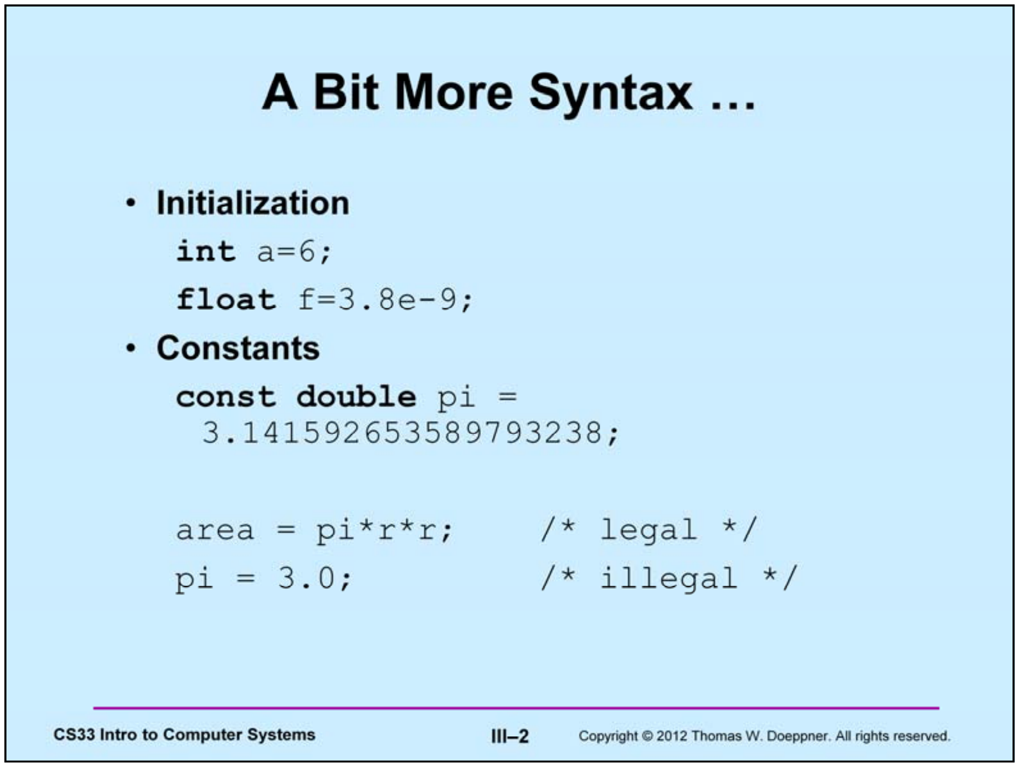
CS 33

More C



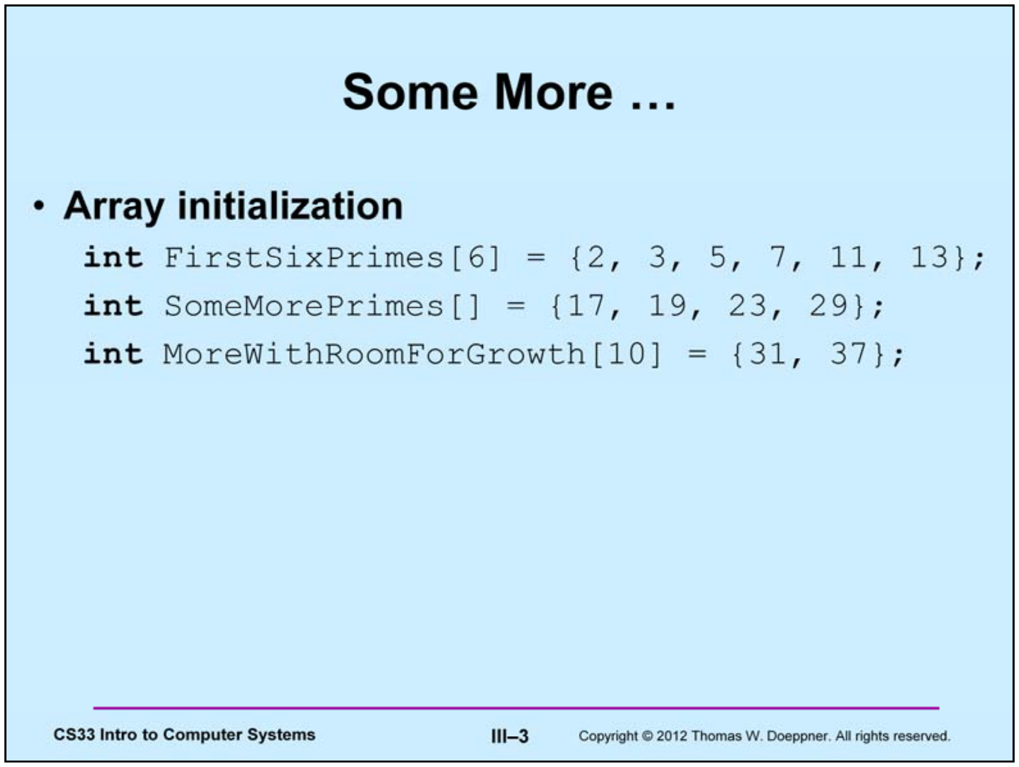
A Bit More Syntax …

- **Initialization**

```
int a=6;
float f=3.8e-9;
```

- **Constants**

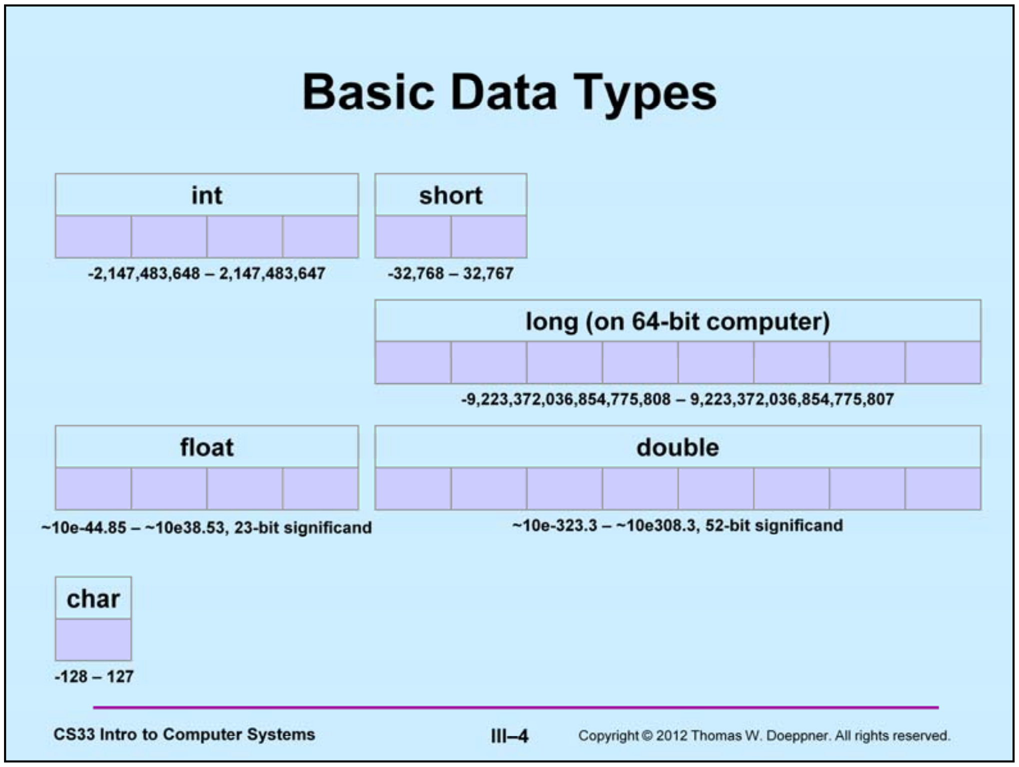
```
const double pi =
 3.14159265358979323846;

area = pi*r*r;     /* legal */
pi = 3.0;          /* illegal */
```



