CS 33

Introduction to C
Part 3

Some of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

Parameter passing

Passing arrays to a function

```
int average(int a[], int size) {
   int i; int sum;
   for(i=0,sum=0; i<size; i++)
     sum += a[i];
   return sum/size;
}
int main() {
   int a[100];
   …
   printf("%d\n",average(a,100));
}
```


Here is another example of passing an array to a function. We need to pass the size of the array as well, assuming the function needs to know the array's size.

Swapping

Write a function to swap two entries of an array

```
void swap(int a[], int i, int j) {
   int tmp;
   tmp = a[j];
   a[j] = a[i];
   a[i] = tmp;
}
```

Selection Sort

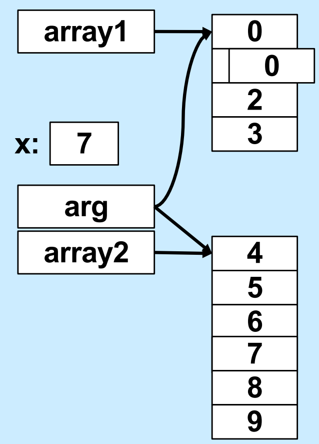
```
void selectsort(int array[], int length){
  int i, j, min;
  for (i = 0; i < length; ++i){
    /* find the index of the smallest item from i onward */
    min = i;
    for (j = i; j < length; ++j) {
      if (array[j] < array[min])
        min = j;
    }
    /* swap the smallest item with the i-th item */
    swap(array, i, min);
  }
  /* at the end of each iteration, the first i slots have the i
     smallest items */
}
```


Note that C uses the same syntax as Java does for conditional (if) statements. In addition to relational operators such as "==", "!=", "<", ">", "<=", and ">=", there are the conditional operators "&&" and "||" ("logical and" and "logical or", respectively).

Arrays and Arguments

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

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




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


In this example, we've declared **array1** and **array2** in **main** and **func**. Both declarations allocate storage for arrays of **int**s. Both **array1** and **array2** refer (by pointing to the first elements) to the storage allocated for the arrays. What memory locations they refer to can't be changed (though the contents of these locations can be changed).

In the definition of **func**, **arg** is an argument that acts as a variable that's initialized with whatever is passed to **func**. In the slide, **func** is called with **array1** as the argument. Thus, **arg** is initialized with **array1**, which means it's initialized with a pointer to the first element of the array referred to by **array1**. But this initial value of **arg** is not permanent -- we're free to change it, as we do when we assign **array2** to **arg**.

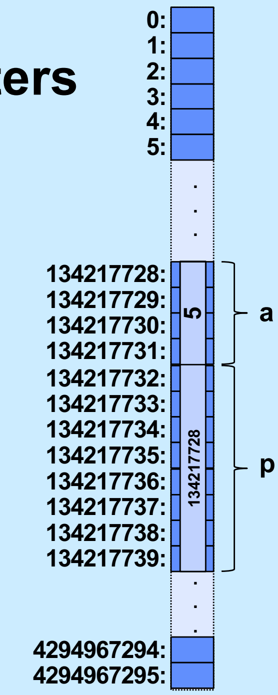
Arrays and Arguments

```
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 %p\n", *p, p);
}
```

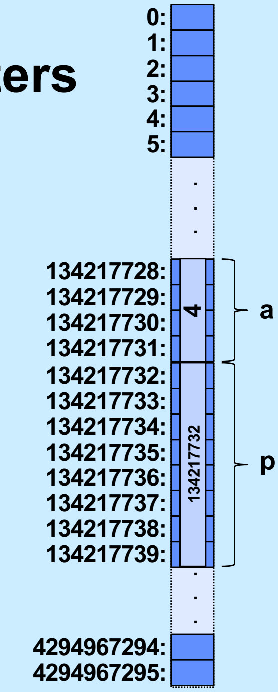
```
$ ./a.out
5 134217728
```



Dereferencing C Pointers

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

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



Operator precedence is hard to remember! ("++" takes precedence over "*".)

Note that even though ***p** is an int, but it's printed as an 8-byte pointer, what's printed is its value as an int. Exactly why this is so (and why it could be a problem) is something we'll discuss in a week or two.

Dereferencing C Pointers

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

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


Here it's clear that the * operator is applied before the ++ operator.

