Numeric Conversions

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

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

Assigning a short to an int will always work, since all possible values of a short can be represented by an int. The reverse doesn’t always work, since there are many more values an int can take on than can be represented by a short.

A float can represent an int in the sense that the smallest and largest ints fall well within the range of the smallest (most negative) and largest floats. However, floats have few significant digits than do ints and thus, when converting from an int to a float, there may well be a loss of precision.

When converting from a float to an int there will not be any loss of precision, but large floats and small (most negative) floats cannot be represented by ints.
x's value will be 2, since the result of the (integer) division of i by j will be 0.
Implicit Conversions (2)

```c
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.
Here we do the int-to-float conversion explicitly; x's value will be 2.5.
“Coercion” is a commonly accepted term for one use of casts. “Intimidation” is not. The concept is more commonly known as a “sidecast”. Coercion means to convert something of one datatype to another. Intimidation (or sidecasting) means to treat an instance one datatype as being another datatype without doing any conversion of the actual data.
Quiz 1

• Will this work?

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

...

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

a) yes
b) no
The `void *` type is an exception to the rule that the type of the target of a pointer must be known.
Dereferencing a pointer must result in a value with a useful type. “void” is not a useful type.
Fun with Functions (1)

```c
void ArrayDouble(int A[], int len) {
    int i;
    for (i=0; i<len; i++)
        A[i] = 2*A[i];
}
```
Here `func` is declared to be a pointer to a function that takes an `int` as an argument and returns an `int`.

What’s the difference between a pointer to a function and a function? A pointer to a function is, of course, the address of the function. The function itself is the code comprising the function. Thus, strictly speaking, if `func` is the name assigned to a function, `func` really represents the address of the function. You might think that we should invoke the function 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 function as the third argument, without prepending “&”. 
Here we define another function that takes a single int and returns an int, and pass it to ArrayBop.
Can we write a version of `swap` that handles a variety of data types?
Note that there is a function 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”.

```c
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;
    }
}
```
Using Generic Swap

```c
short a=1, b=2;
gswap(&a, &b, sizeof(short));

int x=6, y=7;
gswap(&x, &y, sizeof(int));

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

- An array of 3 int *s
  
  ```c
  int *A[3];
  ```

- A func returning an int *, taking an int *
  
  ```c
  int *f(int *);
  ```

- A pointer to such a func
  
  ```c
  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

```c
int *f0(int *a) { *a += 1; return a; }
double *f1(double *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;
}
```

```bash
$ ./funcptr
2
$ 
```
Quiz 2

```c
int *f0(int *a) { *a += 1; return a; }
double *f1(double *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
  ```c
  pf[0] = (void (*)(void *))f0;
  ```
- Sometimes false
  ```c
  long f = 7;
  (void(*)(int))f(2);
  ```
Laziness ...

- Why type the declaration
  ```c
  void *(*f)(void *, void *);
  ```
- You could, instead, type
  ```c
  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;
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) {
    ...
}

Quiz 3

- What’s A?

```c
typedef double X_t[M];
X_t A[N];
```

- a) an array of N doubles
- b) an MxN array of doubles
- c) an NxM array of doubles
- d) a syntax error
CS 33

Data Representation

Part 1
If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide, where w is the number of bits in the word.
We might also want to interpret the contents of a computer word as a signed integer. There are a few options for how to do this. One straightforward approach is shown in the slide, where we use the high-order (leftmost) bit as the “sign bit”: 0 means positive and 1 means negative. However, this has the somewhat weird result that there are two representations of zero. This further means that the computer would have to have two implementations of arithmetic instructions: one for signed arithmetic, the other for unsigned arithmetic.
In ones' complement, a number is positive if its leftmost bit is zero negative otherwise. We negate a number by complementing all its bits. Thus if the leftmost bit is zero, a one in position $i$ of the remaining bits contributes a value of $2^i$ and a zero contributes nothing. But if the leftmost bit is one, a zero in position $i$ contributes a value of $-2^i$ and a one contributes nothing.

Note that the most-significant bit serves as the sign bit. But, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.
There’s only one zero!

Two’s complement is used on pretty much all of today’s computers to represent signed integers.

Note that the high-order (most-significant) bit represents $-2^{w-1}$. All the other bits represent positive numbers.