CSE 374 Lecture 10

(Week 4: C continued)

- Homework 2 due tonight
- Review previous lectures via website
- Office Hours This Week
  - Monday: Cynthia 11-12
  - Wednesday: Simon 4-5
  - Thursday: Dixon 3-4
  - Friday: Andrew 12-1
  - *More available upon request*
- Guest Lecture by Andrew on Friday: Debugging

```
// includes for functions & types
defined elsewhere
#include <stdio.h>
#include "localstuff.h"
// symbolic constants
#define MAGIC 42
// global variables (if any)
static int days_per_month[ ] = { 31,
28, 31, 30, …};
// function prototypes
// (to handle "declare before use")
 void some_later_function(char, int);
// function definitions
void do_this( ) { … }
char *return_that(char s[ ], int n)
{ … }
int main(int argc, char ** argv) { … }
```

Includes declarations & prototypes you might want to share.

Global variables & forward declarations go first.

Source File Structures

Stuff in function definitions is local to those functions

Pointer Review

Pointers point to an address in memory
`&x` returns the address

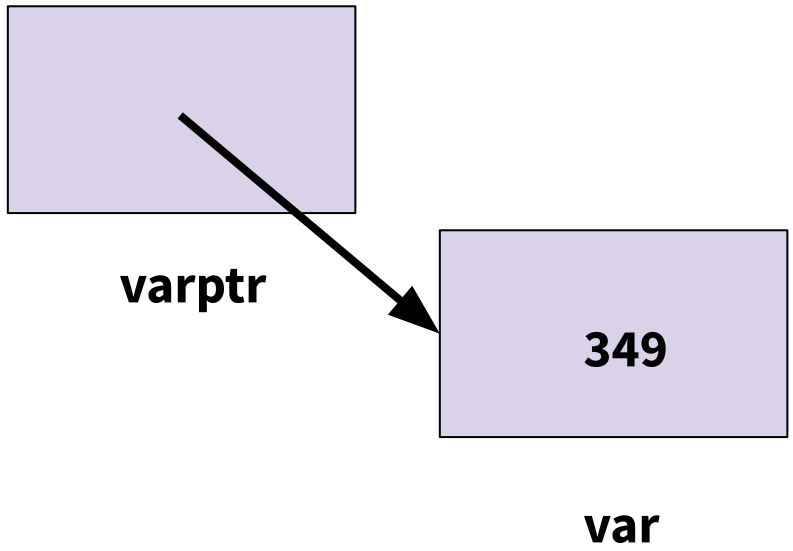
Declare a pointer to a pointer type and it has a specific type/size of memory:

`T *x;` or `T* x`; or `T * x`; or T*x
(T is a type, x is a variable)

An expression to dereference a pointer
`*x` (more generally *expression)

Arrays have an implicit pointer type
`T = x[n]` implies x is of type T*

```
int var = 349;
int *varptr = &var;
```



Pointers to pointers

Levels of pointers make sense:

I.e.: `argv, *argv, **argv`

Or: `argv, argv[0], argv[0][0]`

But

`&(&p)` doesn't make sense

```
void f(int x) {
    int*p = &x;
    int**q = &p;
    // x, p, *p, q, *q, **q
}
```

