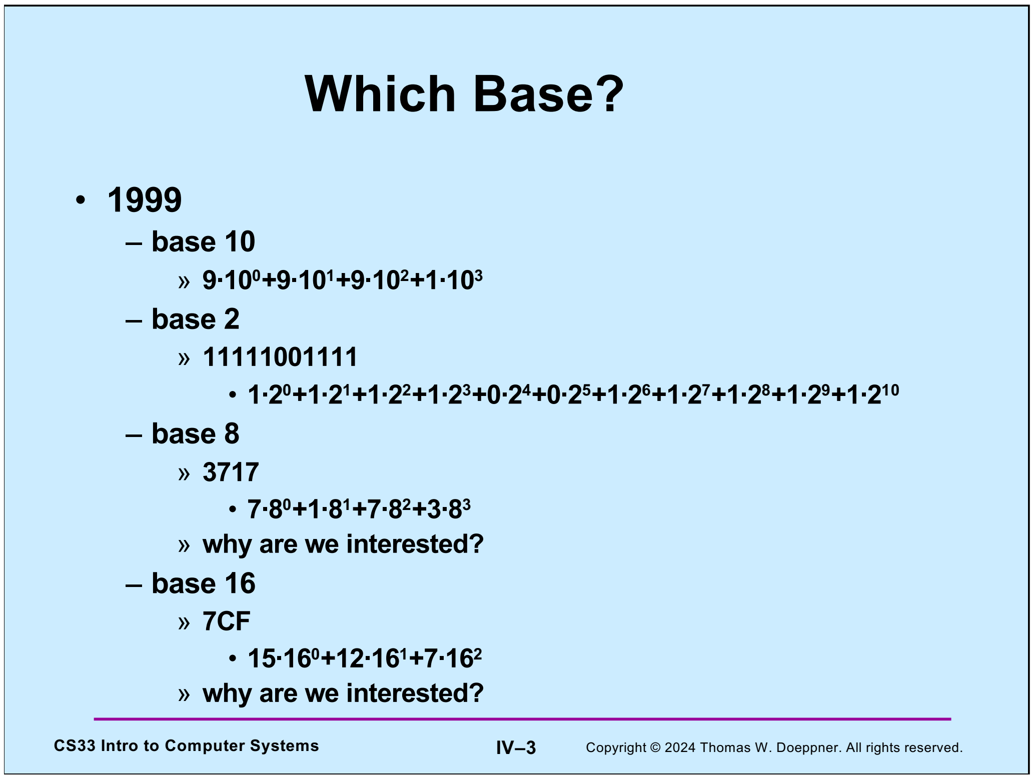
CS 33

Introduction to C
Part 4

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?

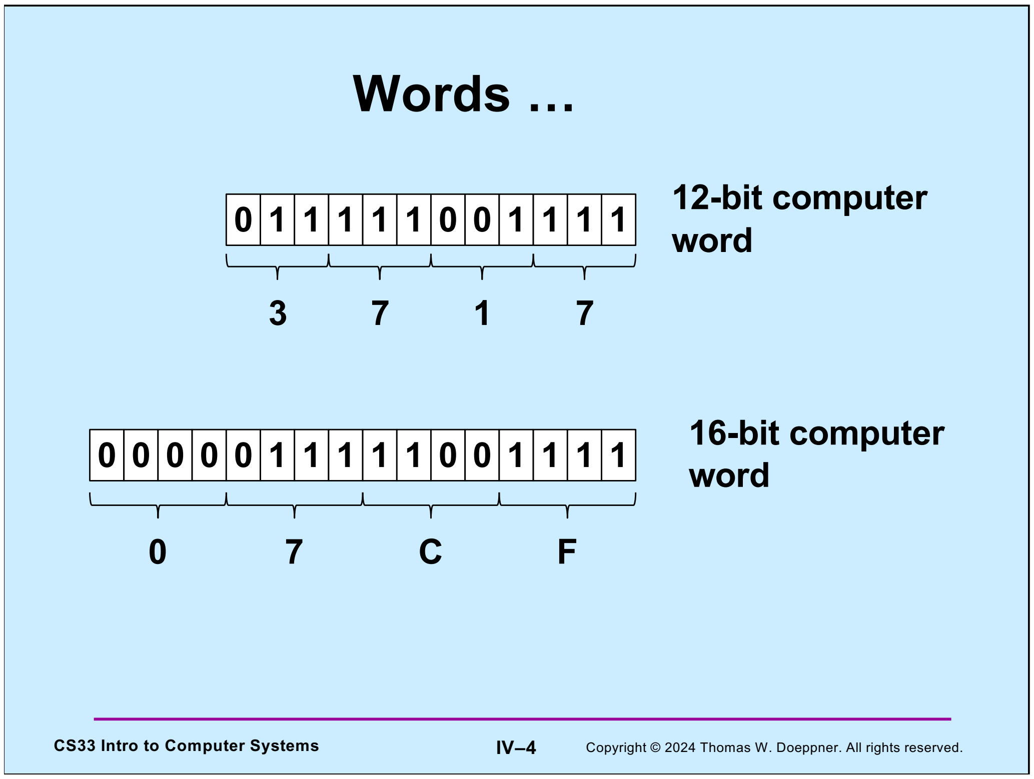
- 1999
  - base 10
    - » $9 \cdot 10^0 + 9 \cdot 10^1 + 9 \cdot 10^2 + 1 \cdot 10^3$
  - base 2
    - » 11111001111
      - $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 + 1 \cdot 2^9 + 1 \cdot 2^{10}$
  - base 8
    - »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

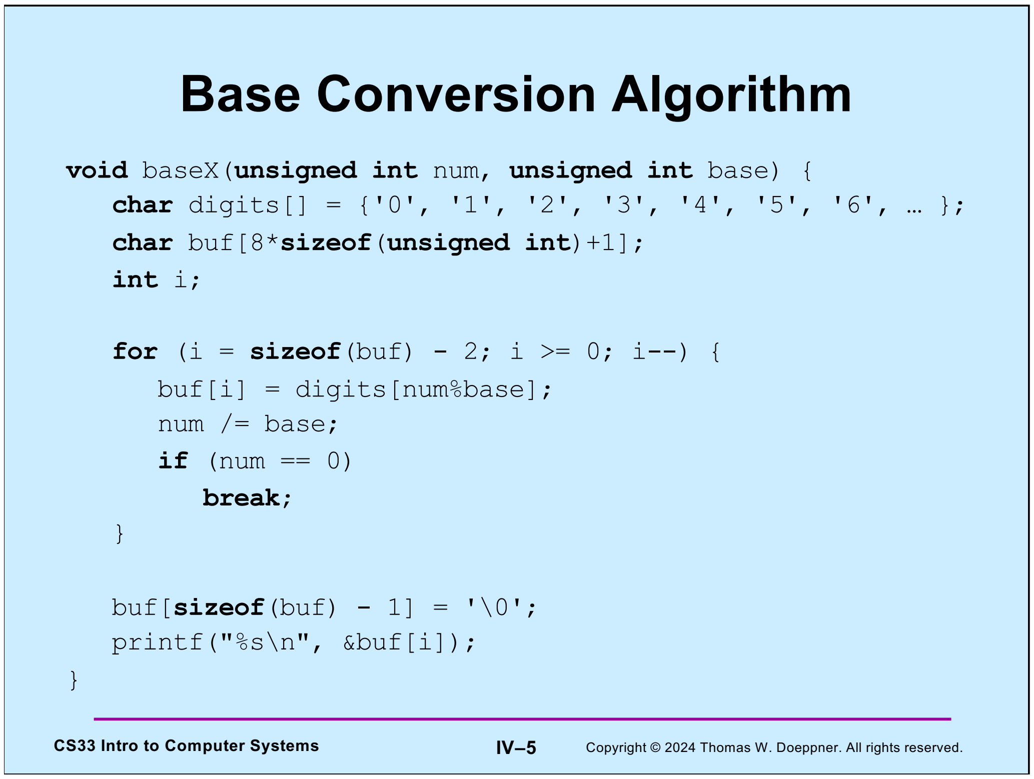
Base 16 is known as “hexadecimal” notation. Note that “hexa” is derived from the Greek language and “decimal” is derived from the Latin language. Many people feel you shouldn’t mix languages when you invent words, but IBM, who coined the term “hexadecimal” in the 1960s, didn’t think their corporate image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



Words …

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.

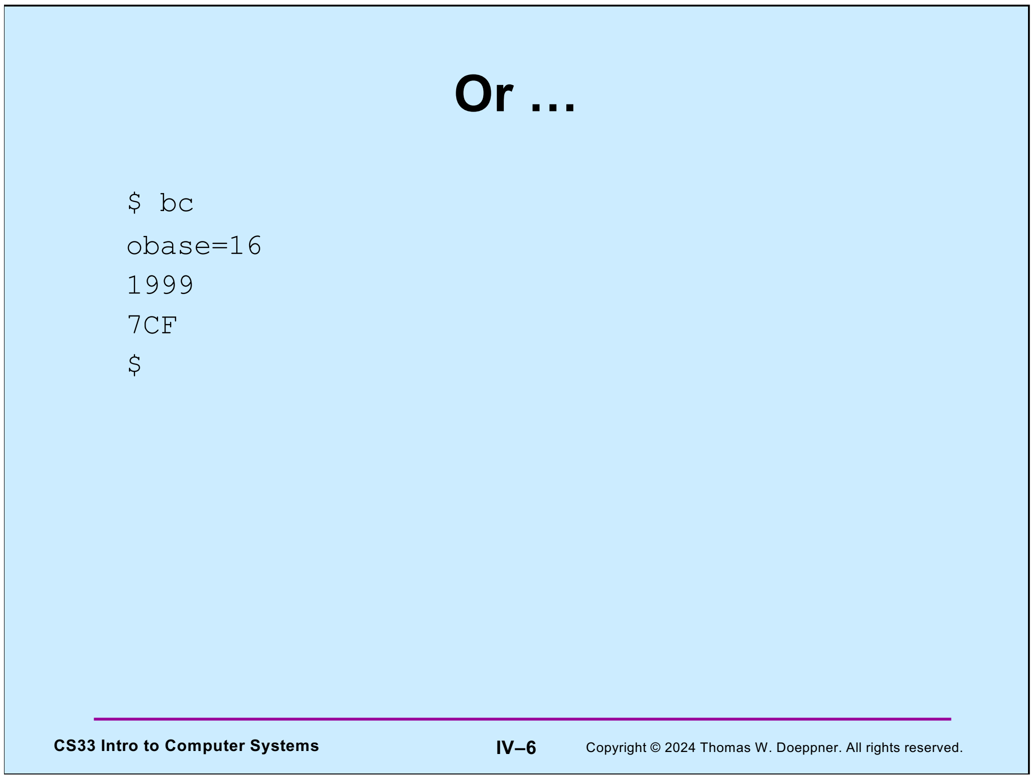
Note that for the moment we consider only unsigned integers: i.e., integers whose values are nonnegative. (We explain signed integers in a week or so.)



Base Conversion 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. We are doing integer division, thus the result of dividing 10 by 3 is 3. (Note that the “…” is not heretofore unexplained C syntax, but is shorthand for “fill this in to the extent needed.”)



