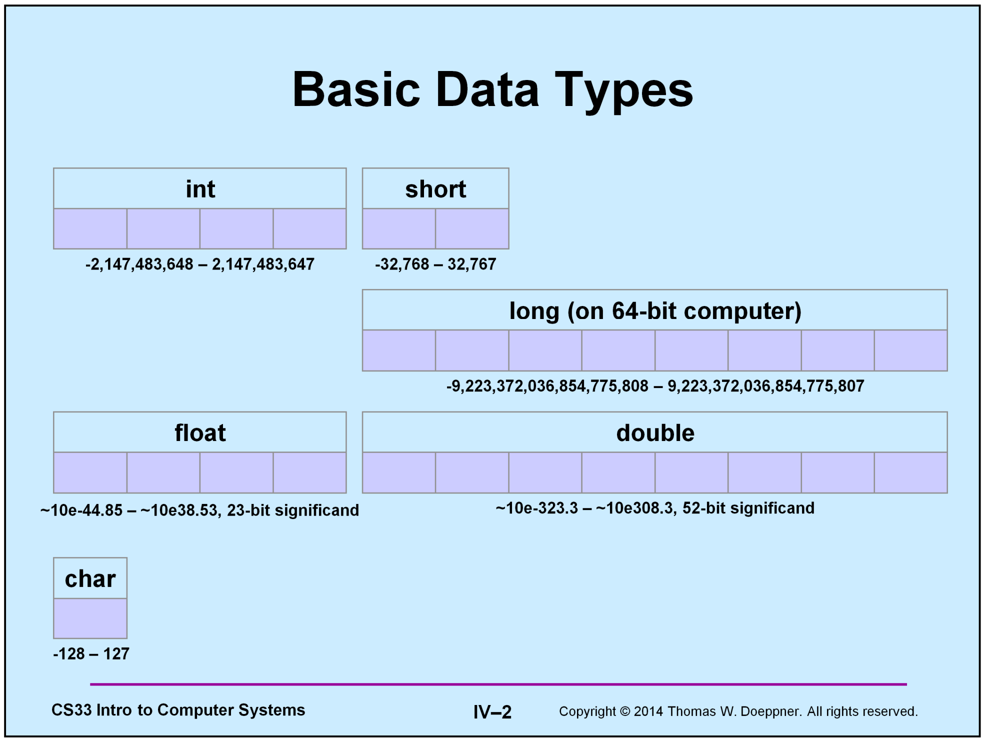
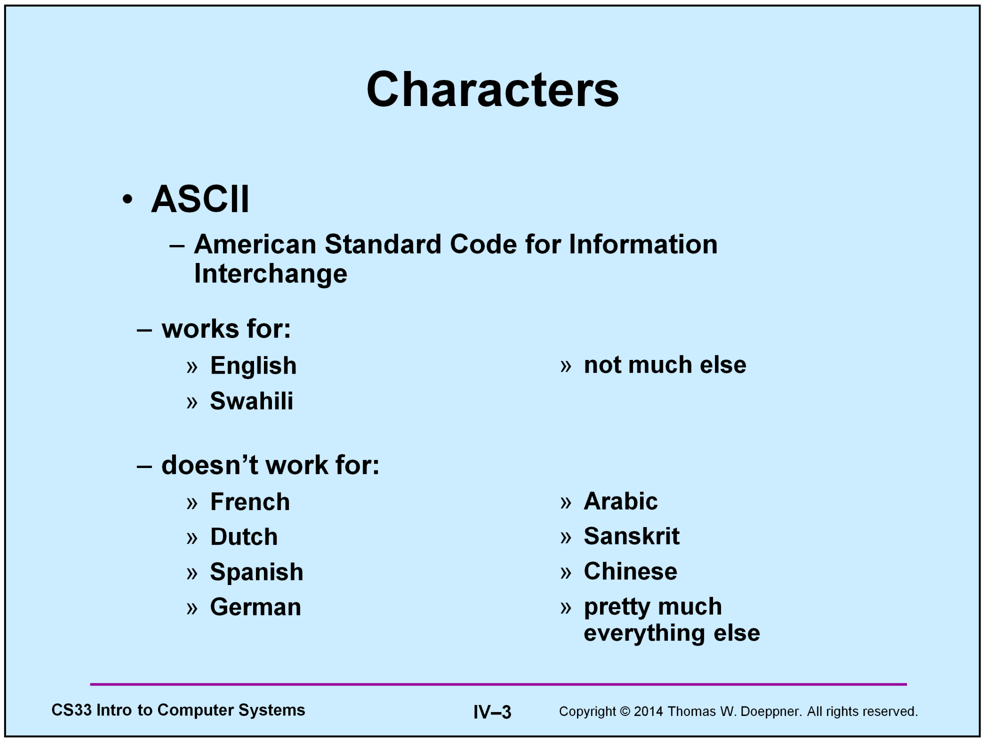
CS 33

Introduction to C
Part 3



Basic Data Types

The floating-point representation is as supported on the Intel X86 architecture. Note that the exponent is base 2, so that the limits given are approximate. We will discuss the representation of the basic data types in much more detail soon.



Characters

- **ASCII**
  - **American Standard Code for Information Interchange**
  - **works for:**
    - » **English**
    - » **Swahili**
    - » **not much else**
  - **doesn't work for:**
    - » **French**
    - » **Dutch**
    - » **Spanish**
    - » **German**
    - » **Arabic**
    - » **Sanskrit**
    - » **Chinese**
    - » **pretty much everything else**

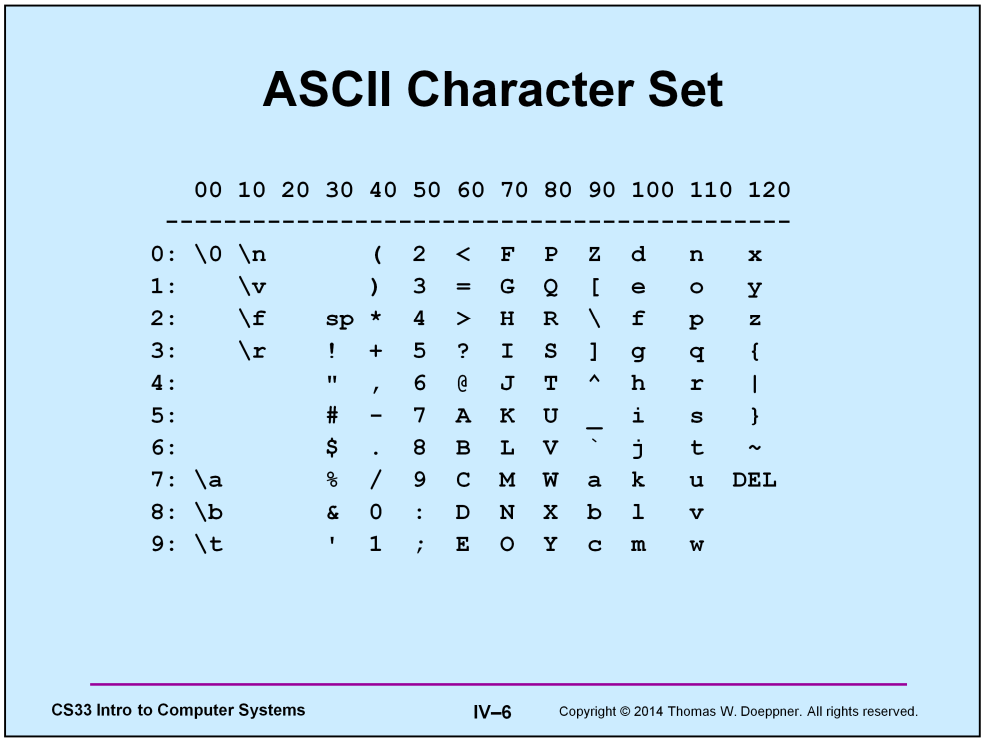
CS33 Intro to Computer Systems IV–3

ASCII is appropriate for English. English-speaking missionaries devised the written form of some languages, such as Swahili, using the English alphabet. What differentiates the English alphabet from those of other European languages is the absence of diacritical marks. ASCII thus has no characters with diacritical marks and works for English, Swahili, and very few other languages.

Who cares!!

You should care …

(but not in this course)



ASCII Character Set

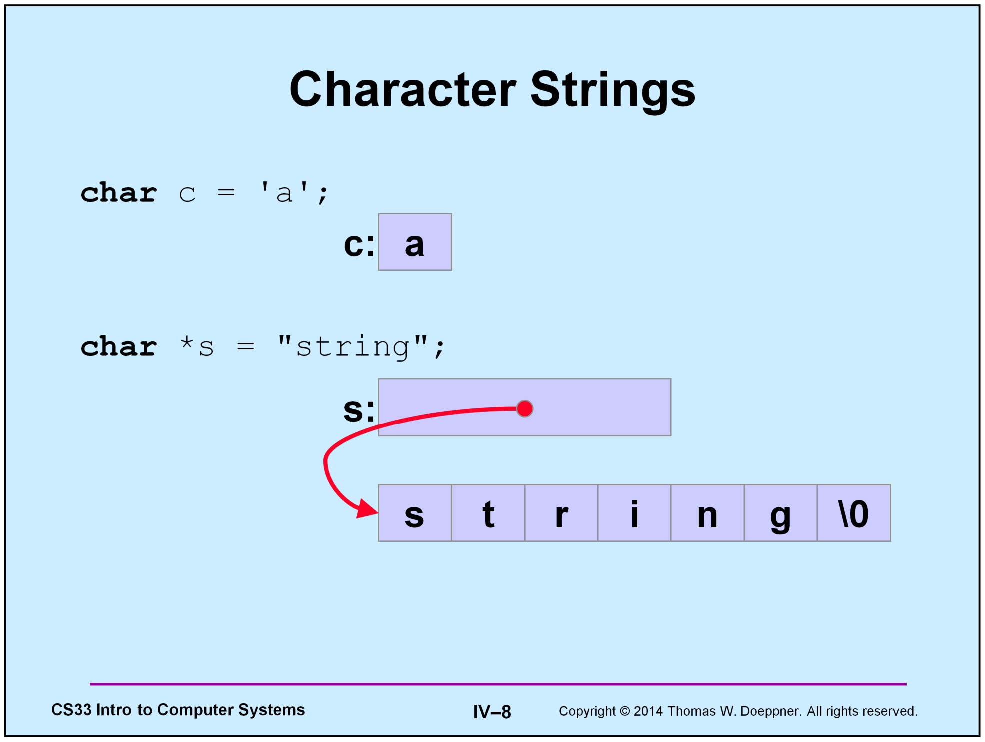
	00	10	20	30	40	50	60	70	80	90	100	110	120
0:	\0	\n			(	2	<	F	P	Z	d	n	x
1:		\v			)	3	=	G	Q	[	e	o	y
2:		\f		sp	*	4	>	H	R	\	f	p	z
3:		\r		!	+	5	?	I	S	]	g	q	{
4:				"	,	6	@	J	T	^	h	r	|
5:				#	-	7	A	K	U	_	i	s	}
6:				$	.	8	B	L	V	`	j	t	~
7:	\a			%	/	9	C	M	W	a	k	u	DEL
8:	\b			&	0	:	D	N	X	b	l	v	
9:	\t			'	1	;	E	O	Y	c	m	w	

ASCII uses only seven bits. Most European languages can be coded with eight bits. Many Asian languages require far more than eight bits.

This table is a bit confusing: it’s presented in column-major order, meaning that it’s laid out in columns. Thus the value of the character ‘0’ is 48, the value of ‘1’ is 49, the value of ‘2’ is 50, the value of ‘3’ is 51, etc.

