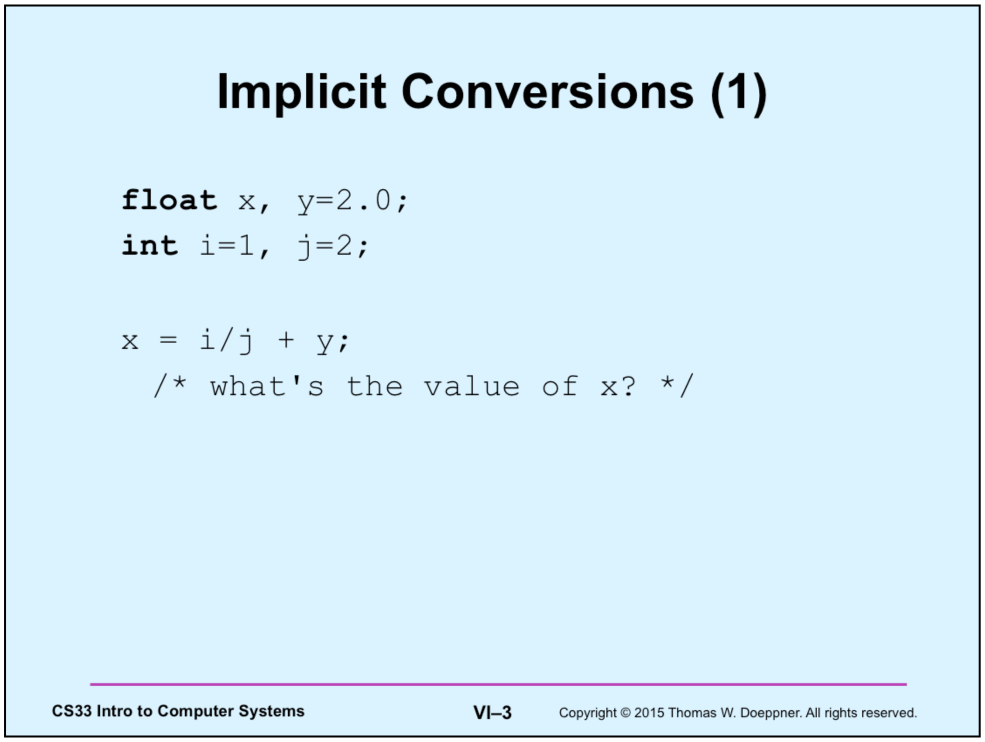
CS 33

Introduction to C
Part 6

Numeric Conversions

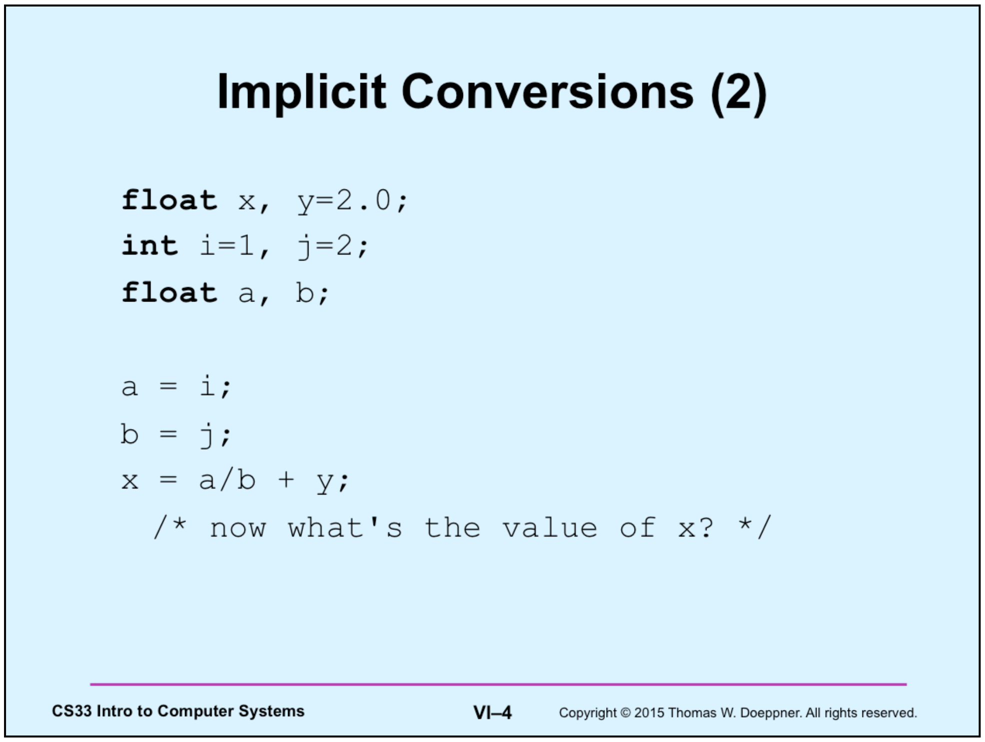
```
short a;
int b;
float c;

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



Implicit Conversions (1)

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