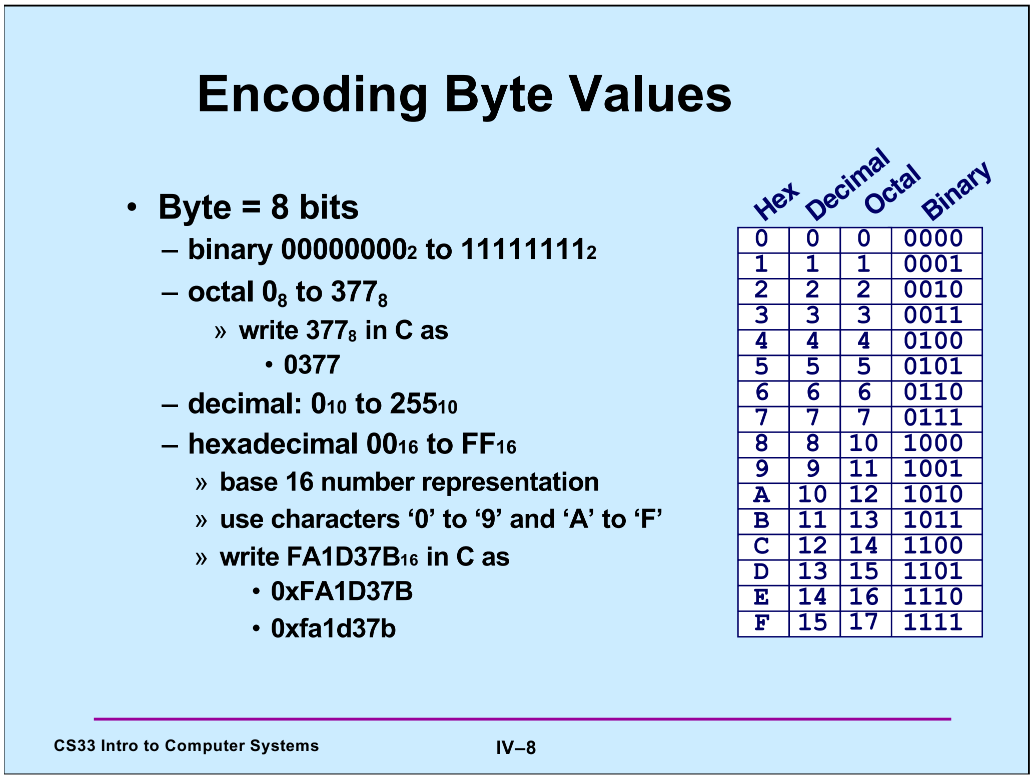
Or …

```
$ 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 be made up of lower-case letters).

Quiz 1

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



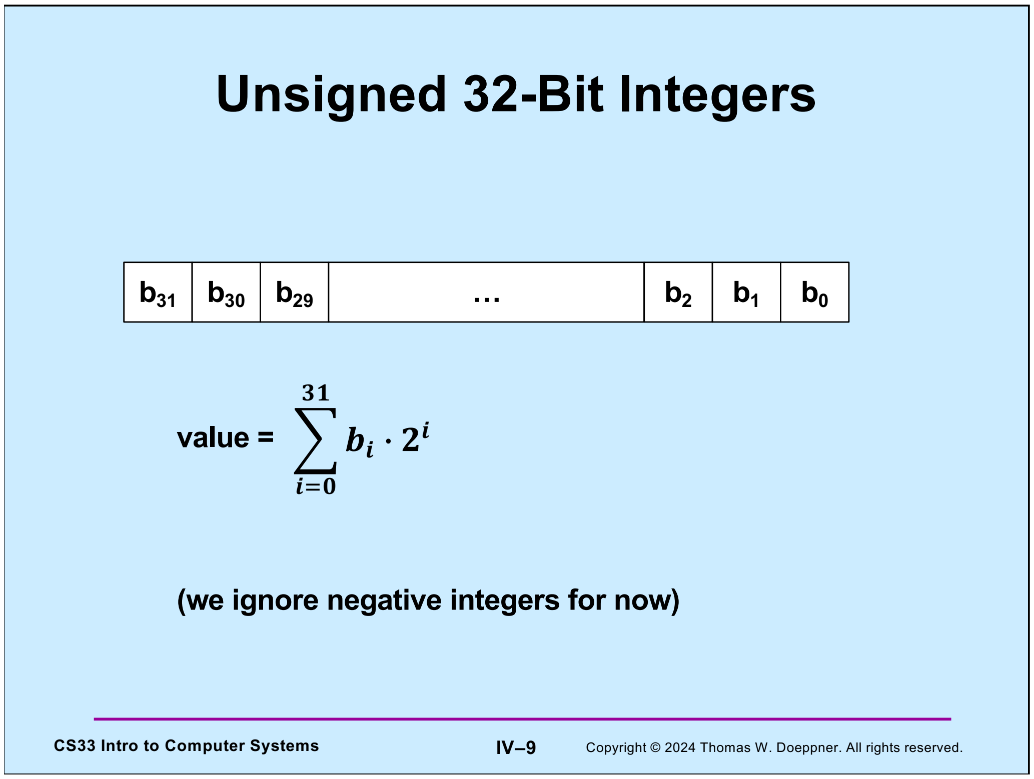
Encoding Byte Values

- Byte = 8 bits
  - binary 00000000_2 to 11111111_2
  - octal 0_8 to 377_8
    - » write 377_8 in C as
      - 0377
  - 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	Octal	Binary
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

Supplied by CMU.

Note that C 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.



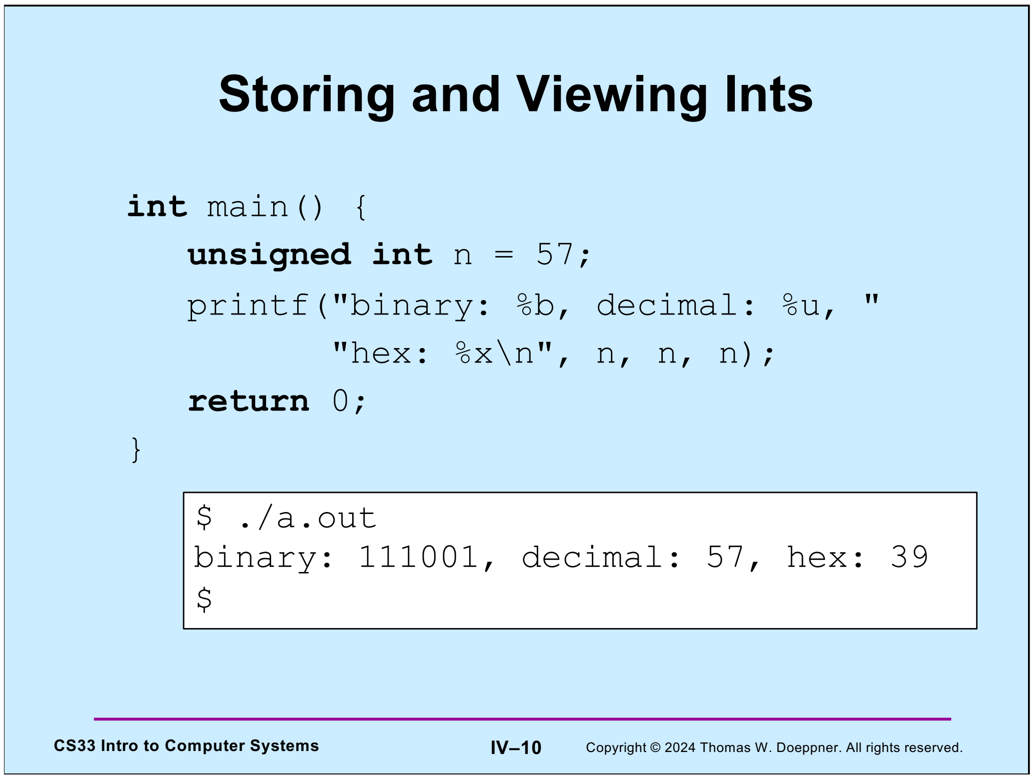
Unsigned 32-Bit Integers

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.



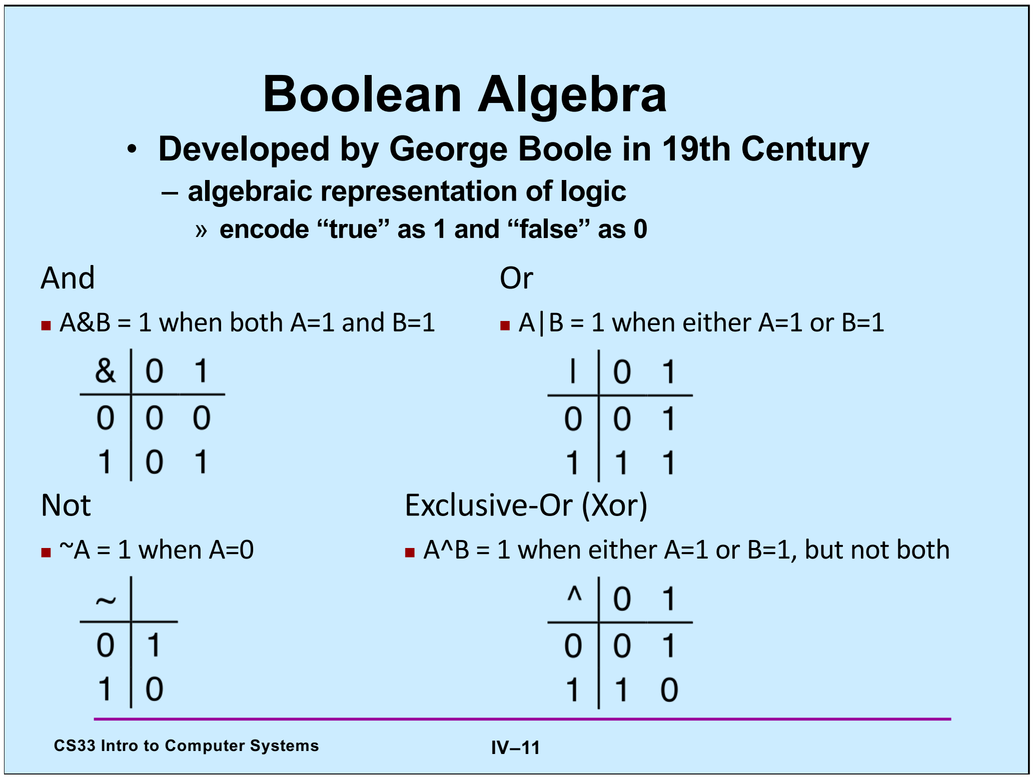
Storing and Viewing Ints

```
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. We could use %o to print a value in octal.

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

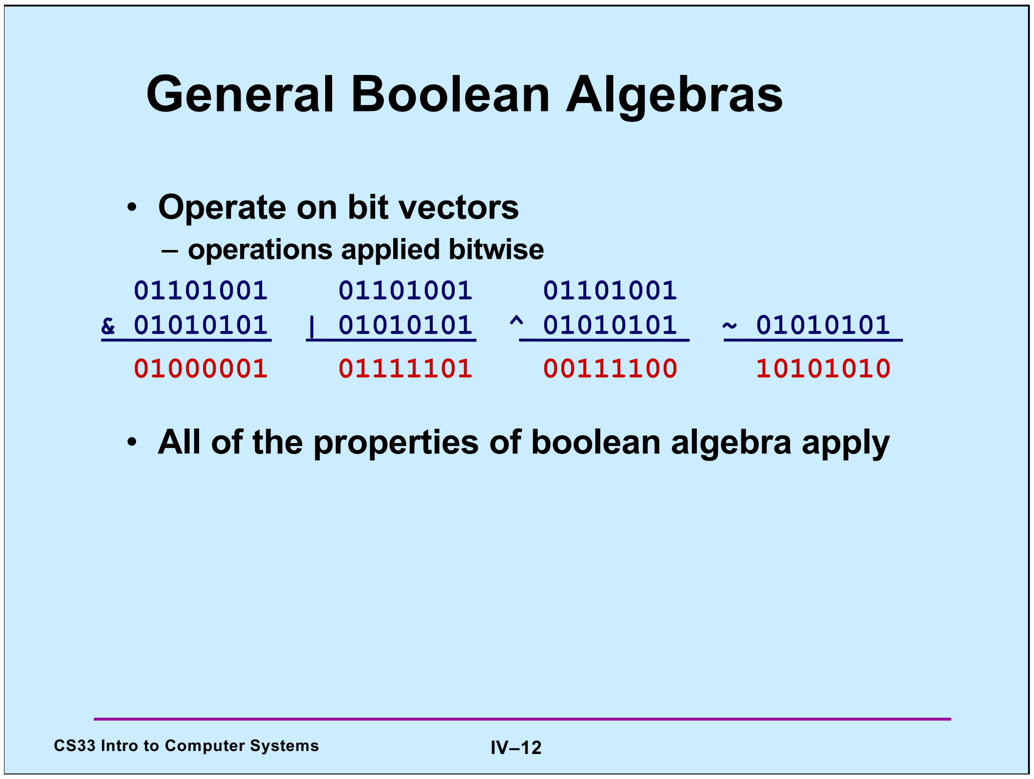
~	
0	1
1	0

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.