Quiz 1

```
int func(int arg[]) {
   arg++;
   return arg[0];
}

int main() {
   int A[3]={10, 11, 12};
   printf("%d\n",
     func(A));
}
```

What's printed?

a) 9
b) 10
c) 11
d) 12

Quiz 2

```
int func(int a[]) {
   int b[5] = {10, 11, 12, 13, 14};
   a = b;
   return a[1];
}

int main() {
   int array[50];
   array[1] = 0;
   printf("result = %d\n",
      func(array));
   return 0;
}
```

This program prints:

a) **0**
b) **10**
c) **11**
d) **nothing: it doesn't compile because of a syntax error**

Note how we initialize the contents of array **b** in **func**.

Quiz 3

```
int func(int a[]) {
   int b[5] = {10, 11, 12, 13, 14};
   a = b;
   return a[1];
}

int main() {
   int array[5] = {9, 8, 7, 6, 5};
   func(array);
   printf("%d\n", array[1]);
   return 0;
}
```

This program prints:

- **a) 7**
- **b) 8**
- **c) 10**
- **d) 11**

The Preprocessor

#include

- **calls the preprocessor to include a file**

What do you include?

- **your own *header* file:**
```
#include "fact.h"
```
  - **look in the current directory**
- **standard *header* file:**
```
#include <assert.h>
#include <stdio.h>
```
  - **look in a standard place**

Contains declaration of *printf* (and other things)

The preprocessor modifies the source code before the code is compiled. Thus, its output is what is passed to gcc's compiler.

Note that one must include **stdio.h** if using **printf** (as well as some other functions) in a program.

On most Unix systems (including Linux, but not OS X), the standard place for header files is the directory /usr/include.

Function Declarations

fact.h

```c
float fact(int i);
```

main.c

```c
#include "fact.h"
int main() {
  printf("%f\n", fact(5));
  return 0;
}
```


It's convenient to package the declaration of functions (and other useful stuff) in header files, such as **fact.h**, so the programmer need simply to include them, rather than reproduce their contents.

The source code for the **fact** function would be in some other file, perhaps as part of a library (a concept we discuss later).

#define

```
#define SIZE 100
int main() {
   int i;
   int a[SIZE];
}
```

`#define`

- **defines a substitution**
- **applied to the program by the preprocessor**

#define

```
#define forever for(;;)
int main() {
   int i;
   forever {
     printf("hello world\n");
   }
}
```


The #define directive can be used for pretty much anything, such as segments of code as shown in the slide. (It's not its concern as to whether the code segments are useful!)

assert

```
#include <assert.h>
float fact(int i) {
  int k, res;
  assert(i >= 0);
  for(res=1,k=1; k<=i; k++)
    res = res * k;
  return res;
}
int main() {
  printf("%f\n", fact(-1));
```

assert

- **verify that the assertion holds**
- **abort if not**

```
$ ./fact
main.c:4: failed assertion 'i >= 0'
Abort
```