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

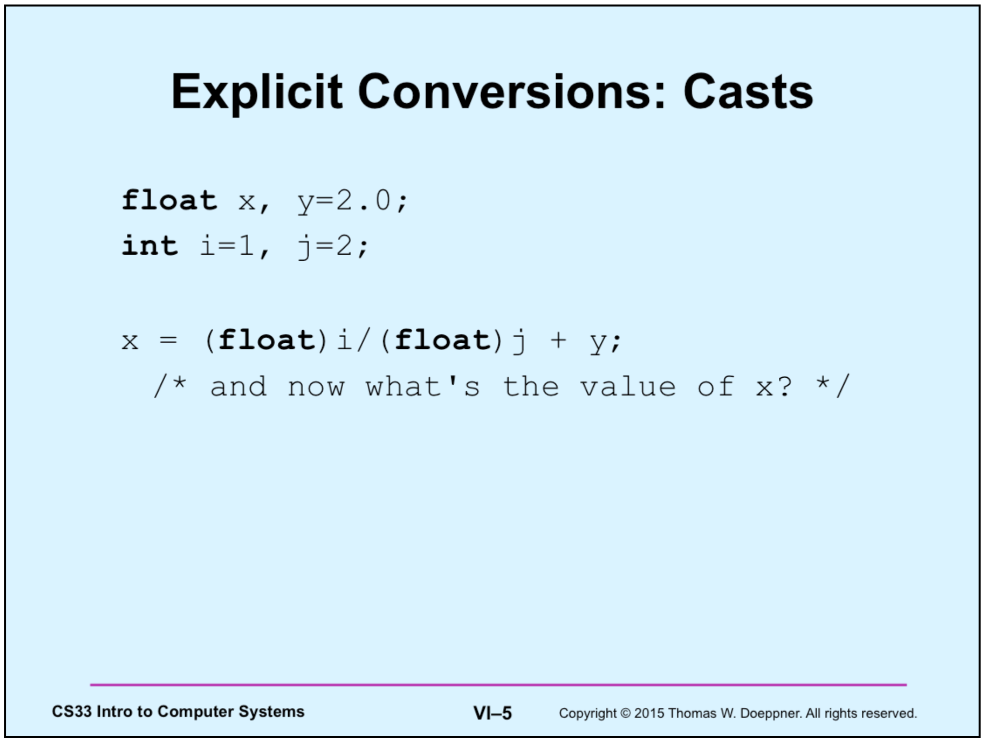
x’s value will be 2, since the result of the (integer) division of i by j will be 0.



Implicit Conversions (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.



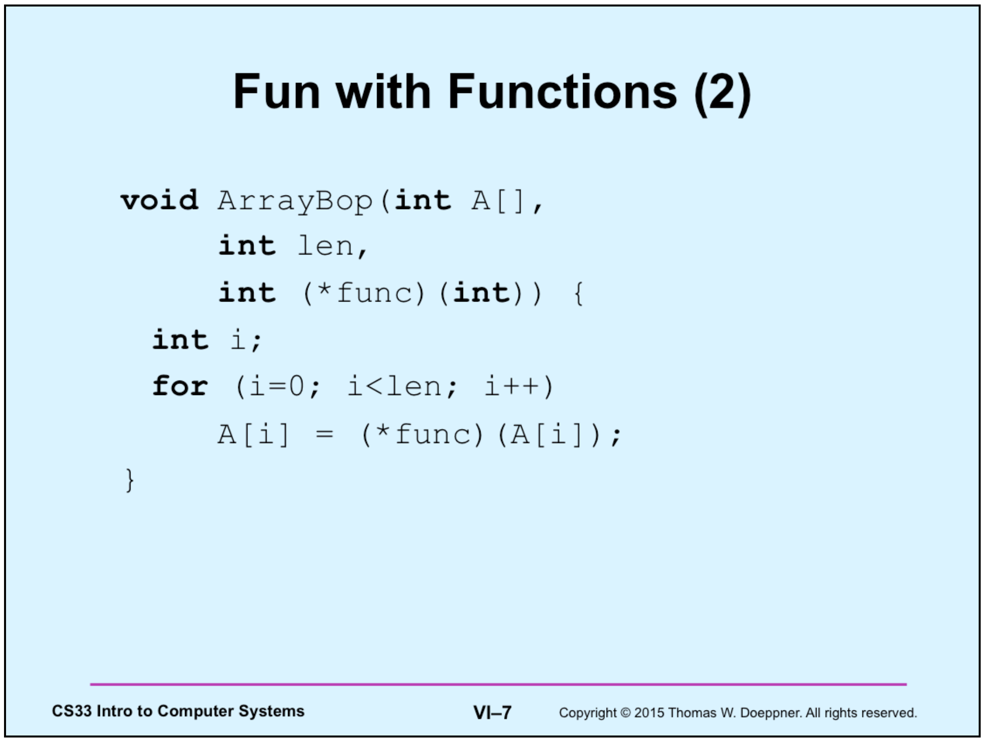
Explicit Conversions: Casts

