Lecture 5: Outline

I. Multi-dimensional arrays
II. Multi-level arrays
III. Structures
IV. Data alignment
V. Linked Lists
Multidimensional arrays: 2D

• Declaration
  
  ```
  int a[3][4];  /*Conceptually 2D matrix with 3 rows 4 columns */
  ```

- Element in row $i$, column $j$ is retrieved as $a[i][j]$
- ‘a’ is a pointer to an integer array of size 4
Representing a 2D array in memory:

```c
int arr[3][4]; /* Conceptually 2D matrix with 3 rows 4 columns */
```

In memory, the elements of `arr` are stored in a contiguous memory block.

[Diagram showing the arrangement of elements]
(arr + i) increments the address of ‘arr’ by how many bytes?

arr

A. i * sizeof(int)

B. i * sizeof(int*)

C. i * sizeof(int) * number of columns
(arr+i) increments the address of ‘arr’ by how many bytes?

A. $i \times \text{sizeof(int)}$

B. $i \times \text{sizeof(int*)}$

C. $i \times \text{sizeof(int)} \times \text{number of columns}$
Express \( \text{arr}[i][j] \) using ‘\text{arr}’ as a pointer

\[
\text{arr}
\]

\[
\begin{array}{cccccc}
\end{array}
\]

A. \( *((\text{arr}+i)+j) \)

B. \( *((*(\text{arr}+i)+j) \)

C. \( *((\text{int}*)(\text{arr}+i)+j) \)

D. \( *((\text{arr}+4*i*\text{sizeof}(\text{int})+j*\text{sizeof}(\text{int})) \)
Express \( arr[i][j] \) using ‘\( arr \)’ as a pointer

\[
arr
\]

\[
\begin{array}{cccccccc}
\end{array}
\]

A. \( *((arr+i) +j) \)

B. \( *((arr+i)+j) \) : This is what is generally used

C. \( *((int*)(arr+i)+j) \)

D. \( *(arr+4*i*sizeof(int)+j*sizeof(int)) \)
Multi-level arrays

• Declaration

  char name_1[]="John";
  char name_2[]="Paul";
  char name_3[]="Rose";

  char * names[]={name_1, name_2, name_3};

What does the following statement print?

  printf("\n Letter: %c\n", names[1][2]);
Multi-level arrays: Representation in memory

char name_1[]="John";
char name_2[]="Paul";
char name_3[]="Rose";
char * names[]={name_1, name_2, name_3};
Multi-level vs multi-dimensional arrays

1. What does names[1][2] give in each case?
2. Which one needs more memory accesses?
3. When would we prefer multi-level arrays?
Pointers in C

• Why use pointers?
  • If we want to pass a huge struct or array, it’s easier to pass a pointer than the whole thing.
  • In general, pointers allow cleaner, more compact code.

• So what are the drawbacks?
  • Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
Derived data types: Structures
C structures: Overview

- A **struct** is a data structure composed of simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

```c
struct point {
    int x;
    int y;
}

void PrintPoint(struct point p) {
    printf( "(%d,%d)" , p.x, p.y);    
}
```
Pointers to structures

• The C arrow operator (\texttt{\textasciitilde\textasciitilde}) dereferences and extracts a structure field with a single operator.

• The following are equivalent:

```c
struct point *p;

printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```
Representation in memory

```c
struct p {
    int y;
    char x;
};
struct p sp;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>y (4 bytes)</td>
</tr>
<tr>
<td>0x104</td>
<td>x (1 byte)</td>
</tr>
<tr>
<td>0x105</td>
<td></td>
</tr>
</tbody>
</table>
Data Alignment

```
struct p {
    char x;
    int y;
};
Struct p sp;
```

Q: How is the above struct laid out in memory?
Alignment Fundamentals

- Processors do not always access memory in byte sized chunks, instead in 2, 4, 8, even 16 or 32 byte chunks.
- Boundaries at which data objects are stored affects the behavior of read/write operations into memory.

Programmer's view of memory

```
0x00
0x01
0x02
0x03
0x04
0x05
```

Processor’s view of memory

```
0x00
0x02
0x04
```
Alignment Fundamentals

• Consider the task of reading 4 bytes from memory starting at 0x00 for processors that read 1, 2, 4 bytes at a time.

• How many memory accesses are needed in each case?
Alignment Fundamentals

• Now consider the task of reading 4 bytes from memory starting at 0x01 for processors that read 1, 2, 4 bytes at a time.

• How many memory accesses are needed in each case?
  • Some processors just would not allow this scenario because it is extra work for the h/w and it affects the performance.