Some More …

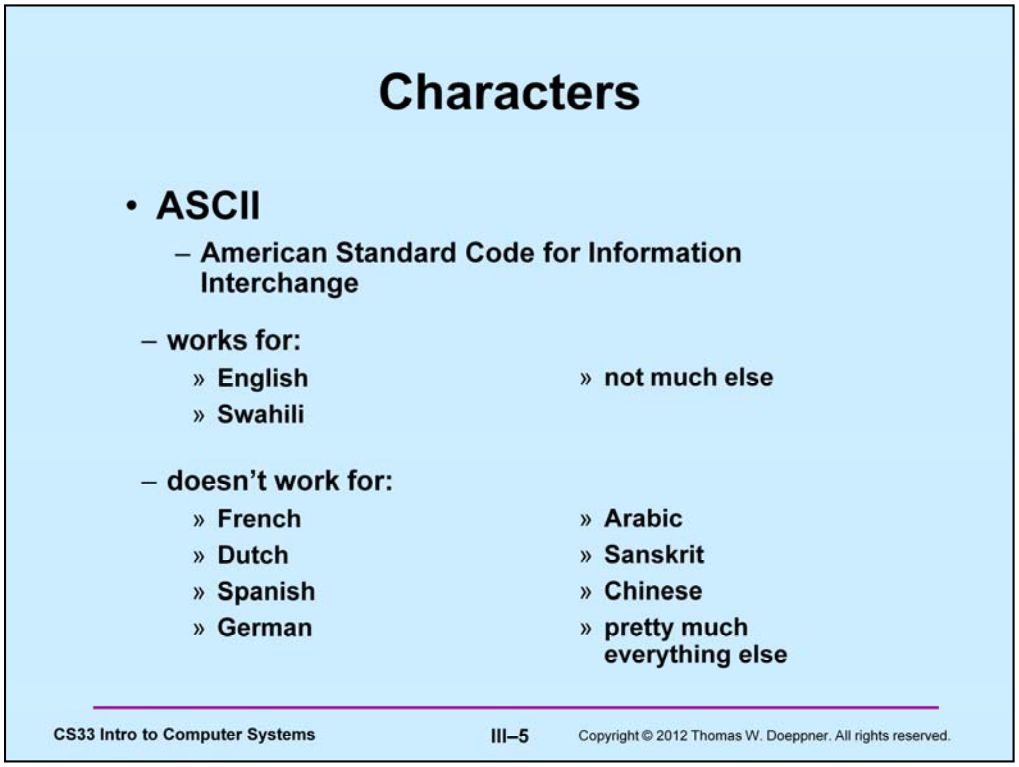
- **Array initialization**

```
int FirstSixPrimes[6] = {2, 3, 5, 7, 11, 13};
int SomeMorePrimes[] = {17, 19, 23, 29};
int MoreWithRoomForGrowth[10] = {31, 37};
```



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

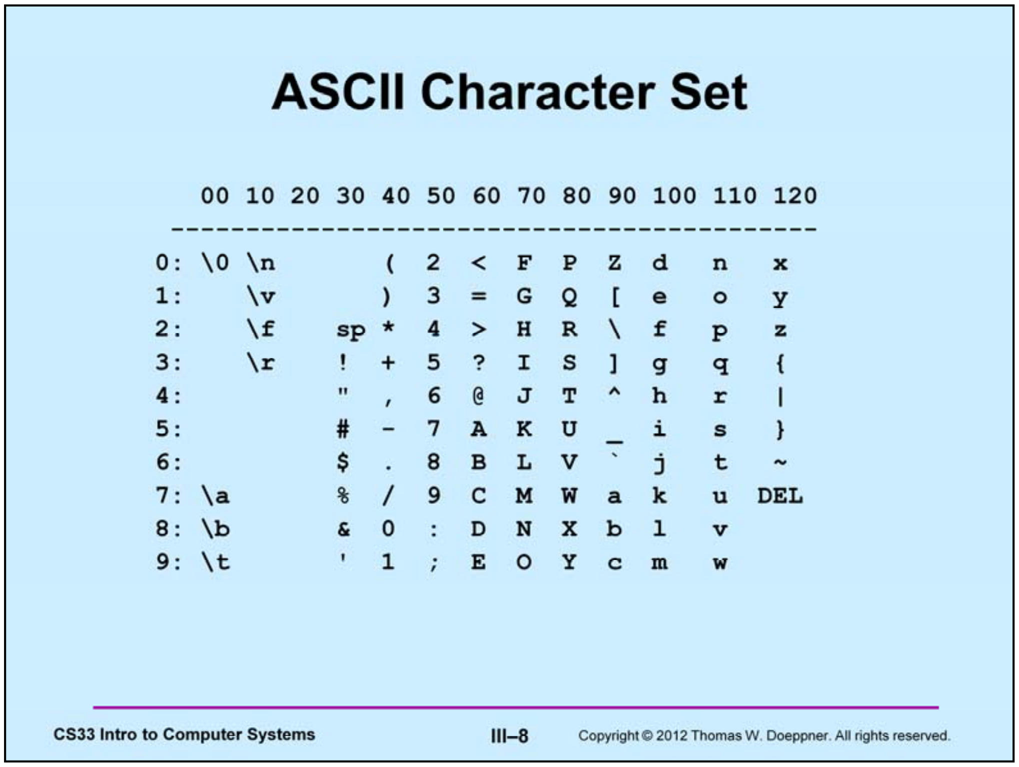
- **ASCII**
  - American Standard Code for Information Interchange
  - works for:
    - » English
    - » Swahili
    - » not much else
  - doesn't work for:
    - » French
    - » Dutch
    - » Spanish
    - » German
    - » Arabic
    - » Sanskrit
    - » Chinese
    - » pretty much everything else

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	

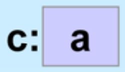
ASCII uses only seven bits. Most European languages can be coded with eight bits. Many Asian languages require far more than eight bits.

*char*s as Integers

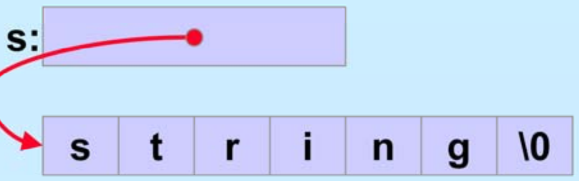
```
char tolower(char c) {
  if (c >= 'A' && c <= 'Z')
      return c + 'a' - 'A';
  else
      return c;
}
```

Character Strings

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



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



Quiz (1)

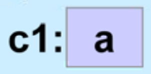
Is there any difference between *c1* and *c2* in the following?

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

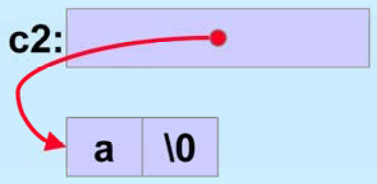
Answer (1)

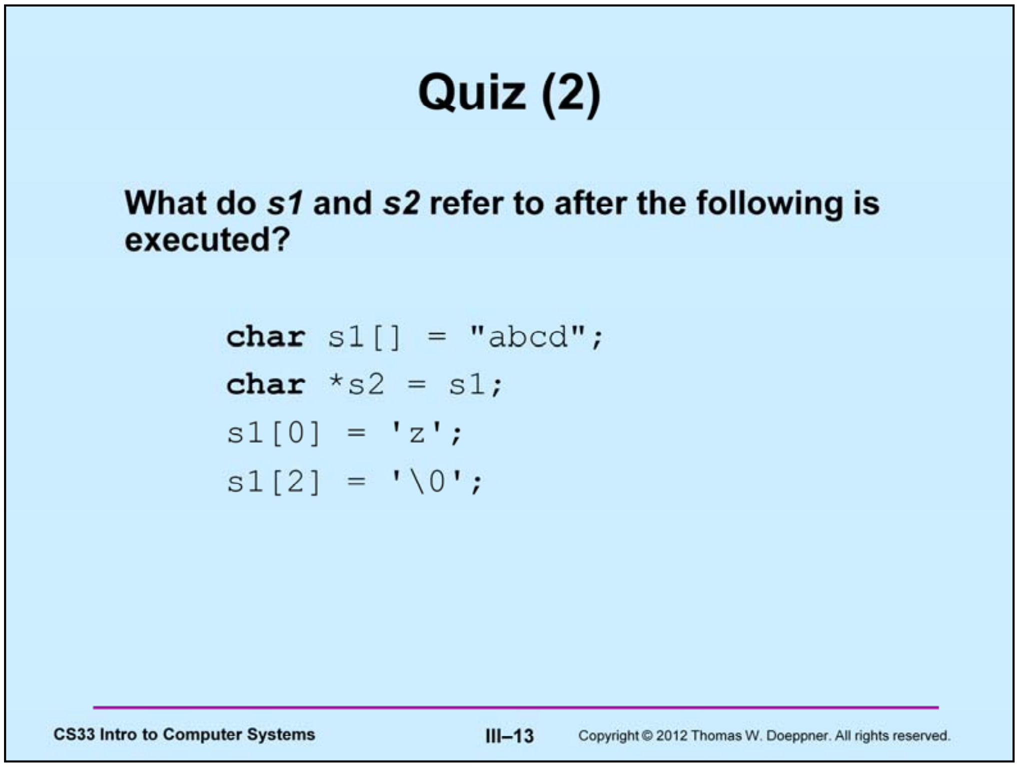
Yes!!