```
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)

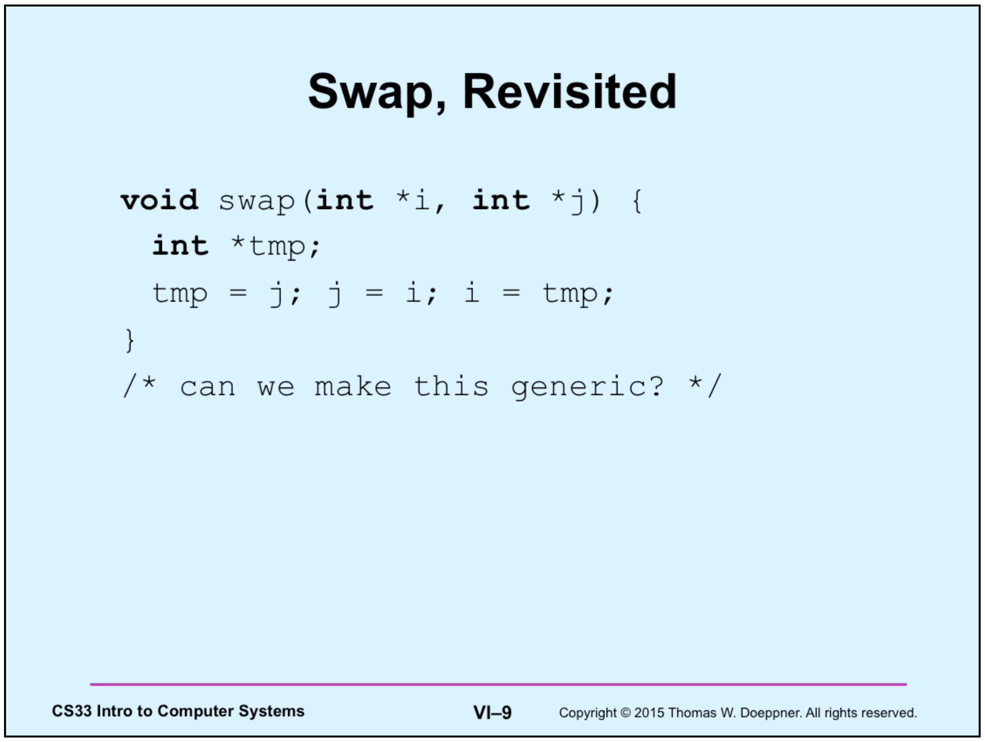
```
void ArrayBop(int A[],
      int len,
      int (*func)(int)) {
  int i;
  for (i=0; i<len; i++)
      A[i] = (*func)(A[i]);
}
```

