

CS 33

Introduction to C Part 3

Meet Your TAs!!

- **Come to CIT 3rd-floor atrium on Thursday at 5**
- **Eat tasty cookies!**
- **Talk to the course staff!**
- **Talk to fellow students in the course!**

Arrays and Parameters

```
int main() {  
    int array1[4] = {0, 1, 2, 3};  
    printf("%d, %d\n", func(array1), array1[1]);  
    return 0;  
}
```

```
void func(int arg[]) {  
    int array2[6] = {4, 5, 6, 7, 8, 9};  
    arg[1] = 0;  
    arg = array2;  
    return arg[3];  
}
```

```
$ ./a.out  
7 0
```

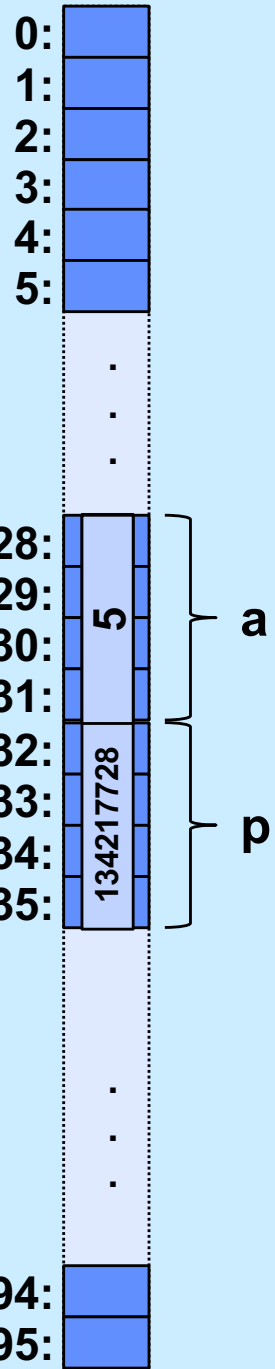
Arrays and Parameters

```
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 */  
}
```

Dereferencing C Pointers

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

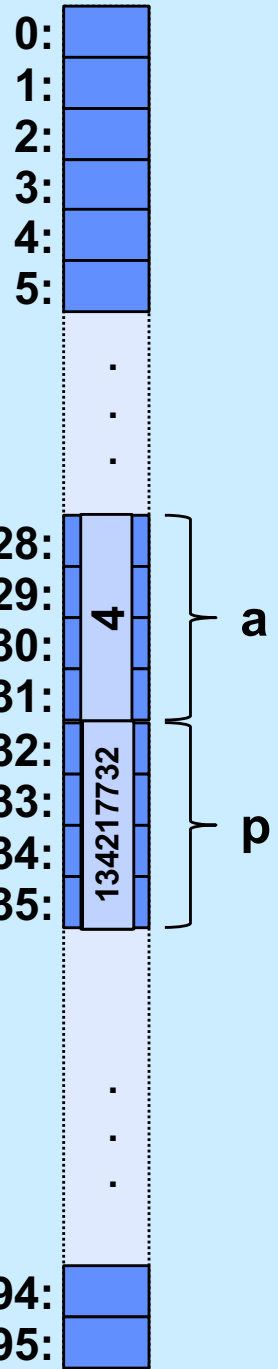
```
$ ./a.out  
5 134217728
```



Dereferencing C Pointers

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

```
$ ./a.out  
134217732 134217732
```



Dereferencing C Pointers

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

```
$ ./a.out  
5 134217728
```

Quiz 1

```
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"

H	e	l	l	o	\0
---	---	---	---	---	----

Strings

```
int main() {  
    printf("%s", "Hello");  
    return 0;  
}
```

```
$ ./a.out  
Hello$
```

Strings

```
int main() {  
    printf("%s\n", "Hello");  
    return 0;  
}
```

```
$ ./a.out  
Hello  
$
```

Strings

```
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

2-D Arrays

- Suppose T is a datatype (such as `int`)
- $T\ n[6]$
 - declares n to be an array of (six) T
 - the type of n is $T[6]$
- Thus $T[6]$ is effectively a datatype
- Thus we can have an array of $T[6]$
- $T\ m[7][6]$
 - m is an array of (seven) $T[6]$
 - $m[i]$ is of type $T[6]$
 - $m[i][j]$ is of type T

