

# ECE 2400 Computer Systems Programming

## Spring 2025

### Topic 4: C Pointers

School of Electrical and Computer Engineering  
Cornell University

revision: 2025-02-10-10-37

<b>1</b>	<b>Pointer Basics</b>	<b>2</b>
<b>2</b>	<b>Call by Value vs. Call by Pointer</b>	<b>4</b>
<b>3</b>	<b>Mapping Conceptual Storage to Machine Memory</b>	<b>7</b>
<b>4</b>	<b>Pointers to Other Types</b>	<b>10</b>
4.1.	Pointers to struct . . . . .	10
4.2.	Pointers to Nothing . . . . .	11
4.3.	Pointers to Pointers . . . . .	13

**zyBooks** The zyBooks logo is used to indicate additional readings and coding labs included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

Copyright © 2025 Anne Bracy. All rights reserved. This handout was prepared by Prof. Anne Bracy at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming (derived from previous handouts prepared and copyrighted by Prof. Christopher Batten). Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

- Pointers refer to the **location** (or **address**) of a variable
- A variable can now “point” to another variable
- Programmers can (1) access what a pointer points to and (2) change a pointer to point to something else
- This is an example of **indirection**, a powerful programming concept

### 1. Pointer Basics

- Pointers require introducing **new types** and **new operators**
- Every type T has a corresponding pointer type T\*
- A variable of type T\* contains a pointer to a variable of type T

```
1 int*   a_ptr;    // pointer to a variable of type int
2 char*  b_ptr;    // pointer to a variable of type char
3 float* c_ptr;    // pointer to a variable of type float
```

- The **address-of** operator (&) evaluates to the location of a variable
- The address-of operator is used to initialize/assign to pointers

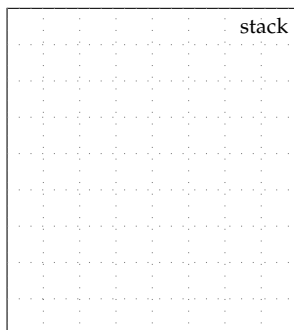
```
1 int  a;          // variable of type int
2 int* a_ptr;      // pointer to a variable of type int
3 a_ptr = &a;      // assign location of a to a_ptr
```

- The **dereference** operator (\*) evaluates to the value of the variable the pointer points to

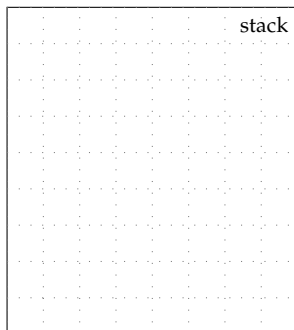
```
1 int  b = 42;      // initialize variable of type int to 42
2 int* b_ptr = &b;  // pointer to a variable of type int
3 int  c = *b_ptr;  // initialize c with what b_ptr points to
```

**Example declaring, initializing,  
RHS dereferencing pointers**

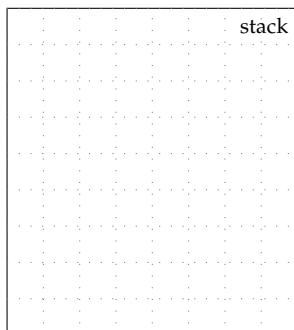
```
□□□ 01 int a = 3;
□□□ 02 int* a_ptr;
□□□ 03 a_ptr = &a;
□□□ 04
□□□ 05 int b = 2;
□□□ 06 int c = b + (*a_ptr);
```

**Example illustrating aliasing**

```
□□□ 01 int a = 3;
□□□ 02 int* a_ptr0 = &a;
□□□ 03 int* a_ptr1 = a_ptr0;
□□□ 04 int c = (*a_ptr0) + (*a_ptr1);
```

**Example declaring, initializing,  
LHS dereferencing pointers**

```
□□□ 01 int a = 3;
□□□ 02 int b = 2;
□□□ 03
□□□ 04 int c;
□□□ 05 int* c_ptr = &c;
□□□ 06 *c_ptr = a + b;
```