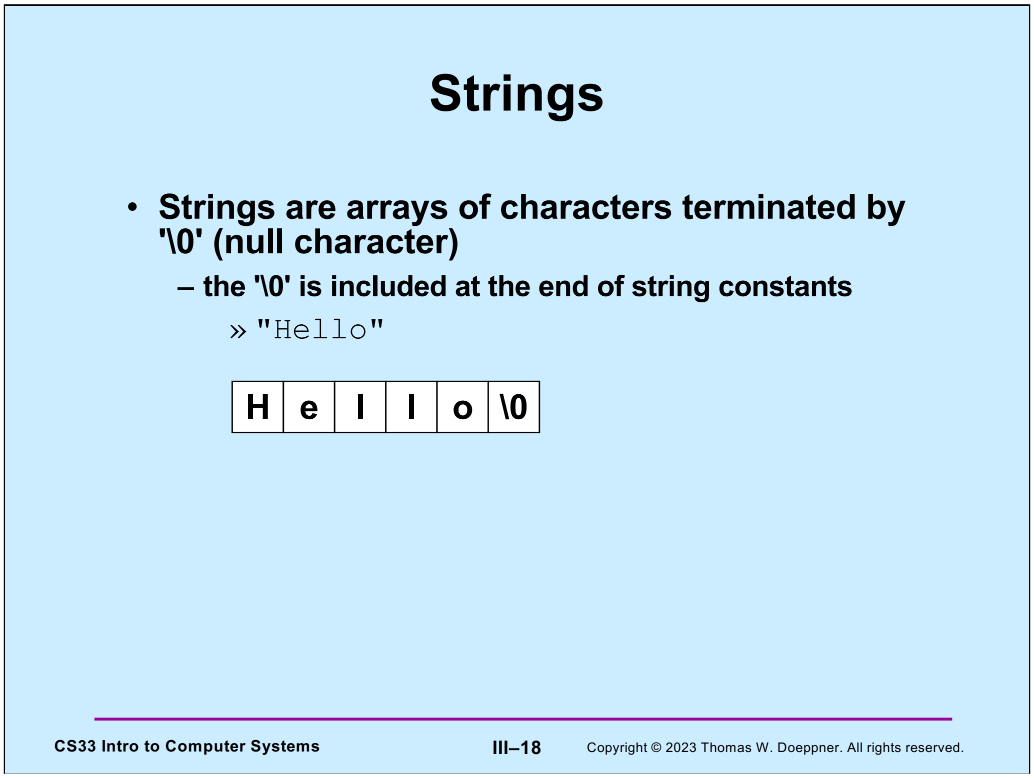
The assert statement is actually implemented as a macro (using #define). One can “turn off” asserts by defining (using #define) NDEBUG. For example,

```
#include <assert.h>
…
#define NDEBUG
…
assert(i>=0);
```

In this case, the assert will not be executed, since NDEBUG is defined. Note that one also can define items such as NDEBUG on the command line for gcc using the –D flag. For example,

```
gcc –o prog prog.c –DNDEBUG
```

Has the same effect as having “#define NDEBUG” as the first line of prog.c.



Note that ‘\0’ is represented as a byte containing all zeroes.

A single character (such as '\0') is enclosed in single quotes and, of course, is just that character. A string of characters (such as "hello") ends with a null character and is enclosed with with double quotes). Its value is the pointer to the string of characters.

Strings

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

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


We use the %s format code to print a string. The “string” is actually the pointer to an array of characters (ending with a ‘\0’), and thus the %s format code expects a pointer.,

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.

Strings

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

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


We've added the newline character to the format specifier of printf – the prompt now appears on the next line. Note that we could get the same effect by putting the newline character at the end of the string, rather than in the format specifier.

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

We can also print a single character at a time. Note the test for the null character, which determines whether we've reached the end of the string.

1-D Arrays

- If T is a datatype (such as int), then

  ```
  T n[6]
  ```

  declares `n` to be an array of six `T`'s
  - the type of each element goes before the identifier
  - the number of elements goes after the identifier
- What is n's type?