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



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

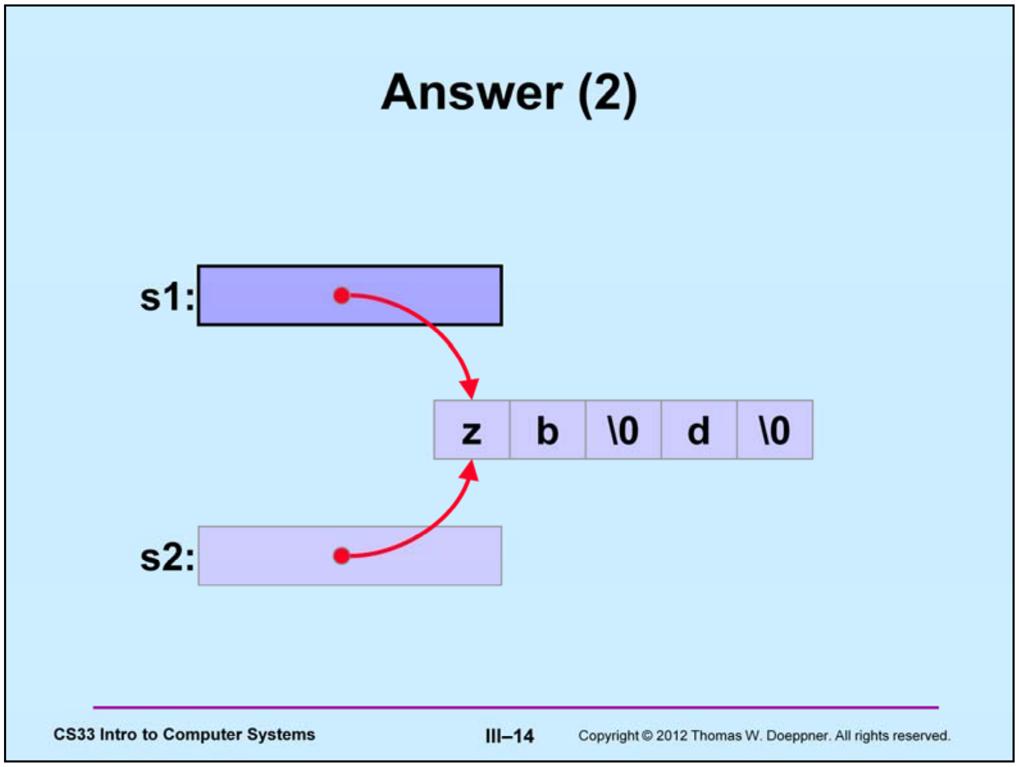




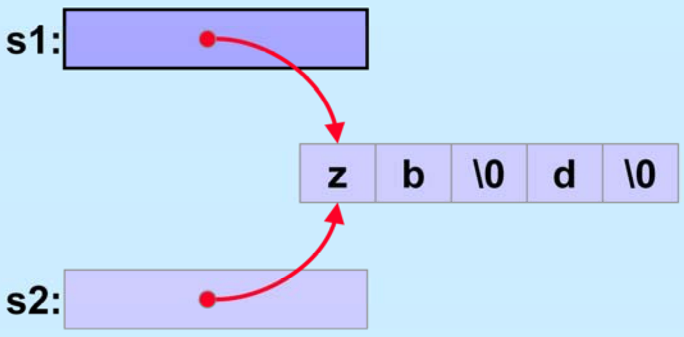
Quiz (2)

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

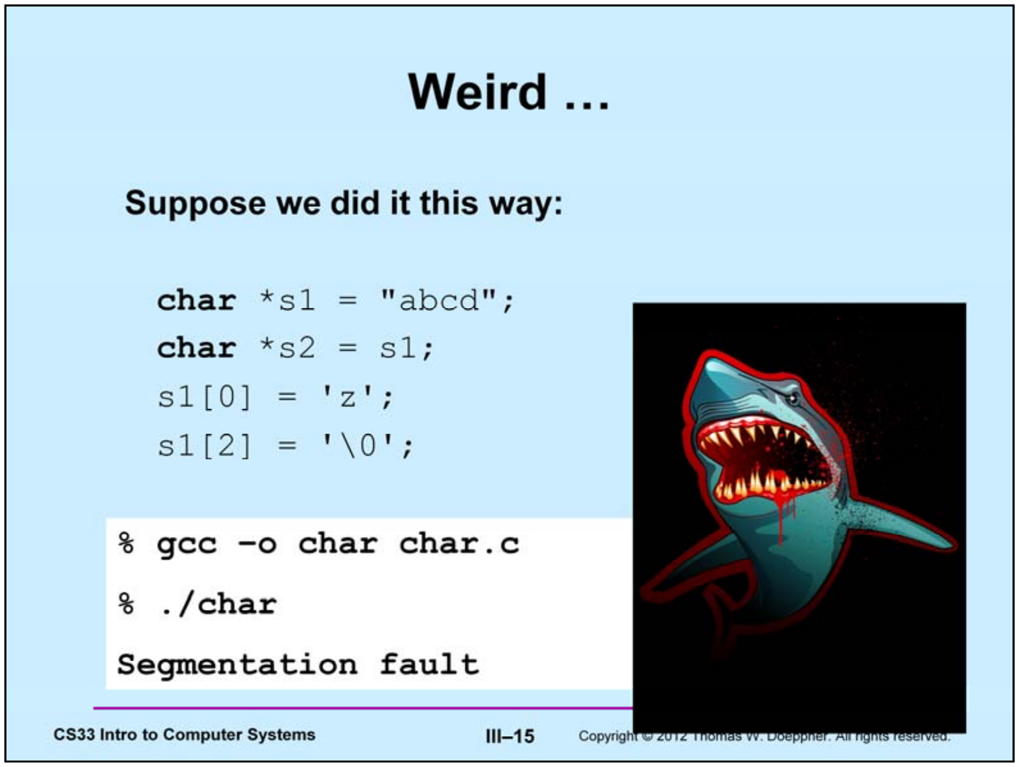
```
char s1[] = "abcd";
char *s2 = s1;
s1[0] = 'z';
s1[2] = '\0';
```



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.



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.

Structures

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

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

*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;
}
```

Alternatively …

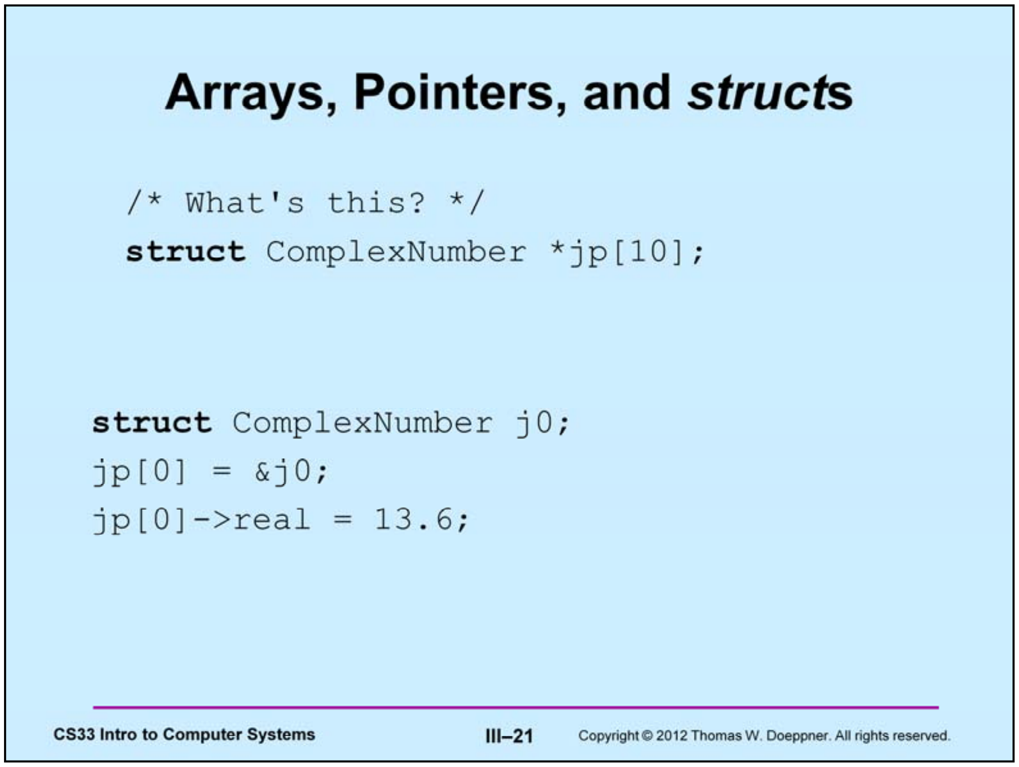
```
void ComplexAdd(
      struct ComplexNumber *a1,
      struct ComplexNumber *a2,
      struct ComplexNumber *result) {
  result->real = a1->real + a2->real;
  result->imag = a1->imag + a2->imag;
  return;
}
```

Using It …

```
struct ComplexNumber j1 = {3.6, 2.125};
struct ComplexNumber j2 = {4.32, 3.1416};
struct ComplexNumber j3;

