ECE 2400 Computer Systems Programming
Fall 2019

Topic 4: C Pointers

School of Electrical and Computer Engineering
Cornell University

revision: 2019-09-16-00-12

1 Pointer Basics

2 Call by Value vs. Call by Pointer

3 Mapping Conceptual Storage to Machine Memory

3.1. Mapping Conceptual Storage to Machine Memory on x86 ★

4 Pointers to Other Types

4.1. Pointers to struct ★

4.2. Pointers to Nothing ★

4.3. Pointers to Pointers ★

4.4. Pointers to Functions ★

Sections marked with a star (★) are not covered in lecture but are instead covered in the online lecture notes. Students are responsible for all material covered in lecture and in the online lecture notes. Material from the online lecture notes will definitely be assessed in the prelim and final exam.

Copyright © 2018 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. 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.
1. Pointer Basics

- In C, pointers are a way of referring to the location (or the address) of a variable
- Pointers enable a variable’s value to be a “pointer” to a completely different variable
- Programmers can access what a pointer points to and redirect the 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 \( T \)
  
  ```
  int* a_ptr; // pointer to a variable of type int
  char* b_ptr; // pointer to a variable of type char
  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
  
  ```
  int a; // variable of type int
  int* a_ptr; // pointer to a variable of type int
  a_ptr = &a; // assign location of a to a_ptr
  ```

- The **dereference** operator (*) evaluates to the value of the variable the pointer points to
  
  ```
  int b = 42; // initialize variable of type int to 42
  int* b_ptr = &b; // pointer to a variable of type int
  int c = *b_ptr; // initialize c with what b_ptr points to
  ```
1. Pointer Basics

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;
```
2. Call by Value vs. Call by Pointer

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

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

```
void sort( int x, int y )
{
    if ( x > y ) {
        int temp = x;
        x = y;
        y = temp;
    }
}

int main( void )
{
    int a = 9;
    int b = 5;
    sort( a, b );
    return 0;
}
```
2. Call by Value vs. Call by Pointer

- **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

```c
void sort( int* x_ptr,
          int* y_ptr )
{
    if ( (*x_ptr) > (*y_ptr) ) {
        int temp = *x_ptr;
        *x_ptr = *y_ptr;
        *y_ptr = temp;
    }
}

int main( void )
{
    int a = 9;
    int b = 5;
    sort( &a, &b );
    return 0;
}
```

https://repl.it/@cbatten/ece2400-T04-ex1
Draw a state diagram corresponding to the execution of this program

```c
void avg( int* result_ptr,
           int x, int y )
{
    int sum = x + y;
    *result_ptr = sum / 2;
}

int main( void )
{
    int a = 10;
    int b = 20;
    int c;
    avg( &c, a, b );
    return 0;
}
```
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;
05 int b = 2;
06 int c;
07 c = b + (*a_ptr);
```

```
stack
```

```
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

```c
void sort( int* x_ptr, int* y_ptr )
{
    if ( (*x_ptr) > (*y_ptr) ) {
        int temp = *x_ptr;
        *x_ptr = *y_ptr;
        *y_ptr = temp;
    }
}

int main( void )
{
    int a = 9;
    int b = 5;
    sort( &a, &b );
    return 0;
}
```

---

**Memory**

(4B word addr)
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

```
typedef struct _node_t
{
  int value;
  struct _node_t* next_ptr;
} node_t;
```

```
int main( void )
{
  // First node
  node_t node0;
  node0.value = 3;

  node_t* node_ptr = &node0;
  (*node_ptr).value = 4;

  // Second node
  node_t node1;
  node1.value = 5;
  node1.next_ptr = &node0;

  node_ptr = &node1;
  (*node_ptr).value = 6;
  *(((*node_ptr).next_ptr)).value = 7;

  return 0;
}
```
• C provides the arrow operator (\(\rightarrow\)) as syntactic sugar
  • \(a \rightarrow b\) is equivalent to \((\ast a) \cdot b\)

```c
int main( void )
{
  ...

  node_t* node_ptr = &node0;
  node_ptr->value = 4;

  ...

  node_ptr = &node1;
  node_ptr->value = 6;
  node_ptr->next_ptr->value = 7;
}
```

### 4.2. Pointers to Nothing

• NULL is defined in \texttt{stdlib.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
```c
#include <stddef.h>
#include <stdio.h>

typedef struct _node_t
{
    int value;
    struct _node_t* next_ptr;
} node_t;

int main( void )
{
    node_t node0;
    node0.value = 3;
    node0.next_ptr = NULL;

    node_t node1;
    node1.value = 4;
    node1.next_ptr = &node0;

    node_t node2;
    node2.value = 5;
    node2.next_ptr = &node1;

    int sum = 0;
    node_t* node_ptr = &node2;
    while ( node_ptr != NULL ) {
        sum += node_ptr->value;
        node_ptr = node_ptr->next_ptr;
    }
    return 0;
}
```
4.3. Pointers to Pointers

```
int a = 3;
int* a_ptr = &a;
int** a_pptr = &a_ptr;
int*** a_ppptr = &a_pptr;
int b = ***a_ppptr + 1;
```