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)

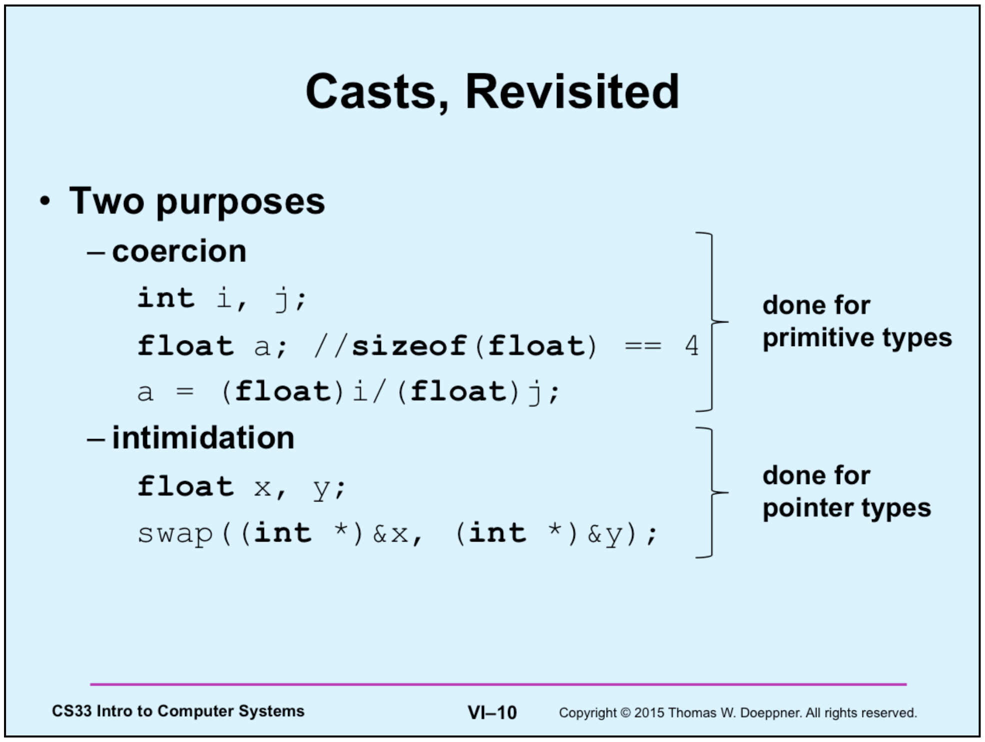
```
int triple(int arg) {
  return 3*arg;
}

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; //sizeof(float) == 4
a = (float)i/(float)j;
```

