Reminder of midterm 1

• Next Thursday, Feb. 14, in class
• Look at Piazza announcement for rules and preparations
A New Example to see effect of E++
(better than the one in previous lecture)

- Purpose of showing increment.c
  - Compiler error if misuse non-lvalue
  - Difference between *p++ and (*p)++

```c
#include <stdio.h>
main() {
    int i=0, j[2], *p, *q;
    p = &i;
    q = j;
    printf("i++ = \t %d\n", i++);
    printf("i is now  \t %d\n", i);
    printf("*p is now  \t %d\n", *p);
    j[0]=2;
    j[1]=3;
    (*q)++ = 0;
}
/*      (i++)++;  */
j[0]=2;
q = j;
printf("j is now \t %p\n", j);
printf("j[0] is now \t %d\n", j[0]);
printf("j[1] is now \t %d\n", j[1]);
printf("q is now \t %p\n", q);
printf("q++ is now \t %d\n", *q);
printf("q is now  \t %p\n", q);
printf("(*q)++ is now \t %d\n", (*q)++);
printf("j[0] is now \t %d\n", j[0]);
printf("j[1] is now \t %d\n", j[1]);
printf("q is now \t %p\n", q);
printf("q++ is now \t %d\n", *q);
printf("q is now  \t %p\n", q);
printf("(*q)++ is now \t %d\n", (*q)++);
printf("j[0] is now \t %d\n", j[0]);
printf("j[1] is now \t %d\n", j[1]);
printf("q is now \t %p\n", q);
printf("q++ is now \t %d\n", *q);
printf("q is now  \t %p\n", q);
printf("(*q)++ is now \t %d\n", (*q)++);
printf("j[0] is now \t %d\n", j[0]);
printf("j[1] is now \t %d\n", j[1]);
```

```c
q = j;
printf("q is now \t %p\n", q);
printf("(*q)++ is now \t %d\n", (*q)++);
printf("q is now \t %p\n", q);
printf("(*q)++ is now \t %d\n", (*q)++);
printf("j[0] is now \t %d\n", j[0]);
printf("j[1] is now \t %d\n", j[1]);
```
• We’ll leave other expressions to next Tuesday’s review lecture
• Next we discuss more complex way to organize and address data
  – Arrays of arrays
• In future lectures, we discuss
  – Nested structures
  – Recurrent (i.e. self-referential) structures
  – Arrays of structures (AOS)
  – Structure of arrays (SOA)
Arrays of arrays

- `int ndigit[4][4]`
  - Declares a 4-element array `ndigit[4]`, in which each element is another 4-element array of integers
- It is the programmer’s responsibility to not go beyond the array bound
  - The compiler by default does not issue warnings
  - Neither does it insert run-time check of array bound, by default
Arrays of arrays

#include <stdio.h> /* main1.c, local array, going out of bound */

main() {
    int i, j, ndigit[4][4];
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            ndigit[i][j] = i+j;
    for (i = 0; i < 10; ++i)
        for (j = 0; j < 4; ++j)
            printf("ndigit[%d][%d] = \t %d\n", i, j, ndigit[i][j]);
}
Declaring global multiple dimension arrays

```c
#include <stdio.h>    /* f.c */
extern int ndigit[][
void f() {
    int i;

    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            printf("ndigit[%d][%d] = 	
%dn", i, j, ndigit[i][j]);

}

/* Purpose: show how different files pass data through global arrays, main2.c */
int ndigit[4][4];
extern void f();
main() {
    int i, j;
    for (i = 0; i < 4; ++i) {
        for (j = 0; j < 4; ++j) {
            ndigit[i][j] = i+j;
        }
        f();
    }
}
```
Let us run the following program and examine the addresses of the array elements

```c
#include <stdio.h> /* main3.c */

main() {
    int i, j, ndigit[4][4];
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            ndigit[i][j] = i+j;
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            printf("ndigit[%d][%d] is at \t %p\n", i, j, &ndigit[i][j]);
}
```
Memory Layout of a 2-dim Integer Array

• In C, the array int i[][] is allocated row-wise
  – We call the first index the row index and the second the column index
  – This allocation scheme is known as row-major allocation
    • All elements of an entire row are allocated consecutively
    • For good performance, it is important that the program accesses the array elements in a consecutive way
    • We will discuss more on data locality and performance issues in later lectures
Array of pointers
• Let’s compile and run the following program

```c
#include <stdio.h> /* main4.c */

main() {

    int i, j, *ndigit[4][4];
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            *ndigit[i][j] = i+j;
}
```

What happened?
ndigit[i][j] does not have any values yet!