Integer, pointer to integer, pointer to pointer to integer

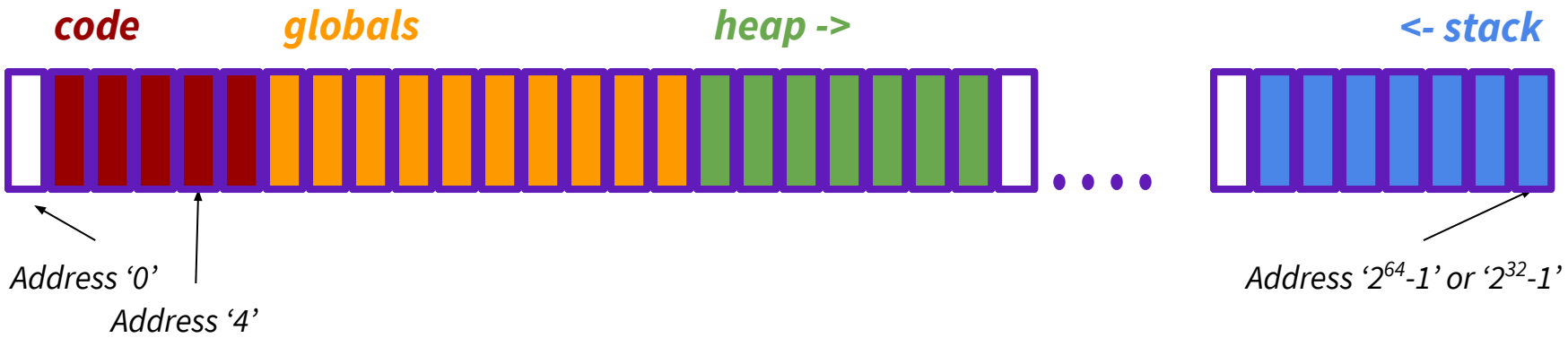
`&p` is the address of '`p`',

`&(&p)` would be the address of the address of `p`, but that value isn't stored separately anywhere and doesn't have an address

Try using `printf ("The address of x is %p\n", &x);`

Storage

- Variables need a place to live in memory
- Get ‘allocated’ a physical space in memory (with an address)
- Size of memory allocation depends on datatype
- Get ‘deallocated’ to release the space in memory

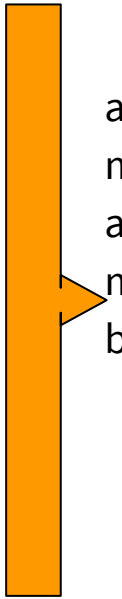


Scope

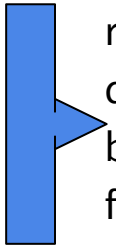
Variables may be accessed by the caller only at certain times - this is scope

Scope and storage are related, but not the same thing

- Global variables
  - Scope: entire program
  - Not desirable (violate encapsulation) But can be OK for truly global data like conversion tables, physical constants, etc.
- Static global variables
  - Scope: containing file
  - Static functions cannot be called from other files
- Static local variables
  - Scope: that function, rarely used
- Local variables (automatic)
  - Scope: that block – With recursion, multiple copies of same variable (one per stack frame/function activation)

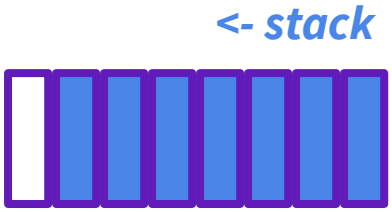


allocated before main, deallocated after main - memory in ‘global’ block



allocated “when reached” deallocated “after block” - memory in frame on stack

The stack



Stack stores active functions & local variables

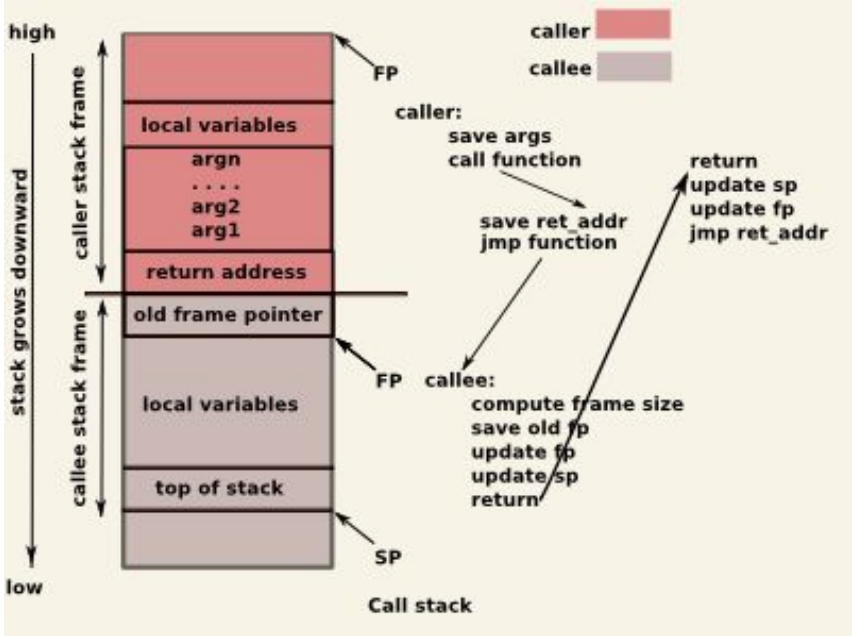
Frames deleted when function returns

Local variables do not persist

Local variables must have defined size

Can not make run-time adjustments

(Arrays must have length)



