C Memory Management

2008-02-01

Hello to Scott Sloan from Superior, WI!

Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Sci-Fi Realities ⇒

…the predictions of the Sci-Fi analysts feel will become reality and when: Biometric Security – voice, iris, retinal scans [2010], Space Tourism [2013], Holodeck [2016], Self-Aware Computers [2019], Domestic Robots [2020]. Get in line!
Review

• Pointers and arrays are **virtually same**

• C knows how to **increment pointers**

• C is an efficient language, with little protection
  • Array bounds not checked
  • Variables not automatically initialized

• (Beware) The cost of efficiency is more overhead for the programmer.
  • “C gives you a lot of extra rope but be careful not to hang yourself with it!”
• Sometimes you want to have a procedure increment a variable?

• What gets printed?

```c
void AddOne(int x) {
    x = x + 1;   }

int y = 5;
AddOne( y);
printf("y = %d\n", y);
```
Pointers (2/4)  ...review...

• Solved by passing in a pointer to our subroutine.
• Now what gets printed?

```c
void AddOne(int *p)
{
    *p = *p + 1;
}

int y = 5;
AddOne(&y);
printf("y = %d\n", y);
```
Pointers (3/4)

• But what if what you want changed is a pointer?

• What gets printed?

```c
void IncrementPtr(int *p)
{
    p = p + 1;
}

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
printf("*q = %d\n", *q);
```
Pointers (4/4)

• Solution! Pass a pointer to a pointer, declared as **h

• Now what gets printed?

```c
void IncrementPtr(int **h)
{
    *h = *h + 1;
}

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);
printf("*q = %d\n", *q);
```

```plaintext
A q q

50 60 70

*q = 60
```
Dynamic Memory Allocation (1/4)

• C has operator `sizeof()` which gives size in bytes (of type or variable)

• Assume size of objects can be misleading and is bad style, so use `sizeof(type)`
  • Many years ago an `int` was 16 bits, and programs were written with this assumption.
  • What is the size of integers now?

• “`sizeof`” knows the size of arrays:
  ```c
  int ar[3]; // Or: int ar[] = {54, 47, 99}
  sizeof(ar) ⇒ 12
  ```
  • ...as well for arrays whose size is determined at run-time:
  ```c
  int n = 3;
  int ar[n]; // Or: int ar[fun_that_returns_3()];
  sizeof(ar) ⇒ 12
  ```
Dynamic Memory Allocation (2/4)

• To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):

```c
ptr = (int *) malloc (sizeof(int));
```

  • Now, `ptr` points to a space somewhere in memory of size `(sizeof(int))` in bytes.
  
  • `(int *)` simply tells the compiler what will go into that space (called a typecast).

• `malloc` is almost never used for 1 var

```c
ptr = (int *) malloc (n*sizeof(int));
```

  • This allocates an array of `n` integers.
Dynamic Memory Allocation (3/4)

• Once `malloc()` is called, the memory location contains garbage, so don’t use it until you’ve set its value.

• After dynamically allocating space, we must dynamically free it:

  ```c
  free(ptr);
  ```

• Use this command to clean up.

  • Even though the program frees all memory on `exit` (or when `main` returns), don’t be lazy!

  • You never know when your `main` will get transformed into a subroutine!
Dynamic Memory Allocation (4/4)

• The following two things will cause your program to crash or behave strangely later on, and cause VERY VERY hard to figure out bugs:
  - `free()`ing the same piece of memory twice
  - calling `free()` on something you didn’t get back from `malloc()`

• The runtime **does not** check for these mistakes
  - Memory allocation is so performance-critical that there just isn’t time to do this
  - The usual result is that you corrupt the memory allocator’s internal structure
  - You won’t find out until much later on, in a totally unrelated part of your code!
Arrays not implemented as you’d think

```c
void foo() {
    int *p, *q, x, a[1]; // a[] = {3} also works here
    p = (int *) malloc (sizeof(int));
    q = &x;

    *p = 1; // p[0] would also work here
    *q = 2; // q[0] would also work here
    *a = 3; // a[0] would also work here

    printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
    printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
    printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```

```
12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 ...
...
... 52 32 2 3 ...