/* examine the contents of arrays of pointers */
#include <stdio.h> /* main5.c */

main() {
    int i, j, *ndigit[4][4];
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            printf("ndigit[%d][%d] = \t %p\n", i, j, ndigit[i][j]);
}

Bad Memory Addresses

ndigit[0][0] = 0x7f5a52a53000
ndigit[0][1] = 0x400379
ndigit[0][2] = 0xa
ndigit[0][3] = 0x8000
ndigit[1][0] = 0x1
ndigit[1][1] = 0x7ffeef9dfc58
ndigit[1][2] = 0x1
ndigit[1][3] = 0x4005b0
ndigit[2][0] = (nil)
ndigit[2][1] = 0x40041b
ndigit[2][2] = 0x7ffeef9dfc58
ndigit[2][3] = 0x4005f5
ndigit[3][0] = 0x7f5a524f56e0
ndigit[3][1] = 0x4005b0
ndigit[3][2] = (nil)
ndigit[3][3] = 0x400450
A different example

```c
#include <stdio.h>   /* main7.c */
main() {
    int i, j, ndigit[4][4], *address[4][4];
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j) {
            ndigit[i][j] = i+j;
            address[i][j]=&ndigit[i][j];
        }

    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            printf("ndigit[%d][%d] = \t %d\n", i, j, *address[i][j]);
}
```
#include <stdio.h>        /* main8.c */
main() {
    int i, j, ndigit[4][4], *address[4];
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j) {
            ndigit[i][j] = i+j;
            address[i]=ndigit[i];
        }
    for (i = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            printf("ndigit[%d][%d] = \t %d\n", i, j,*address[i]++);
}
2-dim Array Initialization

• Bracketed initialization: row by row

```c
#include <stdio.h>     /* arrayinit.c */
main() {
    float y[3][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 } 
    };

    int i, j;
    for (i = 0; i < 3; ++i)
        for (j = 0; j < 3; ++j)
            printf("y[%d][%d] = \t %f
", i, j, y[i][j]);
}
```
#include <stdio.h>

main()
{
float y[3][3] = {1, 3, 5, 2, 4, 6, 3, 5, 7};

    int i, j;
    for (i = 0; i < 3; ++i)
        for (j = 0; j < 3; ++j)
            printf("y[%d][%d] = \t %f\n", i, j, y[i][j]);
}
Avoid Modifying Constant Array by Mistake

```c
#include <stdio.h>    /* try to compile constarray.c */

main() {
    const float y[3][3] = {1, 3, 5 , 2, 4, 6 , 3, 5, 7};

    int i, j;
    for (i = 0; i < 3; ++i)
        for (j = 0; j < 3; ++j)
            y[i][j]=0.0;
}   /* Compiler will report the error */
```
An array of strings

- `char my[2][3] = { "me", "my" };`
  - Remember each string literal has the terminating ‘\0’
  - For fixed sized strings, arrays of arrays of characters work well
- For variable sized strings, char pointers must be used
  
  ```c
  char* you[2] = { "you", "yours" };
  ```

- `my[2][3]` has 6 bytes allocated in memory
- `you[2]` has memory allocated for two pointers
  - 16 bytes if each pointer takes 8 bytes

```c
/* size of pointer */
#include <stdio.h>

main() {
    int i, *j=&i;
    printf("size of pointer j is \t %lu\n", sizeof(j));
}
```
For array my[2][3]

- The pointer value of my[i] can be computed by the compiler
  - Because the rows are stored consecutively in memory
- No need to store a pointer in the memory
Compile and run this program

```c
#include <stdio.h>    /* arraystring.c */
main() {
    int i, j;
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 3; ++j) {
            printf("my[%d][%d] code \t %d\n", i, j, my[i][j]);
            printf("my[%d][%d] = \t %c\n", i, j, my[i][j]);
        }
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 3; ++j) {
            printf("you[%d][%d] code \t %d\n", i, j, *you[i]);
            printf("you[%d][%d] string \t %s\n", i, you[i++], you[i]);
        }
}
```
• More commonly, a 2-dimensional array may be formed dynamically by calling memory management routine such as malloc()

