Some of this lecture is based on material prepared by Pascal Van Hentenryck.
Note, this has been fixed in Fortran, and, since C passes parameters by value, this has never been a problem in C.
The "%u" format code in `printf` means to interpret the item being printed as being unsigned. We'll explain this concept more thoroughly in an upcoming lecture. What's being printed is an address, which can't be negative.
C Pointers

- What is a C pointer?
  - a variable that holds an address
- Pointers in C are “typed” (remember the promises)
  - pointer to an int
  - pointer to a char
  - pointer to a float
  - pointer to <whatever you can define>
- C has a syntax to declare pointer types
  - things start to get complicated...

C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%u\n", p);
}
```

`$ ./a.out`

3221224352
C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%u\n", p);
}
```

Can you guess what &p is?

$ ./a.out
3221224352
C Pointers

• Pointers are typed
  – the type of the objects they point to is known
  – there is one exception (discussed later)

• Pointers are first-class citizens
  – they can be passed to functions
  – they can be stored in arrays and other data structures
  – they can be returned by functions
Swapping

What does this do?

```c
void swap(int *i, int *j) {
    int *tmp;
    tmp = j; j = i; i = tmp;
}

int main() {
    int a = 4;
    int b = 8;
    swap(&a, &b);
    printf("a:%d b:%d\n", a, b);
}
```

```bash
$ ./a.out
a:4 b:8
```
C Pointers

- Dereferencing pointers
  - accessing/modifying the value pointed to by a pointer

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    printf("%d\n", *p);
}
```

```
$ ./a.out
4
5
```
Dereferencing C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    *p += 3;
    printf("%d\n", a);
}
```

$ ./a.out
4
8

Note that “*p” and “a” refer to the same thing after p is assigned the address of a.
“x+=y” means the same as “x = x+y”. Similarly, there are -=, *=, and /= operators.
Swapping

```c
void swap(int *i, int *j) {
    int tmp;
    tmp = *j; *j = *i; *i = tmp;
}
int main() {
    int a = 4;
    int b = 8;
    swap(&a, &b);
    printf("a:%d b:%d\n", a, b);
}
```

$ ./a.out
a:8 b:4
Quiz 1

```c
int doubleit(int *p) {
    *p = 2*(p);
    return *p;
}
int main() {
    int a = 3;
    int b;
    b = doubleit(&a);
    printf("%d\n", a*b);
}
```

What's printed?

a) 0  
b) 12  
c) 18  
d) 36
The pointer `p` points to the first element of the array `a`. Thus `a[0]` and `*p` have identical values.
Adding one to a pointer, rather than increasing its value by one, causes it to refer to the next element. Thus if the size of what it refers to is 4 (which is the case for pointers to ints), adding one to the pointer increases its value by 4.
Pointer Arithmetic

Pointers can be incremented/decremented
– what this does depends on its type

```
int main() {
    int a[7];
    int *p;
    p = &a[0];
}
```

Now p and a have the same value
A pointer to the first element of an array can be used as if it were the array itself. Thus, in this example, there's little difference between how one uses “p” and “a”.

```c
int main() {
    int a[7];
    int *p;
    p = a;
    *p = 33;
    p[1] = 167;
}
```
Pointers and Arrays

\[ p = \&a[0]; \quad \text{can also be written as} \quad p = a; \]
\[ a[i]; \quad \text{really is} \quad *(a+i) \]

- **This makes sense, yet is weird and confusing ...**
  - \( p \) is of type int *\n    - it can be assigned to
      \[ \text{int} \; *q; \]
      \[ p = q; \]
  - a sort of behaves like an int *
    - but it can't be assigned to
      \[ = q \]
Pointers and Arrays

- An array name represents a pointer to the first element of the array
- Just like a literal represents its associated value
  - in:
    \[ x = y + 2; \]
    » “2” is a literal that represents the value 2
  - can’t do
    \[ 2 = x + y; \]
As we've already discussed, arguments to functions are passed by value – this means that the function receives a copy of the argument.
Arrays and Procedures

```c
int proc(int *a, int nelements) {
    // sizeof(a) == sizeof(int *)
    int i;
    for (i=0; i<nelements-1; i++)
        a[i+1] += a[i];
    return a[nelements-1];
}

int main() {
    int array[50] = ...;
    // sizeof(array) == 50*sizeof(int)
    printf("result = %d\n", proc(array, 50));
    return 0;
}
```

Note that the argument to proc is not the entire array, but the pointer to its first element. Thus $a$ is initialized by copying into it this pointer.
Note that one could include the size of the array (“\texttt{int proc(int a[50], int nelements)}”), but the size would be ignored, since it’s not relevant: arrays don’t know how big they are. Thus the \texttt{nelements} argument is very important.
Note how we initialize the contents of array $b$ in $proc$. 

```c
int proc(int a[], int nelements) {
    int b[5] = {0, 1, 2, 3, 4};
    a = b;
    return a[1];
}

int main() {
    int array[50];
    array[1] = 0;
    printf("result = %d\n",
           proc(array, 50));
    return 0;
}
```

This program prints:

a) 0  
b) 1  
c) 2  
d) nothing: it doesn’t compile because of a syntax error
int proc(int a[], int nelements) {
    int b[5] = {0, 1, 2, 3, 4};
    a = b;
    return a[1];
}

int main() {
    int array[5] = {4, 3, 2, 1, 0};
    proc(array, 50);
    printf("%d\n", array[1]);
    return 0;
}