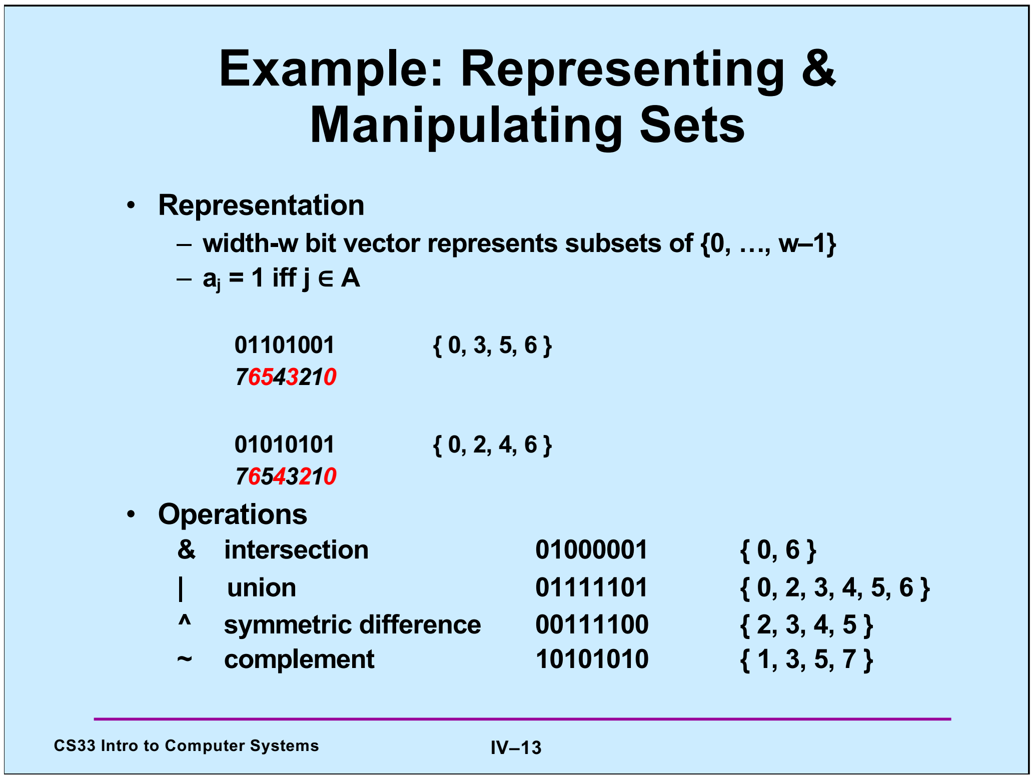
General Boolean Algebras

- 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

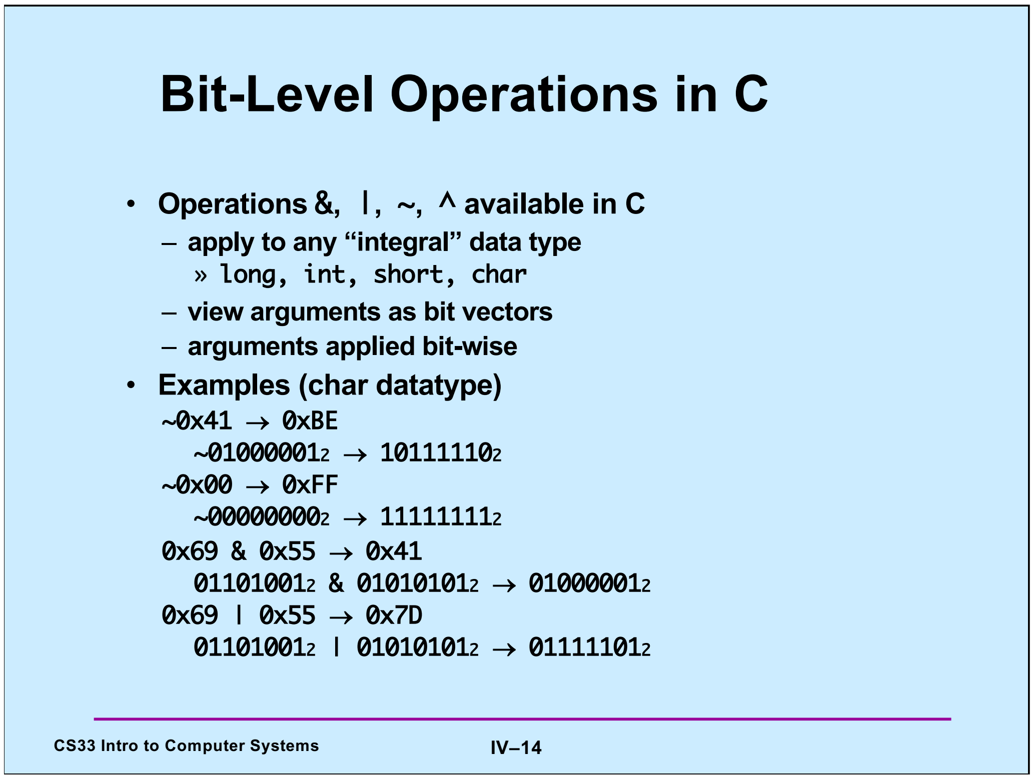
- **Representation**
  - width-w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ iff $j \in A$

01101001	{ 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 }

Supplied by CMU.

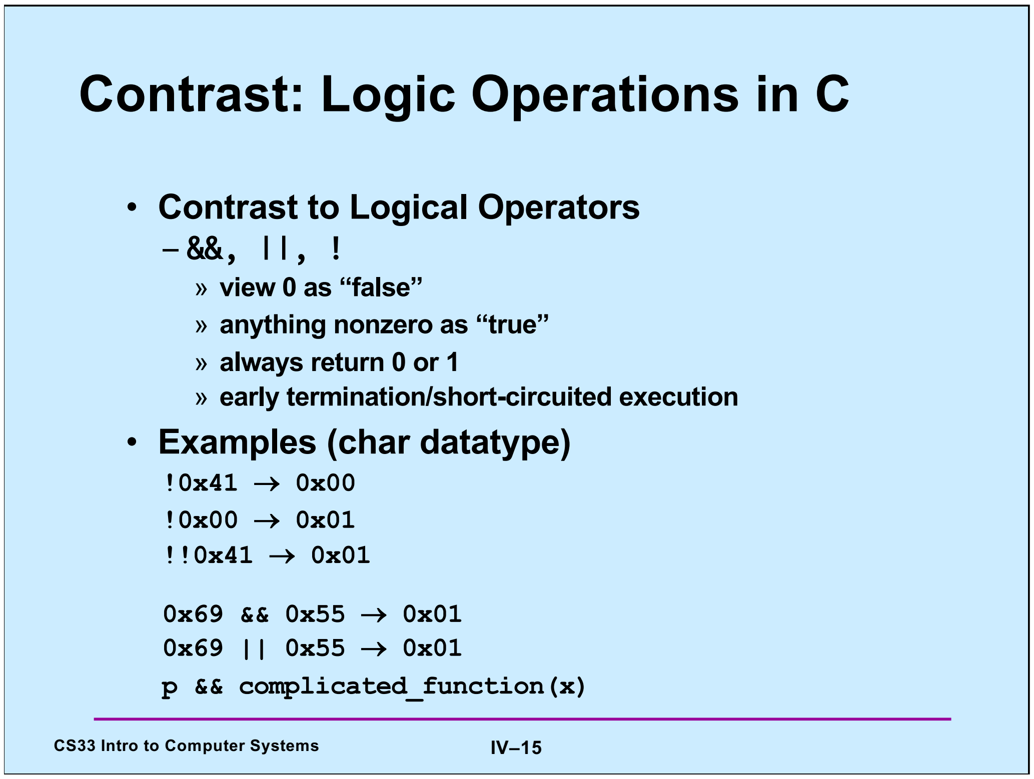


Bit-Level Operations in C

- 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 $\rightarrow$ 0xBE
  ~01000001_2 $\rightarrow$ 10111110_2
~0x00 $\rightarrow$ 0xFF
  ~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

Supplied by CMU.



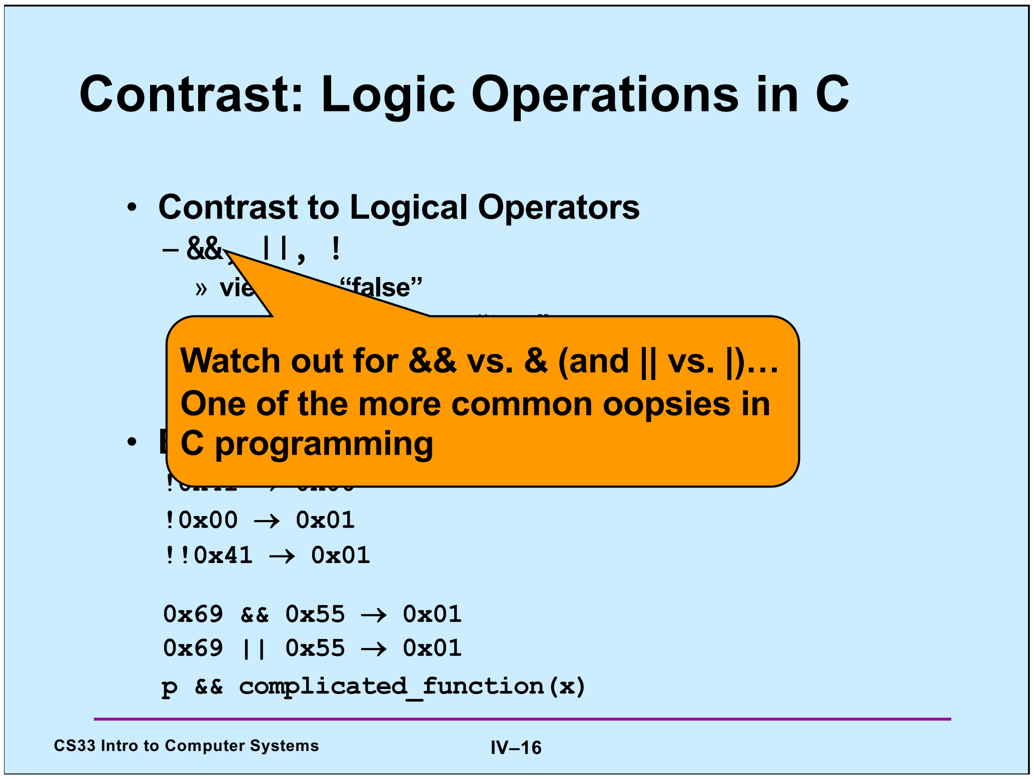
Contrast: Logic Operations in C

- **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 && complicated_function(x)
```

Supplied by CMU.

In the last example, since expressions are evaluated left to right and evaluation stops once the result is known, there's no need to evaluate the complicated function following p if p is false, since we know the final result will be false.



Contrast: Logic Operations in C

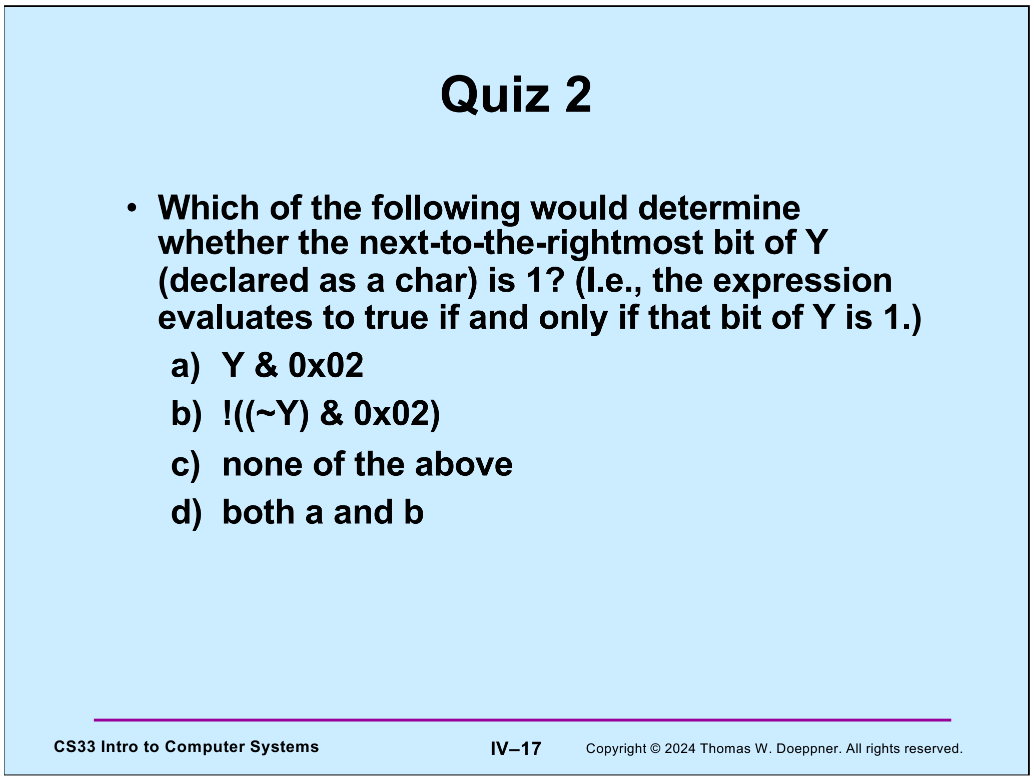
- Contrast to Logical Operators
  - &&, ||, !
    - » vie “false”

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

- E

