CSE 2421

C Pointers – Part 1

Recommended Reading: Pointers on C,
Beginning of Chapter 3 through Section 3.1.3

We're here!!

- POINTERS
  - At last, we arrive at THE MOST DREADED WORD in the lexicon of the C programming student. Pointers are indeed so dreaded that Java has completely done away with pointers and wrapped their functionality into the (admittedly safer) concept of *references*. C++, as a transitional step, has both pointers and references.

C language pointers

- What is covered in the next few slide decks is one of the main differences between C and other languages.
- It's quite possible that you already know some of this material. If so, it will be very easy to tune out.
- *Please try not to do so*.
- This material can go from boring to missing something crucial with respect to the understanding of these concepts at lightning speed.
- Interestingly enough, by paying close attention to this lecture and the next one, you may find the rest of the course to be mostly intuitive and easy to grasp.
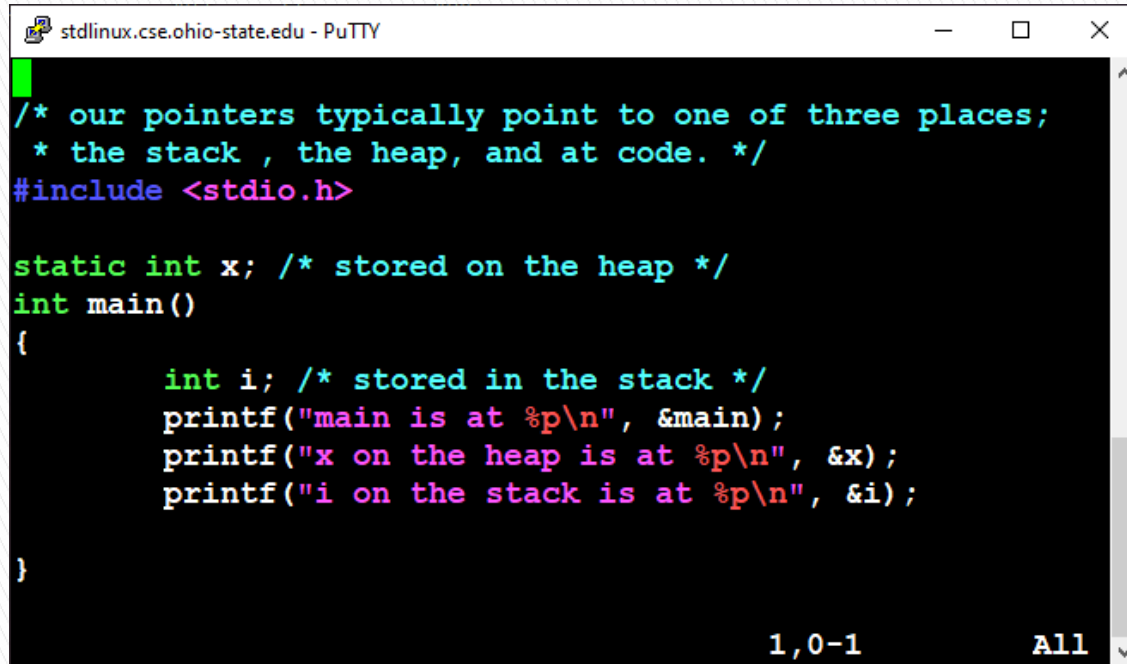
A Pointer Stores a Memory Address

▶ All variables have a memory address where they are stored.
- ◦ Variables have a name e.g. "a"
- ◦ Variables have an address "&a"
- ◦ We can store an address in a special kind of variable

▶ All buildings have a mailing address where they are located.
- ◦ Dreesee Labs is the name of a building
- ◦ 2015 Neil Avenue is the address of that building
- ◦ We can store that address on a mailing label

Variables, Addresses, and Values in Memory

- Values of variables are stored in memory, at a particular location (which is typically unknown to a high-level language programmer).