  - **intimidation**

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

done for primitive types
done for pointer types

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

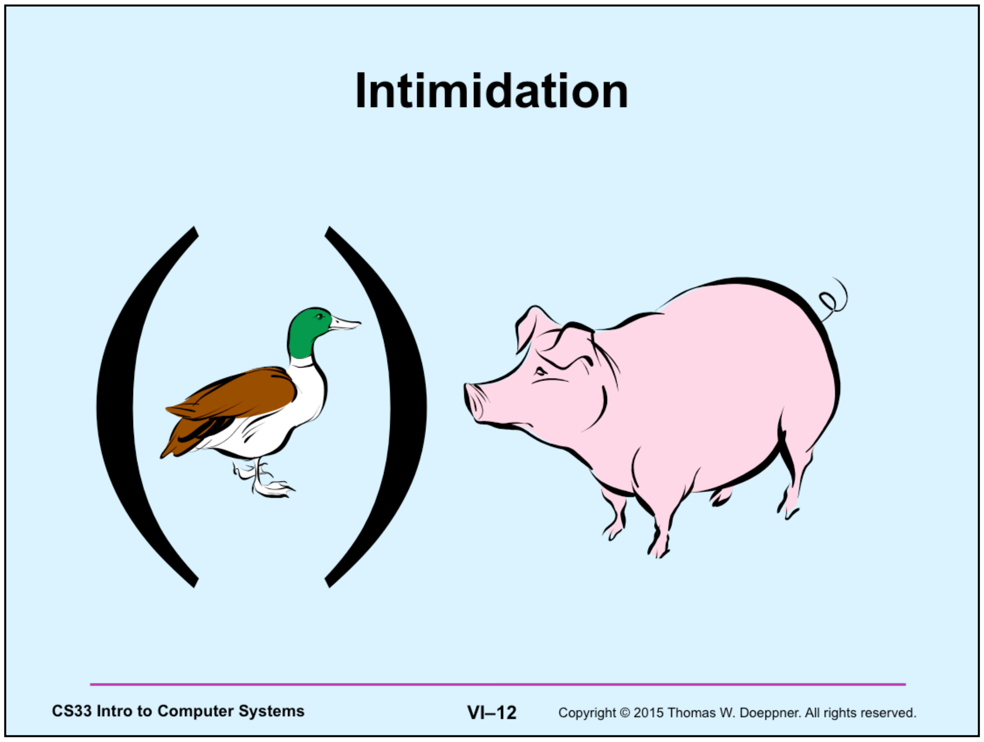
Quiz 1

- **Will this work?**

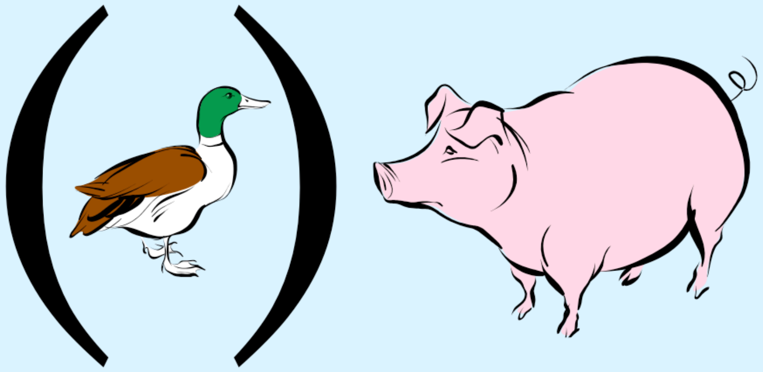
```
double x, y; //sizeof(double) == 8

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

a) yes

b) no



Intimidation



CS33 Intro to Computer Systems VI–12 Copyright © 2015 Thomas W. Doeppner. All rights reserved.