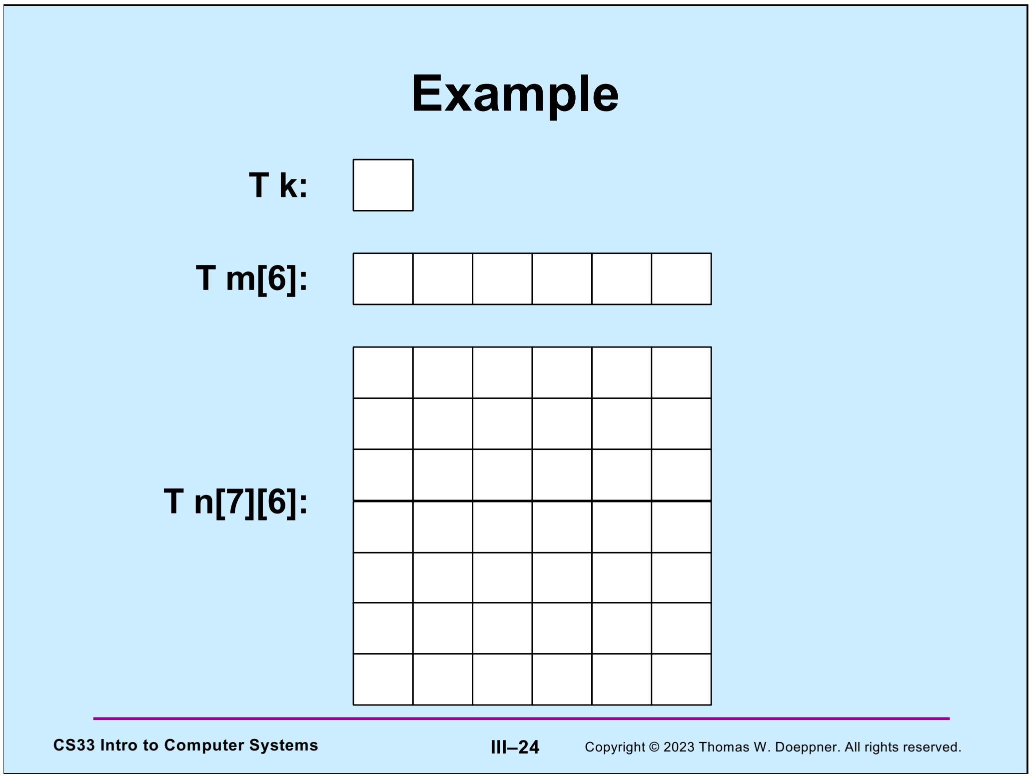
  ```
  T[6]
  ```


To declare something, say n, to be of type "T[6]", we must put the identifier between the element type and the size: T n[6].

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`

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, the type of each array element goes just before the identifier.



At the top we have k, which is of type T. Next, we have m, which is effectively of type T[6], but is an array of 6 T. Finally, we have n, which we may consider to be an array of seven T[6], or a 2-D array (7x6) of T. Each row of n is a 1-D array. Note that the address associated with the variable n is the address of n[0][0].

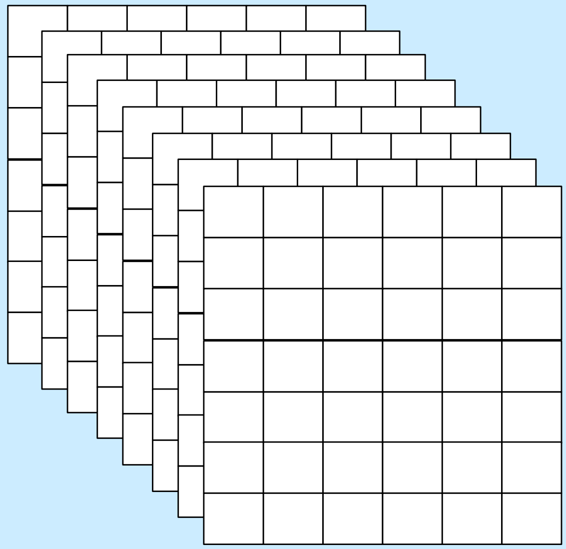
3-D Arrays

- 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`

Example

T m[8][7][6]:



2-D Arrays

```
$ ./a.out
      0       1       2       3
      4       5       6       7
      8       9      10      11
```

```
#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;
}
```


Here we initialize a 2D array, then call a function (described in the next slide) to print it.

