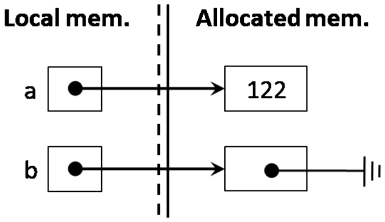
15-122: Principles of Imperative Computation Spring 2024

Recitation 05: A queue_t Interface Wednesday February 14th

Pointer manipulations

Pointers are one of the most useful constructs in languages like C0 and C. The value of an expression of pointer type (like **int**∗) is either a *memory address* or the special value NULL.

Consider the following memory diagram, where a has type **int**∗ and b has type **int**∗∗:



The variable a points to a memory cell that contains the **int** value 122. Instead the variable b points to a memory cell of pointer type that contains NULL — NULL is a valid value for cells of any pointer type (instead, addresses must match the type the cell is declared as). Because the type of b is **int**∗∗, whenever this cell does not contain NULL it must contain the address of a cell to an **int**, i.e., the cell pointed to by b has type is **int**∗.

Give C0 instructions that result in the above diagram (assume you just started coin and the memory is empty).

Pointer manipulations are counterintuitive at first, but with just a little bit of practice, they will become second nature. A key insight is the following:

> ***When you set a pointer equal to another pointer, you make the first pointer point to where the second pointer points.***

Thus, each line of C0 code modifies at most one pointer.

Draw the memory diagram after executing the line of code

```
int* c = a;
```

At this point, `a` and `c` are *aliases*: both variables point to the same memory cell, i.e., they contain the same address.

Draw the memory diagram after executing the line of code

```
*b = a;
```

Now, `*b` (the contents of the cell `b` points to) is also an alias to `a` (and `c`).

Let's define the following struct, nicknamed `student`:

```
struct student_header {
  string name;
  int grade;
};
typedef struct student_header student;
```

It has two fields, `name` of type **`string`** and `grade` of type **`int`**. In C0, structs can only appear in allocated memory.

Draw the memory diagram after executing the line of code

```
student* t = alloc(student);
t->name = "Alex";
t->grade = 100;
```

A wild `struct` appears

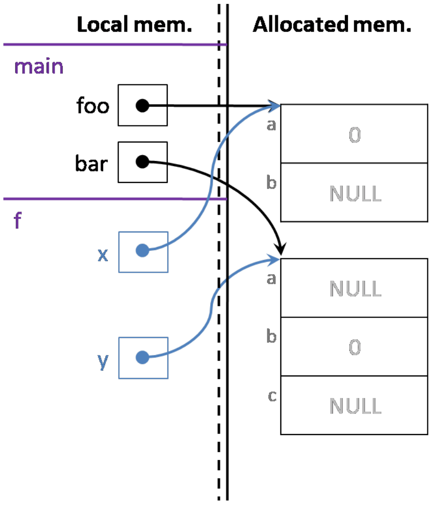
Suppose we have the following in a file:

```
struct X {
  int       a;
  struct Y* b;
};

struct Y {
  int*      a;
  int       b;
  struct X* c;
};

void f(struct X* x, struct Y* y) {
  x->b = y;
  y->c = x;
  y->c->a = 15
  int** d = alloc(int*);
  *d = alloc(int);
  x->b->a = *d;
  *(y->a) = x->a * 8 + 2;
  x->b->b = 1000 * x->a + **d;
  x = NULL;
  y->c = NULL;
}

int main() {
  struct X* foo = alloc(struct X);
  struct Y* bar = alloc(struct Y);
  f(foo, bar);
  return 0;
}
```



Checkpoint 0

Fill out the state of the memory just before and just after `f` returns. What's the value of `bar->b`? (For your own sanity, draw a picture!)

Stack and queue interfaces

Here's the **stack interface** discussed in lecture. It exposes the type `stack_t` and four functions:

```
// typedef ______* stack_t;    /* Abstract type of stacks                      */

bool stack_empty(stack_t S)    /* Check if stack S is empty,              O(1) */
/*@requires S != NULL; @*/ ;

stack_t stack_new()            /* Create a new empty stack,               O(1) */
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, string x) /* Add item x at the top of stack S,       O(1) */
/*@requires S != NULL; @*/
/*@ensures !stack_empty(S); @*/ ;

string pop(stack_t S)          /* Remove and return the top of stack S,   O(1) */
/*@requires S != NULL; @*/
/*@requires !stack_empty(S); @*/ ;
```

The **queue interface** exposes the type `queue_t` and four similar functions:

```
// typedef ______* queue_t;    /* Abstract type of queues                      */

bool queue_empty(queue_t Q)    /* Check if queue Q is empty,              O(1) */
/*@requires Q != NULL; @*/ ;

queue_t queue_new()            /* Create a new empty queue,               O(1) */
/*@ensures \result != NULL; @*/
/*@ensures queue_empty(\result); @*/ ;

void enq(queue_t Q, string e)  /* Add item e at the back of queue Q,      O(1) */
/*@requires Q != NULL; @*/
/*@ensures !queue_empty(Q); @*/ ;

string deq(queue_t Q)          /* Remove and return the front of queue Q, O(1) */
/*@requires Q != NULL; @*/
/*@requires !queue_empty(Q); @*/ ;
```

Checkpoint 1

Write a function to reverse a queue using only functions from the stack and queue interfaces.

```
void reverse(queue_t Q)
//@requires ______________________________________________;
{
______________________________________  // create temp data structure
  while (____________________________________________) {
    ____________________________________________________
  }
  while (____________________________________________) {
    ____________________________________________________
  }
}
```

Checkpoint 2

Write a *recursive* function to count the size of a stack. You may not destroy the stack in the process — the stack's elements (and order) must be the same before and after calling this function. Assume the stack contains elements of type **string**.

```
int size(stack_t S)
//@requires ______________________________________________;
{
  ______________________________________________________
  ______________________________________________________
  ______________________________________________________
  ______________________________________________________
  ______________________________________________________
}
```

Checkpoint 3

Why couldn't this function be used in contracts in C0? Hint: Contracts in C0 can't have side effects.

____