Local variable initialization

Memory allocation and initialization are not the same thing

Unlike Java, you MUST provide a value to initialize a bit of memory

It is possible to access un-initialized bits
unlike Java with sets defaults and checks for initialization
best case scenario: you crash

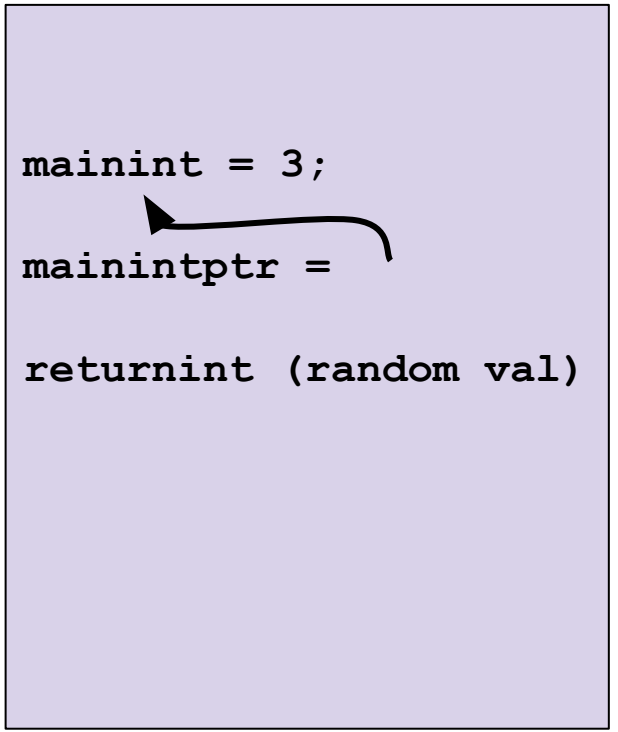
Arguments

Demo

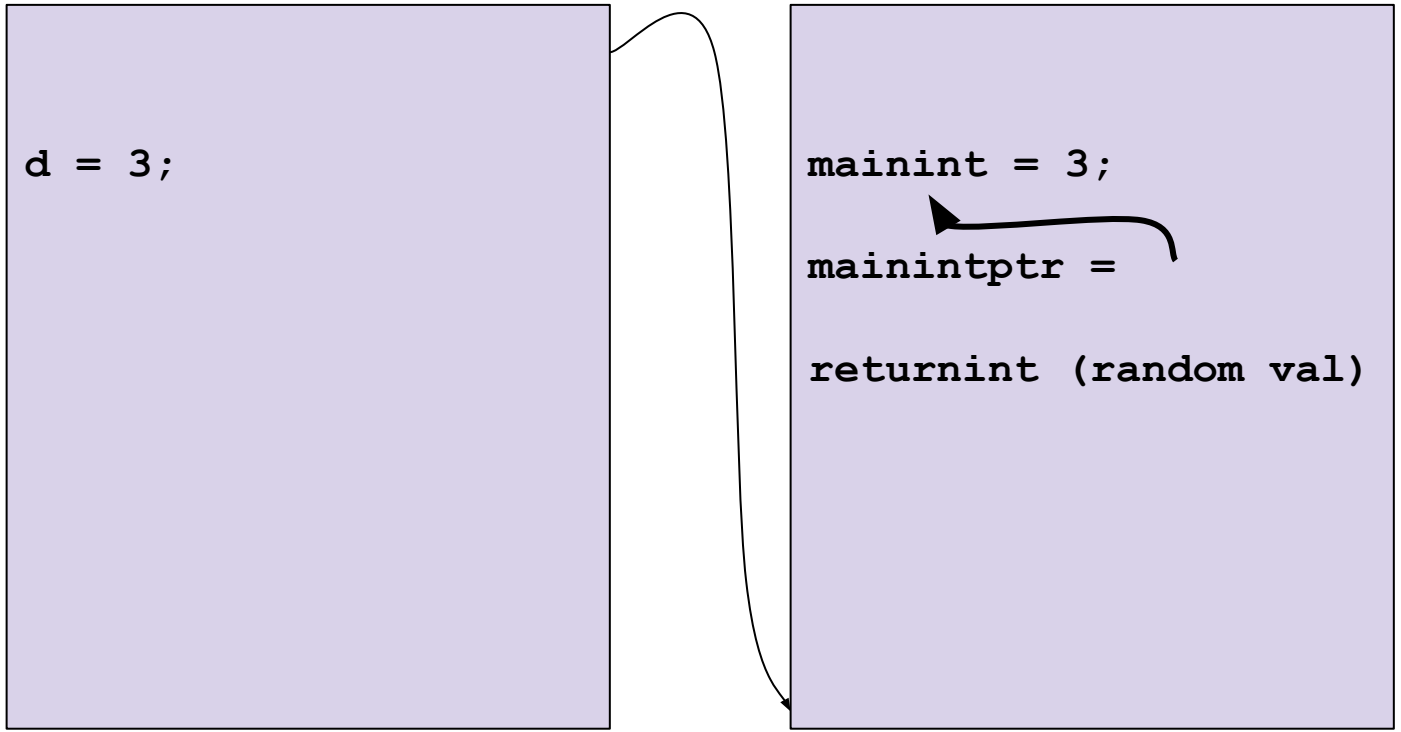
Storage allocation and variable scope is like local variables (i.e. space is part of the function frame added to the stack, and the variable may be used in the function).