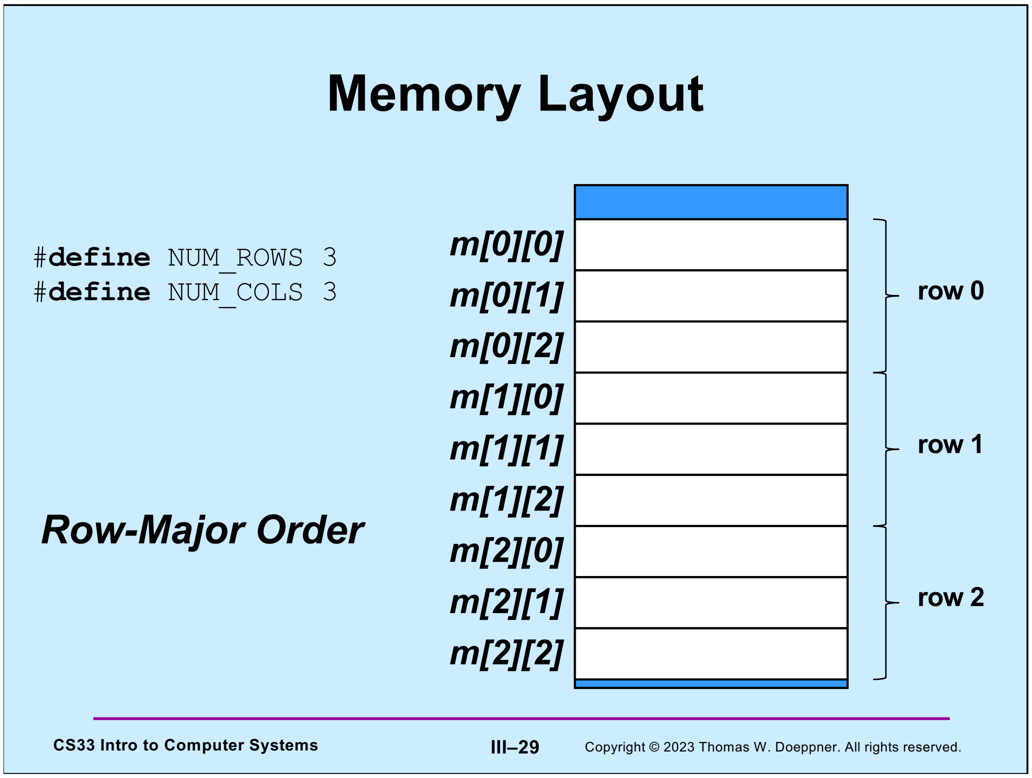
2-D Arrays

It must be told the dimensions

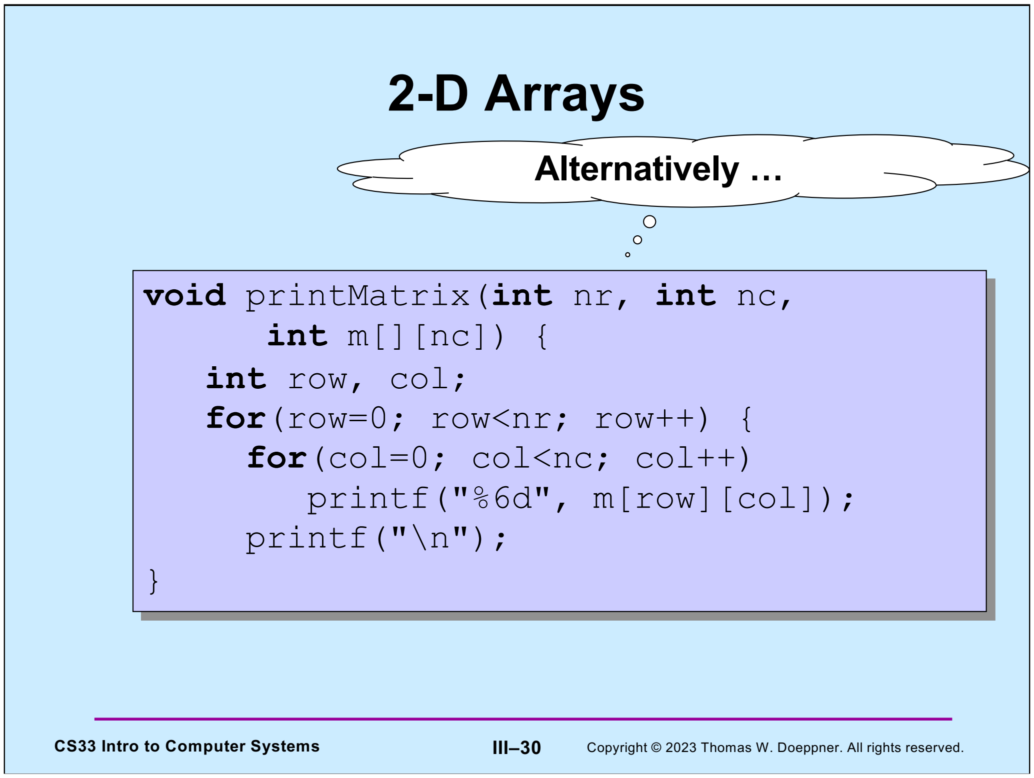
```
void printMatrix(int nr, int nc,
      int m[nr][nc]) {
   int row, col;
   for(row=0; row<nr; row++) {
     for(col=0; col<nc; col++)
        printf("%6d", m[row][col]);
     printf("\n");
   }
}
```


We print the array by rows.