```
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

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

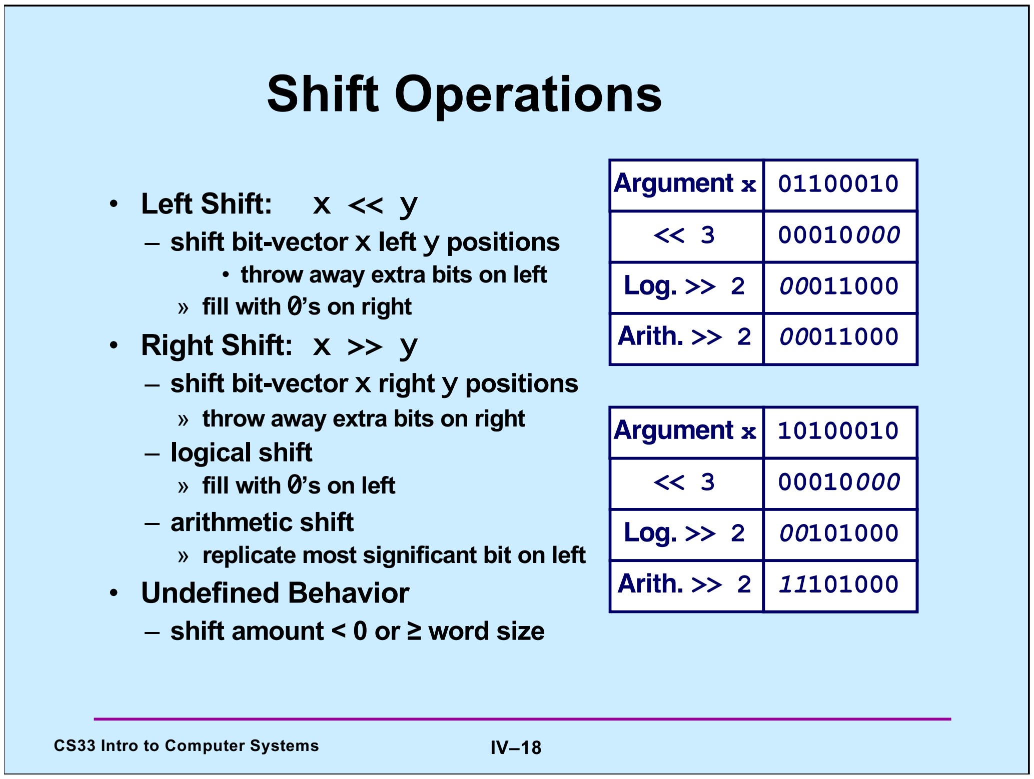
Supplied by CMU.



Quiz 2

- 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

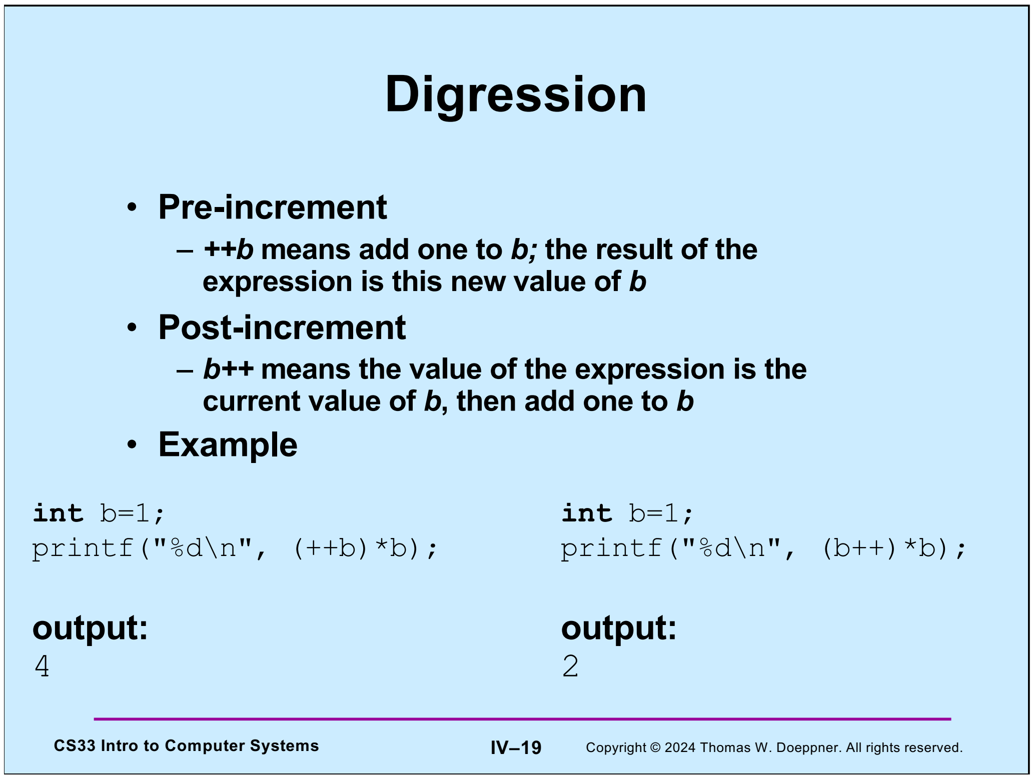
- **Left Shift:** x << y
  - shift bit-vector x left y positions
    - throw away extra bits on left
    - » fill with 0's on right
- **Right Shift:** x >> y
  - shift bit-vector x right y positions
    - » throw away extra bits on right
  - logical shift
    - » fill with 0's on left
  - arithmetic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

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

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 will be an arithmetic right shift. For **unsigned int**s, right shifts are logical right shifts. Why this is so will be explained in the upcoming lecture (it has to do with the representation of negative numbers).



Digression

- **Pre-increment**
  - ***++b* means add one to *b;* the result of the expression is this new value of *b***
- **Post-increment**
  - ***b++* means the value of the expression is the current value of *b*, then add one to *b***
- **Example**

```
int b=1;
printf("%d\n", (++b)*b);
```

output:

```
4
```

```
int b=1;
printf("%d\n", (b++)*b);
```

output:

```
2
```

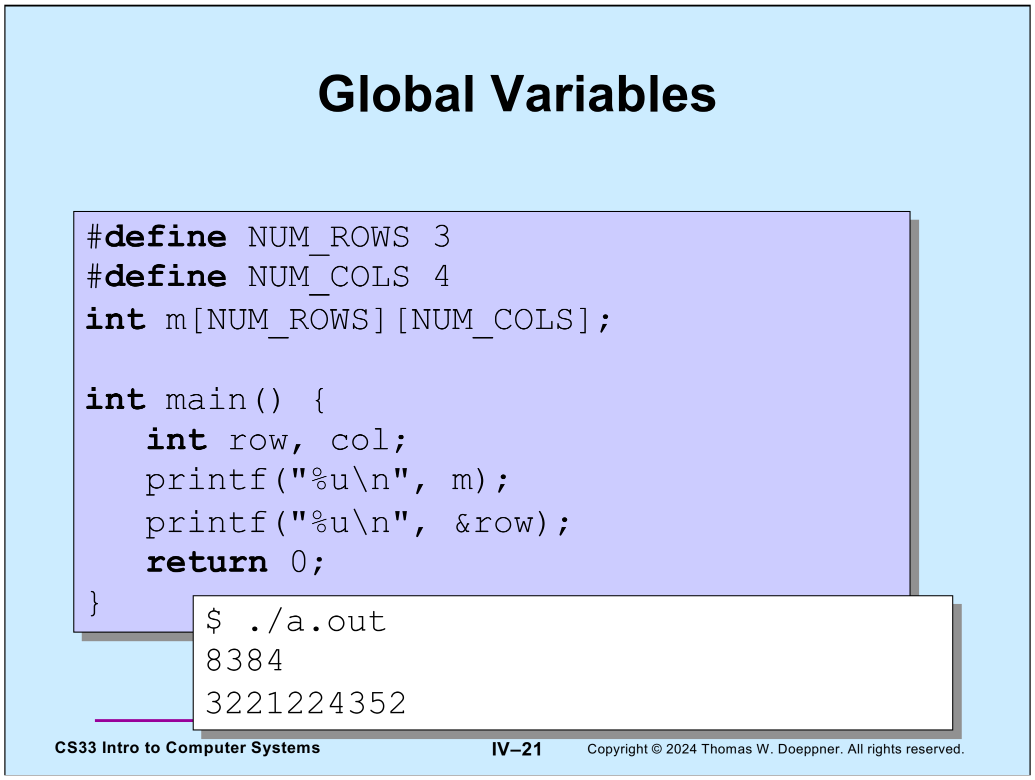
The above applies analogously to --b and b--; these are known as pre-decrement and post-decrement.

While these operators can be (very successfully) used to make code extremely hard to read; if used well, they can actually make code easier to read as well as to write.

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



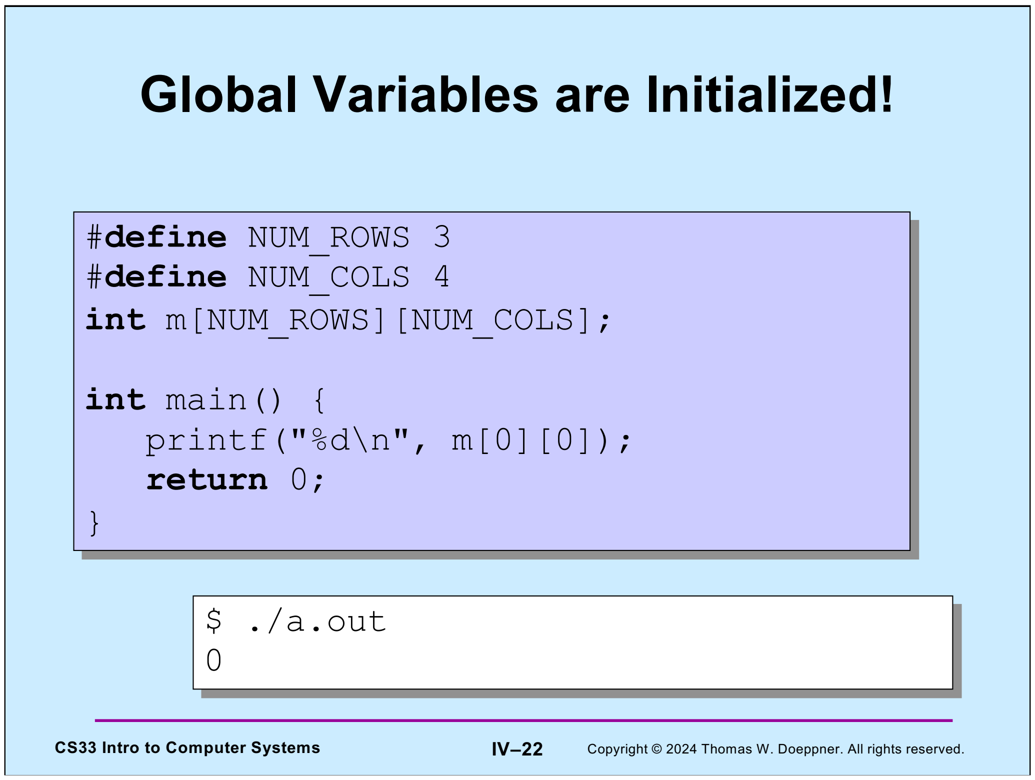
Global Variables

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

int main() {
   int row, col;
   printf("%u\n", m);
   printf("%u\n", &row);
   return 0;
}
```

```
$ ./a.out
8384
3221224352
```

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.

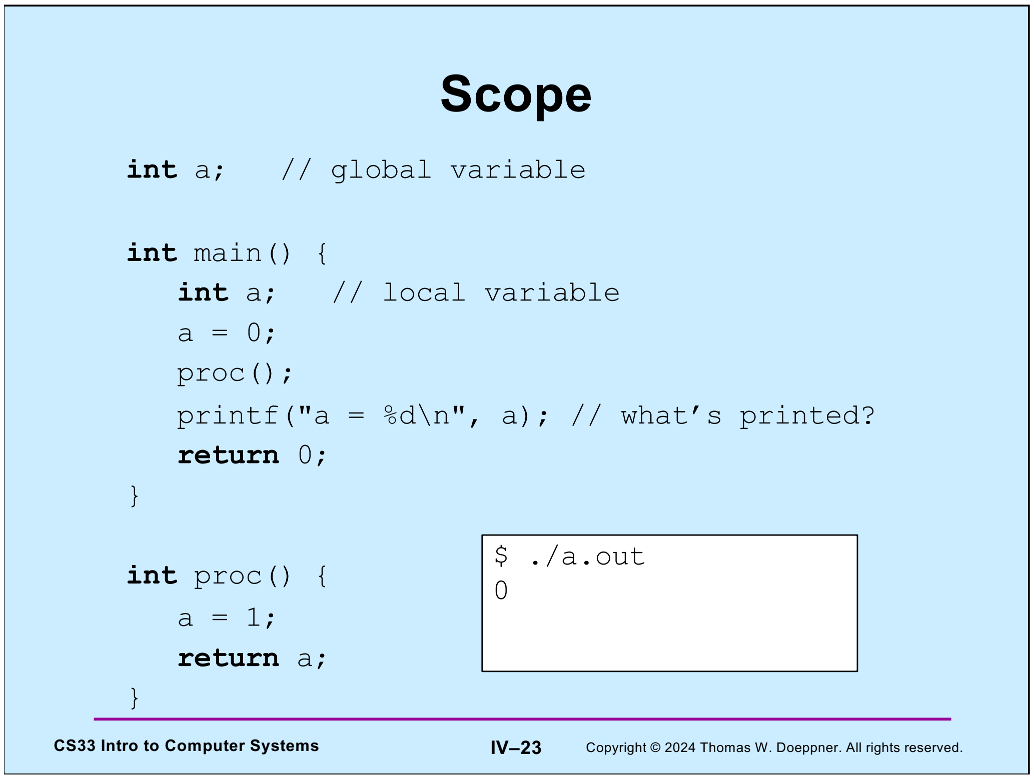


Global Variables are Initialized!

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

int main() {
   printf("%d\n", m[0][0]);
   return 0;
}
```

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

If you don't explicitly initialize a global variable, its initial value is guaranteed to be zero.