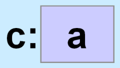
*char*s as Integers

```
char tolower(char c) {
  if (c >= 'A' && c <= 'Z')
      return c + 'a' - 'A';
  else
      return c;
}
```

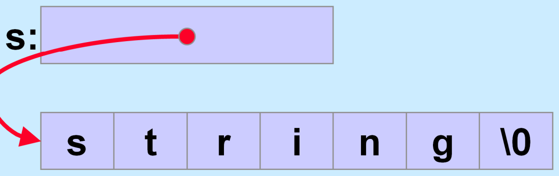


Character Strings

```
char c = 'a';
```



```
char *s = "string";
```



Quiz (1)

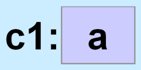
Is there any difference between *c1* and *c2* in the following?

```
char c1 = 'a';
char *c2 = "a";
```

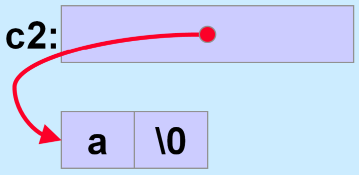
Answer (1)

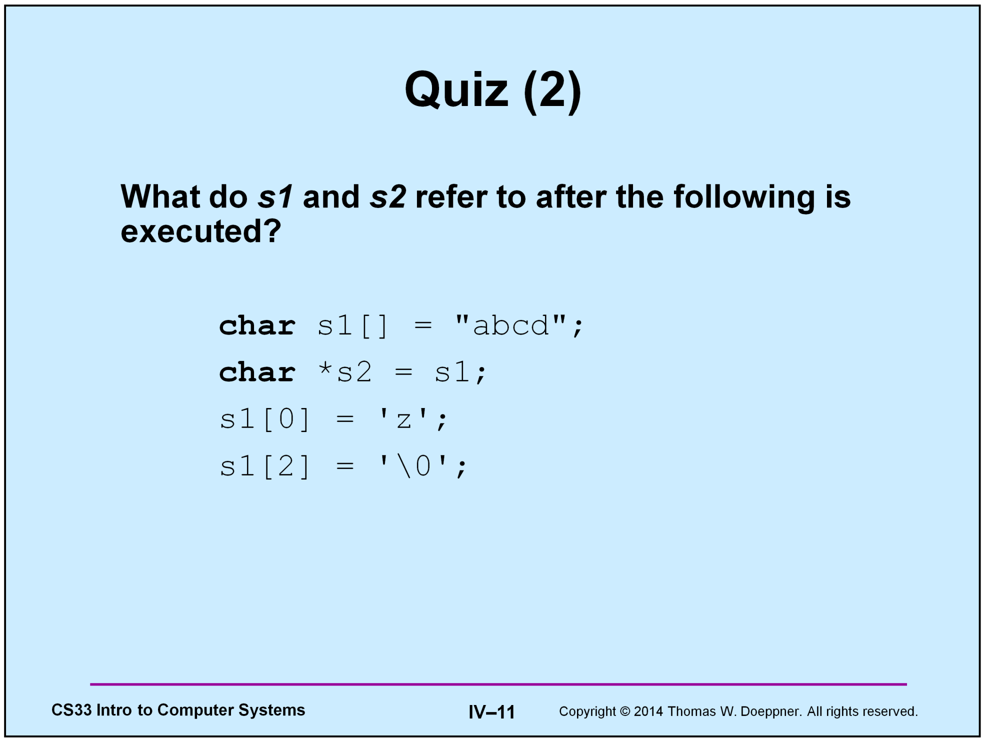
Yes!!

```
char c1 = 'a';
```



```
char *c2 = "a";
```





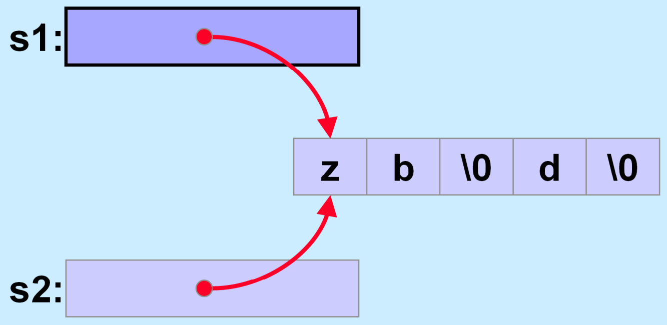
Quiz (2)

What do *s1* and *s2* refer to after the following is executed?

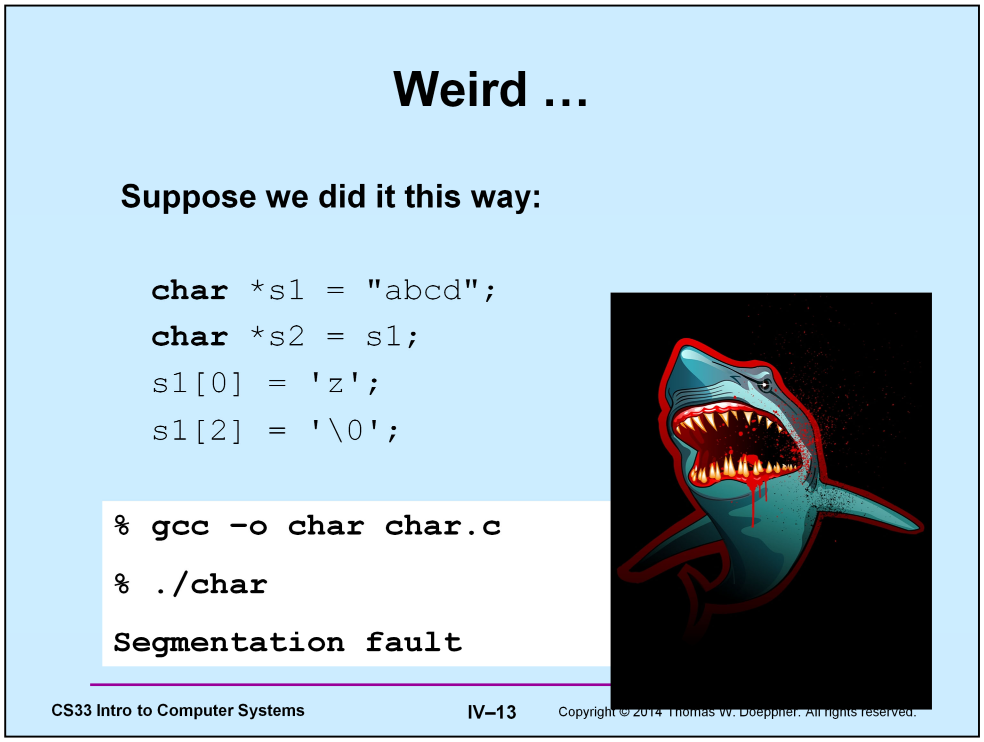
```
char s1[] = "abcd";
char *s2 = s1;
s1[0] = 'z';
s1[2] = '\0';
```

Note that the declaration of s1 results in the allocation of 5 bytes of memory, into which is copied the string “abcd” (including the null at the end).

Answer (2)



Note that if either s1 or s2 is printed (e.g., "printf("%s", s1)), all that will appear is "zb" — this is because the null character terminates the string. Recall that s1 is essentially a constant: its value cannot be changed (it points to the beginning of the array of characters), but what it points to may certainly be changed.



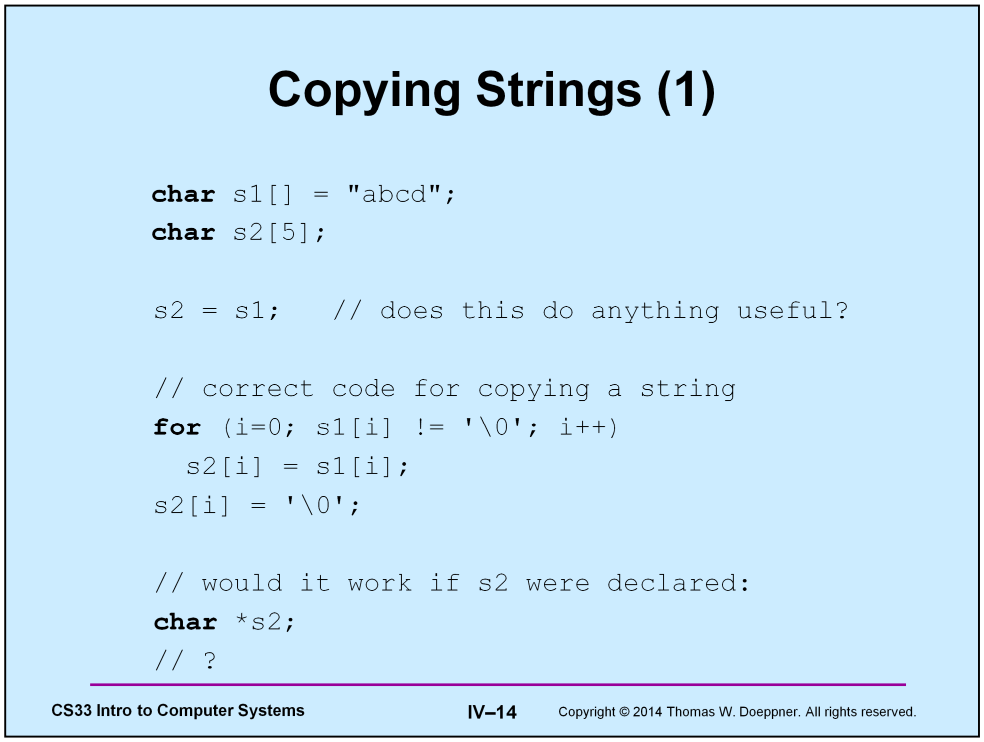
Weird …

Suppose we did it this way:

```
char *s1 = "abcd";
char *s2 = s1;
s1[0] = 'z';
s1[2] = '\0';
```

```
% gcc -o char char.c
% ./char
Segmentation fault
```

String constants are stored in an area of memory that's made read-only, thus any attempt to modify them is doomed. In the example, s1 is a pointer that points to such a read-only area of memory. This is unlike what was done two slides ago, in which the string in read-only memory was copied into read-write memory pointed to by s1.

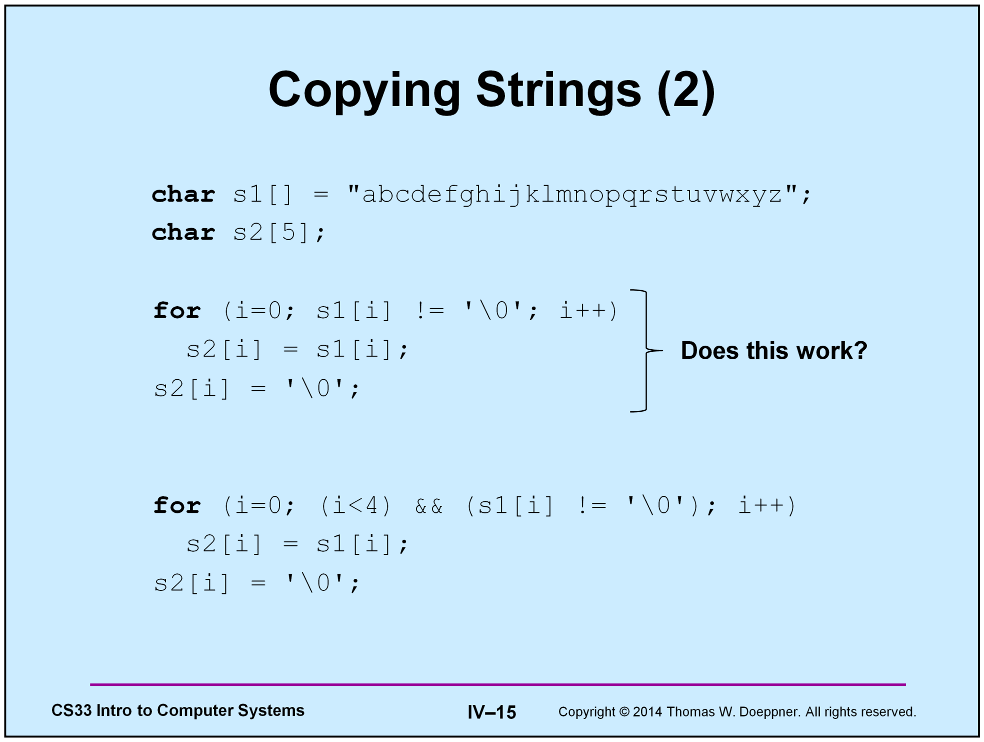


Copying Strings (1)

```
char s1[] = "abcd";
char s2[5];

s2 = s1;   // does this do anything useful?

// correct code for copying a string
for (i=0; s1[i] != '\0'; i++)
  s2[i] = s1[i];
s2[i] = '\0';

// would it work if s2 were declared:
char *s2;
// ?
```

The answer to the first question is no: the assignment will be considered a syntax error, since the value of s2 is constant. What we really want to do is to copy the array pointed to by s1 into the array pointed to by s2.