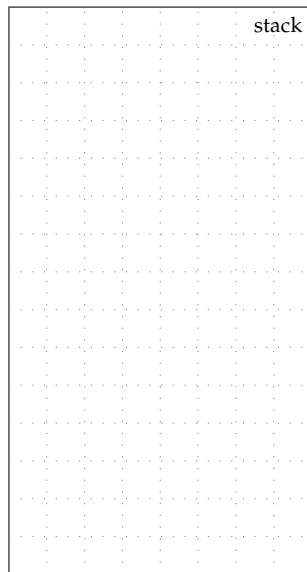


- Be careful – three very different uses of the \* symbol!
  - Multiplication operator     `int a = b * c;`
  - Pointer type                `int* d = &a;`
  - Dereference operator      `int e = *d;`

## 2. Call by Value vs. Call by Pointer

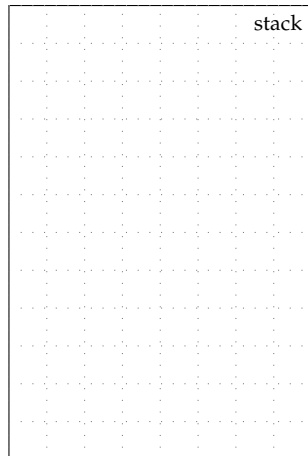
- So far, we have always used **call by value**
- Call by value *copies* values into parameters
- Changes to parameters by callee *are not seen* by caller

```
01 void swap( int x, int y )
02 {
03     int temp = x;
04     x = y;
05     y = temp;
06 }
07
08 int main( void )
09 {
10     int a = 9;
11     int b = 5;
12     swap( a, b );
13     return 0;
14 }
```



- **Call by pointer** uses pointers as parameters
- Callee can read and modify parameters by dereferencing pointers
- Changes to parameters by callee *are seen* by caller

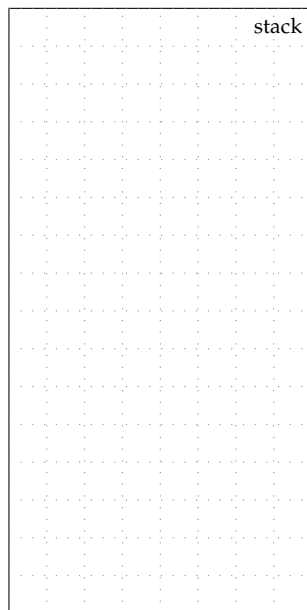
```
01 void swap( int* x_ptr,  
02           int* y_ptr )  
03 {  
04     int temp = *x_ptr;  
05     *x_ptr   = *y_ptr;  
06     *y_ptr   = temp;  
07 }  
08  
09 int main( void )  
10 {  
11     int a = 9;  
12     int b = 5;  
13     swap( &a, &b );  
14     return 0;  
15 }
```



<https://tinyurl.com/zybookch4>

**Draw a state diagram corresponding to the execution of this program**

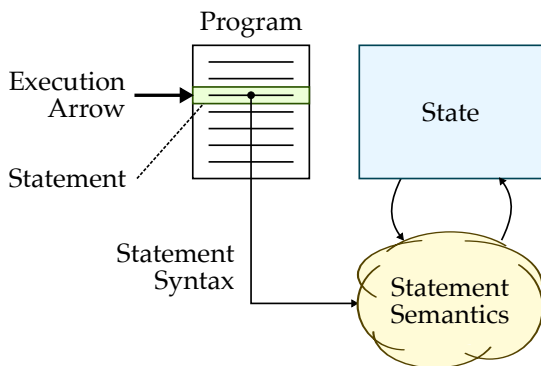
```
01 void avg( int* result_ptr,  
02           int x, int y )  
03 {  
04     int sum = x + y;  
05     *result_ptr = sum / 2;  
06 }  
07  
08 int main( void )  
09 {  
10     int a = 10;  
11     int b = 20;  
12     int c;  
13     avg( &c, a, b );  
14     return 0;  
15 }
```