All arguments passed by value. (i.e. a copy of the value is made and assigned to the variable.)

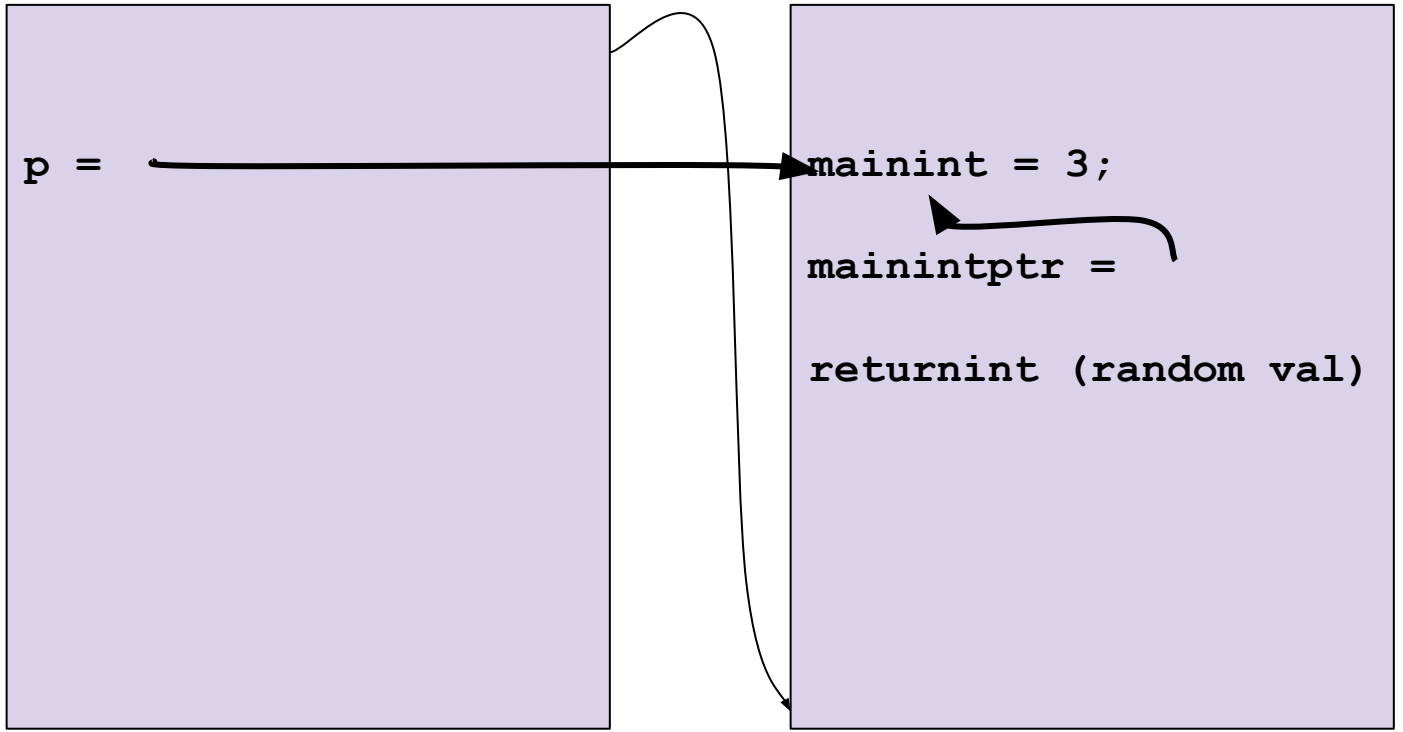
Stack (main)