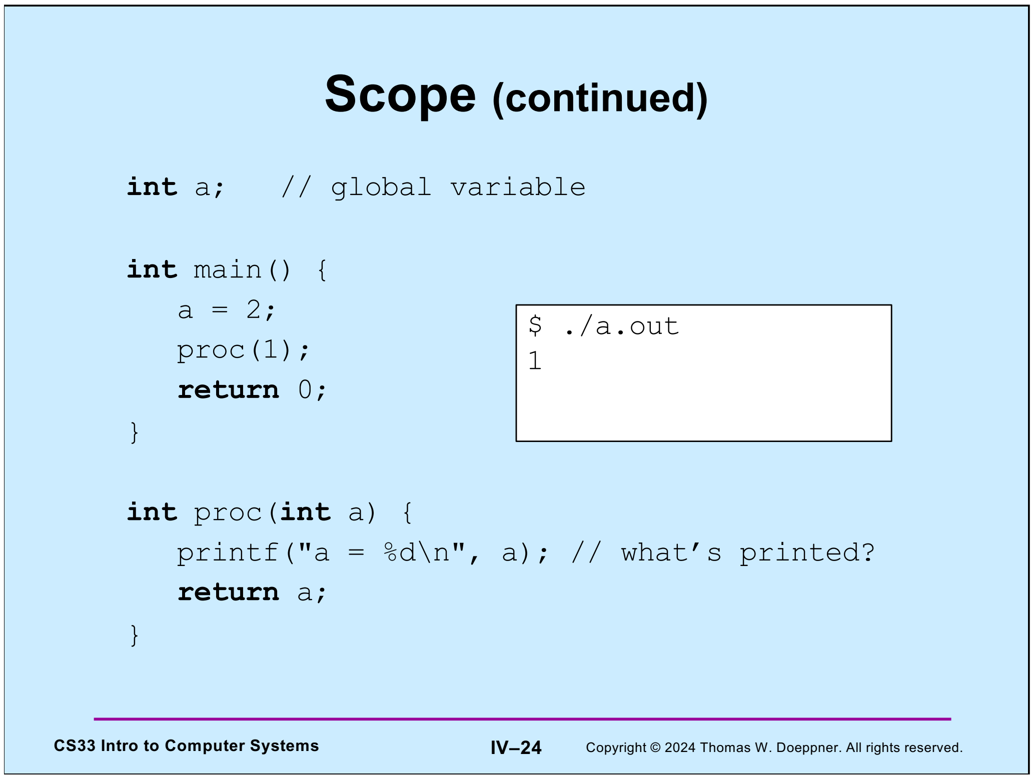


Scope

```
int a;   // global variable

int main() {
   int a;   // local variable
   a = 0;
   proc();
   printf("a = %d\n", a); // what’s printed?
   return 0;
}

int proc() {
   a = 1;
   return a;
}
```

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

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.

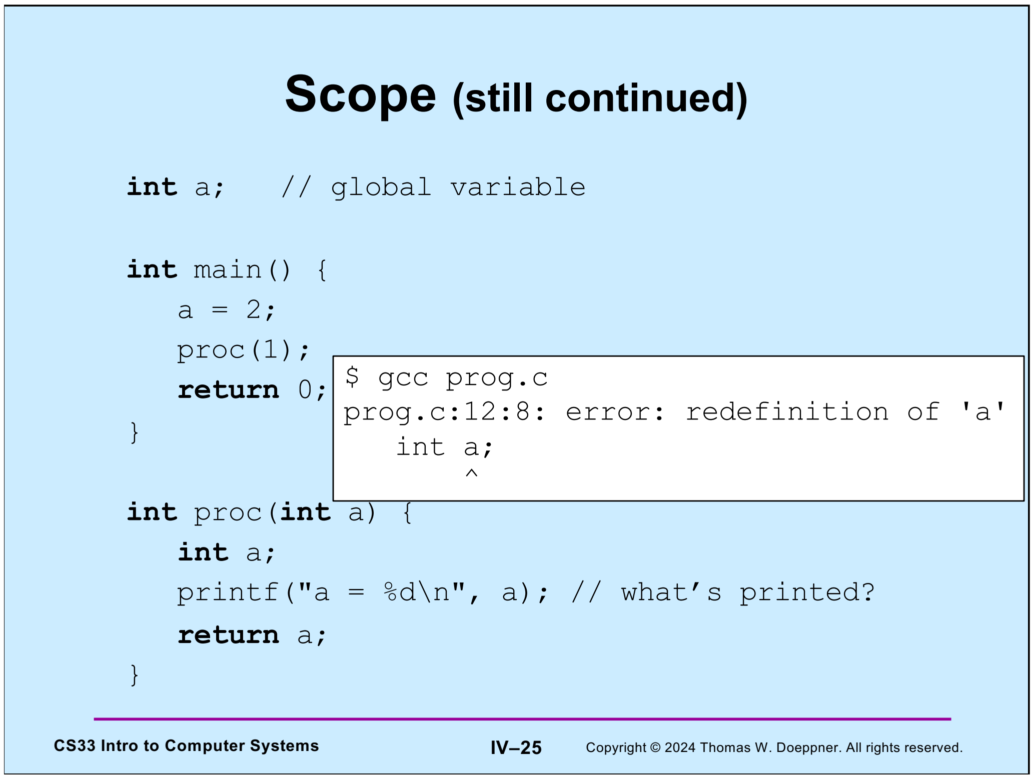


Scope (continued)

```
int a;   // global variable

int main() {
   a = 2;
   proc(1);
   return 0;
}

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

```
$ ./a.out
1
```

Here **a** is declared as a parameter to **proc**, thus references to **a** in **proc** are to the parameter and not to the global variable.



Scope (still continued)

```
int a;   // global variable

int main() {
   a = 2;
   proc(1);
   return 0;
}

int proc(int a) {
   int a;
   printf("a = %d\n", a); // what’s printed?
   return a;
}
```

```
$ gcc prog.c
prog.c:12:8: error: redefinition of 'a'
   int a;
       ^
```

Syntax error: one can’t have a local variable in a scope in which a parameter is declared with the same name.

Scope (more ...)

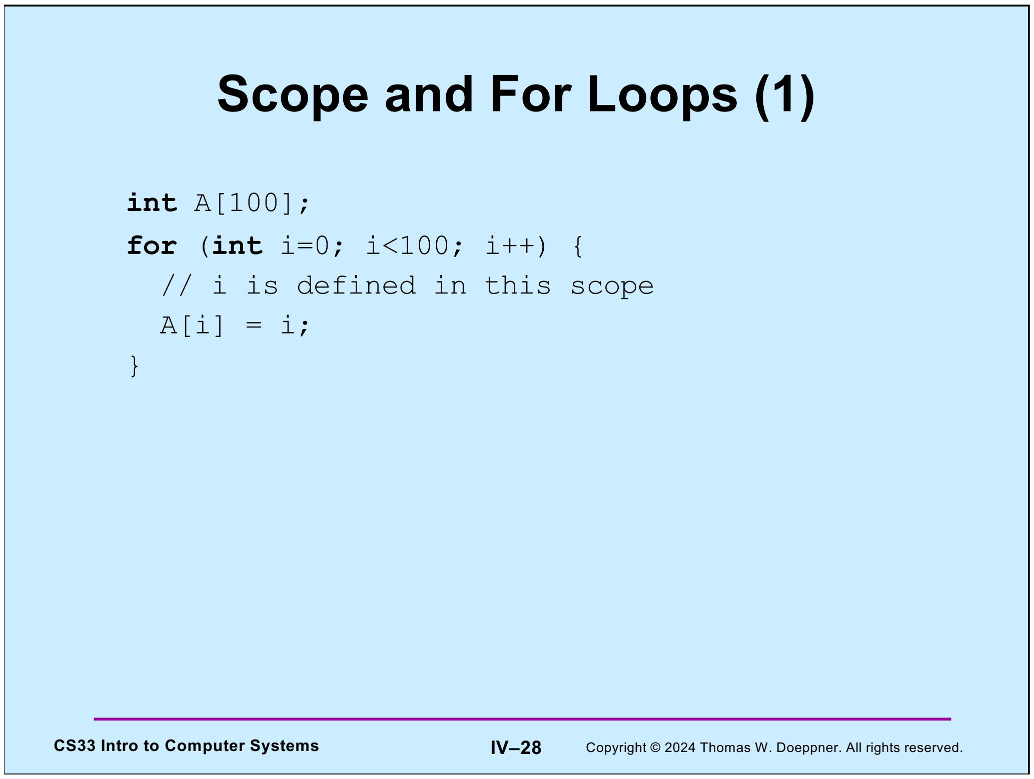
```
int a;   // global variable

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=6;}
   a = b;
   return a+2;
}

int main() {
   {int a = proc(4);}
   printf("a = %d\n", a);
   return 0;
}
```

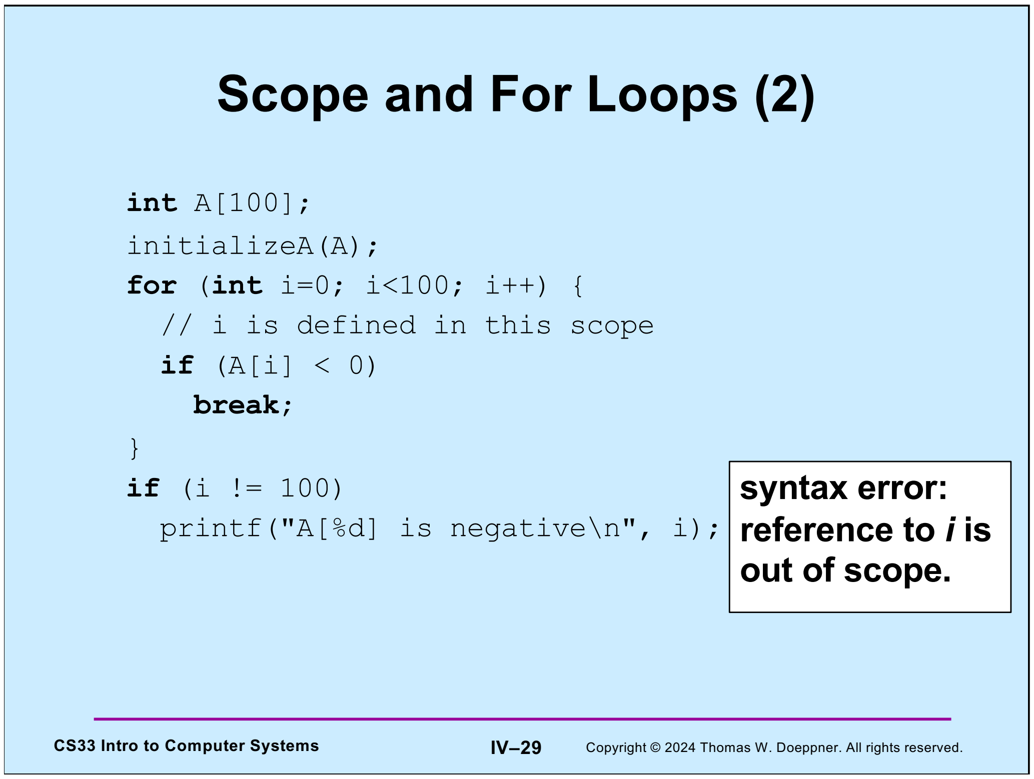
- What's printed?
  a) 0
  b) 4
  c) 6
  d) 8
  e) nothing; there's a syntax error



Scope and For Loops (1)

