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

1 Pointer Basics **2**

2 Call by Value vs. Call by Pointer **4**

3 Mapping Conceptual Storage to Machine Memory **7**

4 Pointers to Other Types **10**

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.

- Pointers refer to the location (or address) of a variable
- A variable can now to "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`

```
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

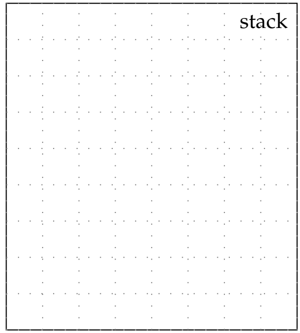
```
int  a;          // variable of type int
int* a_ptr;      // pointer to a variable of type int
a_ptr = &a;      // assign location of a to a_ptr
```

- 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
```

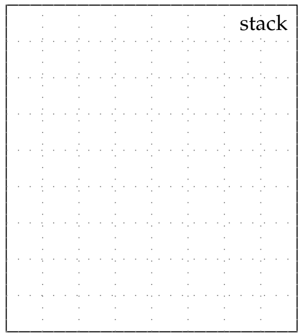
Example declaring, initializing, RHS dereferencing pointers

```
int  a = 3;
int* a_ptr;
a_ptr = &a;

int  b = 2;
int  c = b + (*a_ptr);
```



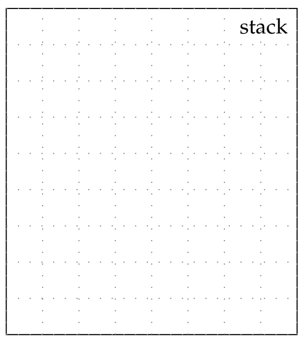
Example illustrating aliasing

```
int  a = 3;
int* a_ptr0 = &a;
int* a_ptr1 = a_ptr0;
int  c = (*a_ptr0) + (*a_ptr1);
```



Example declaring, initializing, LHS dereferencing pointers

```
int  a = 3;
int  b = 2;

int  c;
int* c_ptr = &c;
*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

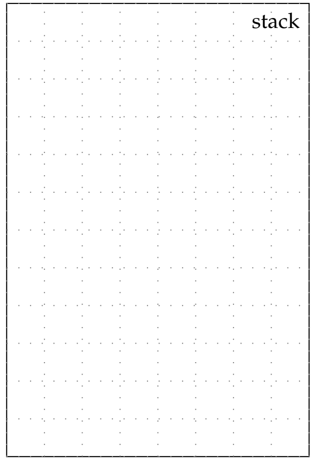
- So far, we have always used call by value
- Call by value *copies* values into parameters
- Changes to parameters by callee *are not seen* by caller

```
void swap( int x, int y )
{
  int temp = x;
  x = y;
  y = temp;
}

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

stack

- 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

```
void swap( int* x_ptr,
           int* y_ptr )
{
  int temp = *x_ptr;
  *x_ptr   = *y_ptr;
  *y_ptr   = temp;
}

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



`https://tinyurl.com/zybookch4`

Draw a state diagram corresponding to the execution of this program

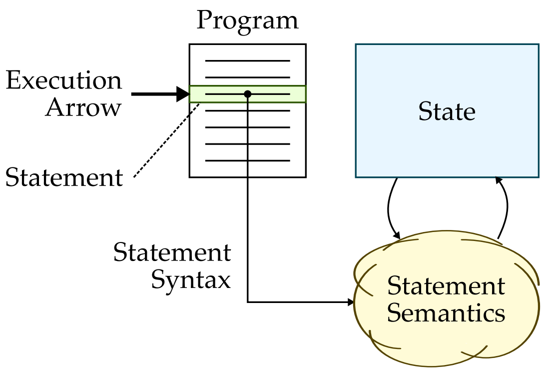
```
void avg( int* result_ptr,
          int x, int y )
{
  int sum = x + y;
  *result_ptr = sum / 2;
}

int main( void )
{
  int a = 10;
  int b = 20;
  int c;
  avg( &c, a, b );
  return 0;
}
```

stack

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

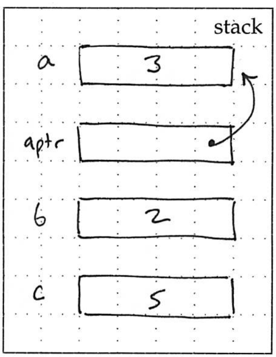
3. Mapping Conceptual Storage to Machine Memory

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

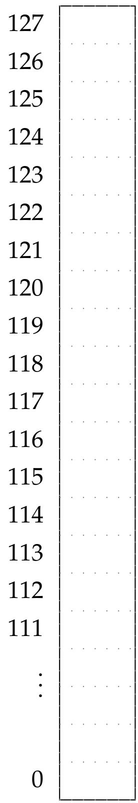


- Can visualize memory using a “byte” or “word” view
- Stack stored at high addresses, stack grows “down”
- As a simplification, assume we only have 128 bytes of memory

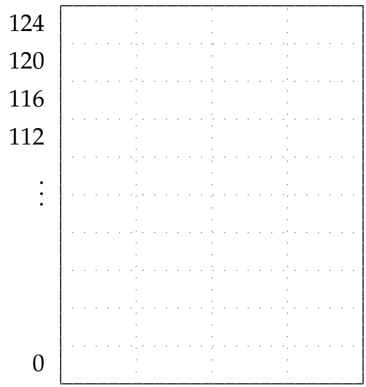
```c
int   a = 3;
int*  a_ptr;
a_ptr = &a;