It would not work if s2 were declared simply as a pointer. The original s2, declared as an array, has 5 bytes of memory associated with it, which is sufficient space to hold the string that’s being copied. Thus the original s2 points to an area of memory suitable for holding a copy of the string. The second s2, being declared as simply a pointer and not given an initial value, points to an unknown location in memory. Copying the string into what s2 points to will probably lead to disaster.



Copying Strings (2)

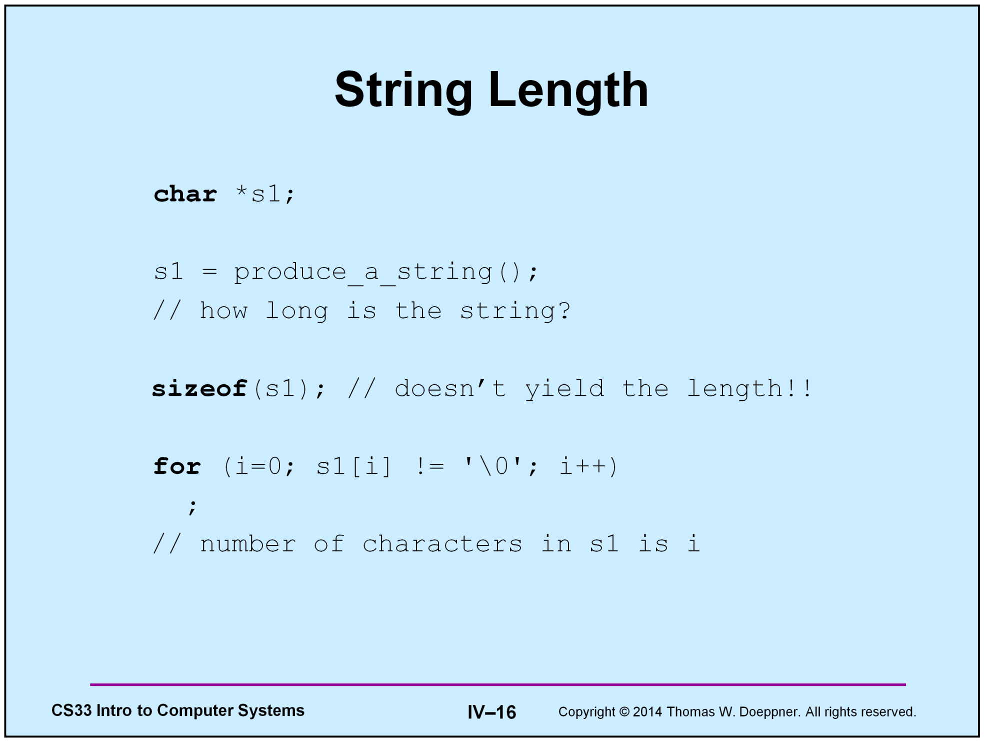
```
char s1[] = "abcdefghijklmnopqrstuvwxyz";
char s2[5];

for (i=0; s1[i] != '\0'; i++)
  s2[i] = s1[i];
s2[i] = '\0';
```

Does this work?

```
for (i=0; (i<4) && (s1[i] != '\0'); i++)
  s2[i] = s1[i];
s2[i] = '\0';
```

The answer, of course, is that it doesn’t work, since there’s not enough room in the array referred to by s2 to hold the contents of the array referred to by s1. Note that “&&” is the AND operator in C.

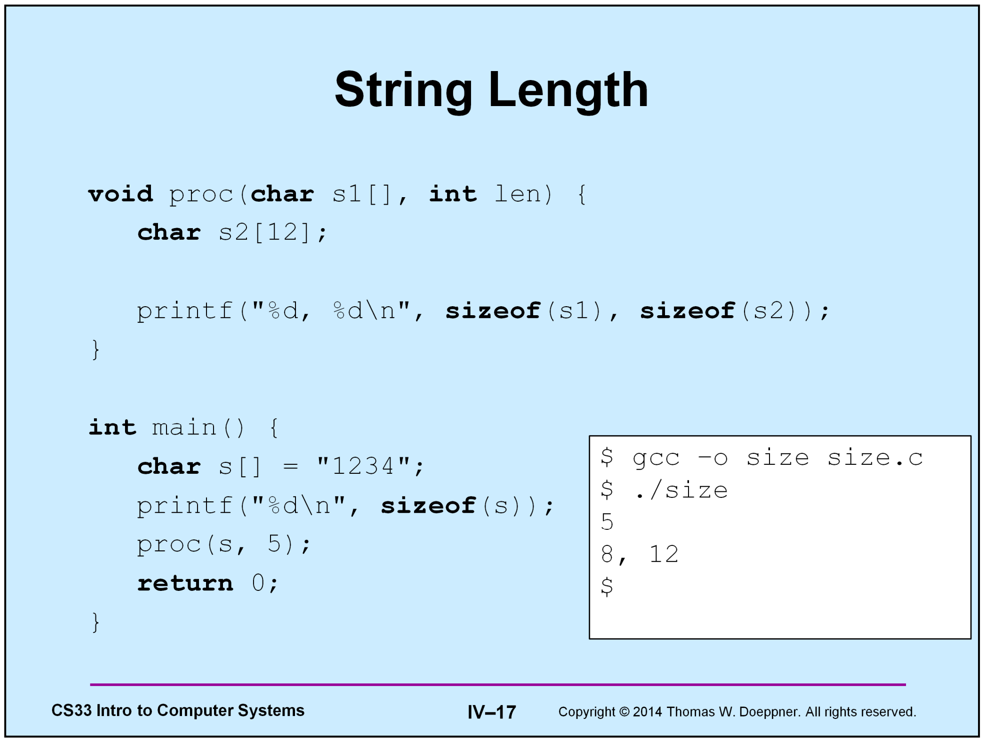
The approach does work (in the sense that it does nothing wrong) because it copies no more than 4 bytes plus a null byte into s2, whose size is 5 bytes.



String Length

```
char *s1;

s1 = produce_a_string();
// how long is the string?

sizeof(s1); // doesn't yield the length!!

for (i=0; s1[i] != '\0'; i++)
  ;
// number of characters in s1 is i
```

sizeof(s1) yields the size of the variable s1, which, on a 64-bit architecture, is 8 bytes.



String Length

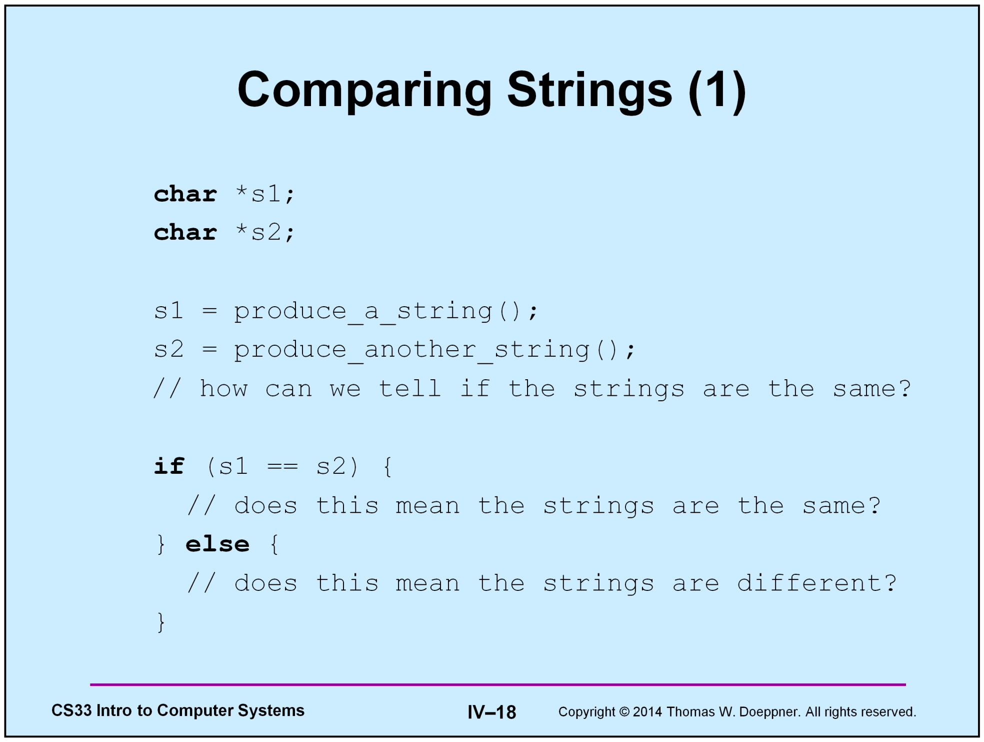
```
void proc(char s1[], int len) {
   char s2[12];

   printf("%d, %d\n", sizeof(s1), sizeof(s2));
}

int main() {
   char s[] = "1234";
   printf("%d\n", sizeof(s));
   proc(s, 5);
   return 0;
}
```

```
$ gcc -o size size.c
$ ./size
5
8, 12
$
```

sizeof(s) is 5 because 5 bytes of storage were allocated to hold its value (including the null).

sizeof(s1) is 8 because it's pointer to a char, and pointers occupy 8 bytes.

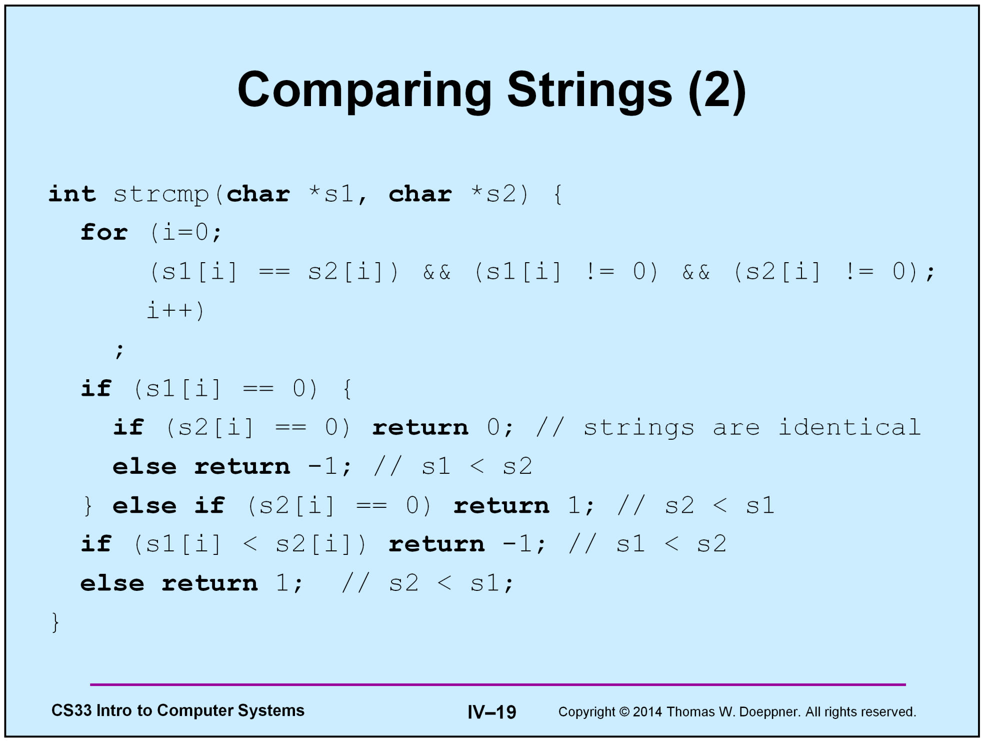
sizeof(s2) is 12 because 12 bytes of storage were allocated for it.



