Pointers

- Variables that store addresses
  - It points to somewhere in the process' virtual address space
  - `&foo` produces the virtual address of `foo`
- Generic definition: `type* name;` or `type *name;`
  - Recommended: do not define multiple pointers on same line: `int *p1, p2;` not the same as `int *p1, *p2;`
  - Instead, use:
```
int *p1;
int *p2;
```
- *Dereference* a pointer using the unary * operator
  - Access the memory referred to by a pointer

Arrays

- Definition: `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size` (*e.g.* `int scores[175];`)
  - Initially, array values are "garbage"
- Size of an array
  - Not stored anywhere – array does not know its own size!
    - `sizeof(array)` only works in variable scope of array definition
  - Recent versions of C (but *not* C++) allow for variable-length arrays
    - Uncommon and can be considered bad practice [*we won't use*]

```
int n = 175;
int scores[n];  // OK in C99
```


Poll Everywhere

pollev.com/cse33320su

- When run, what does this code print?

A. 2
B. 333
C. 999
D. A return address
E. I don't know

```
int main(){
  int64_t* ptr = foo();
  int64_t x = bar(2);
  printf("%d\n, *ptr);
}

int64_t* foo() {
  int64_t x = 333;
  x += bar(x);
  return &x;
}

int64_t bar(int64_t param){
  return param * 2;
}
```


Poll Everywhere

pollev.com/cse33320su

- The code snippets both use a variable-length array. What will happen when we compile with C99?
  - Vote at http://PollEv.com/cse33320su

```
int m = 175;
int scores[m];

void foo(int n) {
  ...
}
```

```
int m = 175;

void foo(int n) {
  int scores[n];
  ...
}
```

A.	Compiler Error	Compiler Error
B.	Compiler Error	No Error
C.	No Error	Compiler Error
D.	No Error	No Error
E.	We're lost…	

So what's the story for arrays?

- Is it call-by-value or call-by-reference?
- Technical answer: a T[ ] array parameter is "promoted" to a pointer of type T*, and the *pointer* is passed by value
  - So it acts like a call-by-reference array (if callee changes the array parameter elements it changes the caller's array)
  - But it's really a call-by-value pointer (the callee can change the pointer parameter to point to something else(!))
    - This is because **`T[i]`** is really **`*(T+i)`** . We aren't changing **`T`**!

```
void copyArray(int src[], int dst[], int size) {
  int i;
  dst = src;   // evil!
  for (i = 0; i < size; i++) {
    dst[i] = src[i];   // copies source array to itself!
  }
}
```