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.

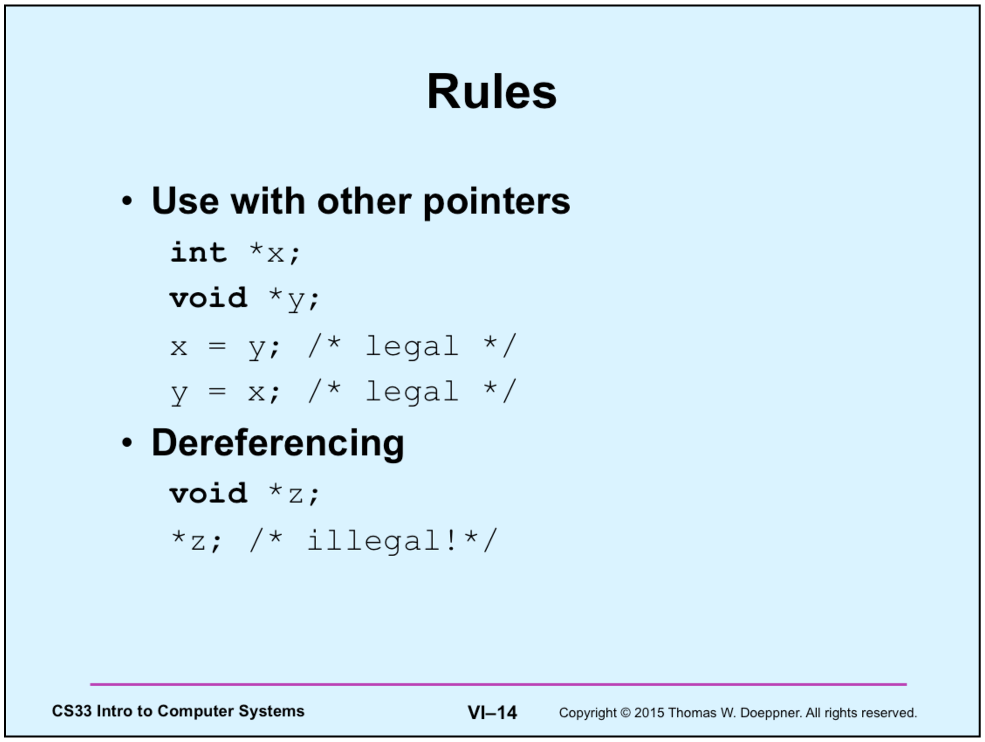
But Chou was that kind of man — he made one believe that pig was duck, because one wanted to believe him, and because he understood the customs of other men and societies and respected them.

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

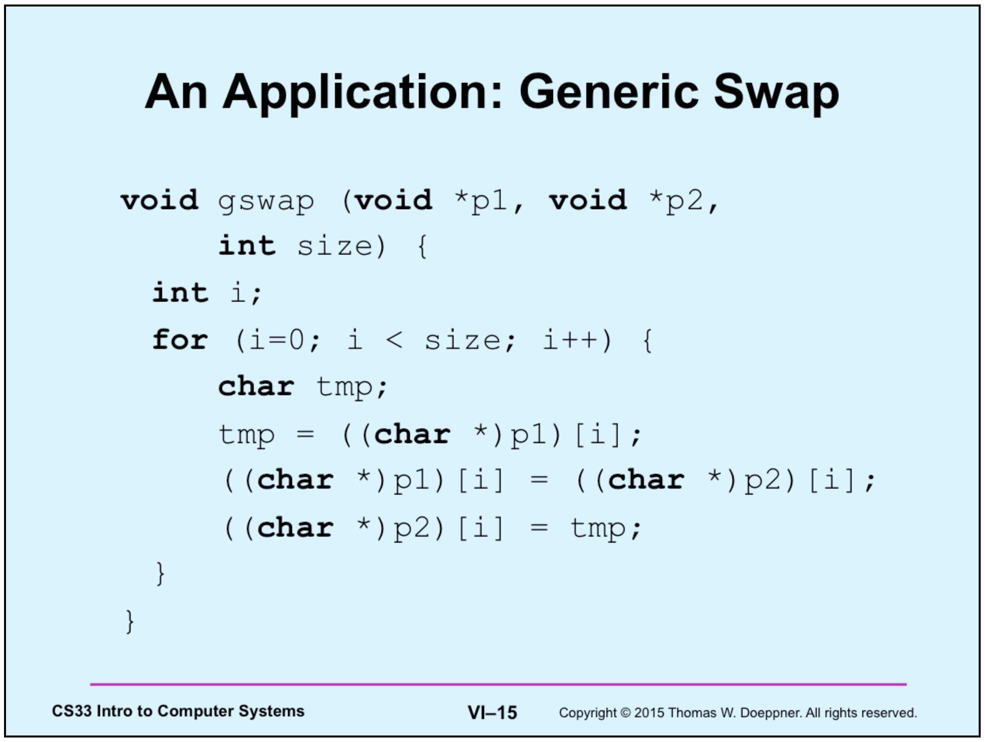
- **Use with other pointers**

```
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.