ComplexAdd(&j1, &j2, &j3);
```

Arrays of *structs*

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



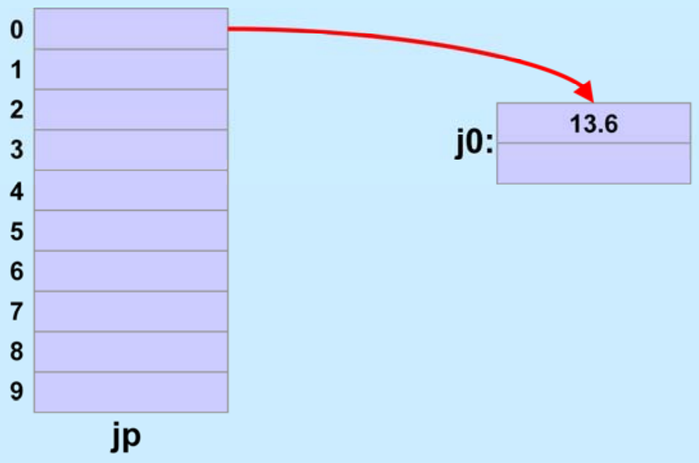
Arrays, Pointers, and *structs*

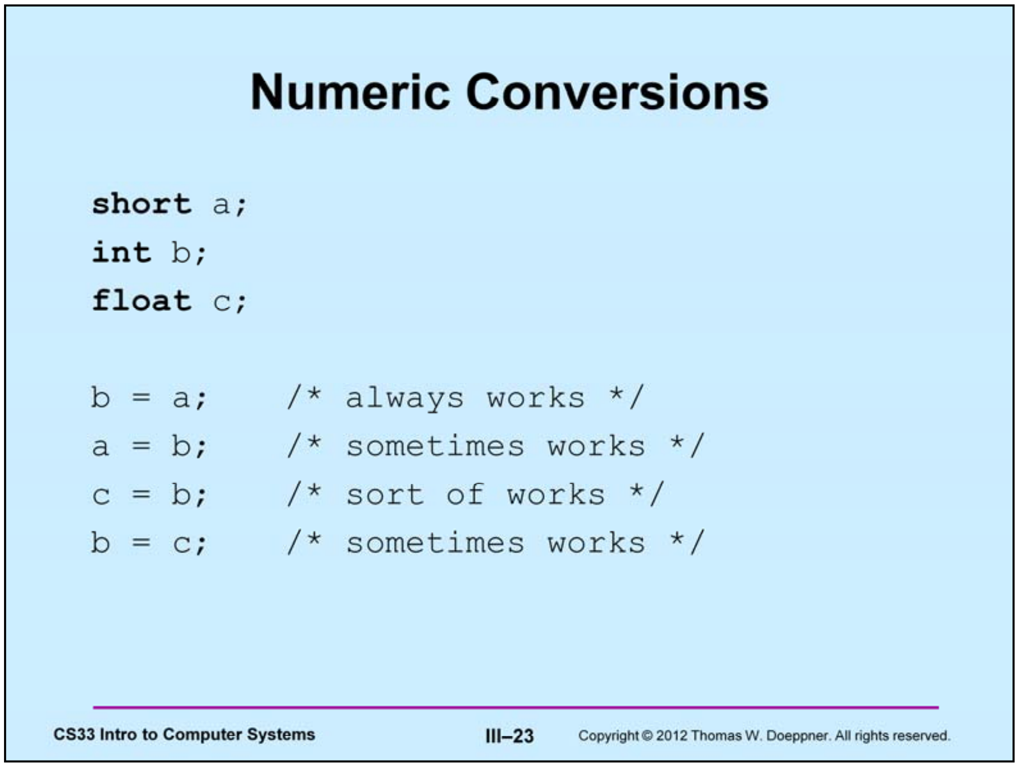
```
/* What's this? */
struct ComplexNumber *jp[10];
```

```
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 ComplexNumbers, rather than a pointer to an array of struct ComplexNumbers.

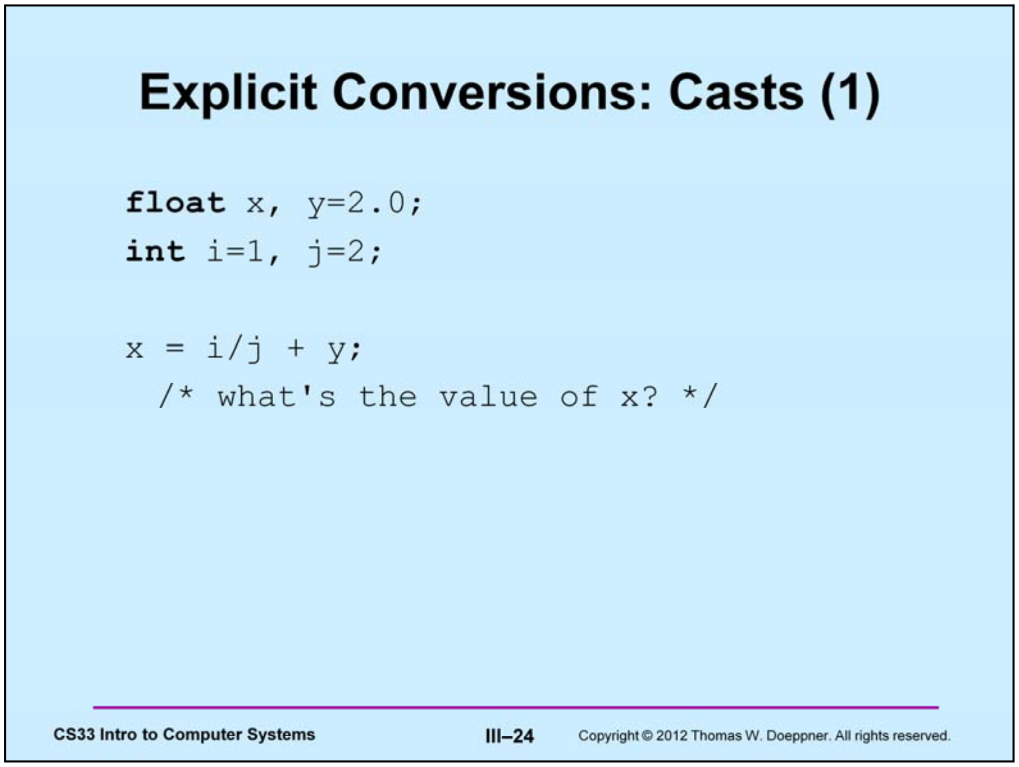
Memory View





Numeric Conversions

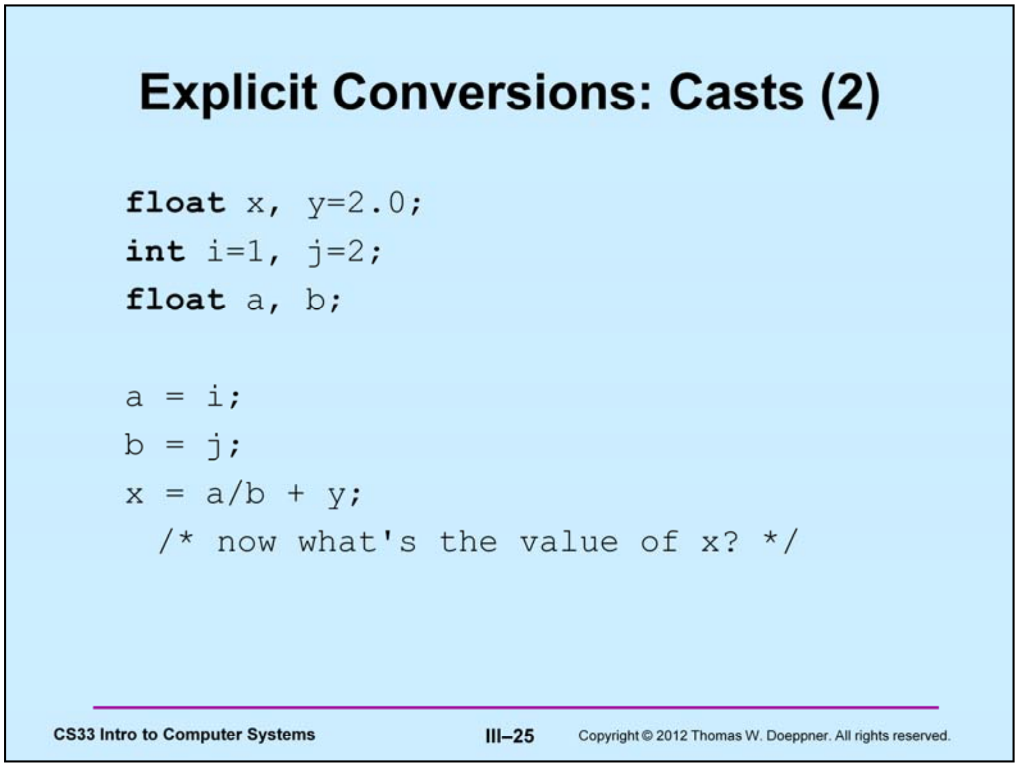
```
short a;
int b;
float c;

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? */
```