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

2-D Arrays

```
#define NUM_ROWS 3
#define NUM_COLS 4
...
int main() {
    int row, col;
    int m[NUM_ROWS][NUM_COLS];
    for(row=0; row<NUM_ROWS; row++)
        for(col=0; col<NUM_COLS; col++)
            m[row][col] = row*NUM_COLS+col;
    printMatrix(NUM_ROWS, NUM_COLS, m);
    return 0;
}
```

```
$ ./a.out
```

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

2-D Arrays

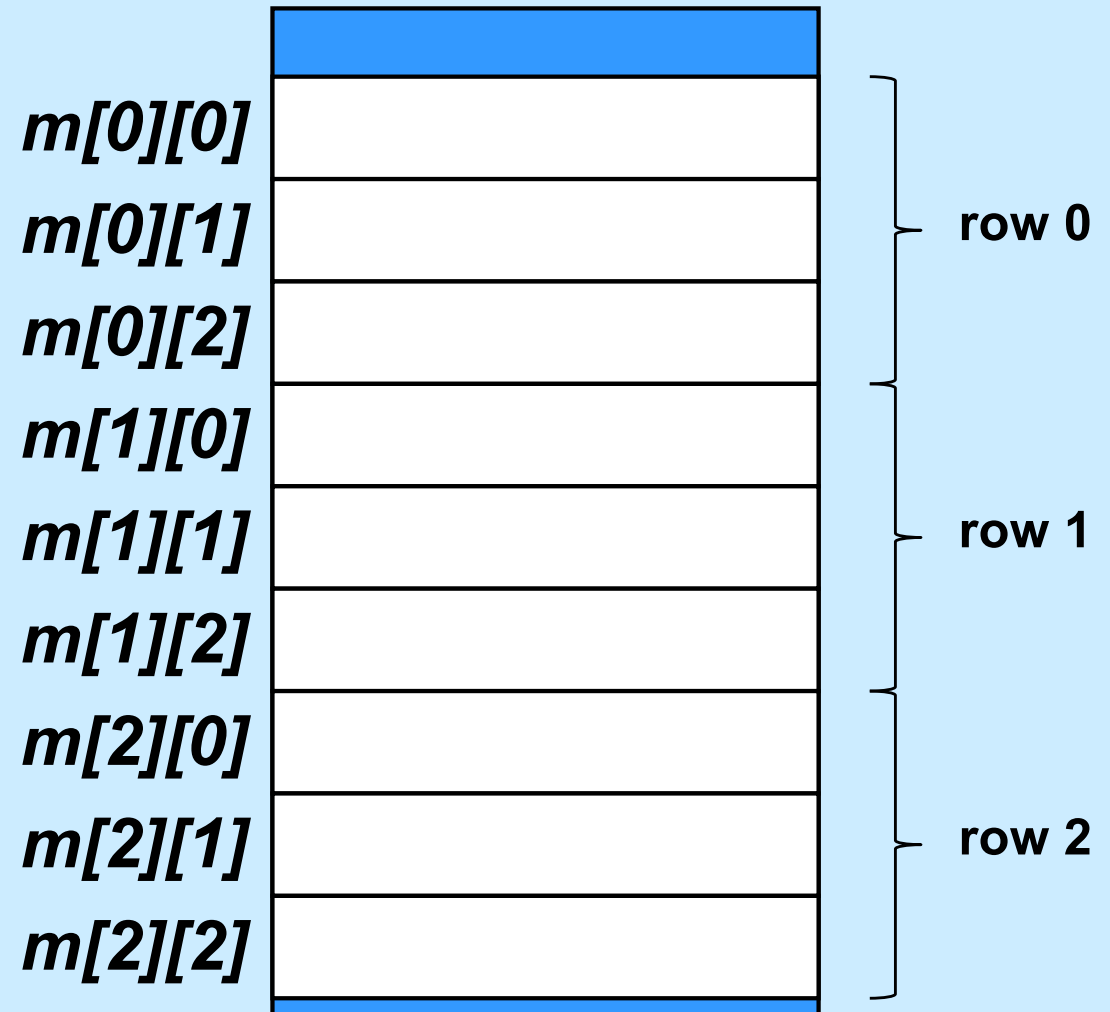
It must be told the dimensions

```
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");
    }
}
```


Memory Layout

```
#define NUM_ROWS 3  
#define NUM_COLS 3
```

Row-Major Order



2-D Arrays

Alternatively ...

```
void printMatrix(int nr, int nc,  
                int m[][nc]) {  
    int row, col;  
    for(row=0; row<nr; row++) {  
        for(col=0; col<nc; col++)  
            printf("%6d", m[row][col]);  
        printf("\n");  
    }  
}
```

