C for Java Programmers

Slides based on those by Philip Papadopoulos
The Generic Pointer (void *)

• Pointers know what data type the point to
  • This makes pointer arithmetic work well.

• What do you do if you don’t know exactly what the data type is, but you still want a reference to it?

• void *
  • pointer to a void data type

• Why would you want this?
  • Think of (void *) as the equivalent of (Object)
  • Every pointer can be safely cast to/from a (void *)
The (dreaded) Null Pointer

• void * ptr = NULL;
  • Set a pointer to all zeros
  • This is the first memory location and is RESERVED

• In If tests
  • if (ptr != NULL) is often shortened to if (ptr)
The most awful “Segmentation Fault”

• What is a segmentation fault?
• Memory (in Unix/Linux) is protected.
  • One process cannot access/change another processes memory
  • Memory is said to be “segmented” so that each process has its own protected area
• Nothing in C enforces what you can put into a pointer, but
  • If it points off to memory that is outside of your processes memory segment you generate a “Segmentation” fault
  • This is a catastrophic error
Syntactic Sugar: the “->” operator

• Having a pointer to a data structure is extremely common in C
• One can access members of the data structure in two different ways (supposing a CONTACT * contactPointer)
  • (*contactPointer).lastname
  OR
  • contactPointer -> lastname

  “lastname member of the structure that contactPointer points to”

• The second form is MUCH preferred for readability.
Pointers to functions: Advanced C

• The syntax for a pointer to function (e.g. a function that returns an int and has a double as an argument
  • int (*funcPtr) (float);

• Let’s pick apart some code to see all this in action
  • pointer arithmetic
  • pointers and arrays
  • pointers to functions
typedef struct contact {
    char lastname[25];
    char firstname[25];
    int areaCode;
    int phoneNumber;
    void (*print)(struct contact *); /* pointer to func that prints */
} CONTACT;

/* prototype declarations */
void formFill(CONTACT * t, char last[], int code);
void printbook(CONTACT book[], int nentries);

a generic print function. Notice it takes a pointer to struct contact

formFill changes, now takes a pointer to the form to fill out
#include <stdio.h>
#include <string.h>
#include "structure2.h"

static void printIt(CONTACT * t) {
    printf("%s (%d)\n", t->lastname, t->areaCode);
}

void formFill(CONTACT * t, char last[], int code) {
    strcpy(t->lastname, last);
    t->areaCode = code;
    t->print = &printIt;
}

void printbook(CONTACT *book, int nentries) {
    int i;
    for (i = 0; i < nentries; i++) {
        printf("book[%d]lastname,(areaCode):",i);
        (book -> print)(book);
        book++;
    }
}
void modifyForm(CONTACT * form, char last[], int code) {
    formFill(form, last, code);
}

int main(int argc, char *argv[]) {
    int i;
    CONTACT phonebook[100];
    formFill(phonebook, "Papadopoulos", 858);
    formFill(phonebook + 1, "Alvarado", 858);
    phonebook[2] = phonebook[1]; /* copy entries */

    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    printbook(phonebook, 3);

    /* Modify entry 2 last name */
    printf("(properly) Modify entry 2\n");
    modifyForm(phonebook + 2, "Smith", 619);
    printbook(phonebook, 3);
}
Print out of the code in previous slides

./structure6
sizeof(CONTACT): 64
sizeof(phonebook): 6400
book[0].lastname, (areaCode): Papadopoulos (858)
(properly) Modify entry 2
book[0].lastname, (areaCode): Papadopoulos (858)
Memory Management

• C puts the onus of dynamic memory management onto the user
  • In java, `new + garbage collector (System.gc())` handles this
  • In C, you explicitly allocate and de-allocated memory as needed

• memory allocators
  • `malloc`
    • Allocate memory from the heap. **Memory is uninitialized.** Program must explicitly deallocate with a call to `free`
  • `calloc`
    • Allocate memory from the heap. **Memory is initialized** to all zero. Program must explicitly deallocate with a call to `free`
  • `alloca`
    • Allocate memory from the stack. **Memory is uninitialized.** Memory is automatically freed when function returns.
Typical Memory Organization