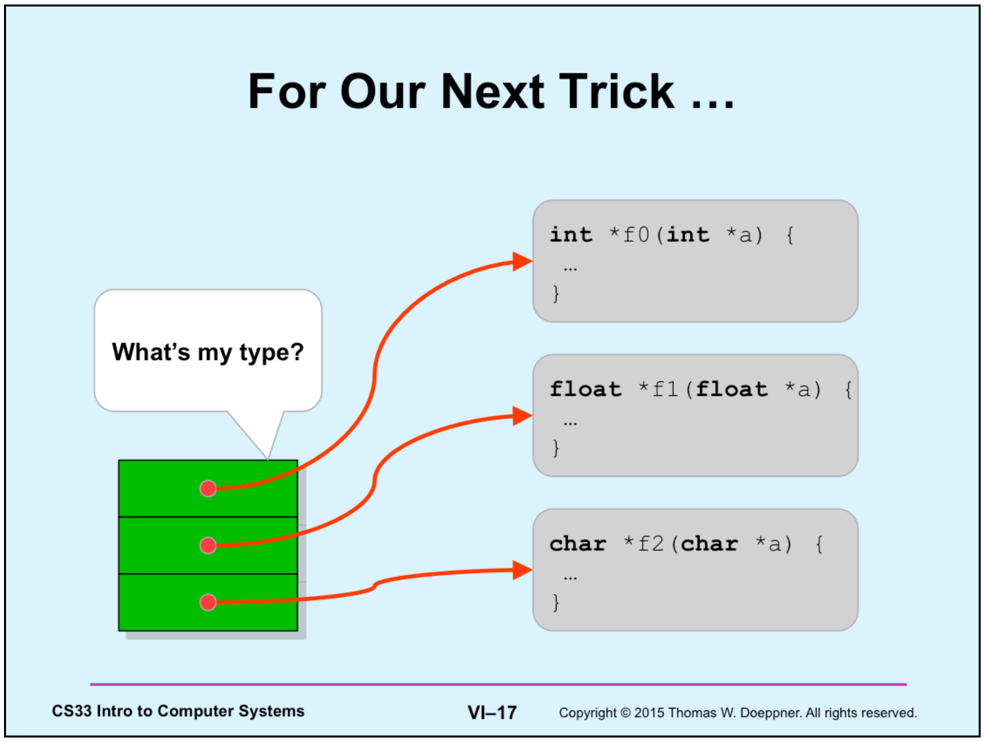
An Application: Generic Swap

```
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, sizeof(int));

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



For Our Next Trick …

What's my type?

```
int *f0(int *a) {
 …
}
```

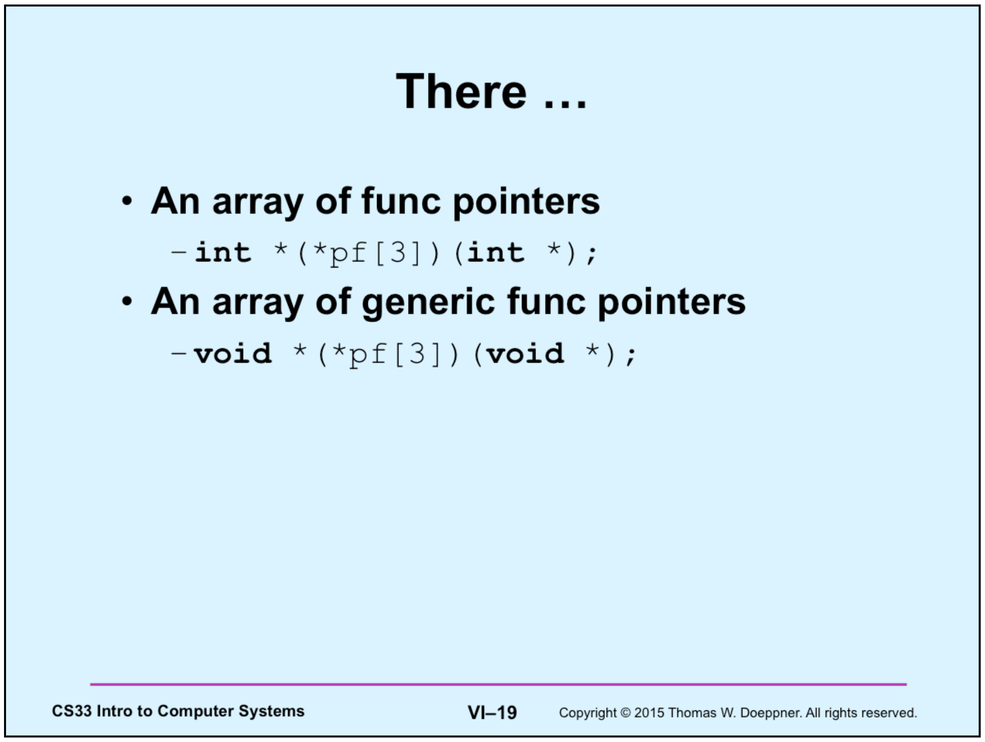
```
float *f1(float *a) {
 …
}
```