• Consecutive rows may not be neighbors in the memory

• We discuss malloc() later
The void type

- The type `void` is not actually a type. It is rather saying “to be determined” or “undefined”
  - `sizeof(void)` is undefined
- A pointer of type `void*` is a generic pointer, that can not be used without first casting it to an actual type
  - The `void*` type is used by functions that return a pointer to a memory area
  - `void* malloc(size_t size);`

- A typical use of malloc is
  ```c
  int* p = (int*) malloc(sizeof(int));
  ```

- Notice the type cast in the above statement
The Heap memory

- Heap:
  - dynamic memory for variables that are created with calls to standard C utility functions `malloc`, `calloc`, `realloc` and are disposed of with calls to `free`
Address space versus physical memory

• A 64-bit machine provides a range of addresses from 0x0000000000000000 to 0xffffffffffffffff
  – 2^32 is covers 4 GB memory, 2^52 covers 4 peta bytes ... ...

• At any given time, not all address ranges are mapped to physical memory
  – Otherwise, with so many processes (i.e., instances of programs) running or loaded to run, we run out of physical memory very fast
What is physical memory

• In a simplified view, the physical memory is
  – The semiconductor DRAM (dynamic random access memory), with the backup of
  – The (magnetic or semiconductor) secondary storage (often also known as the disk device)
  – The secondary storage is usually several orders of magnitude larger than the DRAM, but several orders of magnitude slower
    • The secondary storage is divided between the file system and the swap device (to support the program address space)

• The processor executes a program by accessing instructions and variables in the DRAM
  – The OS swaps instructions/data between DRAM/secondary storage as needed
• OS uses sophisticated algorithms to allocate DRAM and the swap device to each running program
• The allocated swap size is increased as requested
• The UNIX OS allows a C program to directly request a block of “memory” (i.e. a chunk of swap storage) through calls to routine such as `sbrk(size)`
  – The OS automatically round up the requested size to the size that is suitable for allocation
• How to fit a dynamic data structure, e.g. a graph, a linked list, to the allocated chunk of memory
• How to keep track of and reuse the freed parts of the memory
• These will soon become a nightmare to the programmer
• Hence, we are provided with heap management utility routines in C standard library via the header file `<stdlib.h>`
Dynamic memory management interface

- `#include <stdlib.h>`

- `void* calloc(size_t n, size_t s)`
- `void* malloc(size_t s)`
- `void  free(void* p)`
- `void* realloc(void* p, size_t s)`

- Allocate and free dynamic memory
• Using such C routines, the programmer does not need to directly interact with the OS
• Instead, these routines implement a “memory manager” to
  – keep track of the “free” areas of the memory (i.e. the data segment obtained from the swap device)
  – find a suitable piece of the “free” areas when requested
  – put back a “freed” piece of memory back to the free areas
  – Various allocation algorithms have been discussed in literature
  – We can see the exact algorithm used in our utility routines
Warm up your iclickers

• Quizzes 3
Q3.#1

• What number is printed by the following program?

```c
#include <stdio.h>
main() {
    int j[2], *q;
    q=j;
    j[0]=2;
    j[1]=3;
    *q++ = 0;
    printf("%d\n", j[0]); }
```

• (a) 2
• (b) 3
• (c) 0
• (d) none of the above
• Answer is 0
Q3.#2

• What number is printed by the following program?

```c
#include <stdio.h>
main() {
    int j[2], *q;
    q=j;
    j[0]=2;
    j[1]=3;
    *++q = 0;
    printf("%d\n", j[0]); }
```

• (a) 2
• (b) 3
• (c) 0
• (d) none of the above
• Answer is 2
What number is printed by the following program?

```c
#include <stdio.h>
main() {
    int j[2], *q;
    q = j;
    j[0] = 2;
    j[1] = 3;
    printf("%d\n", *q++);
}
```

- (a) 2
- (b) 3
- (c) 0
- (d) none of the above
• Answer is 2