2-D Arrays



Or ...

```
void printMatrix(int nr, int nc,  
                int m[][nc]) {  
    int i;  
    for(i=0; i<nr; i++)  
        printRow(nc, m[i]);  
}
```

```
void printRow(int nc, int a[]) {  
    int i;  
    for(i=0; i<nc; i++)  
        printf("%6d", a[i]);  
    printf("\n");  
}
```

2D as 1D

0	1	2	3
4	5	6	7

=

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

```
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;  
}
```

```
$ ./a.out  
6  
$
```

Quiz 2

Consider the array

`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

Quiz 3

Consider the array

```
int A[3][3];
```

```
int *B = &A[0][0];
```

```
B[8] = 8;
```

– which element of A was modified?

a) A[0][3]

b) A[2][2]

c) A[3][0]

d) none of the above

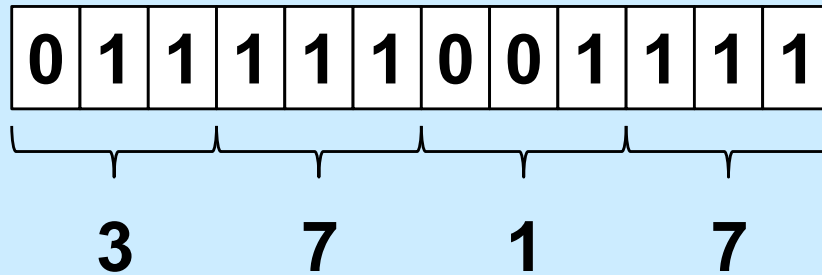
Number Representation

- **Hindu-Arabic numerals**
 - developed by Hindus starting in 5th century
 - » positional notation
 - » symbol for 0
 - adopted and modified somewhat later by Arabs
 - » known by them as “Rakam Al-Hind” (Hindu numeral system)
 - 1999 rather than MCMXCIX
 - » (try doing long division with Roman numerals!)

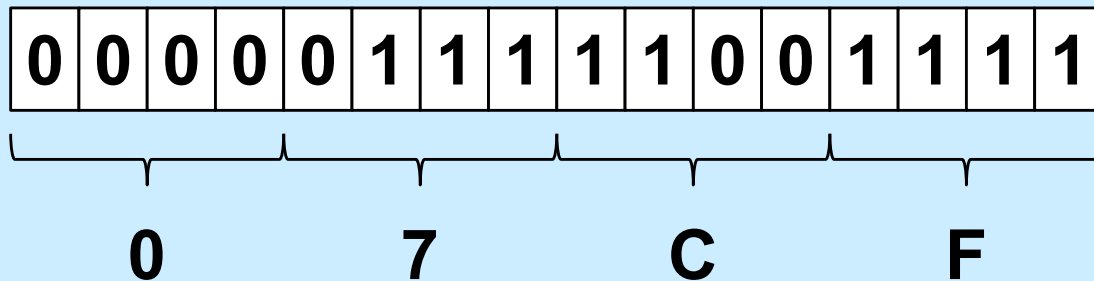
Which Base?

- **1999**
 - **base 10**
 - » $9 \cdot 10^0 + 9 \cdot 10^1 + 9 \cdot 10^2 + 1 \cdot 10^3$
 - **base 2**
 - » 11111001111
 - $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 + 1 \cdot 2^9 + 1 \cdot 2^{10}$
 - **base 8**
 - » 3717
 - $7 \cdot 8^0 + 1 \cdot 8^1 + 7 \cdot 8^2 + 3 \cdot 8^3$
 - » why are we interested?
 - **base 16**
 - » 7CF
 - $15 \cdot 16^0 + 12 \cdot 16^1 + 7 \cdot 16^2$
 - » why are we interested?

Words ...



12-bit computer word



16-bit computer word

Algorithm ...

```
void baseX(unsigned int num, unsigned int base) {
    char digits[] = {'0', '1', '2', '3', '4', '5', '6', ... };
    char buf[8*sizeof(unsigned int)+1];
    int i;

    for (i = sizeof(buf) - 2; i >= 0; i--) {
        buf[i] = digits[num%base];
        num /= base;
        if (num == 0)
            break;
    }

    buf[sizeof(buf) - 1] = '\0';
    printf("%s\n", &buf[i]);
}
```

Or ...

```
$ bc
obase=16
1999
7CF
$
```

Quiz 4

- What's the decimal (base 10) equivalent of 23_{16} ?
 - a) 19
 - b) 33
 - c) 35
 - d) 37