- A location in memory is identified and referenced with **an 8-byte address** (when on a 64-bit address space machine like stdlinux).

- This address is analogous to identifying a house's location via an address

- The "size" (e.g. char, int, etc.) can be thought to be analogous to the size of the lot the house is on.

- This address can also be called *a reference* (but usually is not in C terminology).

Actual Addresses on stdlinux

```
stdlinux.cse.ohio-state.edu - PuTTY                    —  □  ×

/* our pointers typically point to one of three places;
 * the stack , the heap, and at code. */
#include <stdio.h>

static int x; /* stored on the heap */
int main()
{
        int i; /* stored in the stack */
        printf("main is at %p\n", &main);
        printf("x on the heap is at %p\n", &x);
        printf("i on the stack is at %p\n", &i);

}

                                        1,0-1          All
```

main is at
$\qquad$ 0x40052d

x on the heap is at
$\qquad$ 0x601038

i on the stack is at
$\qquad$ 0x7ffdc23e838c

Code to print addresses

Typical addresses for various areas in memory

Address Versus Contents (Value)

Address (hex)	0x100	0x104	0x108	0x10C	0x110
Value (decimal)	112	08	260	00	883

- The above table contains 5 32-bit integer values
- Assuming these were values in our program, remembering where they are in memory, and using addresses to access them, would be cumbersome and error-prone
- That's why we typically use identifiers for variables to associate a name with a memory location where the variable is stored; the compiler and assembler take care of mapping identifiers to addresses, which frees programmers from this burden.

Another way to access data - pointers

▶ We can access values of variables using identifiers which name them in C, and we often do that.

▶ In C, it is also possible to access the value of a variable in memory using *another variable* (can also be a constant) which holds *the 8-byte* address* of the first variable.

▶ The second type of variable mentioned above is **a pointer.**

▶ Some data in C programs can *only* be accessed with pointers (for example, elements of arrays, or characters in strings (more on this later)).

*adddresses are 8-bytes on 64-bit computers, 4-bytes on 32-bit computers

Implications of the Pointer type

- Remember, at the lowest level, data is always just **a bit pattern**; a given bit pattern can have *multiple interpretations*, depending on which type of data it encodes.

- In the CPU (Central Processing Unit), instructions which perform a given type of operation on data of different types are ***different instructions*** (for example, integer addition and floating point addition are performed by different instructions).

- When the compiler generates instructions for the CPU to access a value in memory, it does not make *any assumptions* about how to interpret that value (the bit pattern). You must **explicitly** tell the compiler the type of value stored there (with type in a declaration or with a cast), so that it can generate the appropriate type of instruction for the CPU.

Implications of the Pointer type (cont)

- When using pointers (addresses), the compiler chooses machine instructions to generate for the CPU to execute based upon the type you declared that the pointer points to.

- Another way to think about this is to say, from the compiler's perspective, it is not enough to know *an address* (or even a variable name) to access data.

- The compiler will always ask the question: What **type** of data is stored at this address (or in this variable)?

- Your code must answer this question for the compiler (with a declaration, cast, or both), or it will give you warnings or errors, and, perhaps, wrong information. (The different format strings used by scanf exist for this.)

Address Contents

```
int A = 112;
printf("%d\n", A);   /* Prints 112, as expected */
printf("%f\n", A);   /* Compiles and runs, but */
                     /* strange result: 0.000000 */
```

Variable Name	A	B	C	D	E
Address	0x100	0x104	0x108	0x10C	0x110
Value	112	08	104	00	883

If not using memory in the way it was declared, the compiler *may* attempt to protect you and throw a warning or an error, but not always.

We should explicitly cast (better) to use a value as a non-declared type or make sure that the compiler will implicitly cast correctly.

