CS 61C:
Great Ideas in Computer Architecture

C Pointers

Instructors:
Vladimir Stojanovic & Nicholas Weaver
http://inst.eecs.Berkeley.edu/~cs61c/sp16
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 of memory locations and pointers](image)
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;
  ```

  ```
  p = &x;
  ```

  ```
  x = 3;
  ```

  Note the “*” gets used 2 different ways in this example. In the declaration to indicate that \( p \) is going to be a pointer, and in the `printf` to get the value pointed to by \( p \).

• How get a value pointed to?

  “*” (dereference operator): get the value that the pointer points to

  ```
  printf("p points to %d\n",*p);
  ```
Using Pointer for Writes

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

![Diagram showing the change of value from 3 to 5 through the pointer p.]

*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
```
Pointers and Parameter Passing

• 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;
}
```
Pointers and Structures

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=6pmWojsM_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);

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
Administrivia

• HW0 out, due: Sunday 1/31 @ 11:59:59pm
• Give paper copy of mini-bio to your TA
• Get iClickers and register on bCourses! Participation points start today!
• People with *university-related time conflict* with lectures should contact the head GSIs. We will waive the clicker points but need to document conflict.
• Let head GSIs know about exam conflicts by the end of this week
Agenda

• Pointers
• Arrays in C
C Arrays

• Declaration:

  ```c
  int ar[2];
  ```

  declares a 2-element integer array: just a block of memory

  ```c
  int ar[] = {795, 635};
  ```

  declares and initializes a 2-element integer array
C Strings

• String in C is just an array of characters

```c
char string[] = "abc";
```

• How do you tell how long a string is?
  – Last character is followed by a 0 byte (aka “null terminator”)

```c
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++;
    return n;
}
```
Array Name / Pointer Duality

- **Key Concept**: Array variable is a “pointer” to the first (0\textsuperscript{th}) element
- So, array variables almost identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
- **Consequences**:
  - `ar` is an array variable, but works like a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - Can use pointer arithmetic to conveniently access arrays
C Arrays are Very Primitive

• An array in C does not know its own length, and its bounds are not checked!
  – Consequence: We can accidentally access off the end of an array
  – Consequence: We must pass the array and its size to any procedure that is going to manipulate it

• Segmentation faults and bus errors:
  – These are VERY difficult to find; be careful! (You’ll learn how to debug these in lab)
  – But also “fun” to exploit:
    • “Stack overflow exploit”, maliciously write off the end of an array on the stack
    • “Heap overflow exploit”, maliciously write off the end of an array on the heap
Use Defined Constants

- Array size $n$; want to access from 0 to $n-1$, so you should use counter AND utilize a variable for declaration & incrementation
  - Bad pattern
    ```c
    int i, ar[10];
    for(i = 0; i < 10; i++) { ... }
    ```
  - Better pattern
    ```c
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++) { ... }
    ```

- SINGLE SOURCE OF TRUTH
  - You’re utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: “Don’t Repeat Yourself”
  - And don’t forget the < rather than <=:
    When Nick took 60c, he lost a day to a “segfault in a malloc called by printf on large inputs”: Had a <= rather than a < in a single array initialization!
Pointing to Different Size Objects

- Modern machines are “byte-addressable”
  - Hardware’s memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes
sizeof() operator

• sizeof(type) returns number of bytes in object
  – But number of bits in a byte is not standardized
    • In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
• By definition, sizeof(char)==1
• Can take sizeof(arg), or sizeof(structtype)
• We’ll see more of sizeof when we look at dynamic memory management
Pointer Arithmetic

\[ \text{pointer} + \text{number} \quad \text{pointer} - \text{number} \]

e.g., \( \text{pointer} + 1 \) adds 1 \text{something} to a pointer

\[
\begin{array}{ll}
\text{char} & \ast p; \\
\text{char} & a; \\
\text{char} & b; \\
p & = & \&a; \\
p & += & 1;
\end{array}
\]

In each, \( p \) now points to \( b \)

(Assuming compiler doesn’t reorder variables in memory.

\textbf{Never code like this!!!!})

\[
\begin{array}{ll}
\text{int} & \ast p; \\
\text{int} & a; \\
\text{int} & b; \\
p & = & \&a; \\
p & += & 1;
\end{array}
\]

\textcolor{red}{\text{Pointer arithmetic should be used cautiously}}

\textcolor{red}{\text{Adds } 1 \ast \text{sizeof(char)} \text{ to the memory address}}

\textcolor{red}{\text{Adds } 1 \ast \text{sizeof(int)} \text{ to the memory address}}
Changing a Pointer Argument?

- What if want function to change a pointer?
- What gets printed?

```c
void inc_ptr(int *p)
{
    p = p + 1;
}

int A[3] = {50, 60, 70};
int* q = A;
inc_ptr(q);
printf("*q = %d\n", *q);
```

- *q = 50
- A q
- 50 60 70
• Solution! Pass a pointer to a pointer, declared as **h

• Now what gets printed?

```c
void inc_ptr(int **h)
{
    *h = *h + 1;
}

int A[3] = {50, 60, 70};
int* q = A;
inc_ptr(&q);
printf("*q = %d\n", *q);
```

And In Conclusion, ...

• All data is in memory
  – Each memory location has an address to use to refer to it and a value stored in it

• Pointer is a C version (abstraction) of a data address
  – * “follows” a pointer to its value
  – & gets the address of a value
  – Arrays and strings are implemented as variations on pointers

• C is an efficient language, but leaves safety to the programmer
  – Variables not automatically initialized
  – Use pointers with care: they are a common source of bugs in programs