Memory of a Process

- **Stack**
  - Cannot have holes in it
  - Call stack (Activation Frames) are stored here.
  -alloca() allocates memory here

- **Heap**
  - Can have holes in it
  - malloc(), calloc() allocate memory from this part of memory

Stack grows down

Heap grows up
More on the heap

• Unlike the stack, the Heap can have “holes” in it
  • heap-allocated memory does not have to be “freed” in the order it was allocated.
  • Efficient heap allocators will attempt to reuse previously freed regions of memory to keep the heap from growing too large.
    • Can you think of how a heap data structure might be used for this?

• In Java, Objects are allocated from the heap with the new keyword
What do you get when call malloc()?

• A pointer.
  • actually, a `void *` pointer

• Allocating an array of 200 contacts
  • `CONTACT * cptr = malloc (200 * sizeof(CONTACT));`
  • (cast to a CONTACT pointer is done implicitly)

  OR

• Allocating an array of 200 contacts
  • `CONTACT * cptr = (CONTACT *) malloc (200 * sizeof(CONTACT));`
  • `*(cptr + k)` accesses the k+1 element of the array
  • so does `cptr[k]`
Free – memory deallocation

• When your program no longer needs the memory that was allocated on the heap, you should call `free` to recycle it.
• If you don’t call free, you may exhaust memory.
• When you lose reference to allocated memory e.g.,
  
  ```c
  int * iptr = malloc(10*sizeof(int));
  iptr = malloc (20 * sizeof(int));
  free (iptr);
  ```

  • The reference to the first `malloc`ed block of memory is lost
  • This is called a `memory leak`
  • memory is only recovered when the program exits
How not to give yourself a flat tire

• Returning a pointer to alloca()'ed (or other stack allocated) memory to a caller of a function
  • The memory is “freed” and will be overwritten at the next function call

• Attempting to free memory that wasn’t heap-allocated
  • You must call free with the memory address that was returned by malloc/calloc

```c
int * iptr = malloc(10*sizeof(int));
free ( iptr );  // OK!
free ( iptr + 1);  // ✗
```
realloc()

• one can reallocate existing memory (usually to expand it).

“realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized.”
Singly Linked List in C

• Four files
  • linkedList.h - dataType definition
  • linkedList.c – implementation of singly Linked List of voids
  • reverseList.c – a version of file line reversal using a singly linked list
  • reverseArray.c – a version of file line reversal using an array
/* linkedList.h - header file for simple linked list of strings */
#define OK 0
#define FAIL -1
#define TRUE 1
#define FALSE 0

/* Define Linked List Structure of Node, and List */
struct Node
{
    void *elem; //!< data element
    struct Node *link; //!< link to next node
};
typedef struct Node NODE;
struct SLList
{
    int size; //!< size of the list
    struct Node *head; //!< head of list
};
typedef struct SLList SLLIST;

NODE *createNode(void *val);
void destroyNode(NODE *node, int freeElem);
SLLIST *createList();
void destroyList(SLLIST *list, int freeElem);
int insertElem(SLLIST *list, void *elem);
int insertElemIndex(SLLIST *list, void *elem, int idx);
void *removeElem(SLLIST *list);
void *removeElemIndex(SLLIST *list, int idx);
NODE, SLLIST

• NODE is a typedef of struct Node
• SLLIST is a typedef of struct SLList

• Very similar in concept to Nodes (inner class) of List classes developed earlier in the quarter
  • Node – has a pointer to the actual data (notice use of void *)
  • Node – has a pointer to its next next (link)
  • SLList – has a size (keep track of # of nodes)
  • SLList – has a head (ptr to the head Node of the list)
LinkedList.c – Implements a Linked List