Note that the parameter **m** is dimensioned by the previous parameters **nr** and **nc**. It's important that **nr** and **nc** appear in the parameter list before **m**.



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.



As we mentioned for 1-D arrays, when an array is passed to a function, what is passed is a pointer to its first element. For a 1-D array, say an array of ints, that first element is an int. For a 2-D array, the first element is a 1-D array, since a 2-D array is an array of 1-D arrays. Thus, what's passed to **printMatrix** in the slide is a pointer to the first row of the matrix, which is a 1-D array of ints. (And that 1-D array is passed as a pointer to its first element.)

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");
}
```


Note that m is an array of arrays (in particular, an array of 1-D arrays).

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
$
```


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.

Quiz 4

Consider the array

`int A[3][3];`

- which element is adjacent to `A[2][2]` in memory?
  a) `A[3][3]`
  b) `A[1][2]`
  c) `A[2][1]`
  d) none of the above

Quiz 5

Consider the array

```
int A[4][4];
int *B = &A[0][0];

B[8] = 8;
```

- which element of A was modified?
  a) **A[3][2]**
  b) **A[2][0]**
  c) **A[2][3]**
  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?

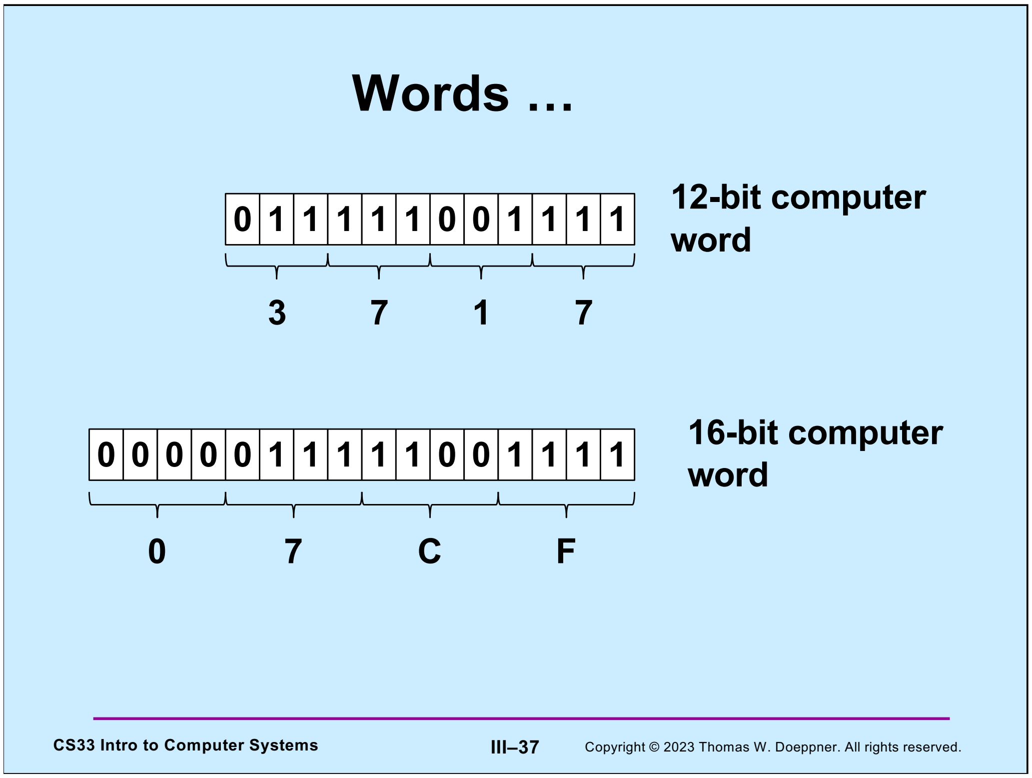
- **1999**
  - **base 10**
    - » $9 \cdot 10^0 + 9 \cdot 10^1 + 9 \cdot 10^2 + 1 \cdot 10^3$
  - **base 2**
    - » **11111001111**
      - $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 + 1 \cdot 2^9 + 1 \cdot 2^{10}$
  - **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?**

Base 2 is known as “binary” notation.

Base 8 is known as “octal” notation.

Base 10 is known as “decimal” notation.