<table>
<thead>
<tr>
<th>Memory</th>
<th>1 byte reads</th>
<th>2 byte reads</th>
<th>4 byte reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x01</td>
<td></td>
<td>0x01</td>
<td>0x02</td>
</tr>
<tr>
<td>0x02</td>
<td></td>
<td>0x02</td>
<td>0x04</td>
</tr>
<tr>
<td>0x03</td>
<td></td>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td></td>
<td>0x04</td>
</tr>
<tr>
<td>0x05</td>
<td></td>
<td>0x05</td>
<td></td>
</tr>
</tbody>
</table>

Additional operations needed to get required bytes.
Alignment requirements

- Data objects should start at memory addresses that are divisible by the size of the data object.
  - short (2 byte) at address divisible by 2 \(0b______0\)
  - int (4 byte) at address divisible by 4 \(0b_____00\)
  - double (8 byte) at address divisible by 8 \(0b_____000\)

```
struct p {
    char x;
    int y;
};
Struct p sp;
```

Q: How is ‘sp’ laid out in memory?

Compiler takes care of zero padding

```cpp
sp
```

```
x 0 0 0 0 y
```

```
0x100 0x104 0x108
```
How big are structs?

• Recall C operator `sizeof()` which gives size in bytes (of type or variable)

Q: How big is `sizeof(struct p)`?

```c
struct p {
  char x;
  int y;
};
```

A. 5 bytes  
B. 8 bytes  
C. 12 bytes
A generic linked list has:
- Multiple data objects (structs) also called nodes
- Each node linked to the next node in the list i.e. it knows the address of the next node
- Nodes located at different memory locations (unlike arrays where they are contiguous)

Advantages compared to arrays
- Multiple data members (of different data types) can be stored in each node
- Nodes can be easily inserted or removed from the list without modifying the whole list
Let’s look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

```c
typedef struct Node node;

struct Node { 
    char *value;
    ____ next;
};

Q: What is the data type of the variable ‘next’?

A. struct Node
B. Node
C. node
D. node *
```
Let’s look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

typedef struct Node node;

struct Node {
    char *value;
    ____ next;
};

Q: What is the data type of the variable ‘next’?

A. struct Node
B. Node
C. node
D. node *
Adding a node to the list

```c
node *list_add(node* head, char *string) {
    //Step 1: Create a new node
    _____________________________;
    new_node
    return new_node;
}
```

Q: How should we declare and initialize `new_node`?

A. `node *new_node=(node*) malloc(sizeof(node));`
B. `node new_node;`
C. `node *new_node=head;`
D. `node *new_node=(node *)malloc(sizeof(head));`
Adding a node to the list

```c
node *list_add(node* head, char *string) {
    //Step 1: Create a new node
    node* new_node = (node*)malloc(sizeof(node));
    return new_node;
}
```

Q: How should we declare and initialize `new_node`?

A. `node *new_node=(node*) malloc(sizeof(node));`
B. `node new_node;`
C. `node *new_node=head;`
D. `node *new_node=(node *)malloc(sizeof(head));`
node *list_add(node* head, char *string)
{
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node));
    //Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}

A. Step 2 is correct.
B. We should use the operator ‘.’ instead of ‘->’
C. Memory is not allocated for ‘value’
Q: What is wrong with step 2?

```c
node *list_add(node* head, char *string) {
    // Step 1: Create a new node
    node *new_node = (node*) malloc(sizeof(node));
    // Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}
```

A. Step 2 is correct.
B. We should use the operator `.` instead of `->`
C. Memory is not allocated for `value`
So far….

```
node *list_add(node* head, char *string) {
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node));
    //Step 2: Fill in its value
    new_node->value = (char*) malloc(strlen(string)+1);
    strcpy(new_node->value, string);
    return new_node;
}
```
What should Step 3 be?

node *list_add(node* head, char *string){
  //Step 1: Create a new node
  node *new_node=(node*) malloc(sizeof(node));
  //Step 2: Fill in its value
  new_node->value = (char*) malloc(strlen(string)+1);
  strcpy(new_node->value, string);
  //Step 3:
  return new_node;
}

A. new_node->next = head;
B. next = head;
C. head = new_node;
D. new_node->next = *head;
node *list_add(node* head, char *string){
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node));
    //Step 2: Fill in its value
    new_node->value = (char*) malloc(strlen(string)+1);
    strcpy(new_node->value, string);
    //Step 3: Link new_node to the head of the list
    new_node->next = head;
    return new_node;  }

A. new_node->next = head;
B. next=head;
C. head=new_node;
D. new_node->next = *head;