**zyBooks** The course zyBook includes a coding lab to implement an in-place square function that uses call-by-pointer.

### 3. Mapping Conceptual Storage to Machine Memory

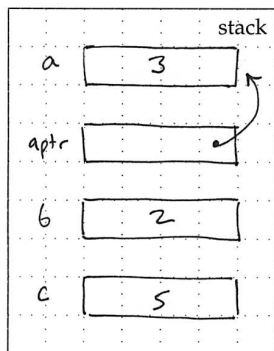
- Our current use of state diagrams is conceptual
- Real machine uses **memory** to store variables
- Real machine does not use “arrows”, uses **memory addresses**



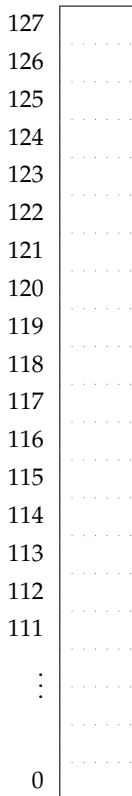
### 3. Mapping Conceptual Storage to Machine Memory

- Can visualize memory using a “byte” or “word” view
- Stack stored at high addresses, stack grows “down”
- As a simplification, assume we only have 128 bytes of memory

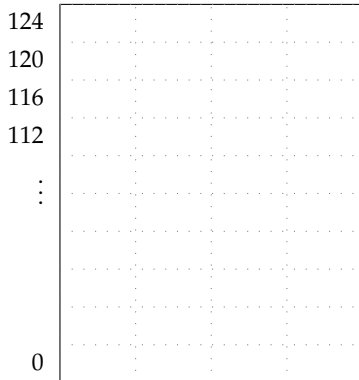
```
01 int a = 3;
02 int* a_ptr;
03 a_ptr = &a;
04
05 int b = 2;
06 int c;
07 c = b + (*a_ptr);
```



**Memory**  
(byte addr)



**Memory**  
(4B word addr)





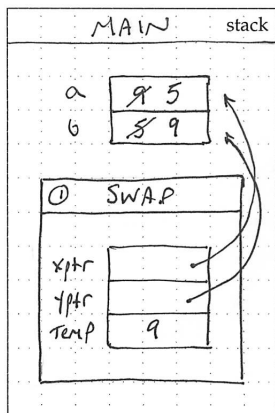
### 3. Mapping Conceptual Storage to Machine Memory

- Both code and stack are stored in 128 bytes of memory
- Stack stored at high addresses, stack grows “down”
- Code stored at low addresses, execution moves “up”
- **Stack Frame**: collection of data on the stack associated with function call including return value, return addr, parameters, local variables

```

0001 void swap( int* x_ptr, int* y_ptr )
0002 {
0003     int temp = *x_ptr;
0004     *x_ptr   = *y_ptr;
0005     *y_ptr   = temp;
0006 }
0007
0008 int main( void )
0009 {
0010     int a = 9;
0011     int b = 5;
0012     swap( &a, &b );
0013     return 0;
0014 }

```



**Memory**  
(4B word addr)

$$\begin{array}{c} 124 \\ 120 \\ 116 \\ 112 \\ 108 \\ 104 \\ \vdots \end{array}$$

36

32

28

24

20

16

12

8

4

0

```
*y_ptr = temp;
*x_ptr = *y_ptr;
int temp = *x_ptr;

...
return 0;
call swap
int b = 5;
int a = 9;
```

## 4. Pointers to Other Types

In addition to pointing to primitive types, pointers can also point to other pointers, to structs, or even functions.

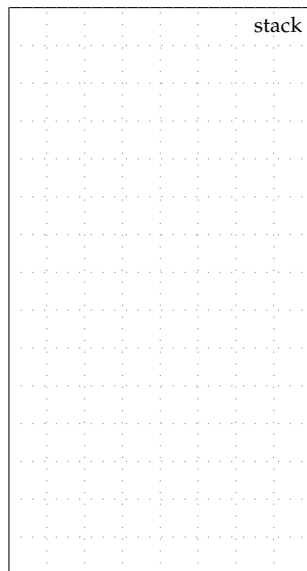
### 4.1. Pointers to struct

- Pointer to a struct is declared exactly as what we have already seen
- Be careful to dereference the pointer first, then access a field

```

01 typedef struct _node_t
02 {
03     int          value;
04     struct _node_t* next_p;
05 }
06 node_t;
07
08 int main( void )
09 {
10     // First node
11     node_t node0;
12     node0.value = 3;
13
14     node_t* head_p = &node0;
15     (*head_p).value = 4;
16
17     // Second node
18     node_t node1;
19     node1.value = 5;
20     node1.next_p = &node0;
21
22     head_p = &node1;
23     (*head_p).value = 6;
24     ((*head_p).next_p).value = 7;
25
26     return 0;
27 }