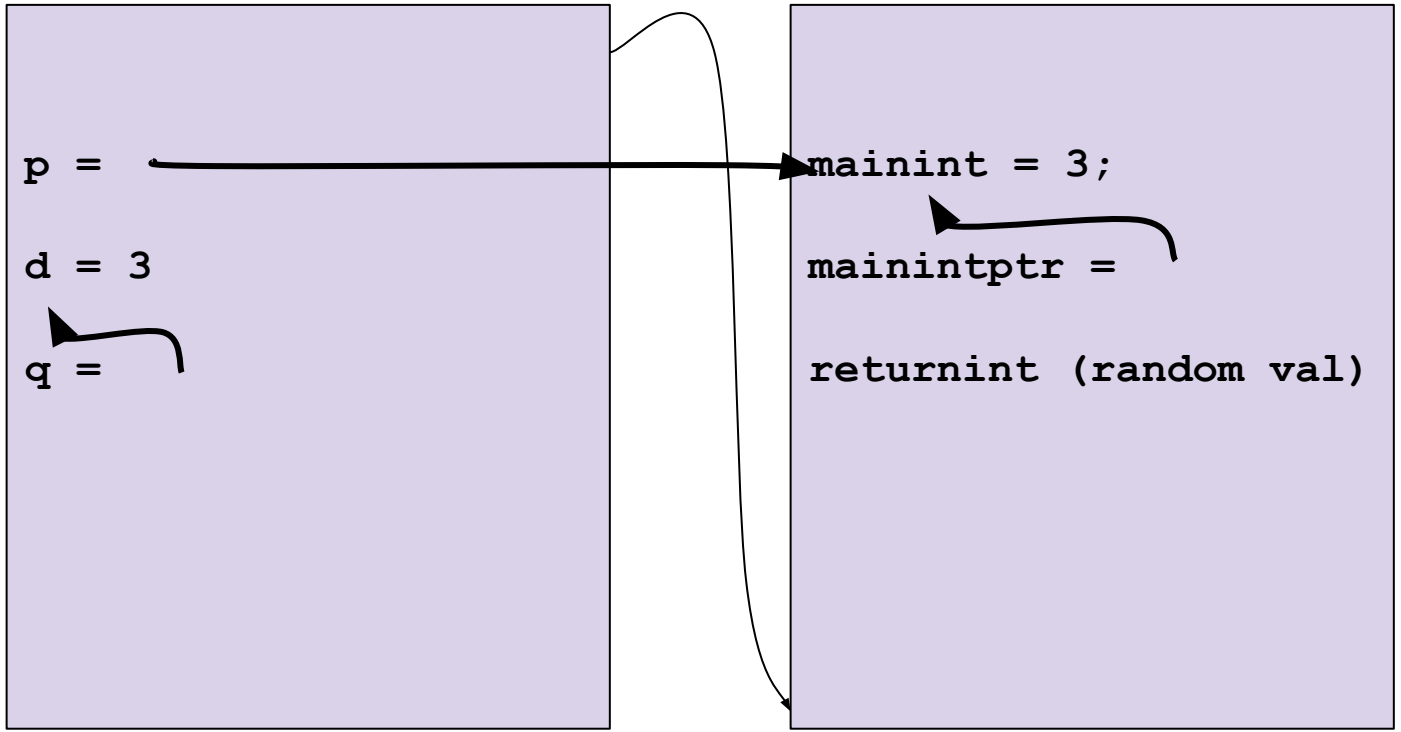
Stack (demoint)



Stack (dempointer)

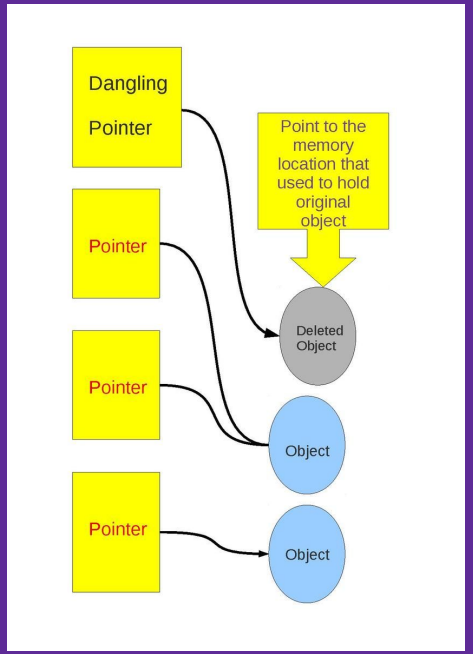


Stack (morepointer)