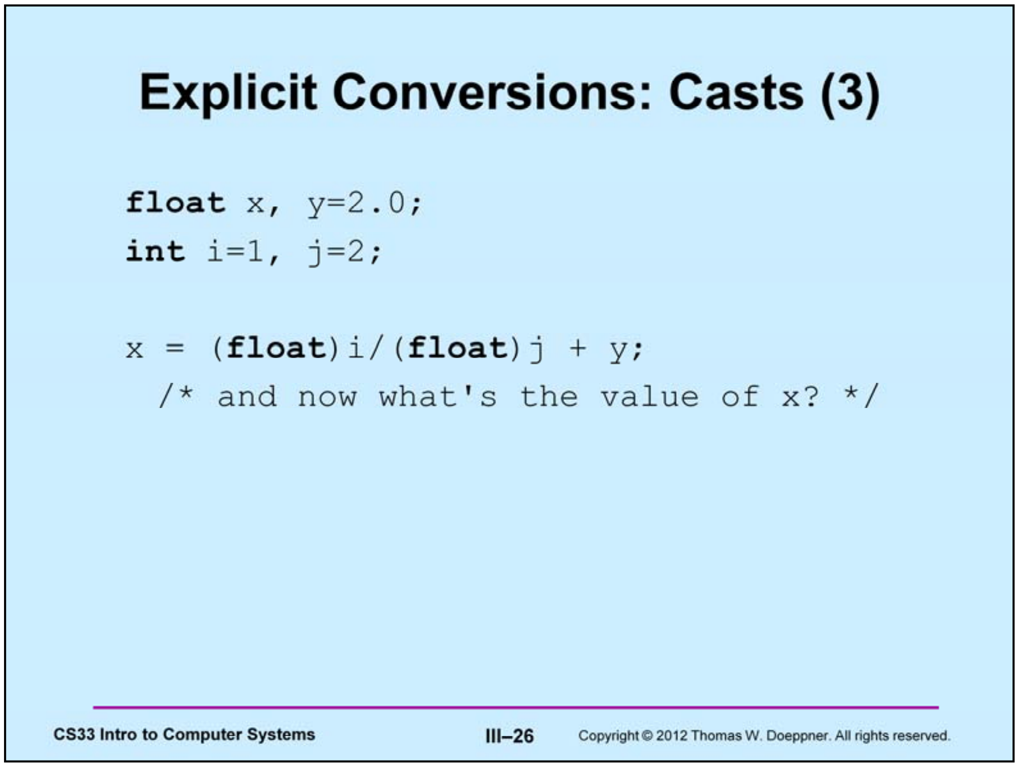
x's value will be 2, since the result of the (integer) division of i by j will be 0.



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.

Transition

- Things have been straightforward
- Now for the fun stuff …

Fun with Functions (1)

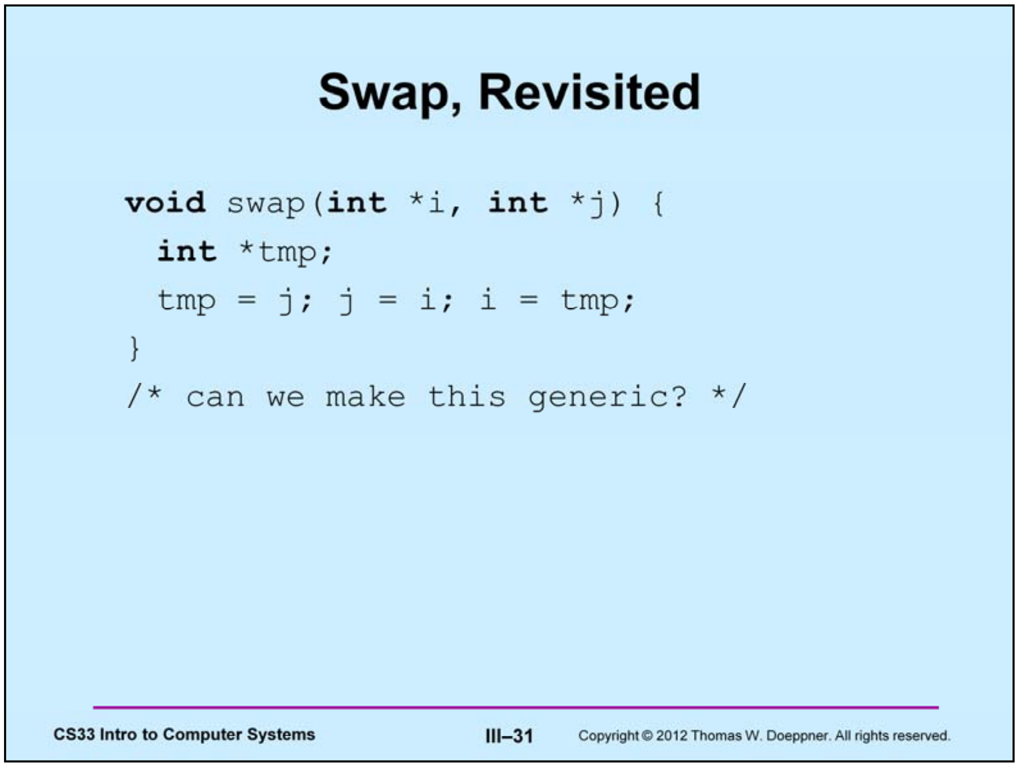
```
void ArrayDouble(int A[], int len) {
  int i;
  for (i=0; i<len; i++)
      A[i] = 2*A[i];
}
```

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]);
}
```

Fun with Functions (3)

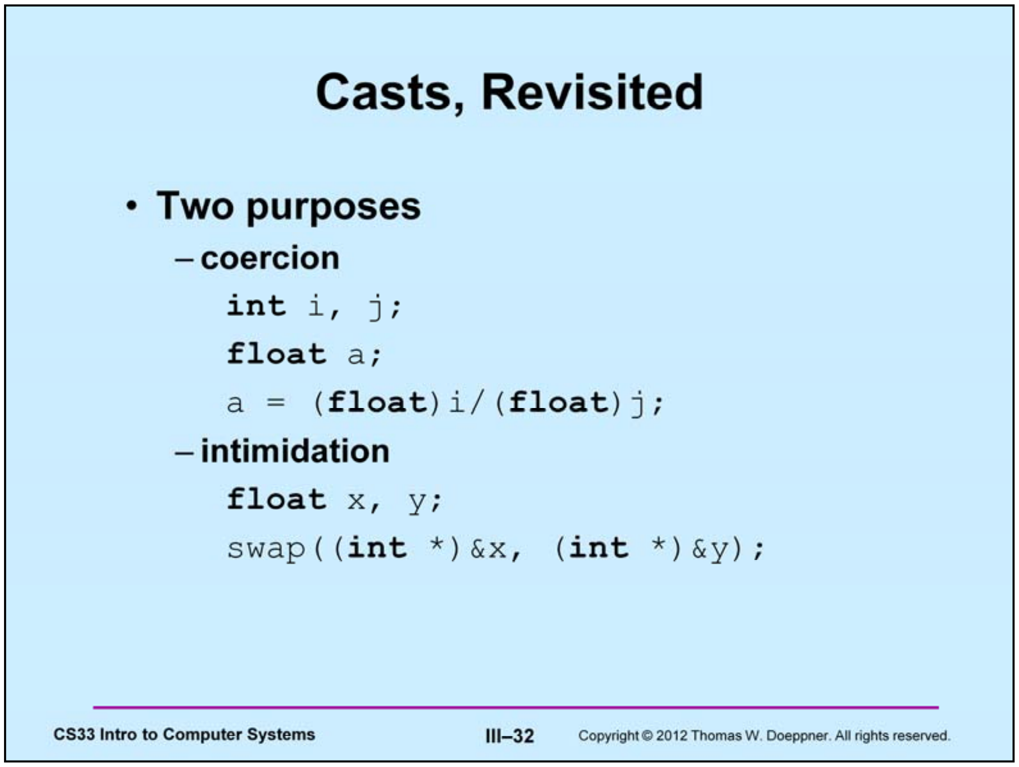
```
int triple(int arg) {
  return 3*arg;
}