Comparing Strings (1)

```
char *s1;
char *s2;

s1 = produce_a_string();
s2 = produce_another_string();
// how can we tell if the strings are the same?

if (s1 == s2) {
  // does this mean the strings are the same?
} else {
  // does this mean the strings are different?
}
```

Note that comparing s1 and s2 simply compares their numeric values as pointers, it doesn't take into account what they point to.



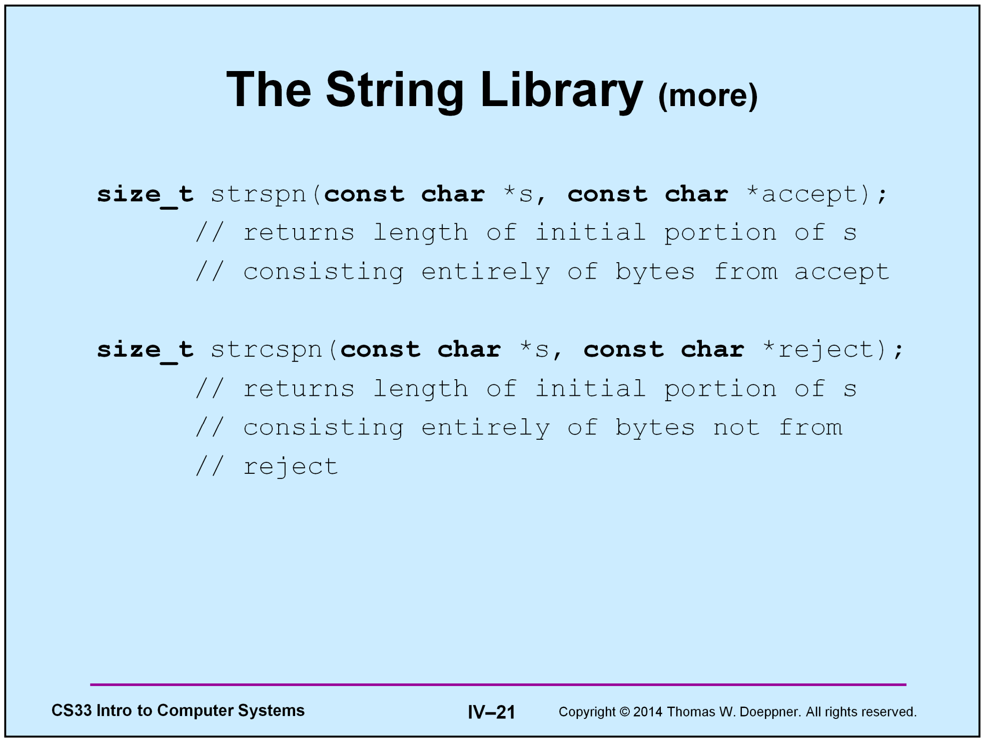
Comparing Strings (2)

```
int strcmp(char *s1, char *s2) {
  for (i=0;
      (s1[i] == s2[i]) && (s1[i] != 0) && (s2[i] != 0);
      i++)
    ;
  if (s1[i] == 0) {
    if (s2[i] == 0) return 0; // strings are identical
    else return -1; // s1 < s2
  } else if (s2[i] == 0) return 1; // s2 < s1
  if (s1[i] < s2[i]) return -1; // s1 < s2
  else return 1;  // s2 < s1;
}
```

The for loop finds the first position at which the two strings differ. The rest of the code then determines whether the two strings are identical (if so, they must be of the same length), and if not, it determines which is less than the other. The procedure returns -1 if s1 is precedes s2, 0 if they are identical, and 1 if s2 precedes s1.

The String Library

```
#include <string.h>

char *strcpy(char *dest, char *src);
  // copy src to dest, returns ptr to dest
char *strncpy(char *dest, char *src, int n);
  // copy at most n bytes from src to dest
int strlen(char *s);
  // return the length of s (not counting the null)
int strcmp(char *s1, char *s2);
  // return -1, 0, or 1 depending on whether s1 is
  // less than, the same as, or greater than s2
int strncmp(char *s1, char *s2, int n);
  // do the same, but for at most n bytes
```



The String Library (more)

```
size_t strspn(const char *s, const char *accept);
      // returns length of initial portion of s
      // consisting entirely of bytes from accept

size_t strcspn(const char *s, const char *reject);
      // returns length of initial portion of s
      // consisting entirely of bytes not from
      // reject
```

These will be useful in upcoming assignments.

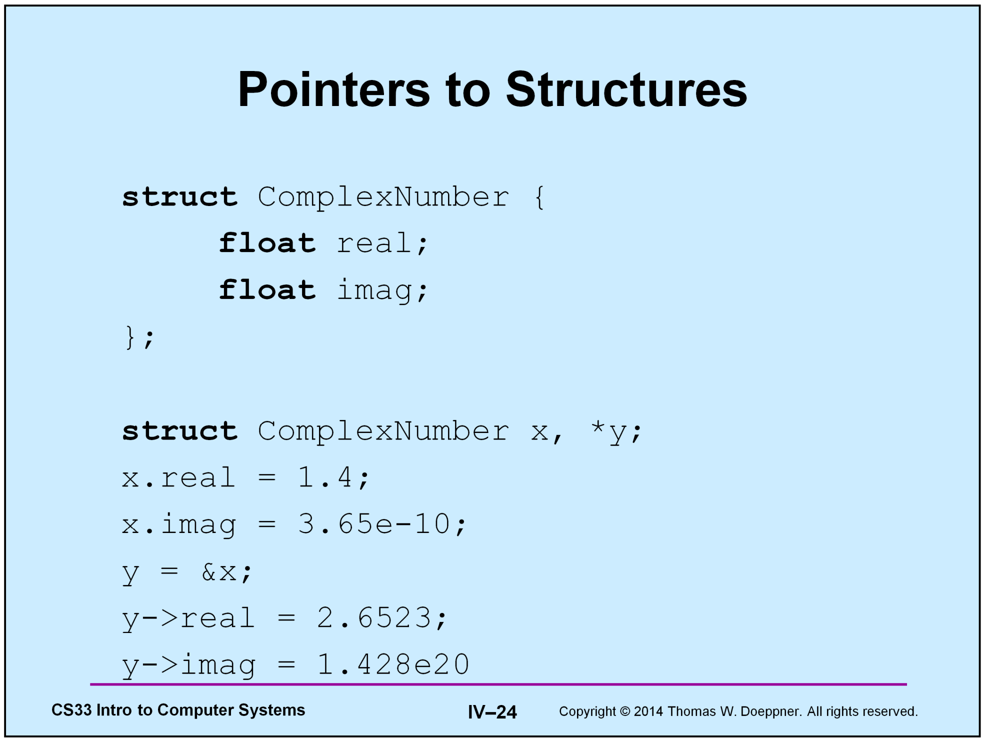
What's Wrong?

```
#include <stdio.h>
#include <string.h>

int main() {
  char s1[] = "Hello World!\n";
  char *s2;
  strcpy(s2, s1);
  printf("%s", s2);
  return 0;
}
```

The pointer s2 has not been given a value, thus we are copying s1 into an unknown location in memory.

Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x;
x.real = 1.4;
x.imag = 3.65e-10;
```



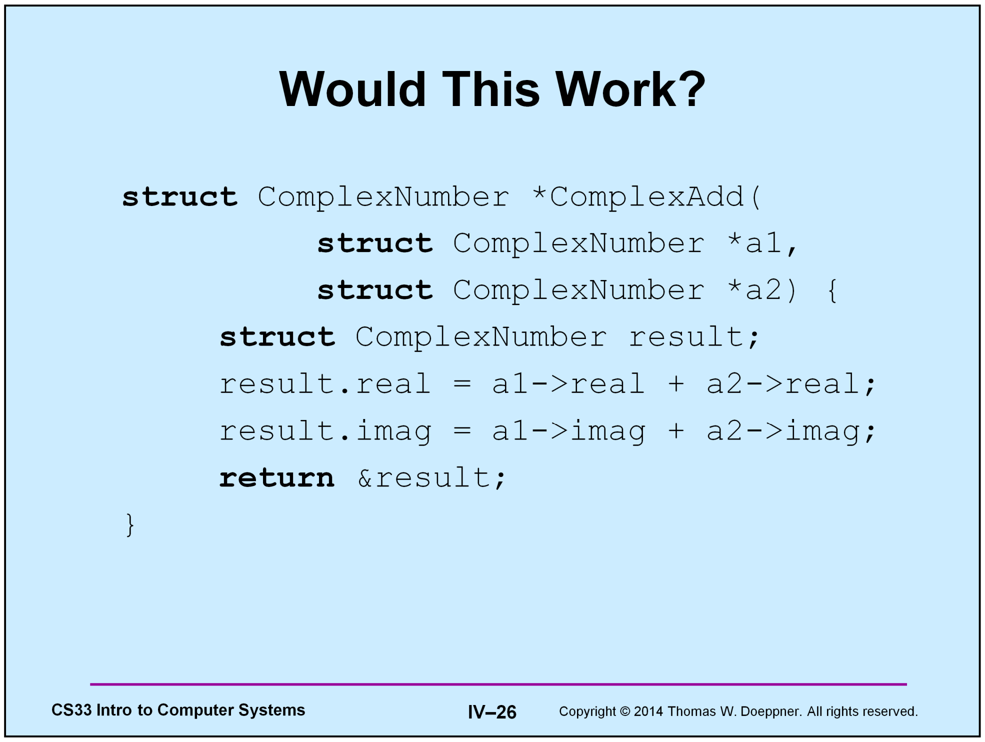
Pointers to Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x, *y;
x.real = 1.4;
x.imag = 3.65e-10;
y = &x;
y->real = 2.6523;
y->imag = 1.428e20
```

Note that when we refer to members of a structure via a pointer, we use the "->" notation rather than the "." notation.

*struct*s and Functions

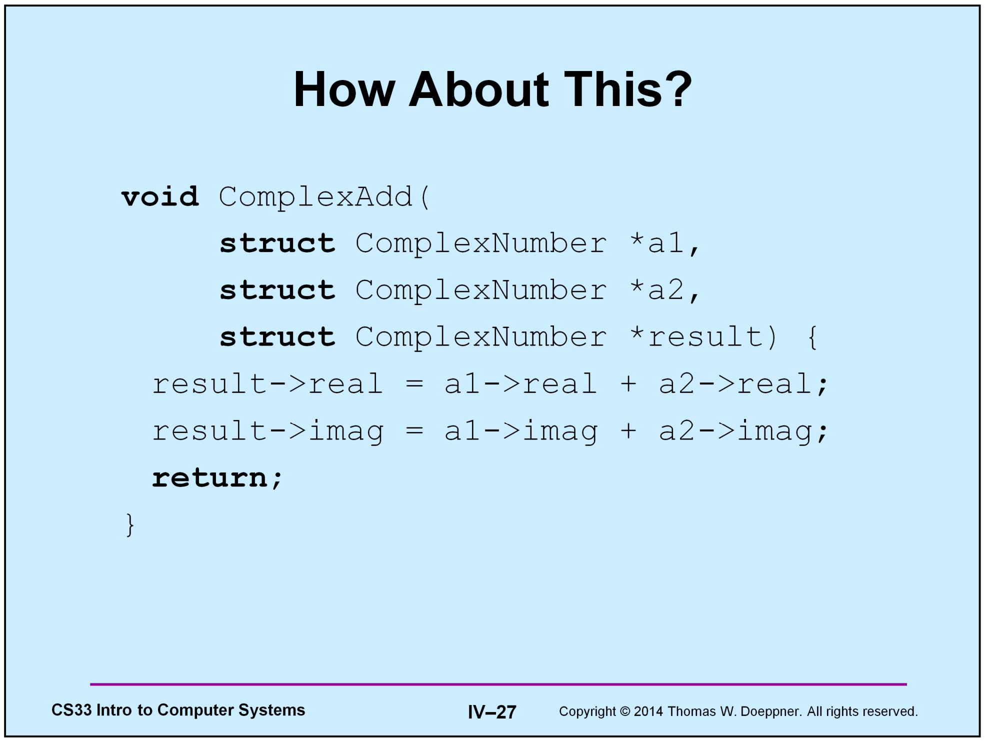
```
struct ComplexNumber ComplexAdd(
          struct ComplexNumber a1,
          struct ComplexNumber a2) {
     struct ComplexNumber result;
     result.real = a1.real + a2.real;
     result.imag = a1.imag + a2.imag;
     return result;
}
```



Would This Work?

```
struct ComplexNumber *ComplexAdd(
          struct ComplexNumber *a1,
          struct ComplexNumber *a2) {
     struct ComplexNumber result;
     result.real = a1->real + a2->real;
     result.imag = a1->imag + a2->imag;
     return &result;
}
```

This doesn't work, since it returns a pointer to result, which would not be in scope once the procedure has returned. Thus the returned pointer would point to an area of memory with undefined contents.



