Lecture 9: C Pointers

CSE 374: Intermediate Programming Concepts and Tools

Lecture Participation Poll #9

Log onto pollev.com/cse374
Or
Text CSE374 to 22333
Administrivia

HW2 due Thursday
HW3 getting posted later today - due Thursday Oct 28th
Reminder Midterm Friday Oct 29th
- last day of material for midterm Monday Oct 25th
- Bash + C programming
Where do computers store data?

- **CPU** – Central Processing Unit – computer circuitry that followed computer instructions in assembly

- **RAM** – Random Access Memory – a computer’s short-term memory where data is stored during program operation
  - When a program ends the memory in use “goes away”

- **Hard disc storage** – a computer’s long-term memory, this is where data is stored when you need to preserve it across re-starts.
  - Data is stored indefinitely
  - Can be modified by different processes
How do computers store data?

- Large sequences of numbers
  - Numbers are representations for electrical switches “transistors” that make up the brains of the CPU

- All data is binary – 1s and 0s
  - A single digit is called a “bit”
  - Bits come in groups of 8 called “bytes”
  - All instructions can be translated into sequences of binary

- Numbers represent other types of data
  - ASCII – each byte represents a letter of the English alphabet
  - Unicode – similar encoding structure to ASCII but covers a wider range of characters including non-English characters, emojis etc...
  - Images – represented by a 2D array of “pixels”
    - Each pixel is represented by 3 numbers: Red, Blue and Green values 0-255

---

<table>
<thead>
<tr>
<th>english</th>
<th>h</th>
<th>e</th>
<th>l</th>
<th>l</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>104</td>
<td>101</td>
<td>108</td>
<td>108</td>
<td>111</td>
</tr>
<tr>
<td>binary</td>
<td>01101000</td>
<td>01100101</td>
<td>01101100</td>
<td>01101100</td>
<td>01101111</td>
</tr>
</tbody>
</table>
Addresses in Memory

- Computer memory operates just like an array – addresses and the spaces they represent
  - Spaces are measured in "bytes" of 8 bits

- Each space in memory is referred to by its address
  - Value 504 stored at address 0x08
  - Address of value 504 stored at 0x38

- A pointer is a data object that holds an address
  - Addresses can point to any type of data because they simply point to any space in memory
  - Like a “contact” object that stores someone’s phone number, doesn’t store the actual person
  - Pointers are also stored in memory
  - Pointers can point to other pointers! <follow down the rabbit hole>
  - Pointers can either point to a single variable or an array
Pointers

Storing in memory an address to another location in memory

```c
int x = 4; // Variable called ‘x’ of type ‘int’ given value ‘4’
int *xPtr = &x; // Variable called ‘xPtr’ of type ‘int pointer’ given value ‘location of x’
int xCopy = *xPtr; // Variable called ‘xCopy’ of type ‘int’ given value ‘value found at address xPtr’
int* noPtr = NULL; // Variable called ‘noPtr; of type ‘int pointer’ given value of ‘null’
```
Pointer and Address Syntax in C

int *ptr; // also works! Programmer preference

int* ptr; // variable of type “pointer to int” without assignment

int x = 123; // an int variable called “x” that stores “123”

ptr = &x; // store the address of “x” in “ptr”

* Means “pointer to type”
  - * placed after type indicates a pointer data type
    - Similar in Java if you add [] after type you declare an array of that type
    - int* means “pointer to int”

& means “address variable”
  - Placing an & before a variable name will give you the address in memory of that variable
Dereferencing Pointers

```c
int x = 123;
int* ptr = &x;
*ptr = 456;
printf("new value of y:%d\n", *ptr);
```

- Placing a * before a pointer **dereferences** the pointer
  - Means “follow this pointer” to the actual data
  - *ptr = <data> will update the data stored at the address the pointer is referring to ie ‘write to memory’
  - *ptr will read the data stored at the address indicated by the pointer
  - Accessing unused addresses causes a ‘segmentation fault’

- A **dangling pointer** is one that points to a dead local variable
  - Data that is no longer in use
  - Dereferencing a dangling pointer is “undefined behavior” (UB)
  - UB means ANYTHING could happen
    - Program could crash(best case), silently fail(worst case)
    - GCC can catch this kind of error with a warning, but not always
Strings in C

char s1[] = {'c', 's', 'e', '\0'};
char s2[] = "cse";
char* s3 = "cse";

All are equivalent ways to define a string in C
There are no “strings” in C, only arrays of characters
- “null terminated array of characters”
char* is another way to refer to strings in C
- Technically is a pointer to the first char in the series of chars for the string

Strings cannot be concatenated in C
printf(“hello, “ + myName + “\n”); // will not work
Strings in C

char s1[] = {'c', 's', 'e', '\0'};
char s2[] = "cse";
char* s3 = "cse";

All are equivalent ways to define a string in C

There are no “strings” in C, only arrays of characters
- “null terminated array of characters”
char* is another way to refer to strings in C
- Technically is a pointer to the first char in the series of chars for the string

Strings cannot be concatenated in C

printf(“hello, “ + myName + “\n”); // will not work
Printf – print format function

- Produces string literals to stdout based on given string with format tags
  - Format tags are stand ins for where something should be inserted into the string literal
  - %s – string with null termination, %d – int, %f – float
  - Number of format tags should match number of arguments
    - Format tags will be replaced with arguments in given order

- Defined in stdio.h

- printf("format string %s", stringVariable);
  - Replaces %s with variable given
  - printf("hello, %s\n", myName);
Arrays

Contiguous blocks in memory
Declare as: `<datatype> arr[<len>]`
EX: `int* arrayOfInts = int arr[10];`
Stores location in memory of first value.
Does NOT store length, user must store and pass around separately.
Not automatically initialized to any value.
I/O: Printf, scanf

Printf and scanf are two I/O functions, prototyped in stdio.h

→ Printf (print-format)
→ int printf(const char *format, ...)
→ ‘Format’ is a string that can contain format tags
→ + additional arguments to match tags
→ Number of arguments better match number of %
→ Corresponding arguments better have the right types (%d, int; %f, float; %e, float (prints scientific); %s, \0- terminated char*; ... Compiler might check, but not guaranteed
  ◆ best case scenario: you crash
→ printf("%s: %d %g\n", p, y+9, 3.0)

→ scanf (gets input, formatted)
→ int scanf(const char *format, ...)
→ ‘Format’ is a string that can contain format tags
→ + additional arguments to match tags - should be pointers to the right data type so input can be stored in them
→ scanf("%d %s", &n, str);
→ scanf("%s %d", &a);
  ◆ %’s ignores string until space, then reads in an integer
Puzzle: What Prints?

```
#include <stdio.h>
void mystery(char *a, int *b, int c)
{
    int *d = b-1;
    c = *b + c;

    *b = c - *d;
    *d = c - *d;
    a[2] = a[b - d];
}

int main (int argc, char **argv)
{
    char ant[4] = "bed";
    int x[3];
    *x = 6;
    x[1] = 7;
    int y = 4;
    int *z = &y;
    *z = *x;
    printf("%d %d %d %s\n", *x, x[1], y, ant);
    mystery(ant, x+1, y);
    printf("%d %d %d %s\n", *x, x[1], y, ant);
}
```
Questions
#include <studio.h>
#include <stdlib.h>
#define EXIT_SUCCESS = 0;

int main (int argc, char** argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return EXIT_SUCCESS;
}