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

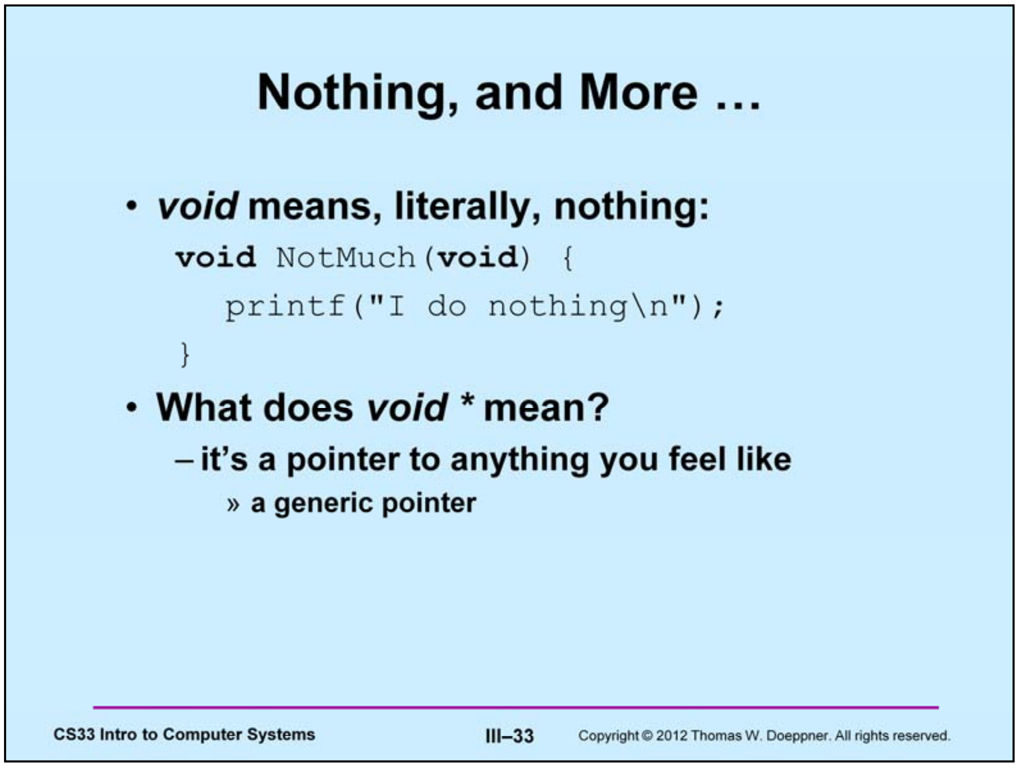
- Two purposes
  - coercion

```
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.

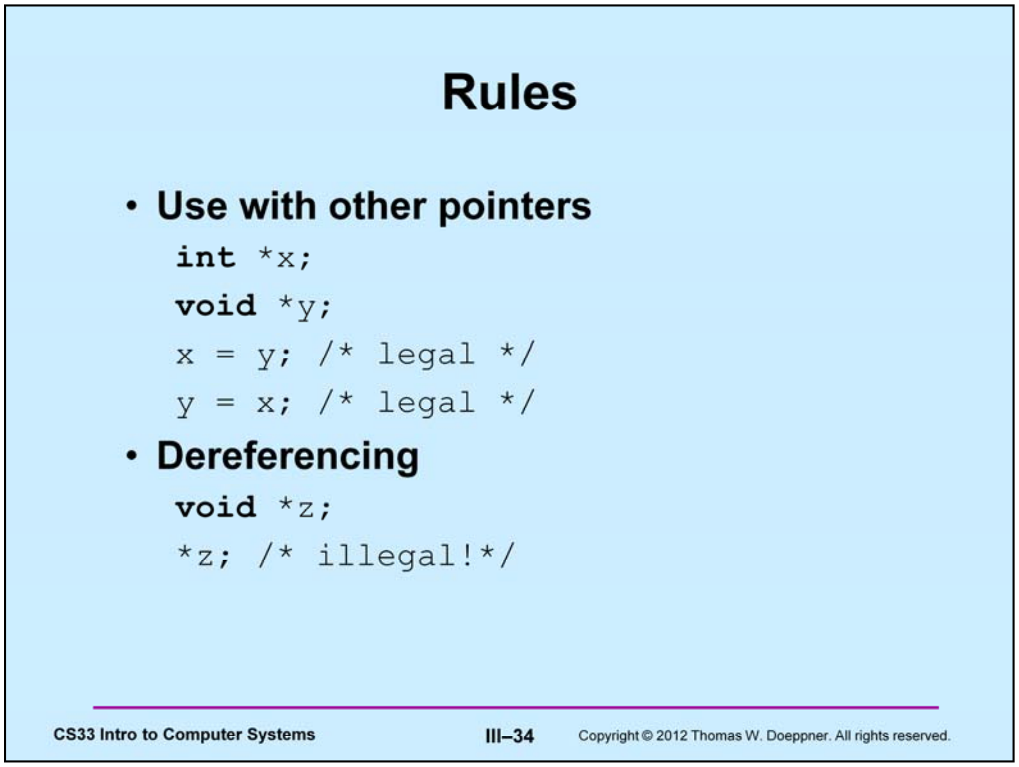


Nothing, and More …

- ***void*** **means, literally, nothing:**

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

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



Rules

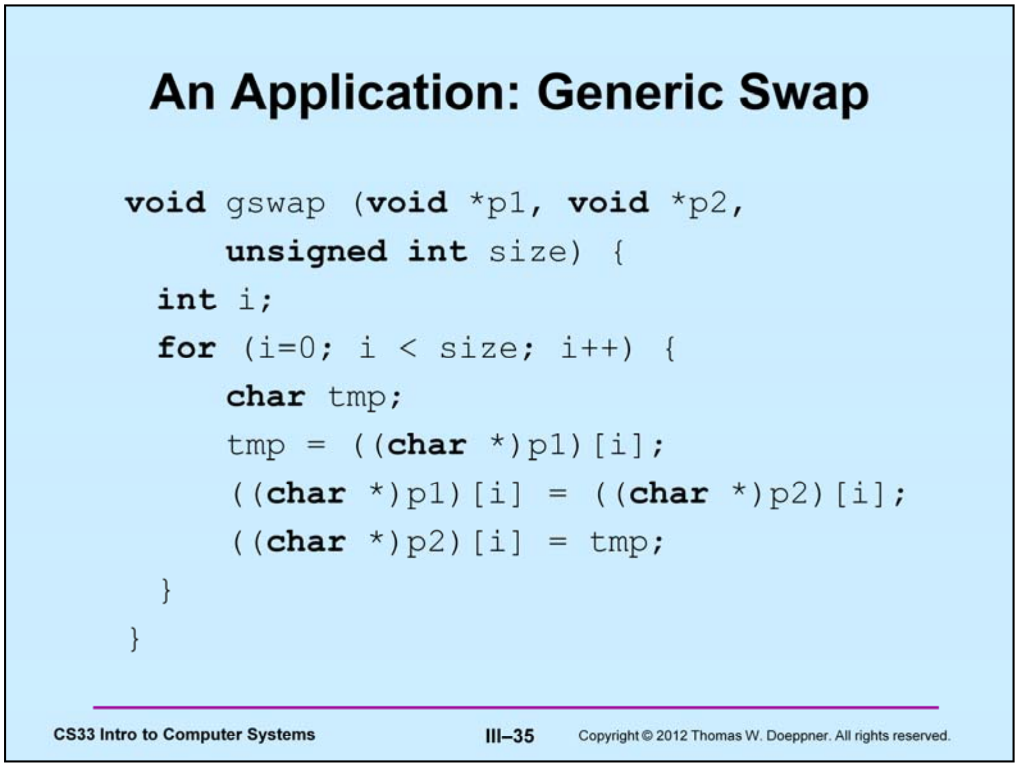
- **Use with other pointers**

```
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,
      unsigned 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”.

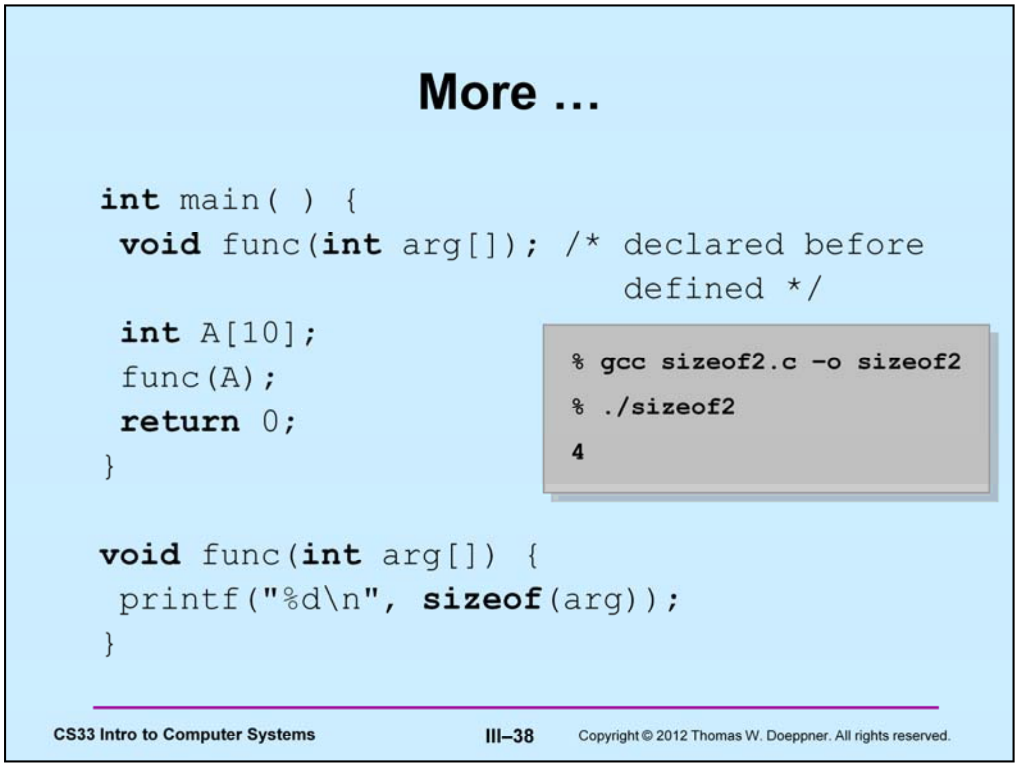
Using Generic Swap

```
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));
```

sizeof

```
int main( ) {
  int A[] = {1, 2, 3};
  char s1[] = "hello";
  char *s2 = "goodbye";

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

