Intro to C: Pointers and Arrays
• Teaching Assistants: Let’s try that again.
• Lectures are recorded. Waitlist/Concurrent Enrollment may have to view recordings. But please assume you are in.
• My office hours: Monday 11-12, 424 SDH.
• People with university-related time conflict with lectures should contact the head GSIs.
• Let head GSIs know about exam conflicts by the end of this week.
Agenda

• Pointers
• Arrays in C
Address vs. Value

- Consider memory to be a single huge array
- Each cell of the array has an address associated with it
- Each cell also stores some value
- For addresses do we use signed or unsigned numbers? Negative address?!
- Don’t confuse the address referring to a memory location with the value stored there
Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable

![Diagram showing memory locations and addresses]
Pointer Syntax

• `int *p;`
  • Tells compiler that variable p is address of an int

• `p = &y;`
  • Tells compiler to assign address of y to p
  • & called the “address operator” in this context

• `z = *p;`
  • Tells compiler to assign value at address in p to z
  • * called the “dereference operator” in this context
Creating and Using Pointers

• How to create a pointer:
  & operator: get address of a variable

int *p, x;     x = 3;

p = &x;

• How get a value pointed to?
  "*" (dereference operator): get the value that the pointer points to

printf("p points to %d\n", *p);

Note the "*" gets used 2 different ways in this example. In the declaration to indicate that \textit{p} is going to be a pointer, and in the \texttt{printf} to get the value pointed to by \texttt{p}.
Using Pointer for Writes

• How to change a variable pointed to?
• Use the dereference operator * on left of assignment operator =

```c
*p = 5;
```
Pointers and Parameter Passing

• Java and C pass parameters “by value”: Procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original
  
  ```java
  void add_one (int x)
  {
    x = x + 1;
  }
  int y = 3;
  add_one(y);
  
  y remains equal to 3
  ```
• How can we get a function to change the value held in a variable?

```c
void add_one (int *p)
{
    *p = *p + 1;
}
int y = 3;

add_one(&y);
```

`y is now equal to 4`
Types of Pointers

- Pointers are used to point to any kind of data (int, char, a struct, etc.)
- Normally a pointer only points to one type (int, char, a struct, etc.).
  - void * is a type that can point to anything (generic pointer)
  - Use void * sparingly to help avoid program bugs, and security issues, and other bad things!
More C Pointer Dangers

• Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!

• Local variables in C are not initialized, they may contain anything (aka “garbage”)

• What does the following code do?

```c
void f()
{
    int *ptr;
    *ptr = 5;
}
```
typedef struct {
    int x;
    int y;
} Point;

Point p1;
Point p2;
Point *paddr;

/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too */
p1 = p2;
Pointers in C

• Why use pointers?
  • If we want to pass a large struct or array, it’s easier / faster / etc. to pass a pointer than the whole thing
  • In general, pointers allow cleaner, more compact code

• So what are the drawbacks?
  • Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    • Most problematic with dynamic memory management—coming up next week
    • Dangling references and memory leaks
Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn’t produce efficient code.
- Computers 25,000 times faster today, compilers better.
- C designed to let programmer say what they want code to do without compiler getting in way.
- Even give compilers hints which registers to use!
- Today’s compilers produce much better code, so may not need to use pointers in application code.
- Low-level system code still needs low-level access via pointers.
Video: Fun with Pointers

https://www.youtube.com/watch?v=6pmWojisM_E
void foo(int *x, int *y)
{
    int t;
    if ( *x > *y ) { t = *y; *y = *x; *x = t; }
}
int a=3, b=2, c=1;
foo(&a, &b);
foo(&b, &c);
foo(&a, &b);
printf("a=%d b=%d c=%d\n", a, b, c);

Result is:
A: a=3  b=2  c=1
B: a=1  b=2  c=3
C: a=1  b=3  c=2
D: a=3  b=3  c=3
E: a=1  b=1  c=1