CS 33

Introduction to C

Part 3
Arrays and Parameters

```c
void func(int arg[]) {
    /* arg points to the caller’s array */
    int local[7];       /* seven ints */
    arg++;              /* legal */
    arg = local;        /* legal */
    local++;            /* illegal */
    local = arg;        /* illegal */
}
```
Operator precedence is hard to remember!
Dereferencing C Pointers

```c
int main() {
    int *p; int a = 4;
    p = &a;
    (*p)++;
    printf("%d %u\n", *p, p);
}
```

```
$ ./a.out
5 3221224356
```
Dereferencing C Pointers

```c
int main() {
    int *p; int a = 4;
    p = &a;
    ++*p;
    printf("%d %u\n", *p, p);
}
```

```
$ ./a.out
5 3221224356
```
Quiz 1

```c
int proc(int arg[]) {  
    arg++;  
    return arg[1];  
}

int main() {  
    int A[3]={0, 1, 2};  
    printf("%d\n",  
            proc(A));  
}
```

What’s printed?

a) 0  
b) 1  
c) 2  
d) indeterminate
Strings

- Strings are arrays of characters terminated by "\0" ("null")
  - the \0 is included at the end of string constants

  » "Hello"

  Hello \0

Note that \0 is represented as a byte containing all zeroes.
Since we didn’t explicitly output a newline character, the prompt for the next command goes on the same line as the string that was printed.
We’ve added the newline character to the format specifier of printf - the prompt now appears on the next line.
We can also print a single character at a time. Note the test for the null character to determine whether we've reached the end of the string.

```c
void printString(char s[]) {
    int i;
    for(i=0; s[i]!='$\0$'; i++)
        printf("%c", s[i]);
}

int main() {
    printString("Hello");
    printf("\n");
    return 0;
}
```

Tells C that this function does not return a value
Note that even though we might think of “int [6]” as being a datatype, to declare “n” to be of that type, we must write “int n[6]” — the size of the array goes just after the identifier.
3-D Arrays

• How do we declare an array of eight $T[7][6]$?
  $T \ p[8][7][6]$
  - $p$ is an array of (eight) $T[7][6]$
  - $p[i]$ is of type $T[7][6]$
  - $p[i][j]$ is of type $T[6]$
  - $p[i][j][k]$ is of type $T$
Here we initialize a 2D array, then call a function (described in the next slide) to print it.
We print the array by rows.

```c
void printMatrix(int nr, int nc,
                int m[nr][nc]) {
    int row, col;
    for(row=0; row<nr; row++) {
        for(col=0; col<nc; col++)
            printf("%6d", m[row][col]);
        printf("\n");
    }
}
```
C arrays are stored in *row-major order*, as shown in the slide. The idea is that the left index references the row, the right index references the column. Thus C arrays are stored row-by-row. Thus to index into a 2D array, we need to know how large each row is (i.e., how many columns there are). But it’s not necessary, for indexing purposes, to know how many rows there are.
In general we don’t need to specify the size of the leftmost dimension of an array argument. In the current 2D example, what’s important is that the compiler know the size of each row so that it can generate code to compute where a particular element is.
Note that m is an array of arrays (in particular, an array of 1-D arrays).
While it’s convenient to think of something as being a 2D array, its elements are stored linearly in memory. Thus, as shown in the slide where we are calling `AccessAs1D` to get the value of `A2D[1][2]`, given a pointer to a 2D array, we can access its elements as if it were a 1D array.

```c
int A2D[2][4];   int A1D[8];

int AccessAs1D(int A[], int Row, int Col, int RowSize) {
    return A[Row*RowSize + Col];
}

int main(void) {
    int A2D[2][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}};
    int *A1D = &A2D[0][0];
    int x = AccessAs1D(A1D, 1, 2, 4);
    printf("%d\n", x);
    return 0;
}
```

Quiz 2

1) Consider the array
   ```c
   int A[3][3];
   ```
   - which element is adjacent to A[0][0] in memory?
     a) A[0][1]
     b) A[1][0]
     c) none of the above

2) int *B = &A[0][0];
   ```c
   B[0] = 8;
   ```
   - which element of A was modified?
     a) A[0][3]
     b) A[2][2]
     c) A[3][0]
     d) none of the above
Global Variables

```c
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
    int row, col;
    for(row=0; row<NUM_ROWS; row++)
        for(col=0; col<NUM_COLS; col++)
            m[row][col] = row*NUM_COLS+col;
    return 0;
}
```

The scope is global; `m` can be used by all functions.
Note that the reference to “m” gives the address of the array in memory.

The point of the slide is that global variables are in a different area of memory than are local variables.
If you don’t explicitly initialize a global variable, its initial value is guaranteed to be zero.
Here we have two declarations for $a$ - one as a global variable and one as a local variable. References to $a$ in `main` are to the local variable, but elsewhere references are to the global variable.
Here \( a \) is declared as a parameter to \( \text{proc} \), thus references to \( a \) in \( \text{proc} \) are to the parameter and not to the global variable.
Syntax error: one can't have a local variable in a scope in which a parameter is declared with the same name.
**Scope (more ...)**

```c
int a; // global variable

int proc() {

    // the brackets define a new scope
    int a;
    a = 6;

    printf("a = %d\n", a); // what's printed?
    return 0;
}

$ ./a.out
0
```
Quiz 3

int a;

int proc(int b) {
    int b=4;
    a = b;
    return a+2;
}

int main() {
    int a = proc(6);
    printf("a = %d\n", a);
    return 0;
}

• What’s printed?
  a) 0
  b) 4
  c) 6
  d) 8
  e) nothing; there’s a syntax error