```
int A[100];
for (int i=0; i<100; i++) {
  // i is defined in this scope
  A[i] = i;
}
```

It's often convenient to declare a for loop's index variable in the for loop, as shown in the slide.

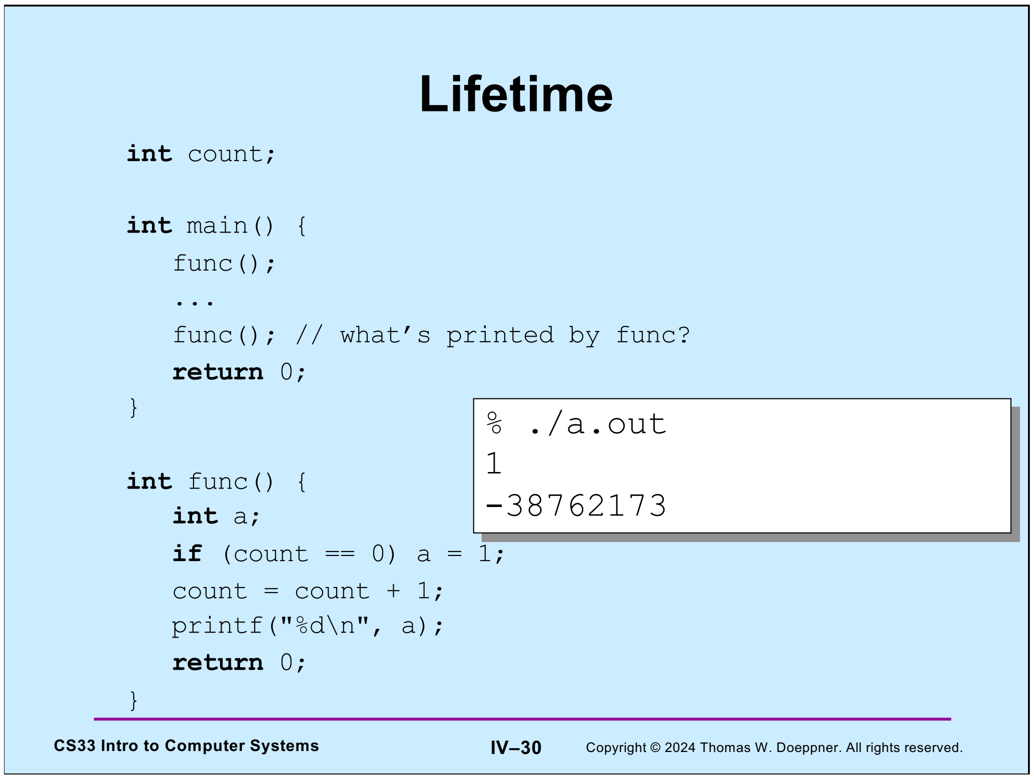


Scope and For Loops (2)

```
int A[100];
initializeA(A);
for (int i=0; i<100; i++) {
  // i is defined in this scope
  if (A[i] < 0)
    break;
}
if (i != 100)
  printf("A[%d] is negative\n", i);
```

syntax error: reference to *i* is out of scope.

But be careful – the scope of such an index variable does not extend outside of the for loop.

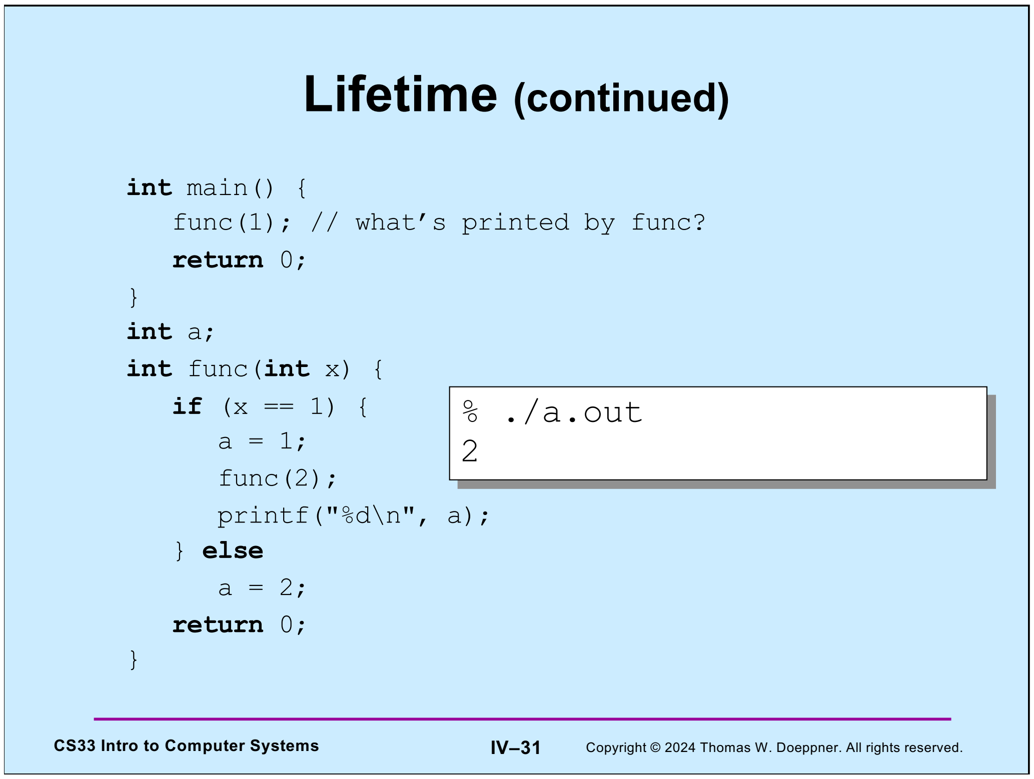


Lifetime

```
int count;

int main() {
   func();
   ...
   func(); // what's printed by func?
   return 0;
}

int func() {
   int a;
   if (count == 0) a = 1;
   count = count + 1;
   printf("%d\n", a);
   return 0;
}
```

```
% ./a.out
1
-38762173
```

Even though **a** is given a value the first time **func** is called, on **func**'s second invocation **a** is not given a value and thus the result that's printed is "undefined". This is because the lifetime of **a** is just for the length of time its scope is active, which is from when the execution of **func** starts to when **func** returns. The **a** in the next invocation of **func** is different from the previous **a**.

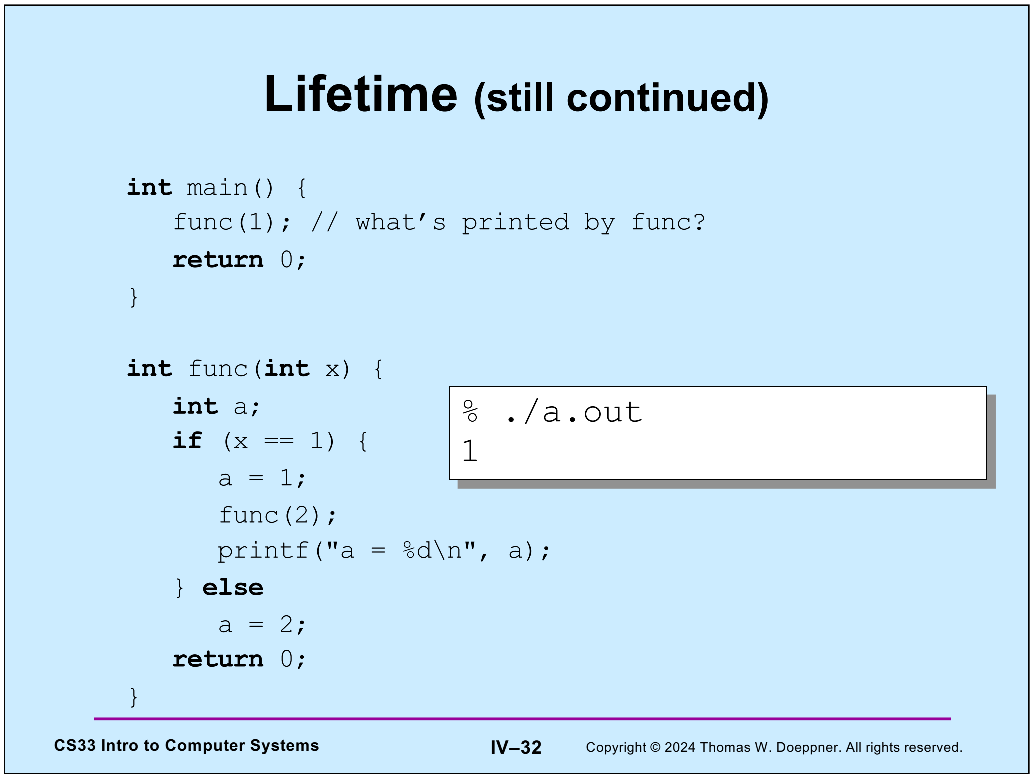


Lifetime (continued)

```
int main() {
   func(1); // what’s printed by func?
   return 0;
}
int a;
int func(int x) {
   if (x == 1) {
      a = 1;
      func(2);
      printf("%d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
2
```

In this case, **a** is global and thus the value set for it in one invocation of **func** is still there for the next invocation – the lifetime of **a** is that of the program itself.

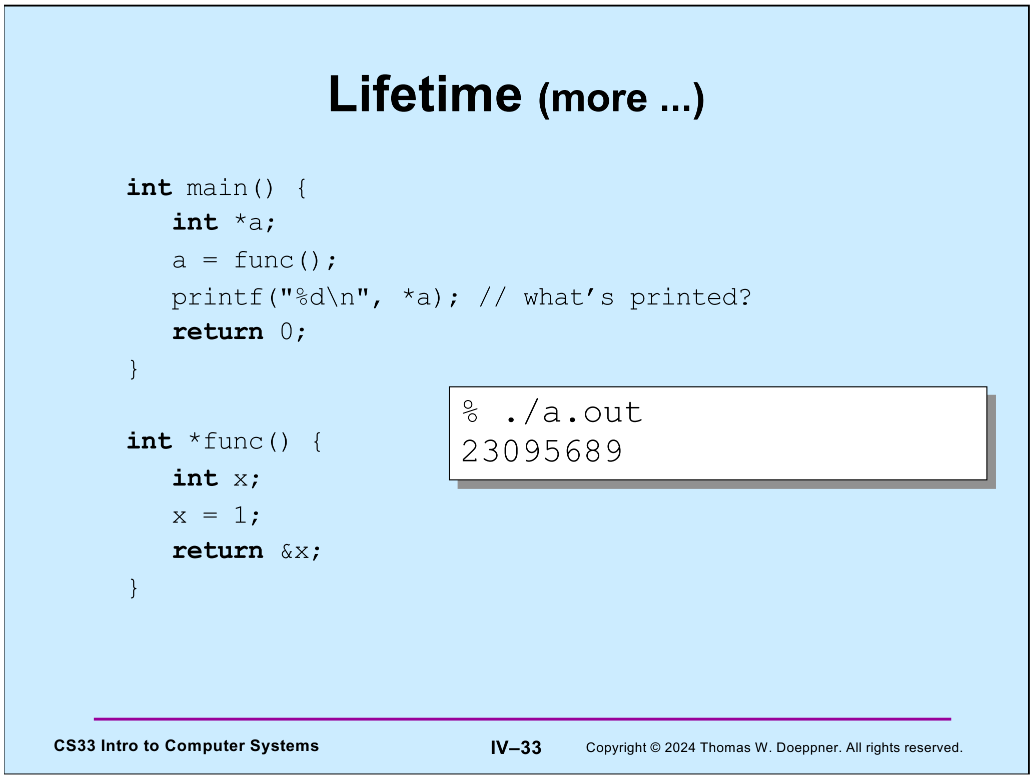


Lifetime (still continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}

int func(int x) {
   int a;
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
1
```

Here **a** is local again. **func** is called (recursively) from within itself: the recursive invocation of **func** modifies a different **a** than is used in the first invocation. Thus, the value printed is 1.