How About This?

```
void ComplexAdd(
     struct ComplexNumber *a1,
     struct ComplexNumber *a2,
     struct ComplexNumber *result) {
  result->real = a1->real + a2->real;
  result->imag = a1->imag + a2->imag;
  return;
}
```

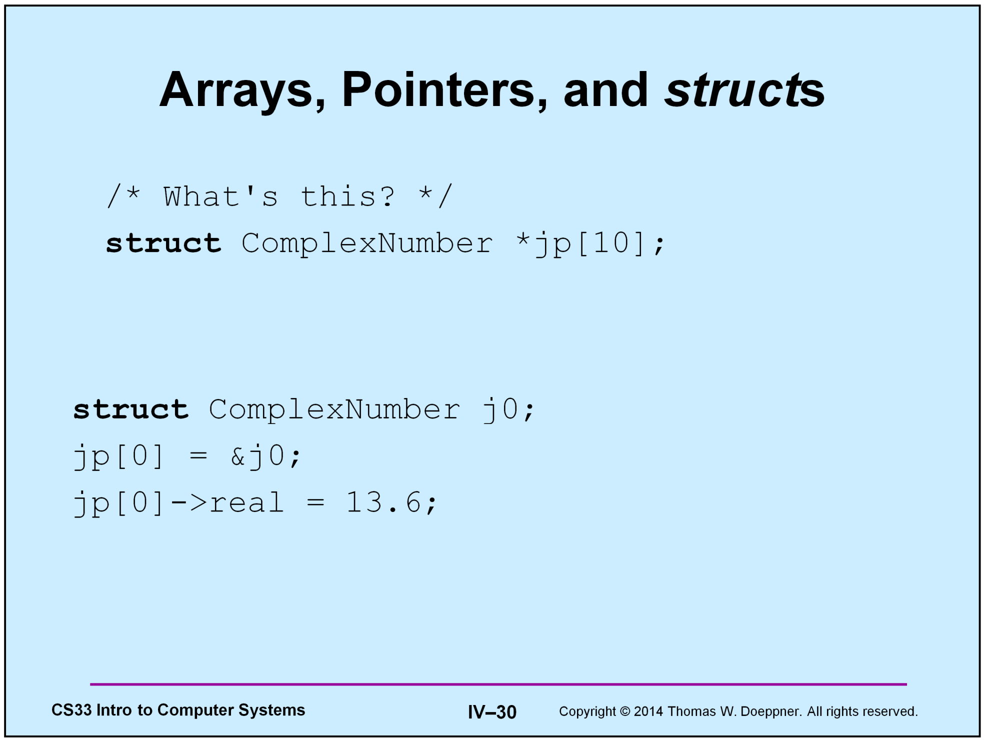
This works fine: the caller provides the location to hold the result.

Using It …

```
struct ComplexNumber j1 = {3.6, 2.125};
struct ComplexNumber j2 = {4.32, 3.1416};
struct ComplexNumber sum;

ComplexAdd(&j1, &j2, &sum);
```

Arrays of *struct*s

```
struct ComplexNumber j[10];
j[0].real = 8.127649;
j[0].imag = 1.76e18;
```



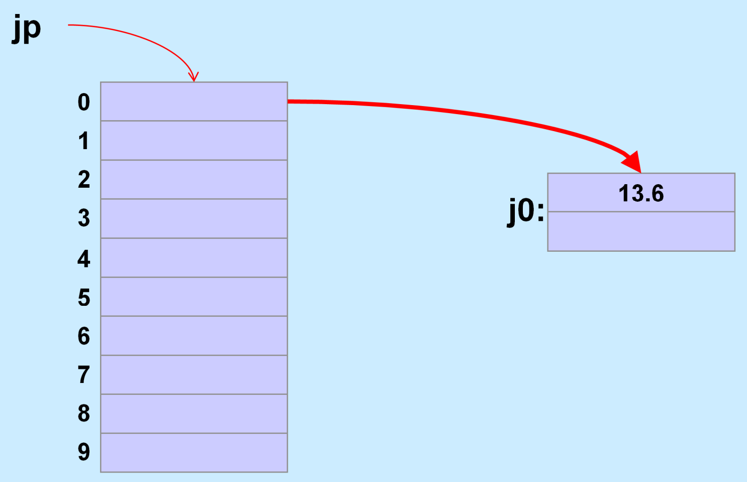
Arrays, Pointers, and *struct*s

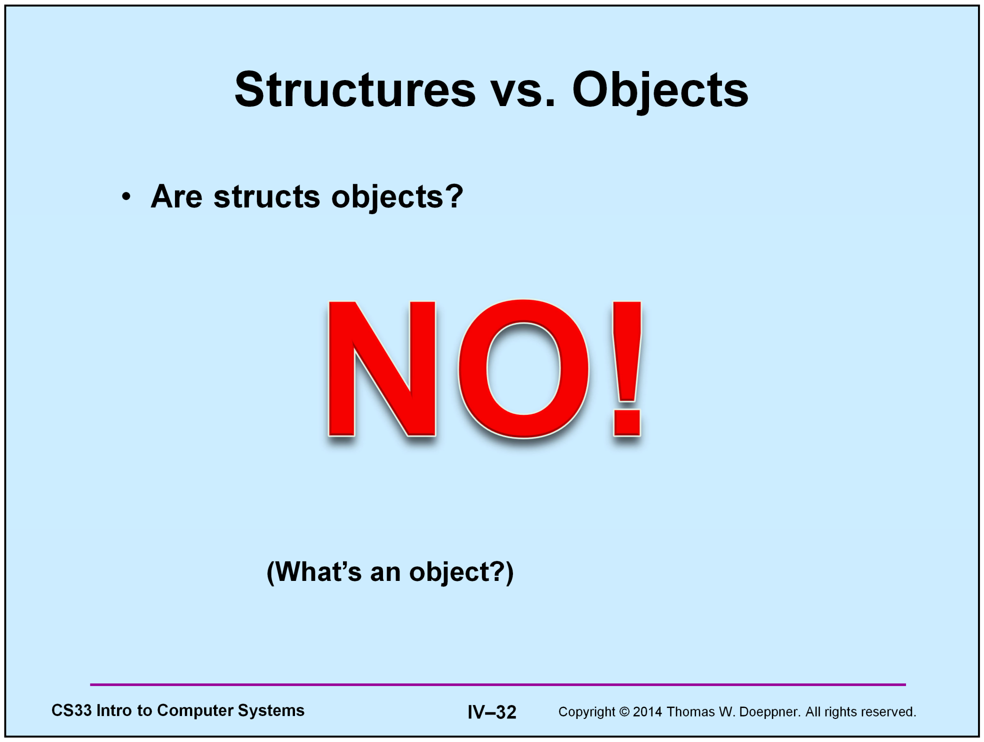
```
/* What's this? */
struct ComplexNumber *jp[10];
```

```
struct ComplexNumber j0;
jp[0] = &j0;
jp[0]->real = 13.6;
```

Subscripting (i.e., the “[]” operator) has a higher precedence than the “*” operator. Thus jp is an array of pointers to *struct ComplexNumber*s, rather than a pointer to an array of *struct ComplexNumber*s.

Memory View





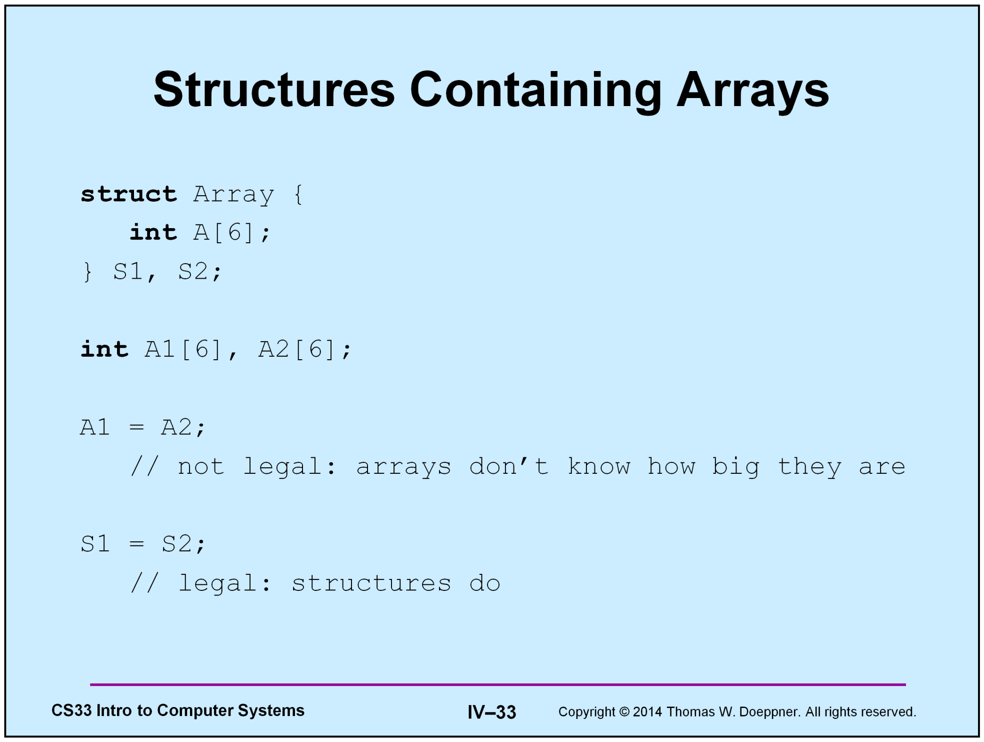
Structures vs. Objects

- Are structs objects?

NO!