int   b = 2;
int   c;
c = b + (*a_ptr);
```



Memory
(byte addr)

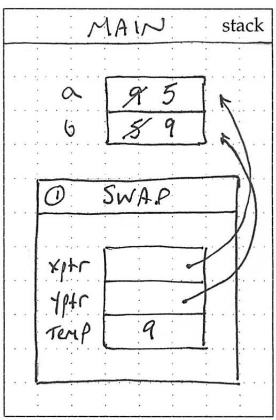


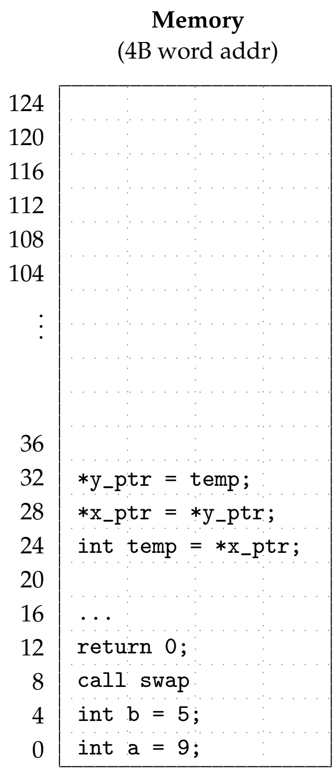
Memory
(4B word addr)



- 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

```
void swap( int* x_ptr, int* y_ptr )
{
  int temp = *x_ptr;
  *x_ptr   = *y_ptr;
  *y_ptr   = temp;
}

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





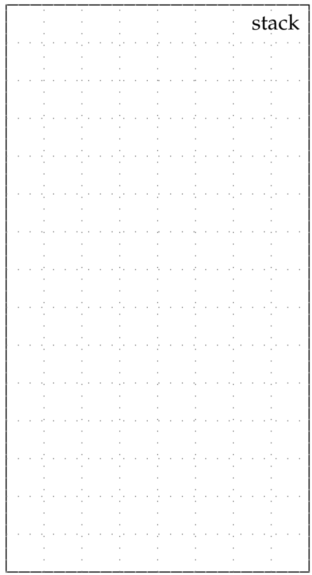
4. Pointers to Other Types

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_p;
}
node_t;

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

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

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

  head_p = &node1;
  (*head_p).value                 = 6;
  (*((*head_p).next_p)).value = 7;

  return 0;
}
```



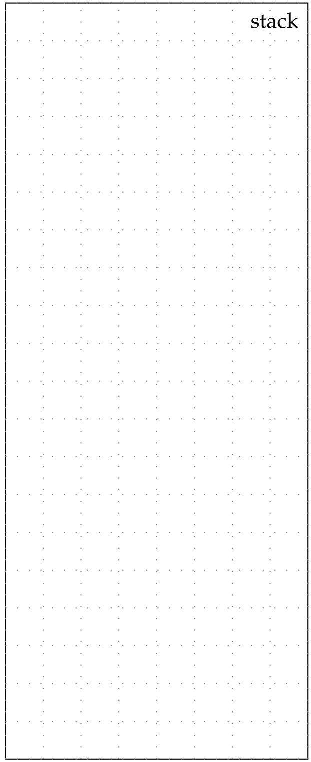
- C provides the arrow operator (`->`) as syntactic sugar
- `a->b` is equivalent to `(*a).b`

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

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

  ...

  head_p = &node1;
  head_p->value         = 6;
  head_p->next_p->value = 7;
}
```

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

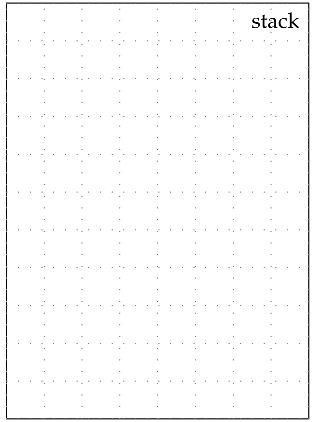
```c
#include <stddef.h>

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

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

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

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

  int sum = 0;
  node_t* curr_p = &node2;
  while ( curr_p != NULL ) {
    sum += curr_p->value;
    curr_p = curr_p->next_p;
  }
  return 0;
}
```



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

```
int     a      = 3;
int*    a_ptr  = &a;
int**   a_pptr = &a_ptr;
int***  a_ppptr = &a_pptr;

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.