Address Contents

Variable Name	A	B	C	D	E
Address (hex)	0x100	0x104	0x108	0x10C	0x110
Value (Decimal)	112	08	260 (x104)	00	883

```
int A = 112;
int B = 8;
int C = 260;          /* = 0x104;          */
int D = 0;
int E = 883;

printf("%d\n", (unsigned int) A); /* 112 */
printf("%f\n", (float) A);        /* 112.00000 */
```

These will both work!

You must tell the compiler how to generate instructions to interpret memory when you want to use a value in a way different from the way in which in was declared.

Pointer definition

- A pointer is a variable (usually) or a constant that contains the address of (that is, a reference to) another variable or of a function or of some generic place in memory

- The type of a pointer is: pointer to the type of data to which it points.

- So, for example, the type of a pointer which points to an integer is pointer to integer.

Pointers - Two related operators

- *** (the unary *dereference operator*)** is used in the declaration of a pointer type

  - int *p; This means variable p is a pointer that points to an integer.

  - Note: *p is read "dereference p" (when one wants to speak of it properly), and means "what is pointed to by p." Thus, the declaration says that what is pointed to by p is an integer; therefore, p is *a pointer to an integer* (holds the address of an integer), or we can say, *an int pointer*. I've heard many refer to *p as "star p" or "splat p" or "the value at p".

  - The dereference operator can be used to access data by **indirection**. When we use a pointer to access a value rather than accessing the value directly through a variable which holds that value, we access it *indirectly* through the pointer variable which holds the address of the value (more on this below).

Pointers - Two related operators (cont.)

- **& (the unary)** *address operator* gives the "address of" a piece of data; this is **always** *a constant.* The constant that represents the address is determined by the compiler. Suppose we have:

      int *p;
      int c = 10;

      p = &c;   /* This statement assigns to the variable p the */
                /* address of c. So, p points to c   */

Example

Variable Name	A	B	C	D	E
Address	x100	x104	x108	x10C	x110
Value	112	08	260 (x104)	00	883

```
int B = 8;
int *C;      /* Declare C to be a pointer to int */
C = &B;      /* and assign it the address of B  */
```

Now, C is an integer pointer that points to B, so we can access B either through its name (identifier), or, using *indirection*, through C (See next slide).

Access using indirection

- Suppose we wanted to print the value of B.
- We can access the value using the identifier B, or using indirection, with *C:

```
printf("%i", B);
printf("%i", *C);
```

Both of these statements output the value of B; the second one accesses the value using indirection.

Pointers cont.

- Every pointer points to a specific data type
    - An exception is *a void pointer* (a generic pointer); a pointer to void holds the address of a value of *any type,* but can't be dereferenced (i.e. cannot be used to get the "contents of" another memory location through indirection) without casting. We will learn more about the use of pointers to void soon.
- PLEASE do not say "this is a pointer in my program." Instead, say "this is a pointer to an integer" or "this is a pointer to a float", or "this is a pointer to void", etc.
    - This is not just being picky. Making use of pointers in C programs without all kinds of strange errors requires **always** paying attention to the type of data to which the pointer points!
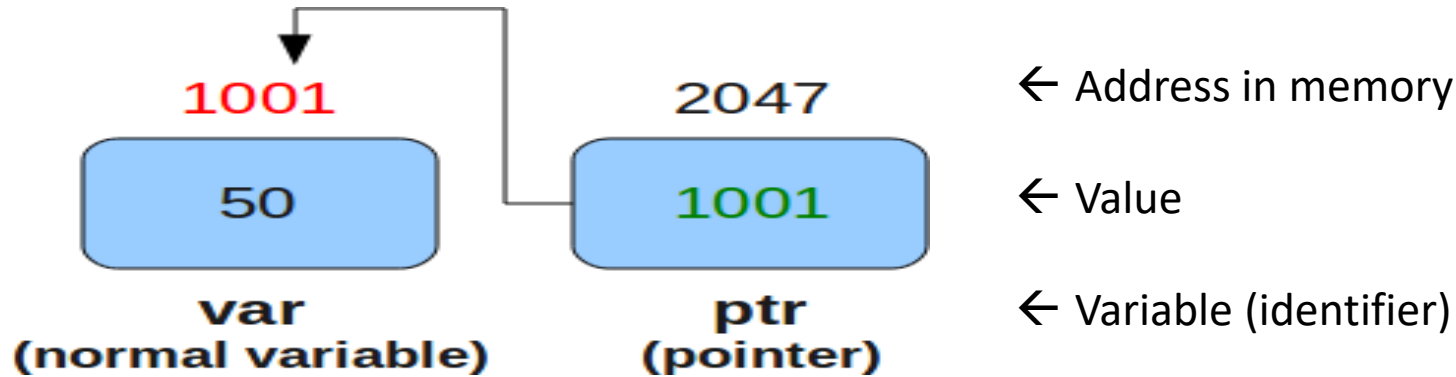
Examples:
```
unsigned int *p;    /* p is a pointer to unsigned int*/
char *c;            /* c is a char pointer */
void *x;            /* x is a void pointer */
int **y;            /* y is a pointer to an integer pointer */
```