```



- C provides the arrow operator ( $\rightarrow$ ) as syntactic sugar
- $a \rightarrow b$  is equivalent to  $(*a).b$

```
1  int main( void )
2  {
3      ...
4
5      node_t* head_p = &node0;
6      head_p->value  = 4;
7
8      ...
9
10     head_p = &node1;
11     head_p->value      = 6;
12     head_p->next_p->value = 7;
13 }
```

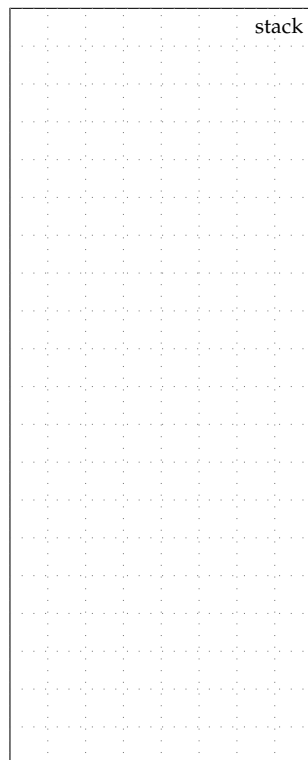
## 4.2. Pointers to Nothing

- NULL is defined in `stddef.h` to be a pointer to nothing
- NULL can be used to indicate “there is no answer” or “error”
- Simply write NULL in state diagrams
- In previous example, NULL can mean there is no next node

```

0000 01 #include <stddef.h>
0000 02
0000 03 typedef struct _node_t
0000 04 {
0000 05     int          value;
0000 06     struct _node_t* next_p;
0000 07 }
0000 08 node_t;
0000 09
0000 10 int main( void )
0000 11 {
0000 12     node_t node0;
0000 13     node0.value = 3;
0000 14     node0.next_p = NULL;
0000 15
0000 16     node_t node1;
0000 17     node1.value = 4;
0000 18     node1.next_p = &node0;
0000 19
0000 20     node_t node2;
0000 21     node2.value = 5;
0000 22     node2.next_p = &node1;
0000 23
0000 24     int sum = 0;
0000 25     node_t* curr_p = &node2;
0000 26     while ( curr_p != NULL ) {
0000 27         sum += curr_p->value;
0000 28         curr_p = curr_p->next_p;
0000 29     }
0000 30     return 0;
0000 31 }

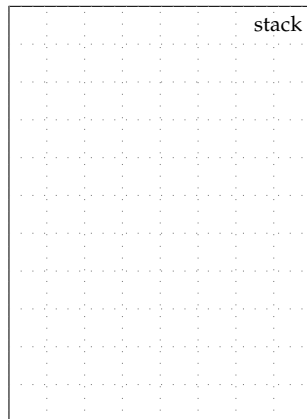
```



**zyBooks** The course zyBook includes a coding lab to implement a function to find the maximum value in a chain of nodes.

### 4.3. Pointers to Pointers

```
□□□ 01 int    a      = 3;
□□□ 02 int*   a_ptr   = &a;
□□□ 03 int**  a_pptr  = &a_ptr;
□□□ 04 int*** a_ppptr = &a_pptr;
□□□ 05
□□□ 06 int b = ***a_ppptr + 1;
```



**zyBooks** Code is also stored in memory, so a *function pointer* points to code. The course zyBook includes more information on function pointers, which are complicated but critical for understanding some of the more sophisticated programming paradigms later in the course. The course zyBook also includes a coding lab to implement a `count_if` function that uses a function pointer as a parameter to decide what elements to count in a chain of nodes.