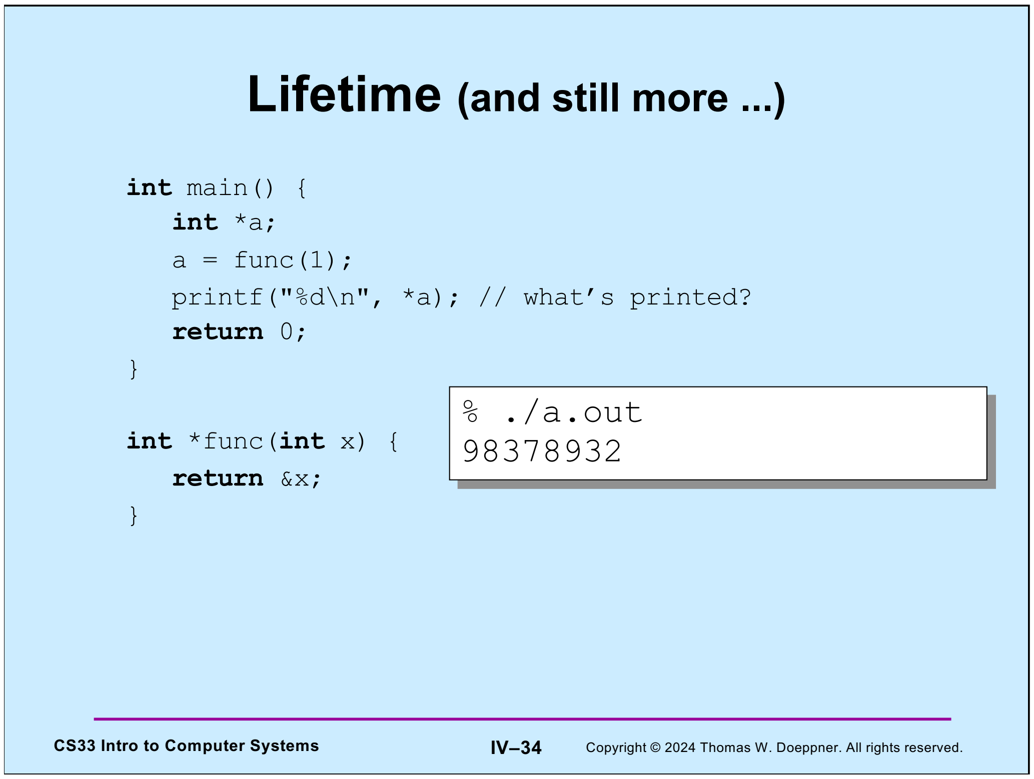


Lifetime (more ...)

```
int main() {
   int *a;
   a = func();
   printf("%d\n", *a); // what’s printed?
   return 0;
}

int *func() {
   int x;
   x = 1;
   return &x;
}
```

```
% ./a.out
23095689
```

When a function returns, its local variables become out of scope and no longer active – the lifetime of local variables is from the instant the function is called to when it returns. Thus, a pointer to a local variable refers to an undefined value if the variable is of a function invocation that is no longer active.



Lifetime (and still more ...)

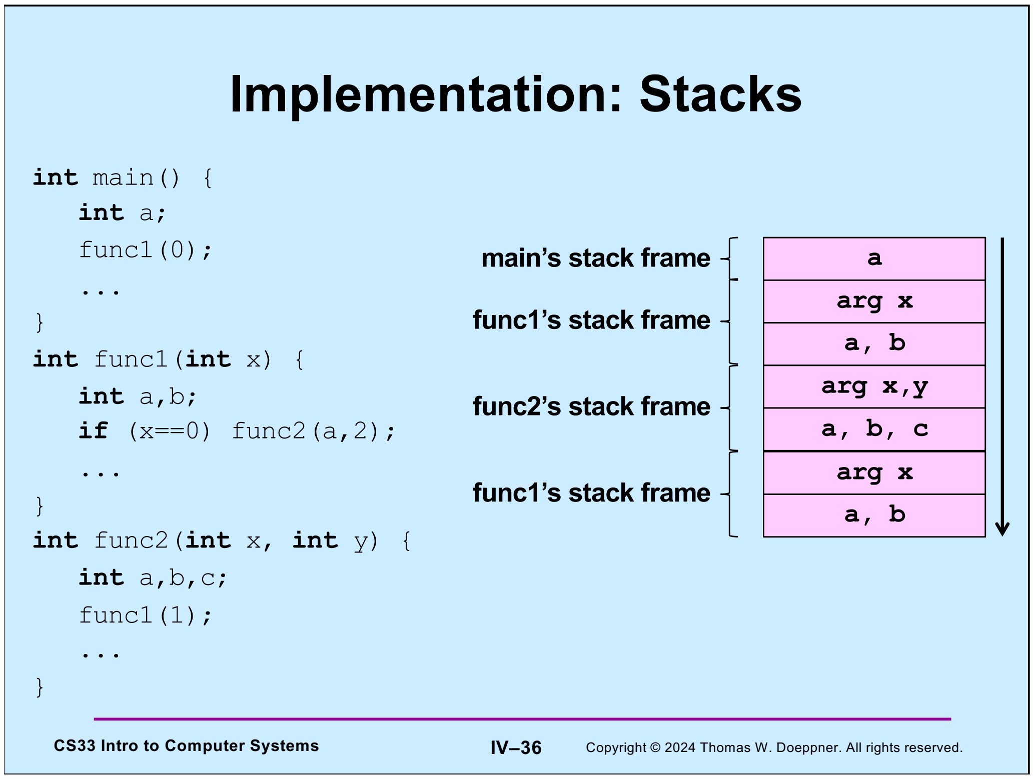
```
int main() {
   int *a;
   a = func(1);
   printf("%d\n", *a); // what’s printed?
   return 0;
}

int *func(int x) {
   return &x;
}
```

```
% ./a.out
98378932
```

Similarly, the lifetime of function arguments is the same as the lifetime of the function.

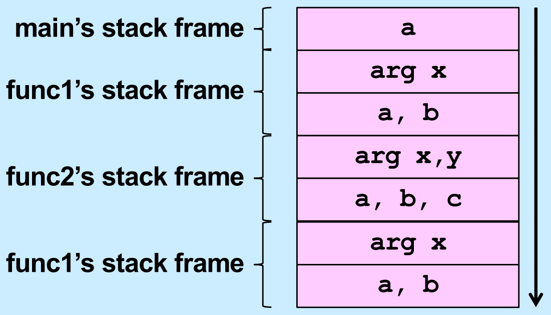
Rules

- Global variables exist for the duration of program's lifetime
- Local variables and arguments exist for the duration of the execution of the function
  - from call to return
  - each execution of a function results in a new instance of its arguments and local variables



Implementation: Stacks

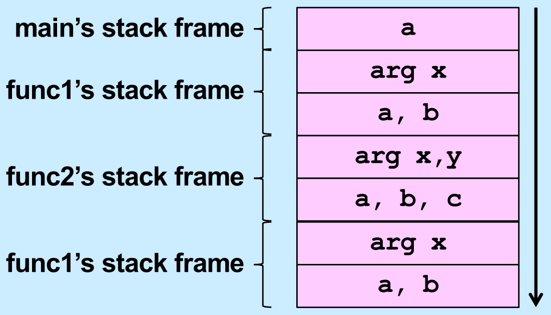
```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Function calling in C (and in most other languages) is implemented on stacks. Associated with an invocation of a function is a stack frame, which contains, among other things, its arguments and local variables. When a function is called, a stack frame for it is pushed onto the stack. When it returns, its stack frame is popped off the stack.

Implementation: Stacks

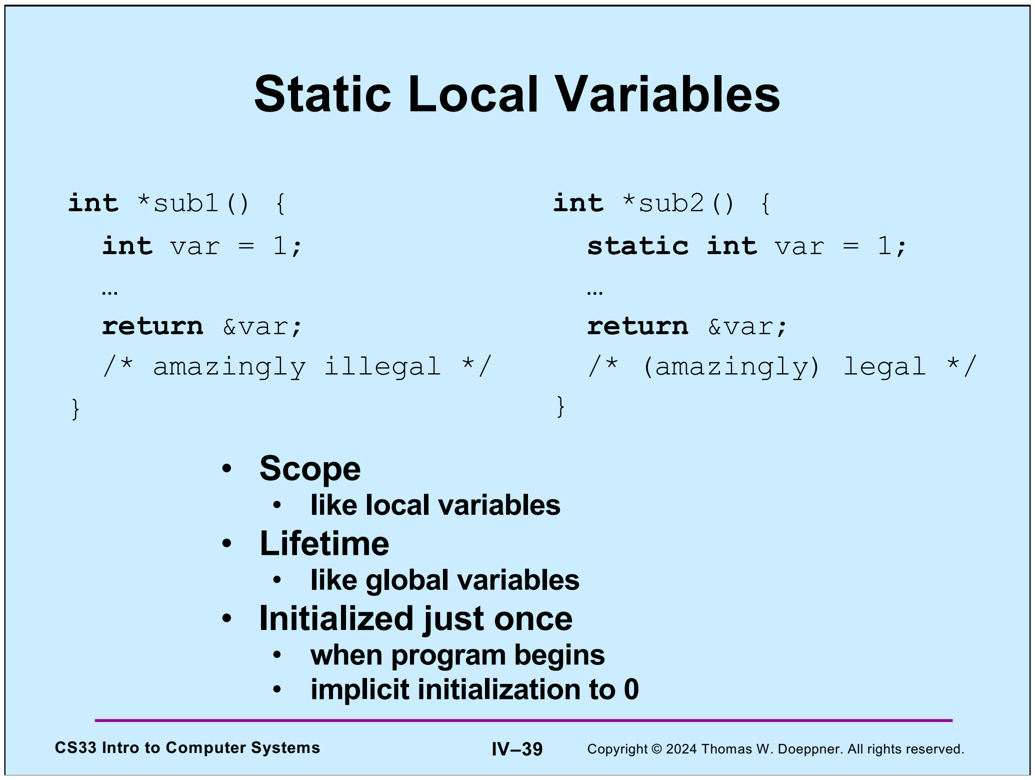
```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Quiz 4

```
void func(int a) {
   int b=2;
   if (a == 1) {
      func(2);
      printf("%d\n", b);
   } else {
      b = a*(b++)*b;
   }
}
int main() {
   func(1);
   return 0;
}
```

- What's printed?
  a) 0
  b) 1
  c) 2
  d) 4



Static Local Variables

```
int *sub1() {
  int var = 1;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var = 1;
  …
  return &var;
  /* (amazingly) legal */
}
```

- **Scope**
  - **like local variables**
- **Lifetime**
  - **like global variables**
- **Initialized just once**
  - **when program begins**
  - **implicit initialization to 0**

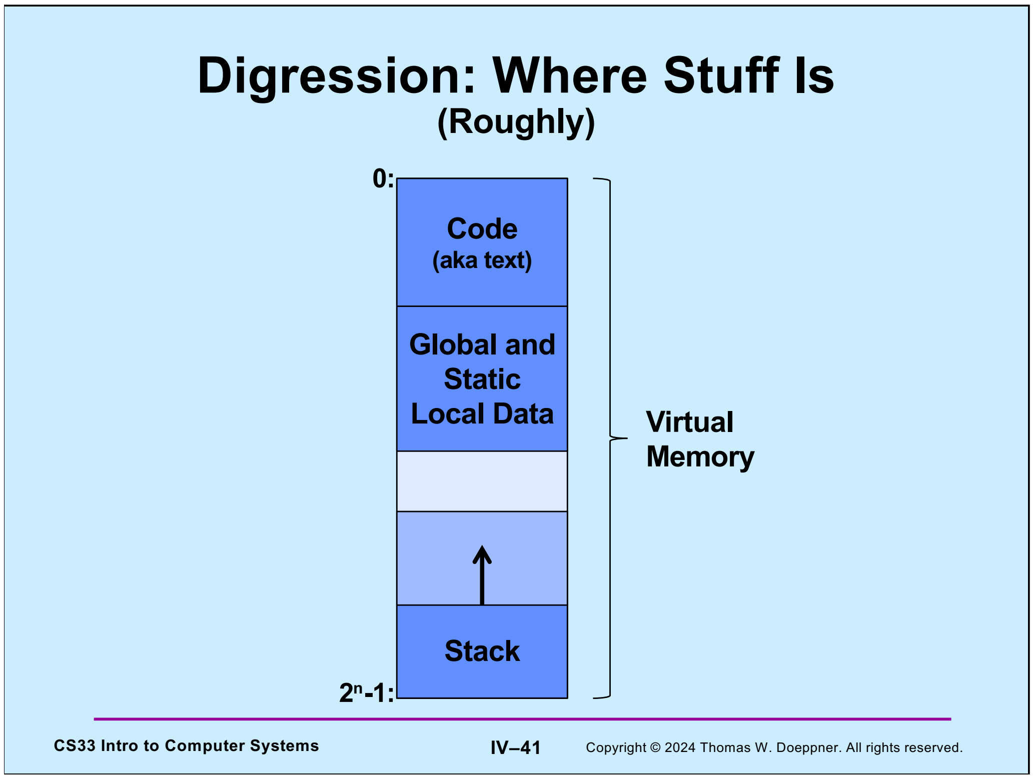
Static local variables have the same scope as other local variables, but their values are retained across calls to the functions they are declared in. Like global variables, uninitialized static local variables are implicitly initialized to zero. Initialization happens just once, when the program starts up. Thus in **sub2**, **var** is set to 1 when the program starts, and not every time **sub2** is called.