```
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) { *a += 1; return a; }
float *f1(float *a) { *a += 1; return a; }
char *f2(char *a) { *a += 1; return 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);
  printf("%d\n", *p);
  return 0;
}
```

```
$ ./funcptr
2
$
```

Quiz 2

```
int *f0(int *a) { *a += 1; return a; }
float *f1(float *a) { *a += 1; return a; }
char *f2(char *a) { *a += 1; return 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[1](&x); // was pf[0]
  printf("%d\n", *p);
  return 0;
}
```

What is printed?

- a) 2
- b) 2.5
- c) something different from the above
- d) nothing: syntax error

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

```
long 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

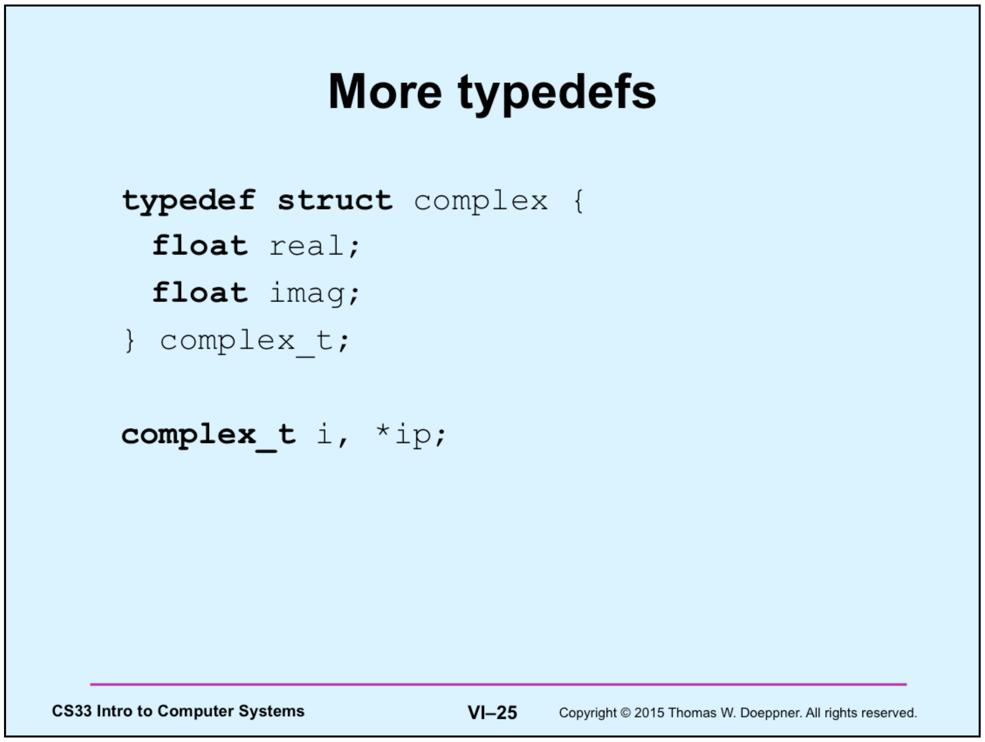
- Allows one to create new names for existing types

```
typedef int *IntP_t;
```

```
IntP_t x;
```

– means the same as

```
int *x;
```



More typedefs

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

complex_t i, *ip;
```

A standard convention for C is that names of datatypes end with "_t". Note that it's not necessary to give the struct a name (we could have omitted the "complex" following "struct").

And …

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

MyFunc_t f;

// you must do its definition the long way

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