(What’s an object?)

```
for (;;)
  printf(“C does not have objects!\n”);
```



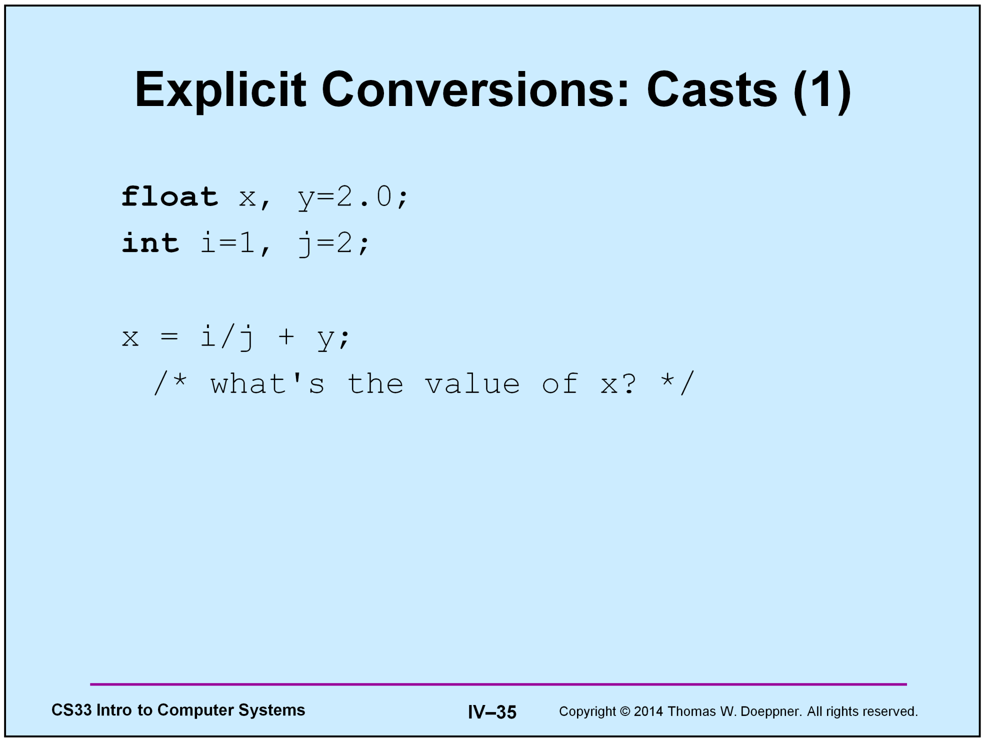
Structures Containing Arrays

```
struct Array {
   int A[6];
} S1, S2;

int A1[6], A2[6];

A1 = A2;
   // not legal: arrays don’t know how big they are

S1 = S2;
   // legal: structures do
```

This seems pretty weird at first glance. But keep in mind that the name of an array refers to the address its first element, and does not represent the entire array. But the name of a structure refers to the entire structure.

Numeric Conversions

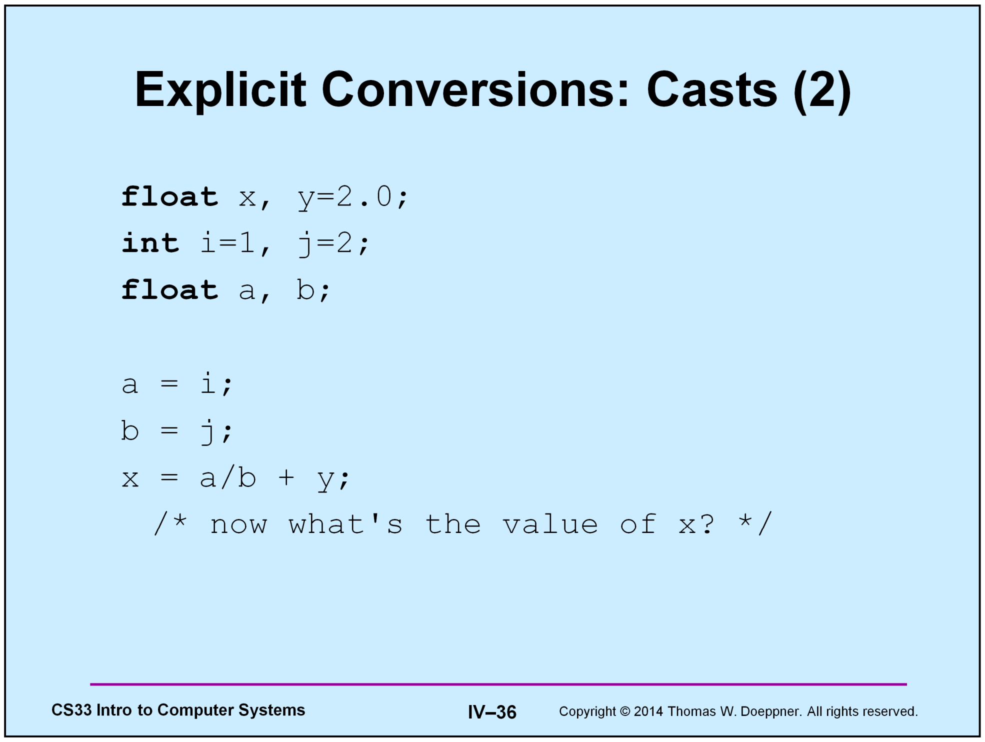
```
short a;
int b;
float c;

b = a;    /* always works */
a = b;    /* sometimes works */
c = b;    /* sort of works */
b = c;    /* sometimes works */
```



Explicit Conversions: Casts (1)

```
float x, y=2.0;
int i=1, j=2;

x = i/j + y;
  /* what's the value of x? */
```

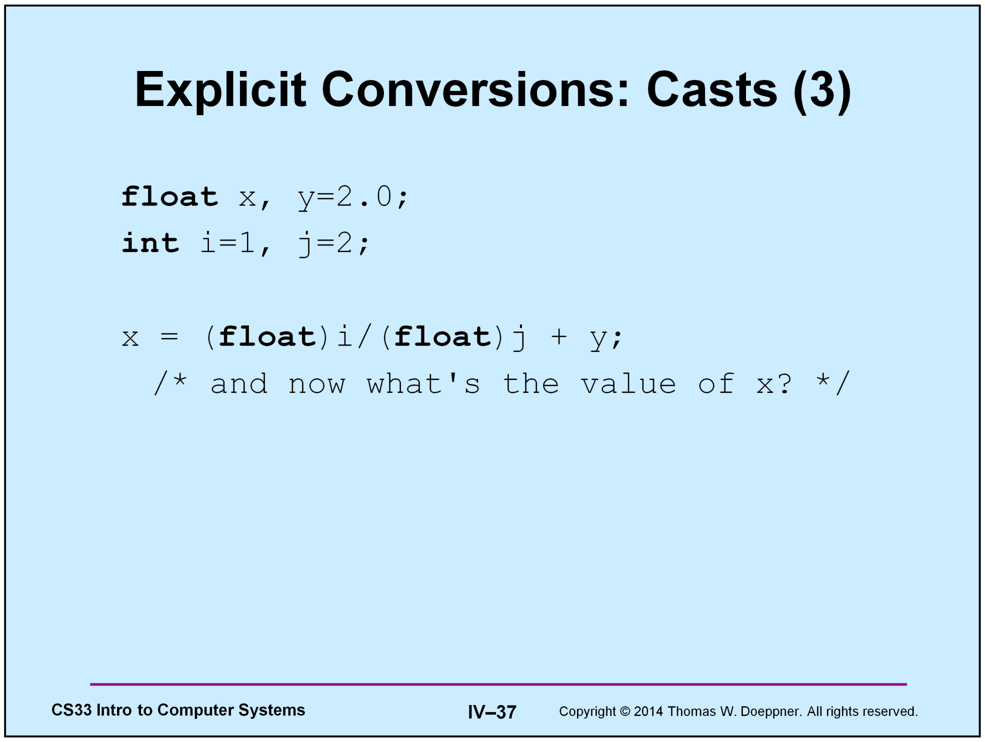
x's value will be 2, since the result of the (integer) division of i by j will be 0.



Explicit Conversions: Casts (2)

```
float x, y=2.0;
int i=1, j=2;
float a, b;

a = i;
b = j;
x = a/b + y;
  /* now what's the value of x? */
```

Here the values of i and j are converted to float before being assigned to a and b, thus the value assigned to x is 2.5.



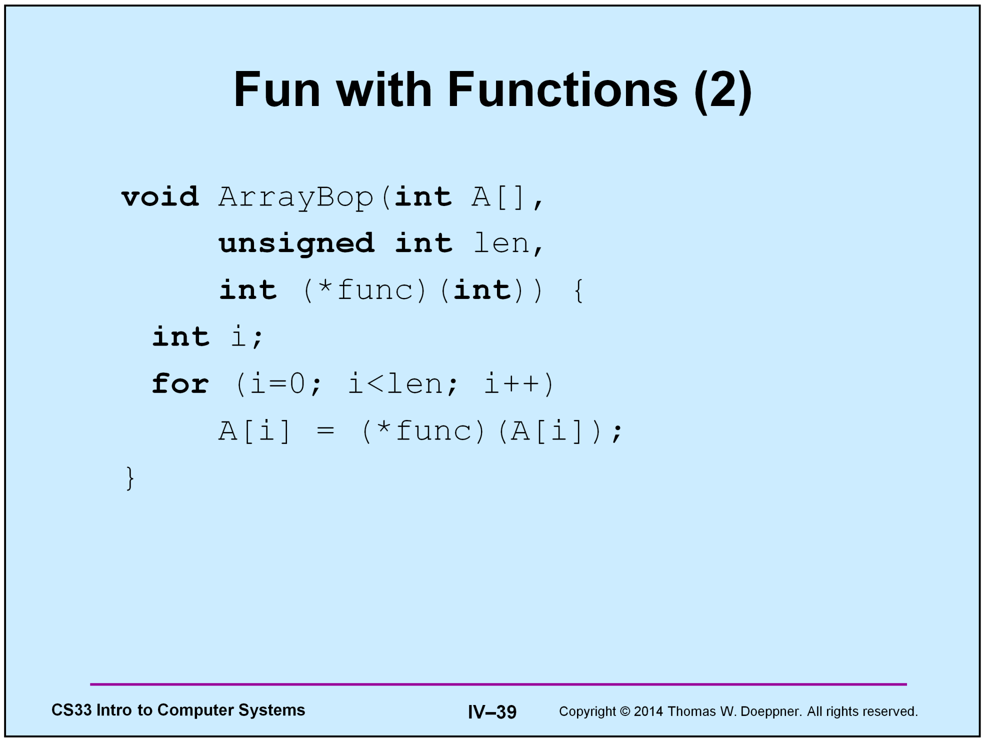
Explicit Conversions: Casts (3)

```
float x, y=2.0;
int i=1, j=2;

x = (float)i/(float)j + y;
  /* and now what's the value of x? */
```

Here we do the int-to-float conversion explicitly; x’s value will be 2.5.

Fun with Functions (1)

```
void ArrayDouble(int A[], int len) {
  int i;
  for (i=0; i<len; i++)
      A[i] = 2*A[i];
}
```



Fun with Functions (2)

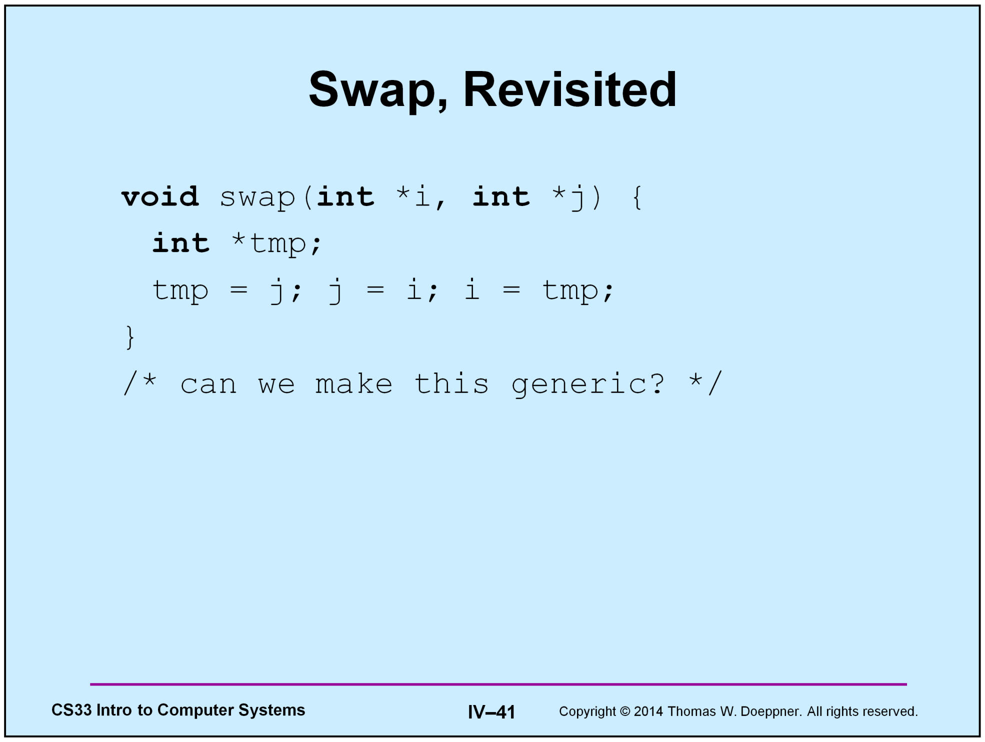
```
void ArrayBop(int A[],
     unsigned int len,
     int (*func)(int)) {
  int i;
  for (i=0; i<len; i++)
     A[i] = (*func)(A[i]);
}
```

Here *func* is declared to be a pointer to a procedure that takes an *int* as an argument and returns an *int*.