Pointers Defined cont.

- unsigned int *x;
- Read as: "declare x as a pointer to an unsigned integer"
- Interpretation: declare x as a variable that holds the numeric (8-byte) address of a location in memory at which bits are stored that we intend to manipulate as an unsigned (4-byte) integer

Pointer introduction

1001 ← Address in memory

50

← Value 1001

var
(normal variable)

ptr
(pointer)

2047

← Variable (identifier)

```
char var = 50, var2 = 0;  /* access var directly – through its name  */
char *ptr;        /*  declare ptr to be a pointer to char */
ptr = &var;     /*  *ptr points to var   (i.e. ptr contains 1001 ) */
*ptr = 1;        /*  Access var using indirection, that is, through ptr  */
                 /*  so var now equals what?                      */
var2 = *ptr;   /*  Access var using indirection, that is, through ptr  */

What value does the variable var2 have?  It is set to 1!
```

Pointers Cont.

- * (unary, not the arithmetic operator) is a dereferencing operator when applied to pointers
  - When applied to a pointer, it accesses the data (i.e., the bits in memory) the pointer points to
  - * in front of a pointer variable means "get (or set) the value at that address" i.e. "contents of" (what the pointer points to)
    - "get" if it is on the right side of an equal sign
    - "set" if it is on the left side of an equal sign

  - Reading data through indirection:
    int a;
    int b = 25;
    int *p;
    p = &b;
    a = *p; means get the value at the address stored in p and assign it to a

  - Writing data through indirection:
    *p = 12; means set the 4 bytes (because p is an integer *) starting at the address stored in p to the value 12

Declaring Pointers

- int* ptr_a;
- int *ptr_a;   /* far preferred */

- Both are valid, but the first style leads to mistakes:
  - int*   ptr_b, ptr_c, ptr_d
    - ptr_b is a pointer to integer but ptr_c and ptr_d are integers

  - int*ptr_b, *ptr_c, *ptr_d
    - 3 pointers to integer are declared here

- Generally, the operand of the dereference operator is *the identifier or expression which follows*, so there **must** be a separate dereference operator for each pointer variable you wish to declare.

Pointers cont.

- Example: y = *int_ptr + 1 takes whatever int_ptr points at, adds 1, and assigns the result to y
- Other ways to increment by 1:
  - *int_ptr += 1 ➔ *int_ptr = *int_ptr + 1
  - ++*int_ptr
  - (*int_ptr)++
    - The parentheses are necessary; without them, the expression would increment int_ptr (so that it points to the following address in memory) instead of what it points to, because post-fix increment has higher precedence than the dereference operator, *, so without parentheses, the compiler will treat the expression as *(int_ptr++)

Pointers cont.