Base 16 is known as “hexadecimal” notation. Note that “hexa” is derived from the Greek language and “decimal” is derived from the Latin language. Many people feel you shouldn’t mix languages when you invent words, but IBM, who coined the term “hexadecimal” in the 1960s, didn’t think their corporate image could withstand “sexadecimal”.



Note that a byte consists of two hexadecimal digits, which are sometimes known as "nibbles". A 32-bit computer word would then have eight nibbles; a 64-bit computer word would have sixteen nibbles.

Note that for the moment we consider only unsigned integers: i.e., integers whose values are nonnegative. (We explain signed integers in a week or two.)

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]);
}
```


This function prints the base **base** representation of **num**. The "%" operator yields the remainder. E.g., "10%3" evaluates to 1: the remainder after dividing 10 by 3. (Note that the "…" is not heretofore unexplained C syntax, but is shorthand for "fill this in to the extent needed.")

Or …

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


“bc” (it stands for basic calculator, or perhaps better calculator) is a standard Unix command that handles arbitrary-precision arithmetic. Among its features is the ability to specify which base to use for input and output of numbers. The default base for both input and output is ten. Setting **obase** to 16 sets the base for output to 16. Similarly, one can change the base for input numbers by setting **ibase**. Note that names of digits beyond 9 are upper-case letters (to avoid syntax issues when using variables, which are constrained to using lower-case letters).

Quiz 6

- What's the decimal (base 10) equivalent of 25_{16}?
  a) 19
  b) 35
  c) 37
  d) 38

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

Supplied by CMU.

Note that C also supports numbers written in octal (base-8) notation. They are written with a leading 0. Thus 016 is the same as 14, which is the same as 0xe.

Unsigned 32-Bit Integers

b_{31}	b_{30}	b_{29}	…	b_2	b_1	b_0

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

(we ignore negative integers for now)

If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide for 32-bit integers. Thus integers are represented in binary (base-2) notation in the computer. We'll discuss representing negative integers in an upcoming lecture.

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
$
```


Here n is an **unsigned int** whose value is 57 (expressed in base 10). As we've seen, it's represented in the computer in binary. When we print its value using **printf**, we choose to view it in the base specified by the format code. %b means binary, %u means decimal (assuming an unsigned int), and %x means hexadecimal.

Note, in the arguments for **printf**, that the format string is in two parts. C allows you to do this: "string 1 " "string 2" is treated the same as "string1 string2".

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

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

Supplied by CMU.

General Boolean Algebras

- **Operate on bit vectors**
  - operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- **All of the properties of boolean algebra apply**

Supplied by CMU.

Example: Representing & Manipulating Sets

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

01101001 *76543210*	{ 0, 3, 5, 6 }
01010101 *76543210*	{ 0, 2, 4, 6 }

- **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 }

Supplied by CMU.

Bit-Level Operations in C

- Operations &, |, ~, ^ 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)

```
~0x41 → 0xBE
  ~01000001₂ → 10111110₂
~0x00 → 0xFF
  ~00000000₂ → 11111111₂
0x69 & 0x55 → 0x41
  01101001₂ & 01010101₂ → 01000001₂
0x69 | 0x55 → 0x7D
  01101001₂ | 01010101₂ → 01111101₂
```


Supplied by CMU.

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 && (x || y) && ((x & z) | (y & z))
```


Supplied by CMU.

In the last example, there's no need to evaluate the complicated expression following p if p is false, since we know the final result will be false.

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - &&, ||, !
    - » vie … “false”

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

```
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && (x || y) && ((x & z) | (y & z))
```


Supplied by CMU.

Quiz 7

- 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) none of the above
  d) both a and b

Recall that a char is an 8-bit integer.

Shift Operations

- **Left Shift:** x << y
  - shift bit-vector x left y positions
    - throw away extra bits on left
    - » fill with 0's on right
- **Right Shift:** x >> 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 ≥ word size

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

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

Supplied by CMU.

Why we need both logical and arithmetic shifts should be clear by the end of an upcoming lecture. If one is applying a right shift to an **int**, it is an arithmetic right shift. Why this is so will be explained in the upcoming lecture (it has to do with the representation of negative numbers). Though we haven't yet explained the datatype "unsigned int" (which we will soon), when a right shift is applied to an **unsigned int**, it is a logical shift.