What's the difference between a pointer to a procedure and a procedure? A pointer to a procedure is, of course, the address of the procedure. The procedure itself is the code comprising the procedure. Thus, strictly speaking, if *func* is the name assigned to a procedure, *func* really represents the address of the procedure. You might think that we should invoke the procedure by saying "**func*", but it's understood that this is what we mean when we say "*func*". Thus when one calls *ArrayBop*, one supplies the name of the desired procedure as the third argument, without prepending "&".

Fun with Functions (3)

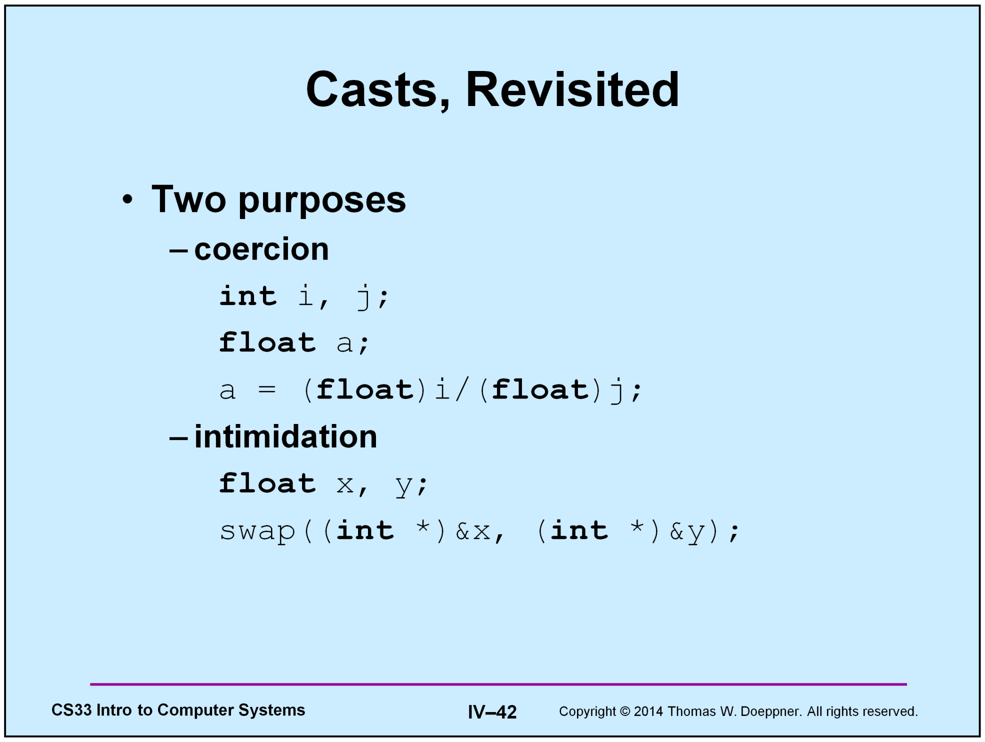
```
int triple(int arg) {
  return 3*arg;
}

int main() {
  int A[20];
  … /* initialize A */
  ArrayBop(A, 20, triple);
  return 0;
}
```



Swap, Revisited

```
void swap(int *i, int *j) {
  int *tmp;
  tmp = j; j = i; i = tmp;
}
/* can we make this generic? */
```

Can we write a version of swap that handles a variety of data types?



Casts, Revisited

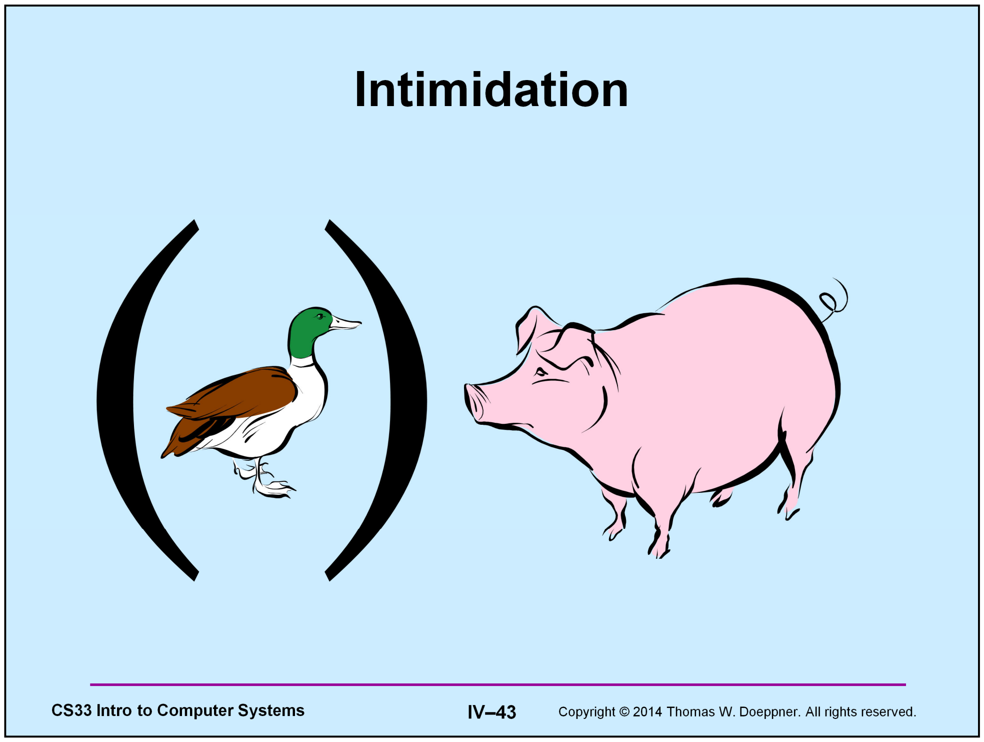
- Two purposes
  - coercion

```
int i, j;
float a;
a = (float)i/(float)j;
```

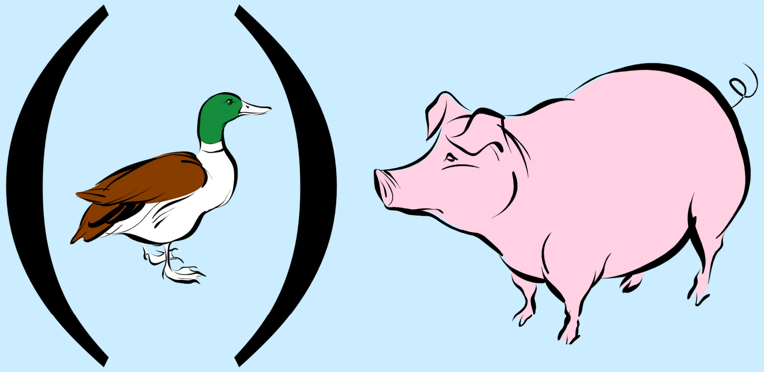
  - intimidation

```
float x, y;
swap((int *)&x, (int *)&y);
```

"Coercion" is a commonly accepted term for one use of casts. "Intimidation" is not. The concept is also known as a "sidecast".



Intimidation



CS33 Intro to Computer Systems IV–43

From “In Search of History” by Theodore H. White:

‘Ch’ing, ch’ing,’ said Chou En-lai, the host — ‘Please, please,’ gesturing with his chopsticks at the pig, inviting the guest to break the crackle first. I flinched, not knowing what to do, but for a moment I hung onto my past. I put my chopsticks down and explained as best I could in Chinese that I was Jewish and that Jews were not allowed to eat any kind of pig meat. The group, all friends of mine by then, sat downcast and silent, for I was their guest, and they had done wrong.

Then Chou himself took over. He lifted his chopsticks once more, repeated, ‘ch’ing, ch’ing,’ pointed the chopsticks at the suckling pig and, grinning, explained: ‘Teddy,’ he said (as I recall it now, for I made no notes that evening), ‘this is China. Look again. See. Look. It looks to you like pig. But in China, this is not pig — this is a duck.’ I burst out laughing, for I could not help it; he laughed, the table laughed, I plunged my chopsticks in, broke the crackle, ate my first mouthful of certified pig, and have eaten of pig ever since, for which I hope my ancestors will forgive me.

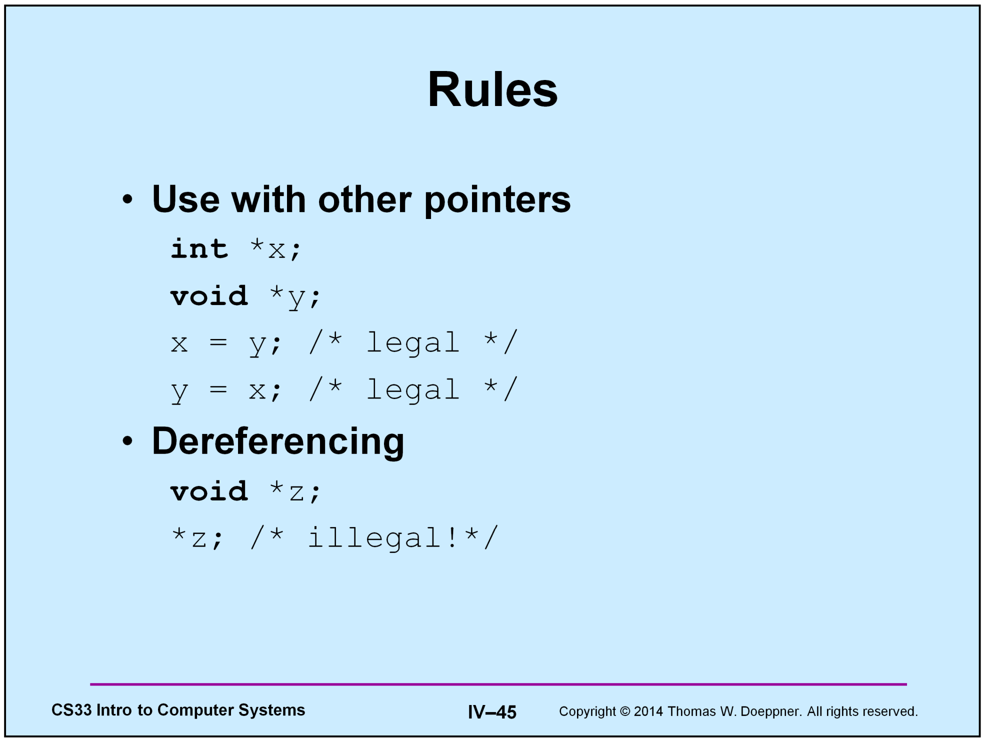
But Chou was that kind of man — he made one believe that pig was duck, because one wanted to believe him, and because he understood the customs of other men and societies and respected them.