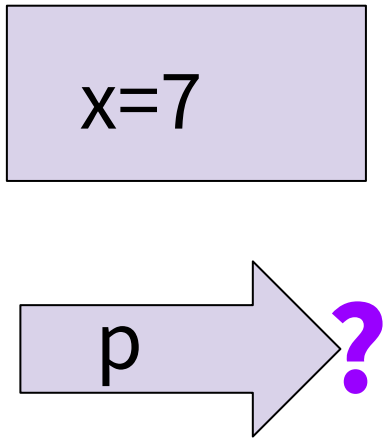
Dangling pointers

Pointers referring to memory that has been released (*Demo*)

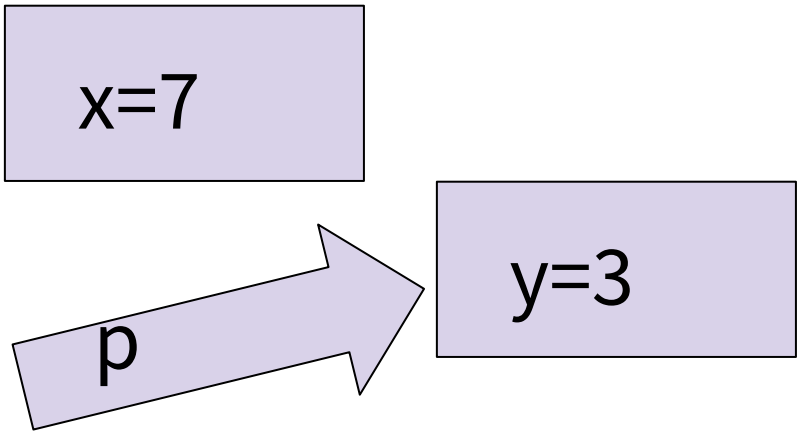


Garbage collecting languages (like Java) only delete memory that is unreachable to avoid this problem.

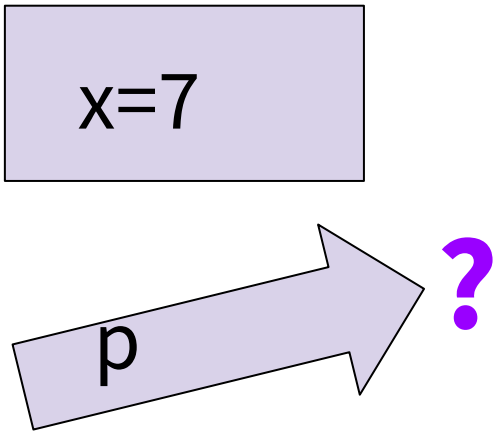
Dangling Pointers (line 4)



Dangling Pointers (line 7)



Dangling Pointers (line 13)



Arrays again

"A reference to an object of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T."

http://c-faq.com/aryptr/aryptrequiv.html

Right: x is the array, which decays to a pointer to an int and &x returns a pointer to the entire array.

```
void f1(int* p) {  // takes a pointer
  *p = 5;
}
int* f2() {
    int x[3];     // x on stack, is pointer
    x[0] = 5;
    (&x)[0] = 5; // address of x, points to
                 // same place but different T
    *x = 5;      // put value at location x
    *(x+0) = 5;  // Also put value at x
    f1(x);
    f1(&x);     // wrong - watch types!
    x = &x[2]; // No!  X isn't really a pointer
    int *p = &x[2];
    return x; // correct type, but is a
              // dangling pointer
}
```

Pointer arithmetic

- If p has type T* or T[] and *p has type T
- If p points to one item of type T, p+1 points to a place in memory for the next item of type T
  - So, p[0] is one item of type T, p+i = p[i]
- T[] always has type T*, even if it is declared as T[]
  - Implicit array promotion

    Result: Arrays are always passed by reference, not by value. (The information passed is the address of where the values are stored.)

https://chortle.ccsu.edu/CPuzzles/CPuzzlesMain.html