• createNode( ...)

```c
/** Create a node in the list
 * @param val pointer to data referenced by this node
 *      val - the */
NODE * createNode(void * val)
{
    NODE * rNode;
    rNode = (NODE *) malloc(sizeof(NODE));
    rNode->elem = val;
    rNode->link = (NODE *) NULL;
    return rNode;
}
```
/** destroy a Node
 * @param node - the NODE to destroy
 * @param freeElem - if nonzero, free the element referenced
 */
void destroyNode(NODE * node, int freeElem)
{
    if (! node ) return;
    if (freeElem && node -> elem)
        free( node -> elem );
    free(node);
}
LinkedList.c – Implements a Linked List

• `createList( ...)`

```c
/** Create a Singly Linked List
 * This uses a sentinel for the head node */
SLLIST * createList()
{
    SLLIST *rval;
    rval = (SLLIST *) malloc(sizeof(SLLIST));
    rval -> size = 0;
    rval -> head = createNode( (void *) NULL);
    return rval;
}
```

Explicit memory allocation

Sentinel head node
LinkedList.c – Implements a Linked List

- `insertElemIndex( ... )`

```c
int insertElemIndex(SLLIST * list, void * elem, int idx)
{
    int i;
    if (!list || !elem) return FAIL;
    NODE * new;
    NODE * head = list->head;
    for (i = 0; i < idx; i++)
        head = head->link;
    if (!head) return FAIL;
    new = createNode(elem);
    new->link = head->link;
    head->link = new;
    list->size ++;
    return OK;
}
```

- Iterate to correct location in list
- Create a new NODE
- Update links and size of list
void * removeElemIndex(SLLIST * list, int idx)
{
    int i;
    if (! list) return (void *) NULL;
    NODE * r; NODE * head = list->head;
    void * rval;
    for (i = 0; i < idx; i ++)
        head = head->link;
    if (!head) return (void *) NULL;

    if (r = head -> link){
        rval = r->elem;
        head->link = r->link;
        destroyNode(r, FALSE);
        list -> size --;
        return rval;
    }
    return (void *) NULL;
}
reverseList.c – reverse print lines of a text file

• Major sections
  • Read the command line arguments
    • Fail if not the correct number
  • Open the file specified on the command line
    • Fail if file cannot be opened for reading
  • Read in file line-by-line
    • Add to list at position 0
  • Iterate through list
    • print lines
reverseList.c – uses a Linked List

• main code loop

```c
SLLIST list = createList();

while (fgets(linebuf, MAXLINE-1, rfile) != NULL) {
    linebuf[MAXLINE-1] = '\0'; // force string termination
    str = malloc(strlen(linebuf)+1);
    strcpy(str, linebuf);
    insertElem(list, str);
}

NODE * n;
 n = list -> head;
while ( (n = n -> link))
    printf("%s", n->elem);
destroyList(list, TRUE);
```

Read file, line-by-line until the end
Allocate storage for the string just
Insert into list (this version inserts at 0)
Iterate through the list
destroy the list
How does this differ Java?

• Explicit memory allocation/deallocation
• pointers ~= references
• SLLIST implements list of generic elements
  • Not TYPED, could mix pointers to all kinds of things (that’s good and bad)
  • Closest to having a list of Objects
Compiling

• Option 1 (all at once)
  • `cc -o reverseLink linkedList.c reverseLink.c`

• Option 2 (first linkedList, then reverseLink)
  • `cc -c linkedList.c`  // creates `linkedList.o` object file
  • `cc -o reverseLink linkedList.o reverseLink.c`

• Option 3 (create object files, then link)
  • `cc -c linkedList.c`
  • `cc -c reverseLink.c`
  • `cc -o reverseList reverseList.o linkedList.o`
Running

$ ./reverseList
reverse <filename>
$ ./reverseList somefile
Could not open file: somefile
$ ./reverseList short.txt
Line 4
Line 3
Line 2
Line 1
$ cat short.txt
Line 1
Line 2
Line 3
Line 4
$
Make

• Subject of another class, but a Makefile is supplied.
• to compile and run the first example, Should be able to do
  • make first
  • ./first