Nothing, and More …

- *void* means, literally, nothing:

```
void NotMuch(void) {
   printf("I return nothing\n");
}
```

- What does *void* * mean?
  - it's a pointer to anything you feel like
    - » a generic pointer



Rules

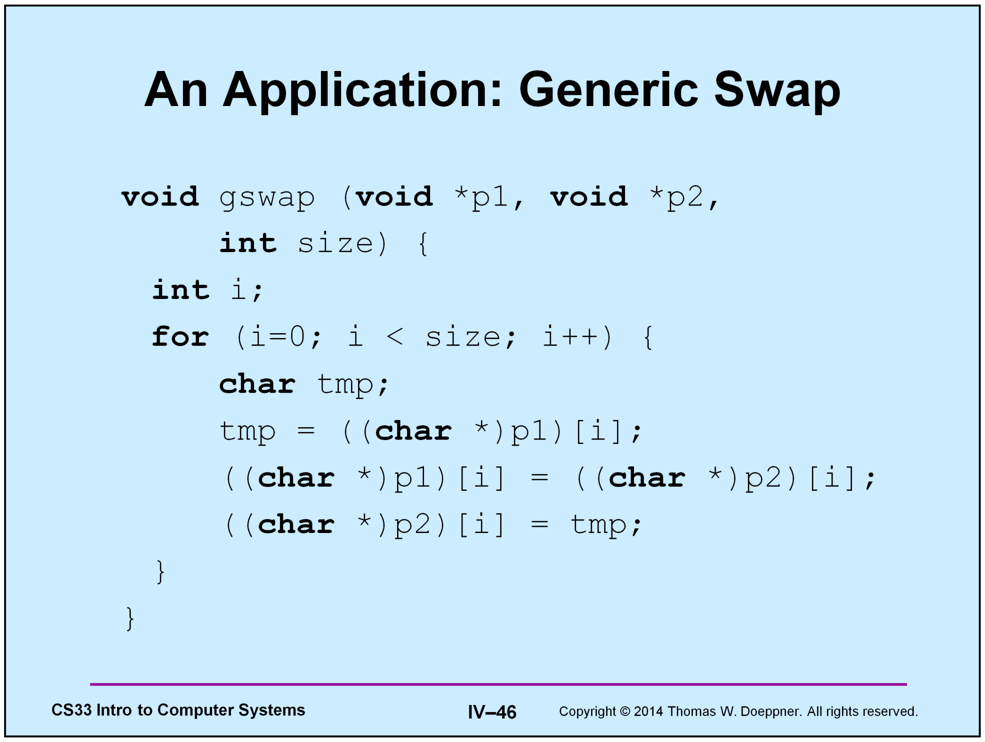
- **Use with other pointers**

```
int *x;
void *y;
x = y; /* legal */
y = x; /* legal */
```

- **Dereferencing**

```
void *z;
*z; /* illegal!*/
```

Dereferencing a pointer must result in a value with a useful type. “void” is not a useful type.



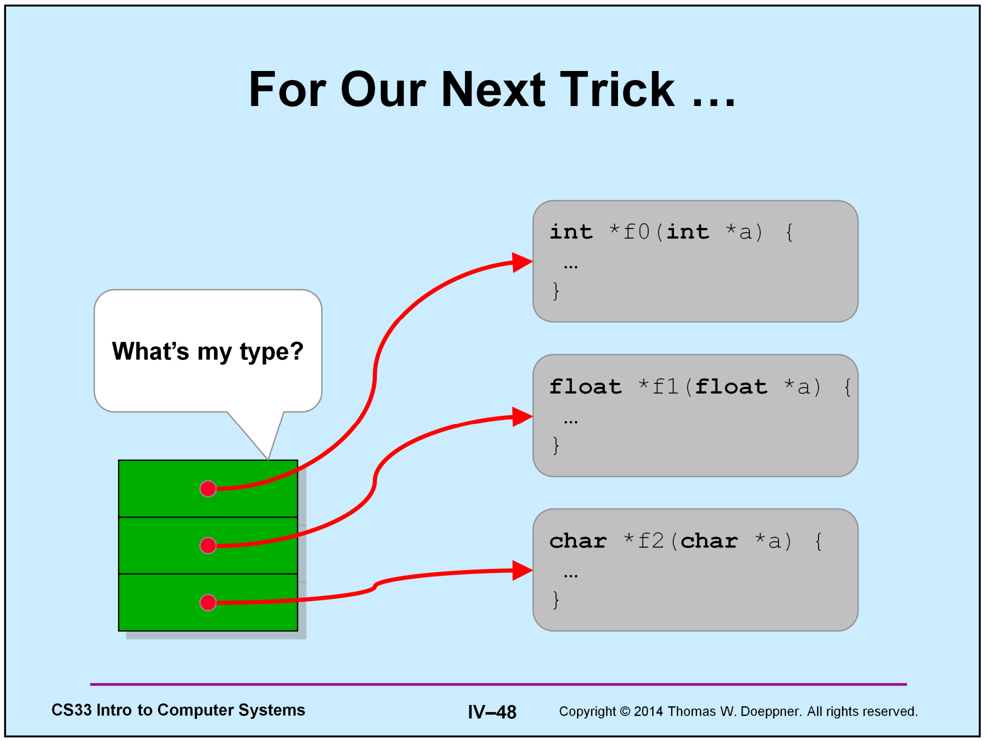
An Application: Generic Swap

```
void gswap (void *p1, void *p2,
     int size) {
  int i;
  for (i=0; i < size; i++) {
     char tmp;
     tmp = ((char *)p1)[i];
     ((char *)p1)[i] = ((char *)p2)[i];
     ((char *)p2)[i] = tmp;
  }
}
```

Note that there is a procedure in the C library that one may use to copy arbitrary amounts of data. It’s called memcpy. To see its documentation, use the Linux command “man memcpy”.

Using Generic Swap

```
short a, b;
gswap(&a, &b, sizeof(short));

int x, y;
gswap(&x, &y, 4);

int A[] = {1, 2, 3}, B[] = {7, 8, 9};
gswap(A, B, sizeof(A));
```



For Our Next Trick …

```
int *f0(int *a) {
 …
}
```

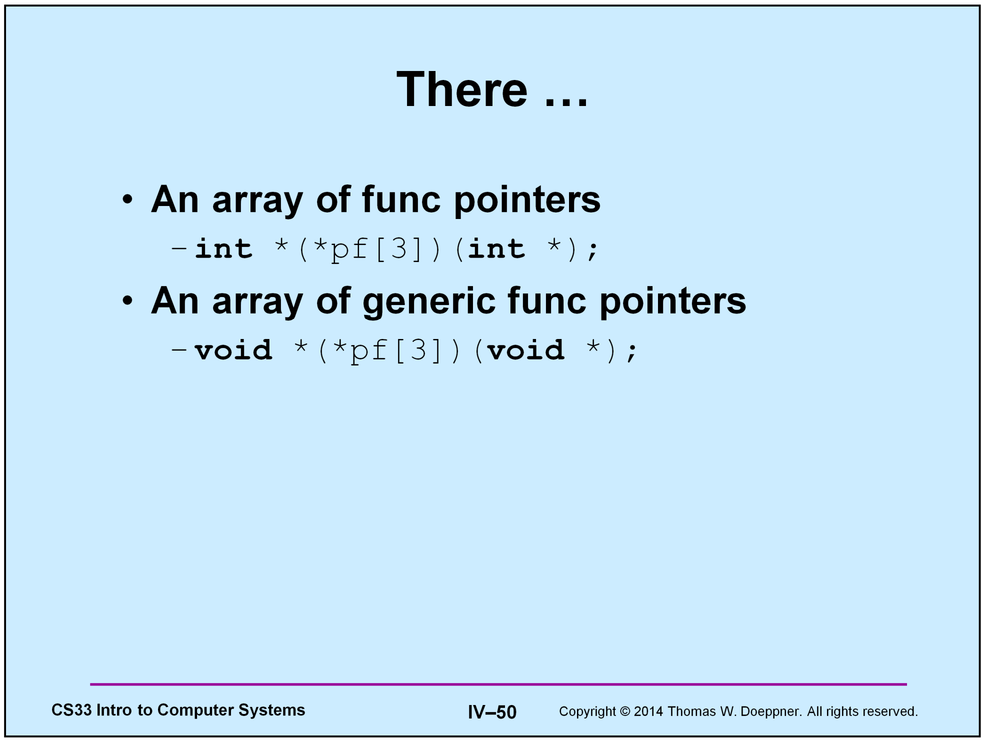
```
float *f1(float *a) {
 …
}
```

```
char *f2(char *a) {
 …
}
```

What we want to come up with is an array, each of whose elements is a function that takes a single pointer argument and returns a pointer value. However, the type of what the pointer points to might be different for each element. What is the type of the resulting array?

Working Our Way There …

- An array of 3 ints
  - `int A[3];`
- An array of 3 int *s
  - `int *A[3];`
- A func returning an int *, taking an int *
  - `int *f(int *);`
- A pointer to such a func
  - `int *(*pf)(int *);`



There …

- An array of func pointers
  - `int *(*pf[3])(int *);`
- An array of generic func pointers
  - `void *(*pf[3])(void *);`

Note that we can't make the function pointers so generic that they may have differing numbers of arguments.

Using It

```
int *f0(int *a) { … }
float *f1(float *a) { … }
char *f2(char *a) { … }
int main() {
  int x = 1;
  int *p;
  void *(*pf[3])(void *);
  pf[0] = (void *(*)(void *))f0;
  pf[1] = (void *(*)(void *))f1;
  pf[2] = (void *(*)(void *))f2;
  p = pf[0](&x);
}
```

Casts, Yet Again

- They tell the C compiler:
  "Shut up, I know what I'm doing!"
- Sometimes true

```
pf[0] = (void *(*)(void *))f0;
```

- Sometimes false

```
int f = 7;
(void(*)(int))f(2);
```

Laziness …

- Why type the declaration

```
void *(*f)(void*, void *);
```

- You could, instead, type

```
MyType f;
```

- (If, of course, you can somehow define *MyType* to mean the right thing)

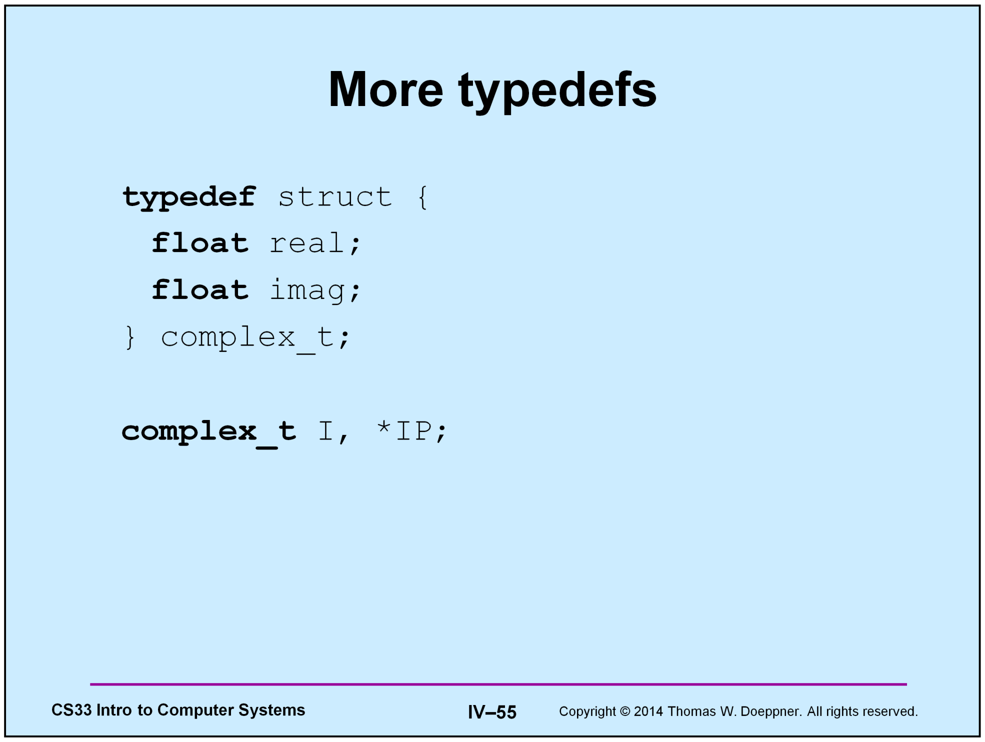
typedef

- Allows one to create new names for existing types

```
typedef int *IntP;



IntP x;
```

  – means the same as

```
int *x;
```



More typedefs

```
typedef struct {
  float real;
  float imag;
} complex_t;

complex_t I, *IP;
```

A standard convention for C is that names of datatypes end with “_t”.

And …

```
typedef void *(*MyFunc)(void *, void *);

MyFunc f;

/* you must do its definition the long
   way */

void *f(void *a1, void *a2) {
  …
}
```

And Finally …

- What's a possible use of …

```
void **
```

?