```
% gcc sizeof.c -o sizeof
% ./sizeof
12, 6, 4
```

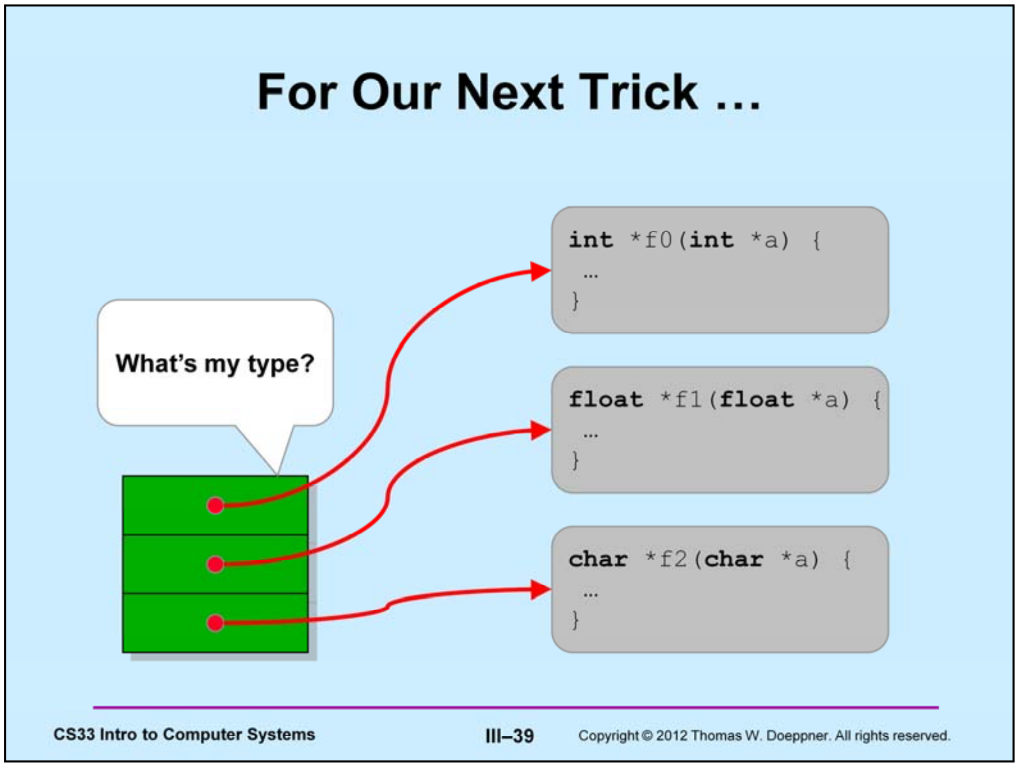


More …

```
int main( ) {
 void func(int arg[]); /* declared before
                          defined */
 int A[10];
 func(A);
 return 0;
}

void func(int arg[]) {
 printf("%d\n", sizeof(arg));
}
```

```
% gcc sizeof2.c -o sizeof2
% ./sizeof2
4
```

Note that even if func's argument were declared "int arg[10]", the value printed would still be 4. Array bounds really are meaningless for arrays passed to procedures.



For Our Next Trick …

```
int *f0(int *a) {
 …
}
```

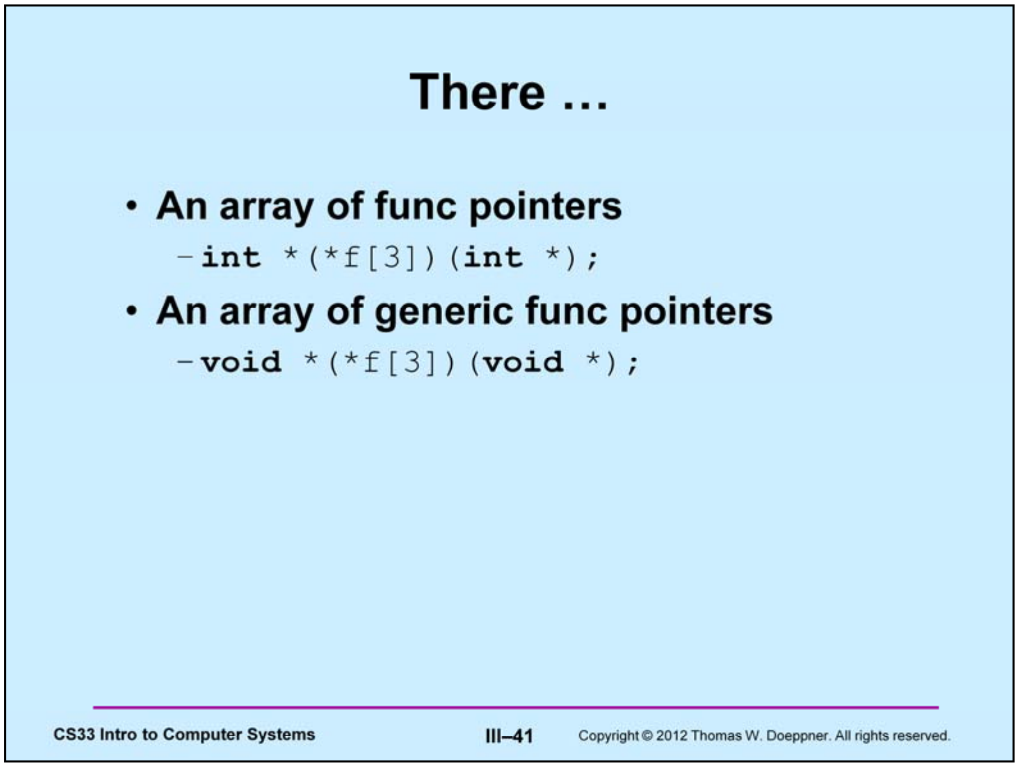
```
float *f1(float *a) {
 …
}
```

```
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 …

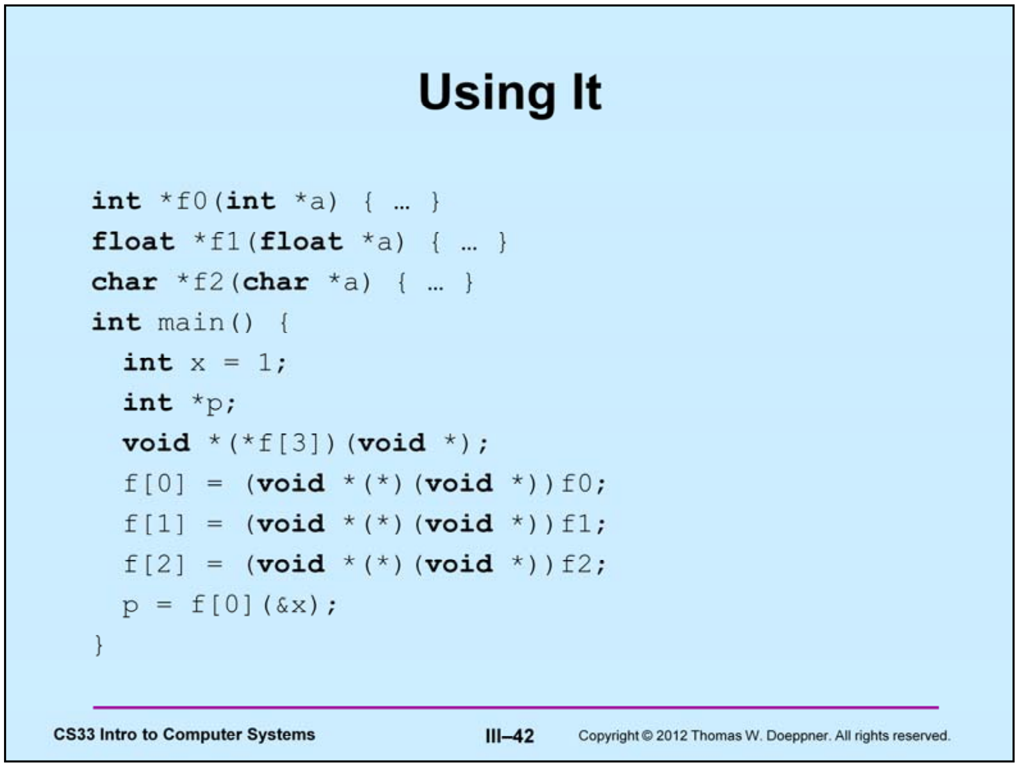
- An array of 3 ints
  - `int A[3];`
- An array of 3 int *s
  - `int *A[3];`
- A func returning an int *, taking an int *
  - `int *f(int *);`
- A pointer to such a func
  - `int *(*f)(int *);`



