Announcements

• No reading quiz today.

• Lab05 final deadline extended.
  • Due Saturday 11:59pm.
Overview

• Accessing *things* via an offset
  – Arrays, Structs, Unions

• How complex structures are stored in memory
  – Multi-dimensional arrays & Structs
Recall: Arrays in Memory

```c
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```
Recall: Two-dimensional Arrays

• Why stop at an array of ints? How about an array of arrays of ints?

    int twodims[3][4];

• “Give me three sets of four integers.”

• How should these be organized in memory?
Recall: Statically Allocated 2D Array

- **Matrix:** 3 rows, 4 columns

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**twodim[1][3]:**

```
base addr + row offset + col offset

twodim + 1*ROWSIZE*4 + 3*4
```

```
0xf260 + 16 + 12 = 0xf27c
```
char *arr;
arr = malloc(sizeof(char)*ROWS*COLS);
for(i=0; i< ROWS; i++) {
    for(j=0; j< COLS; j++) {
        arr[i*COLS+j] = i+j;
    }
}

Heap: all ROW*COLS buckets are contiguous (allocated by a single malloc)
all buckets can be access from single base address (addr)
Recall: Dynamically Allocated 2D Array

```c
cchar *arr;
arr = malloc(sizeof(char)*ROWS*COLS);
for(i=0; i< ROWS; i++) {
    for(j=0; j< COLS; j++) {
        arr[i*COLS+j] = i+j;
    }
}
```

Heap: all ROW*COLS buckets are contiguous (allocated by a single malloc)
all buckets can be access from single base address (addr)

Pro: Memory efficient
Con: access can be annoying
Recall: Dynamically Allocated 2D Array

```c
char **arr;
arr = malloc(sizeof(char*)*3); // array of 3 char*'s
for(i=0; i<3; i++) {
    arr[i] = malloc(sizeof(char)*5);
    for(j=0; j<5; j++) {
        arr[i][j] = i+j;
    }
}
```

Heap: each malloc’ed array of 5 chars is contiguous, but three separately malloc’ed arrays, not necessarily → each has separate base address

Pro: arr[i][j] notation

Con: allocation is annoying
Characters and Strings

• A string is a character array, one after another, with a null terminator (numerical 0) at the end.

• Examples:

  char name[20] = “Martin”;

<table>
<thead>
<tr>
<th>M</th>
<th>a</th>
<th>r</th>
<th>t</th>
<th>i</th>
<th>n</th>
<th>\0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td>[18]</td>
</tr>
</tbody>
</table>
Strings in C

• C String library functions: `#include <string.h>`
  – Common functions (strlen, strcpy, etc.) make strings easier
  – Less friendly than Python strings

• Things to remember about strings:
  – Allocate enough space for null terminator!
  – If you’re modifying a character array (string), don’t forget to set the null terminator!
  – If you see crazy, unpredictable behavior with strings, check these two things!
int main(int argc, char** argv) { ...

The integer is the argument count, the number of arguments from the command line including the name of the program.

The pointer to pointer to char (i.e. array of character pointers, i.e. array of strings) contains all the command line arguments.
int main(int argc, char** argv) {

Example: ./a.out 10 11 monkey 200
argc will be 5
argv[0] will be “./a.out”
argv[1] will be “10” (the two character string)
argv[2] will be “11”
argv[3] will be “monkey”
argv[4] will be “200”
Recall: Composite Data Types

• Combination of one or more existing types into a new type. (e.g., an array of multiple ints, or a struct)

• Example: a queue
  – Might need a value (int) plus a link to the next item (pointer)

```c
struct queue_node{
    int value;
    struct queue_node *next;
}
```
Recall: Structs

• Laid out contiguously by field
  – In order of field declaration (required by C standard).
  – May require some padding, for alignment.

```c
struct student{
    int age;
    float gpa;
    int id;
};

struct student s;
```

![Memory layout diagram]
Recall: Data Alignment:

• Where (which address) can a field be located?

• **char (1 byte):** can be allocated at any address:
  
  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, …

• **short (2 bytes):** must be aligned on 2-byte addresses:
  
  0x1230, 0x1232, 0x1234, 0x1236, 0x1238, …

• **int (4 bytes):** must be aligned on 4-byte addresses:
  
  0x1230, 0x1234, 0x1238, 0x123c, 0x1240, …
Unions

• Declared like a struct, but only contains one field, rather than all of them.

• Struct: field 1 and field 2 and field 3 …

• Union: field 1 or field 2 or field 3 …

• Intuition: you know you only need to store one of N things, don’t waste space.
Unions

```c
struct my_struct {
    char ch[2];
    int i;
    short s;
}

union my_union {
    char ch[2];
    int i;
    short s;
}
```

my_struct in memory

my_union in memory
Unions

my_union u;

u.i = 7;

union my_union {
    char ch[2];
    int i;
    short s;
}

Same memory used for all fields!
Unions

```c
my_union u;

u.i = 7;

u.s = 2;
```

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!
Unions

```c
my_union u;

u.i = 7;

u.s = 2;

u.ch[0] = 'a';

Reading i or s here would be bad!
```

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

```
my_union in memory
```
Unions

my_union u;

u.i = 7;

u.s = 2;

u.ch[0] = 'a';

Reading i or s here would be bad!

u.i = 5;
Unions

• You probably won’t use these often.

• Use when you need mutually exclusive types.

• Can save memory.