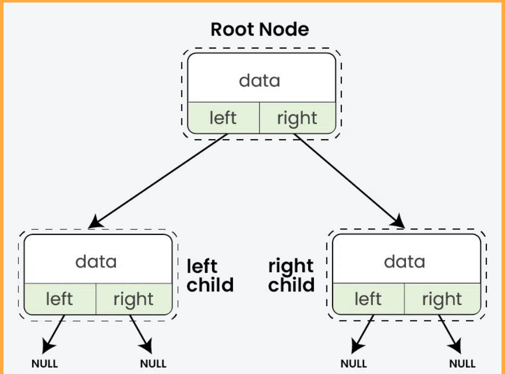



```c
// Define the structure of a node
// in the linked list
struct Node {
    int data;
    struct Node* next;
};
// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    // Insert elements
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    printf("Linked List: ");
    printList(head);
    return 0;
}
```

Application: construct a binary tree

A binary tree is a non-linear hierarchical data structure in which each node has at most two children known as the left child and the right child.

It can be visualized as a hierarchical structure where the topmost node is called the root node and the nodes at the bottom are called leaf nodes or leaves.



Binary Tree Representation

```c
// Define the structure for a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
// Function to insert a node in a binary tree
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        // If the tree is empty, create a new node
        root = createNode(data);
    } else if (data <= root->data) {
        // Insert in the left subtree
        root->left = insert(root->left, data);
    } else {
        // Insert in the right subtree
        root->right = insert(root->right, data);
    }
    return root;
}
// Main function to demonstrate the binary tree operations
int main() {
    struct Node* root = NULL;  // Create an empty tree
    // Insert elements into the binary tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    // Perform different tree traversals
    printf("In-order traversal: ");
    inorderTraversal(root);
    printf("\n");
}
```

Unions

```c
union int_or_double { //declare the union type fields
    int integer;
    double with_dot;
};

int main(){
    union int_or_double u;
    u.integer = 42; //access to u as a variable of type int
    printf("%d\n", u.integer);
    u.with_dot = 3.14; //access to u as a variable of type double
    printf("%lf\n", u.with_dot);
    return 0;
}
```

- At a given time, a variable of that type will contain a value of type `int` **or** of type `double` (but not the two at the same time)
- The choice of a name of a field like `integer` or `with_dot` specifies the way one wants to read/write in the union.

a `union` is a data structure that can hold different data types.

However, in a union, all members share the same memory location, meaning that at any given time, only one member can store a value.

So, the size of the `union` is determined by the size of its largest member. (in the example, it is 8 bytes)

In short, the `union` allows you to define a data type whose value can switch between different existing data types.