There …

- An array of func pointers
  - `int *(*f[3])(int *);`
- An array of generic func pointers
  - `void *(*f[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 *(*f[3])(void *);
  f[0] = (void *(*)(void *))f0;
  f[1] = (void *(*)(void *))f1;
  f[2] = (void *(*)(void *))f2;
  p = f[0](&x);
}
```

Casts, Yet Again

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

```
f[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

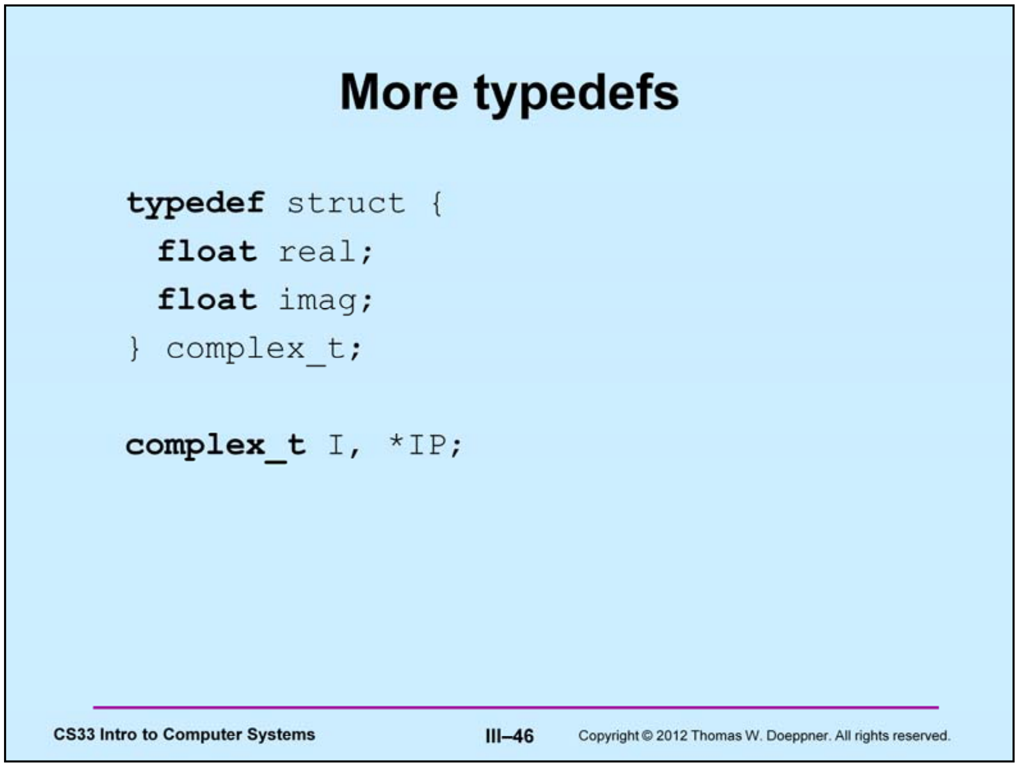
- Allows one to create new names for existing types

```
typedef int *IntP;
```

```
IntP x;
```

- means the same as

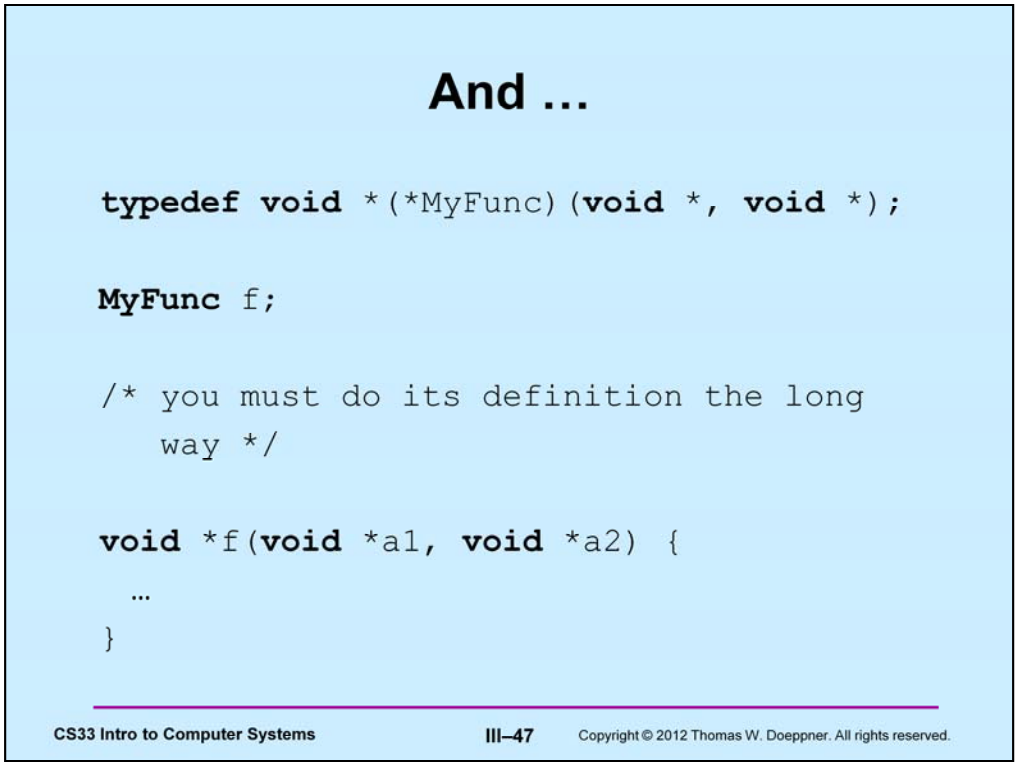
```
int *x;
```



More typedefs

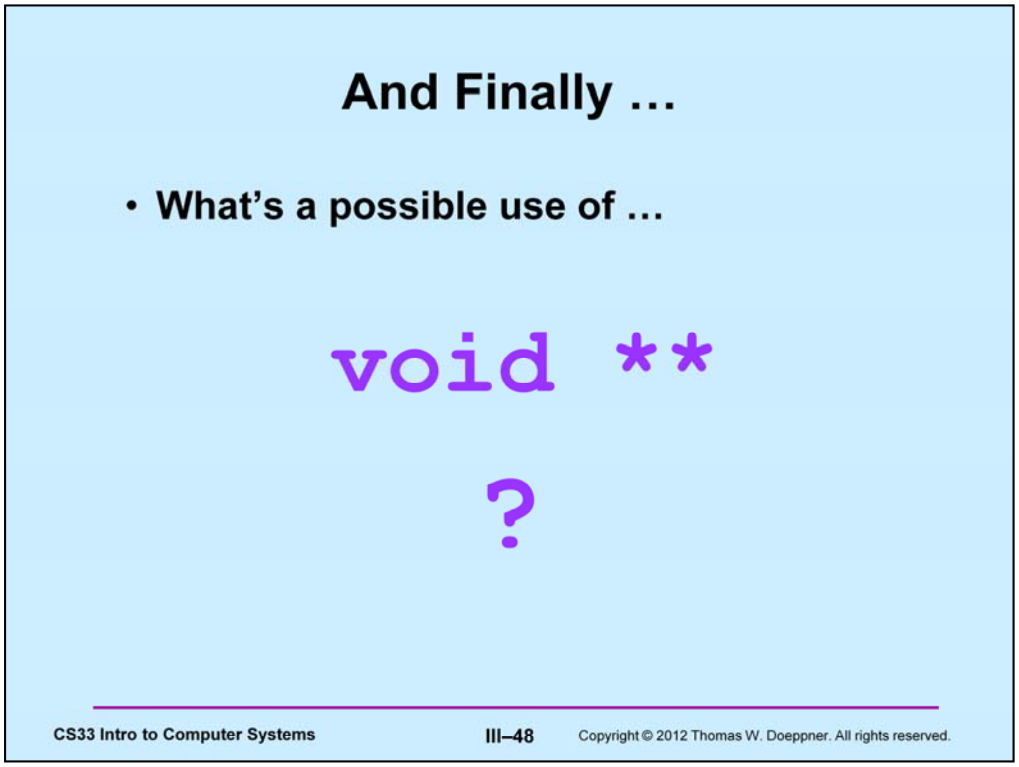
```
typedef struct {
  float real;
  float imag;
} complex_t;

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 **
```

?