Encoding Byte Values

- **Byte = 8 bits**
 - binary 00000000_2 to 11111111_2
 - decimal: 0_{10} to 255_{10}
 - hexadecimal 00_{16} to FF_{16}
 - » base 16 number representation
 - » use characters '0' to '9' and 'A' to 'F'
 - » write $FA1D37B_{16}$ in C as
 - `0xFA1D37B`
 - `0xfa1d37b`

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Unsigned 32-Bit Integers



$$\text{value} = \sum_{i=0}^{31} b_i \cdot 2^i$$

(we ignore negative integers for now)

Storing and Viewing Ints

```
int main() {  
    unsigned int n = 57;  
    printf("binary: %b, decimal: %u, "  
          "hex: %x\n", n, n, n);  
    return 0;  
}
```

```
$ ./a.out  
binary: 111001, decimal: 57, hex: 39  
$
```

Boolean Algebra

- **Developed by George Boole in 19th Century**
 - algebraic representation of logic
 - » encode “true” as 1 and “false” as 0

And

- $A \& B = 1$ when both $A=1$ and $B=1$

$\&$	0	1
0	0	0
1	0	1

Or

- $A | B = 1$ when either $A=1$ or $B=1$

$ $	0	1
0	0	1
1	1	1

Not

- $\sim A = 1$ when $A=0$

\sim	
0	1
1	0

Exclusive-Or (Xor)

- $A \wedge B = 1$ when either $A=1$ or $B=1$, but not both

\wedge	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on bit vectors
 - operations applied bitwise

01101001	01101001	01101001	
<u>& 01010101</u>	<u> 01010101</u>	<u>^ 01010101</u>	<u>~ 01010101</u>
01000001	01111101	00111100	10101010

- All of the properties of boolean algebra apply

Example: Representing & Manipulating Sets

- Representation

- width- w bit vector represents subsets of $\{0, \dots, w-1\}$
- $a_j = 1$ iff $j \in A$

01101001	{ 0, 3, 5, 6 }
76543210	

01010101	{ 0, 2, 4, 6 }
76543210	

- Operations

&	intersection	01000001	{ 0, 6 }
	union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	symmetric difference	00111100	{ 2, 3, 4, 5 }
~	complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- Operations $\&$, $|$, \sim , \wedge available in C
 - apply to any “integral” data type
 - » long, int, short, char
 - view arguments as bit vectors
 - arguments applied bit-wise
- Examples (char datatype)
 - $\sim 0x41 \rightarrow 0xBE$
 $\sim 01000001_2 \rightarrow 10111110_2$
 - $\sim 0x00 \rightarrow 0xFF$
 $\sim 00000000_2 \rightarrow 11111111_2$
 - $0x69 \& 0x55 \rightarrow 0x41$
 $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
 - $0x69 | 0x55 \rightarrow 0x7D$
 $01101001_2 | 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
 - **&&, ||, !**
 - » view 0 as “false”
 - » anything nonzero as “true”
 - » always return 0 or 1
 - » early termination/short-circuited execution
- **Examples (char datatype)**
 - !0x41 → 0x00**
 - !0x00 → 0x01**
 - !!0x41 → 0x01**

 - 0x69 && 0x55 → 0x01**
 - 0x69 || 0x55 → 0x01**
 - p && *p (avoids null pointer access)**

Contrast: Logic Operations in C

- Contrast to Logical Operators

- `&&`, `||`, `!`

- » view “false”

**Watch out for `&&` vs. `&` (and `||` vs. `|`)...
One of the more common oopsies in
C programming**

- `!0x41 → 0x00`

- `!0x00 → 0x01`

- `!!0x41 → 0x01`

- `0x69 && 0x55 → 0x01`

- `0x69 || 0x55 → 0x01`

- `p && *p` (avoids null pointer access)

Quiz 5

- Which of the following would determine whether the next-to-the-rightmost bit of Y (declared as a char) is 1? (I.e., the expression evaluates to true if and only if that bit of Y is 1.)
 - a) `Y & 0x02`
 - b) `!((~Y) & 0x02)`
 - c) both of the above
 - d) none of the above

Shift Operations

- **Left Shift:** $x \ll y$
 - shift bit-vector x left y positions
 - throw away extra bits on left
 - » fill with 0's on right
- **Right Shift:** $x \gg y$
 - shift bit-vector x right y positions
 - » throw away extra bits on right
 - logical shift
 - » fill with 0's on left
 - arithmetic shift
 - » replicate most significant bit on left
- **Undefined Behavior**
 - shift amount < 0 or \geq word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000