Quiz 5

```
int sub() {
  static int svar = 2;
  int lvar = 1;
  svar += lvar;
  lvar++;
  return svar;
}

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

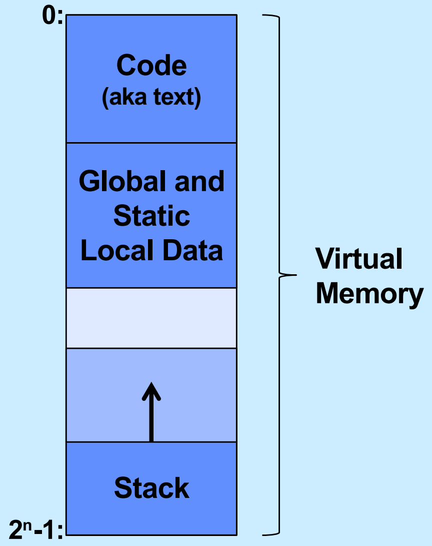
What is printed?

a) 2
b) 3
c) 4
d) 5



Digression: Where Stuff Is

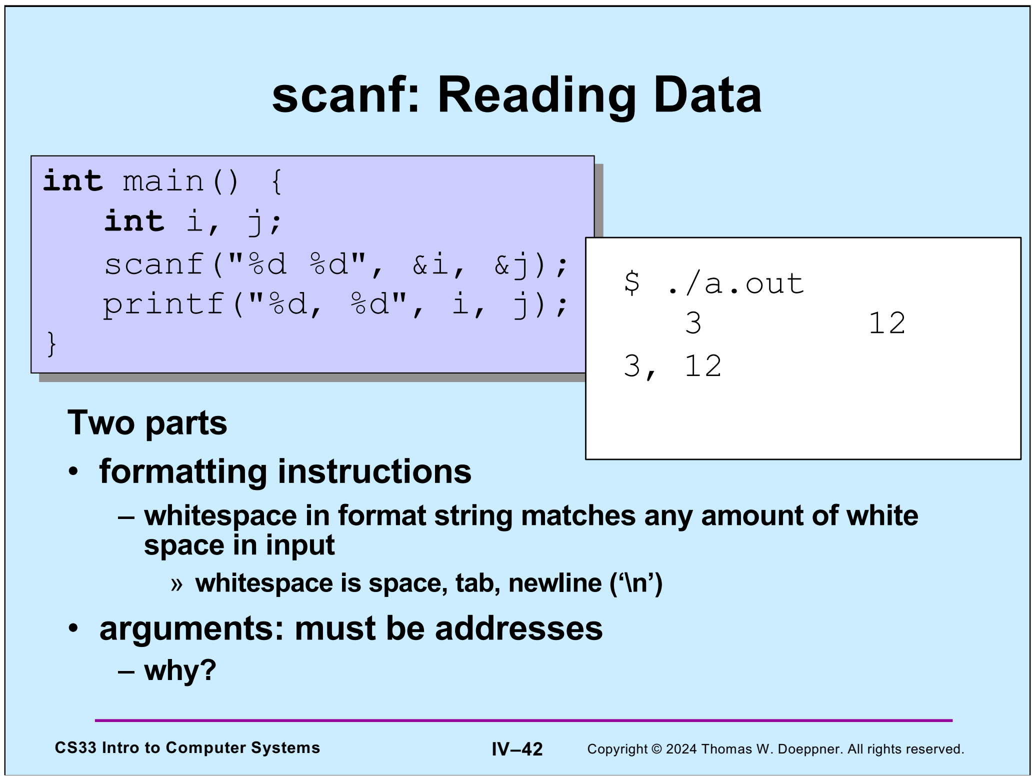
(Roughly)



Let's step back and revisit our concept of virtual memory. All of a program, both code and data, resides in virtual memory. We begin to explore how all of this is organized. This is neither a complete nor a totally accurate picture, but serves to explain what we've seen so far. Executable code (also known, historically, as text) resides at the lower-addressed regions of virtual memory. After it comes a region of memory that contains global and static local data. At the high-addressed end of the address space is memory reserved for the stack. The stack itself starts at the high end of this region and grows (in response to function calls, etc.). If the end of the stack reaches the end of the region of memory reserved for it, a segmentation fault occurs and the program terminates.

This is clearly very rough. As we learn more about how computer systems work, we'll fill in more and more of the details.

Note that here our diagram of memory has lower addresses at the top, higher addresses at the bottom. Soon we'll turn this around and draw it the other way, with higher addresses at the top, lower addresses at the bottom – it generally makes more sense to do it this way.



scanf: Reading Data

```
int main() {
   int i, j;
   scanf("%d %d", &i, &j);
   printf("%d, %d", i, j);
}
```

```
$ ./a.out
   3         12
3, 12
```

Two parts

- **formatting instructions**
  - **whitespace in format string matches any amount of white space in input**
    - » **whitespace is space, tab, newline ('\n')**
- **arguments: must be addresses**
  - **why?**

The function **scanf** is called to read input, doing essentially the reverse of what **printf** does. Its first argument is a format string, like that of **printf**. Its subsequent arguments are pointers to locations where the input should be copied (after format conversion as specified in the format string). Note that we must have pointers for these arguments, not simple values, since arguments are passed by value. (Make sure you understand why this is important!)

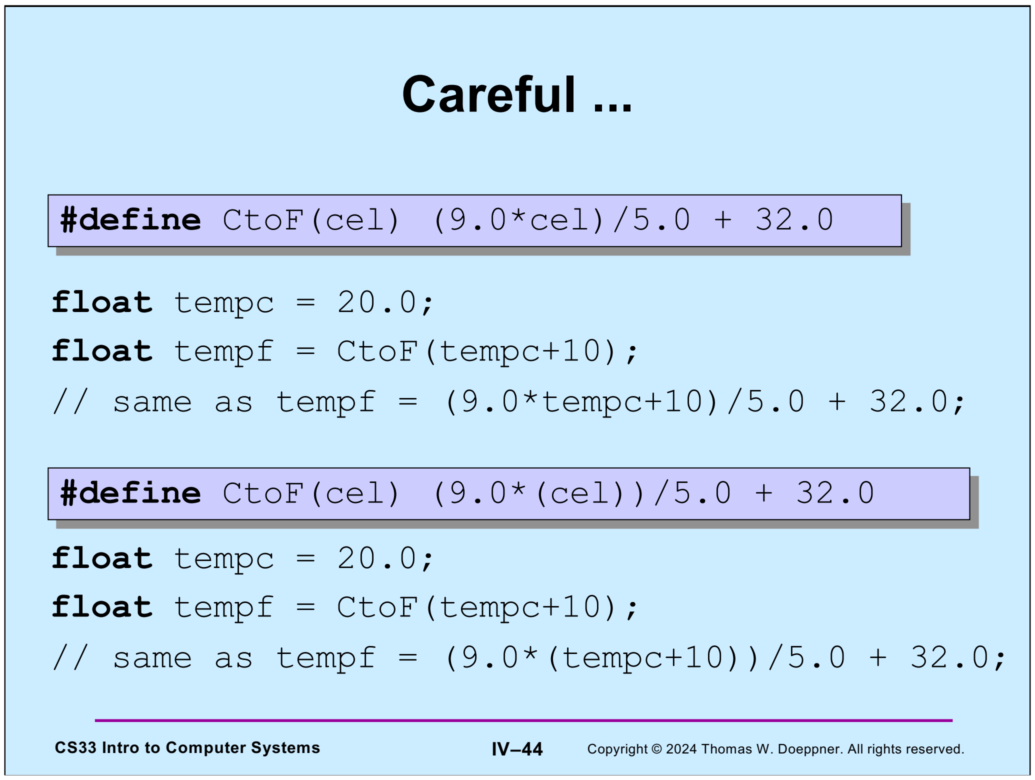
The format conversion done is the reverse of what **printf** does. For example, **printf**, given the **%d** format code, converts the machine representation of an integer into its string representation in decimal notation. **scanf** with the same format code takes the string representation of a number in decimal notation and converts it to the machine representation of an integer.

#define (again)

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0
```

Simple textual substitution:

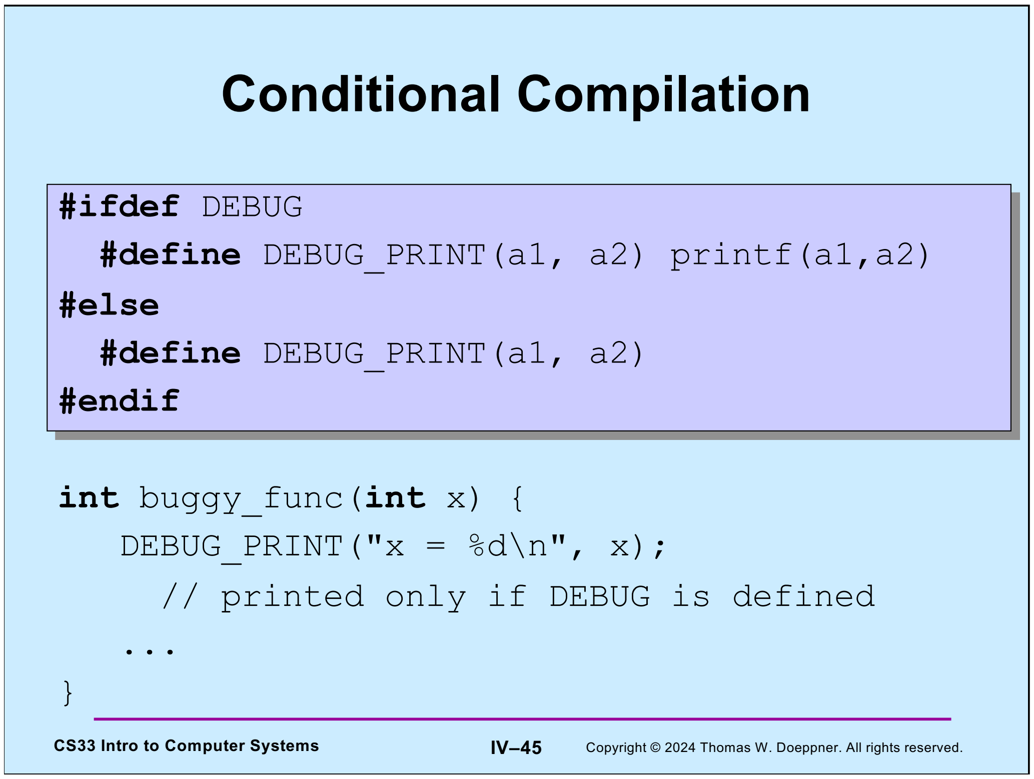
```
float tempc = 20.0;
float tempf = CtoF(tempc);
// same as tempf = (9.0*tempc)/5.0 + 32.0;
```



Careful ...

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0

float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*tempc+10)/5.0 + 32.0;

#define CtoF(cel) (9.0*(cel))/5.0 + 32.0

float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*(tempc+10))/5.0 + 32.0;
```

Be careful with how arguments are used! Note the use of parentheses in the second version.



Conditional Compilation

```
#ifdef DEBUG
  #define DEBUG_PRINT(a1, a2) printf(a1,a2)
#else
  #define DEBUG_PRINT(a1, a2)
#endif
```

```
int buggy_func(int x) {
   DEBUG_PRINT("x = %d\n", x);
     // printed only if DEBUG is defined
   ...
}
```

One can define DEBUG simply by using the statement

#define DEBUG

or by supplying the flag –D=DEBUG to gcc.

Note that in addition to #ifdef (which should be read "if defined"), there's also #ifndef (which should be read "if not defined"). Thus the code in slide could also be written

```
#ifndef DEBUG
  #define DEBUG_PRINT(a1, a2)
#else
  #define DEBUG_PRINT(a1, a2) printf(a1,a2)
#endif
```