- Pointers are variables so they can be used without dereferencing.
  - Example:
    - int x, *iq, *ip=&x;
      /*declares 3 variables, 2 of which are pointers to integers */

      iq = ip;
      /* Copies the contents of ip (an address) into iq,
         making iq point to whatever ip points to */

    - **IMPORTANT NOTE:**
      int x;
      int *ip = &x;

      is equivalent to:

      int x;
      int *ip;
      ip = &x; /* &x is assigned to ip, *NOT* *ip */

Pointers to Pointers

- int **x;   /* assume integer is 32 bits */
- Read as: declare x as a pointer to a pointer to an integer (or a pointer to an integer pointer)
- Interpretation: Declare x as an 8-byte variable that holds the numeric address at which is another 8-byte numeric address at which are 32 bits (4-bytes) that we intend to manipulate as a signed integer

Pointer Casting

- int i = 5;
- char *y = (char *)(&i);
- Declares i as an integer and puts 5 in that location, then declares y as a pointer to a character and assigns to y the address of i cast to (interpreted as) a pointer to a character.
- &i and y are both the same numeric value pointing to the same piece of memory
- Dereferencing y without casting hereafter generates instructions that operate on chars at i's memory address instead of integers
- Note that, in casts, the dereference operator *follows* the type name: (int *) OR (float *) OR (char *) etc.

What did that say????

- int i= 5;
- char *y = (char*)(&i);
- Really? Both of these point to the value 5???
- At least on *SOME* machines. CSE servers are ones where it will be true because CSE servers use "little endian" byte ordering. I don't have access to a "big endian" server to test.
- Read Section 2.1.3 in Bryant/O'Hallaron
- If you want to do this type of thing, better to test the result on your server to insure your code does what you think it is doing.
- We'll work more with "endian" later in the semester.

Check out this test program

```c
#include <stdio.h>
int main()
{
    int i = 0x02030405;
    char *y;
    y = (char*)(&i);

    printf("The address of i is %p\n", &i);
    printf("  the value of y is %p\n", y);

    printf(" y   points to the value %i\n", *y);
    printf(" y+1 points to the value %i\n", *(y+1));
    printf(" y+2 points to the value %i\n", *(y+2));
    printf(" y+3 points to the value %i\n", *(y+3));

    return(0);
}
```

Output is:

```
[kirby.249@cse-sl4 slides]$ endian
The address of i is 0x7ffc31a67684
  the value of y is 0x7ffc31a67684
 y   points to the value 5
 y+1 points to the value 4
 y+2 points to the value 3
 y+3 points to the value 2
[kirby.249@cse-sl4 slides]$
```

Pointers … more!

- Every pointer points to a specific data type.
    - The only exception: "pointer to void" is used to hold the address of any type but cannot be dereferenced without a cast (more later)
- If ip points to the integer x, (ip holds &x) then *ip can occur in any context where x could
    - Example: *ip = *ip + 10 is equivalent to x=x+10; This increments the value stored at the address in ip by 10
- The unary operators * and & have higher precedence than arithmetic operators

RECAP

- ## In a declaration
  - ◦ * says "I am a pointer" that points to a certain type of value
- ## In a statement
  - ◦ & means "get the address of a variable"
  - ◦ * means "access (get/read or set/write) the value at the address stored in the variable which follows the dereference operator"

RECAP – Pointer Rules

- No matter how complex a pointer structure gets, the list of rules remains short:
    - A pointer stores a reference to its pointee. The pointee, in turn, stores something useful.
    - The dereference operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been assigned to refer to a pointee. Most pointer bugs involve violating this one rule.
    - Allocating a pointer does not automatically assign it to refer to a pointee. Assigning the pointer to refer to a specific pointee is a separate operation which is easy to forget.
    - Assignment between two pointers makes them refer to the same pointee which introduces sharing.

- NOTE: A "pointee" is a variable whose address is assigned to be the value of a pointer.