p q x a unnamed-malloc-space

*p:1, p:52, &p:24
*q:2, q:32, &q:28
*a:3, a:36, &a:36
```
Binky Pointer Video (thanks to NP @ SU)

Pointer Fun with Binky

by Nick Parlante
This is document 104 in the Stanford CS Education Library — please see cslibrary.stanford.edu for this video, its associated documents, and other free educational materials.

Copyright © 1999 Nick Parlante. See copyright panel for redistribution terms. Carpe Post Meridiem!
Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

1. Kid meets giant Texas people exercising zen-like yoga. – Rolf O
2. Kind men give ten percent extra, zestfully, youthfully. – Hava E
3. Kissing Mentors Gives Testy Persistent Extremists Zealous Youthfulness. – Gary M
4. Kindness means giving, teaching, permeating excess zeal yourself. – Hava E
5. Killing messengers gives terrible people exactly zero, yo
6. Kindergarten means giving teachers perfect examples (of) zeal (&) youth
7. Kissing mediocre girls/guys teaches people (to) expect zero (from) you
8. Kinky Mean Girls Teach Penis-Extending Zen Yoga
10. Kissing me gives ten percent extra zeal & youth! – Dan G (borrowing parts)
Which are guaranteed to print out 5?

I:    main() {
      int *a-ptr; *a-ptr = 5; printf("%d", *a-ptr); }

II:   main() {
      int *p, a = 5;
      p = &a; ...
      /* code; a & p NEVER on LHS of = */
      printf("%d", a); }

III:  main() {
      int *ptr;
      ptr = (int *) malloc (sizeof(int));
      *ptr = 5;
      printf("%d", *ptr); }

<table>
<thead>
<tr>
<th>I</th>
<th>II</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1:</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2:</td>
<td>-</td>
<td>YES</td>
</tr>
<tr>
<td>3:</td>
<td>-</td>
<td>YES</td>
</tr>
<tr>
<td>4:</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>5:</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>6:</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>7:</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
“And in Conclusion…”

- Use handles to change pointers
- Create abstractions with structures
- Dynamically allocated heap memory must be manually deallocated in C.
  - Use malloc() and free() to allocate and deallocate memory from heap.
Reference slides

You ARE responsible for the material on these slides (they’re just taken from the reading anyway); we’ve moved them to the end and off-stage to give more breathing room to lecture!
C structures: Overview

• A struct is a data structure composed from simpler data types.
  • Like a class in Java/C++ but without methods or inheritance.

```c
struct point {  /* type definition */
    int x;
    int y;
};

void PrintPoint(struct point p)
{  As always in C, the argument is passed by "value" – a copy is made.
    printf("(%d,%d)", p.x, p.y);
}

struct point p1 = {0,10};  /* x=0, y=10 */
PrintPoint(p1);
```
C structures: Pointers to them

- Usually, more efficient to pass a pointer to the struct.
- The C arrow operator (\texttt{->}) dereferences and extracts a structure field with a single operator.
- The following are equivalent:

\begin{verbatim}
struct point *p;
    /* code to assign to pointer */
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
\end{verbatim}
How big are structs?

• Recall C operator `sizeof()` which gives size in bytes (of type or variable)

• How big is `sizeof(p)`?

```c
struct p {
    char x;
    int y;
};
```

• 5 bytes? 8 bytes?
• Compiler may word align integer `y`
Linked List Example

• Let’s look at an example of using structures, pointers, malloc(), and free() to implement a linked list of strings.

```c
/* node structure for linked list */
struct Node {
    char *value;
    struct Node *next;
};
```

Recursive definition!
typedef simplifies the code

```c
struct Node {  
    char *value;
    struct Node *next;
};

/* "typedef" means define a new type */
typedef struct Node NodeStruct;

... OR ...

typedef struct Node {  
    char *value;
    struct Node *next;
} NodeStruct;

... THEN ...

typedef NodeStruct *List;
typedef char *String;

/* Note similarity! */
/* To define 2 nodes */

struct Node {  
    char *value;
    struct Node *next;
} node1, node2;
```
**Linked List Example**

/* Add a string to an existing list */

List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));
    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}

{ 
    String s1 = "abc", s2 = "cde";
    List theList = NULL;
    theList = cons(s2, theList);
    theList = cons(s1, theList);
    /* or, just like (cons s1 (cons s2 nil)) */
    theList = cons(s1, cons(s2, NULL));
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
**Linked List Example**

/* Add a string to an existing list, 2nd call */

List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}

node:

```
| ? | "?????"
```

list:

```
| ... | NULL
```

s:

```
| "abc"
```
Linked List Example

/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
Linked List Example

/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc(strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}