How about them A’s!! Go Oaktown!!

CS61C - Machine Structures

Lecture 4
C Structures
Memory Management

2002-09-05

⇐ Dan Garcia
(www.cs.berkeley.edu/~ddgarcia)

Dave Patterson ⇒
(www.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/
Overview

° Review
° C Strings (finishing up)
° Pointers to pointers
° C Structures
° Dynamic Memory Allocation
° C Memory Management
Review

° Pointers and arrays are **virtually same**
  • Array names are pointers to 1\textsuperscript{st} element.

° C knows how to **increment pointers**

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

° One common mistake is to forget to allocate an extra byte for the null terminator.

° More generally, C requires the programmer to manage memory manually (unlike Java or C++).

• When creating a long string by concatenating several smaller strings, the programmer must insure there is enough space to store the full string!

• What if you don’t know ahead of time how big your string will be?
C String Standard Functions

- `int strlen(char *string);`
  - compute the length of string

- `int strcmp(char *str1, char *str2);`
  - return 0 if `str1` and `str2` are identical (how is this different from `str1 == str2`?)

- `int strcpy(char *dst, char *src);`
  - copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.
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 to 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 to 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 = \n", *q);
```

```c
#define A 50
#define q 60
#define 70
```

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

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
printf("*q = \n", *q);
```
° Solution! Pass a pointer to a pointer, called a handle, 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);
```

```
A q q
50 60 70
```
int main(void) {
    int A[] = {5, 10};
    int *ptr = A;

    printf("%u %d %d %d\n", ptr, *ptr, A[0], A[1]);
    ptr = ptr + 1;
    printf("%u %d %d %d\n", ptr, *ptr, A[0], A[1]);
    *ptr = *ptr + 1;
    printf("%u %d %d %d\n", ptr, *ptr, A[0], A[1]);
}

If the first printf outputs 100 5 5 10, what will the other two printf output?

A: 101 10 5 10 then 101 11 5 11
B: 104 10 5 10 then 104 11 5 11
C: 101 <other> 5 10 then 101 <3-others>
D: 104 <other> 5 10 then 104 <3-others>
E: One of the two printf's causes an ERROR
F: I surrender!
Buffer Page “A’s refuse to lose!”

° Oakland As win 20th straight!

A’s blow 11-run lead and win on bottom of ninth homer!

° The longest win streak in AL history!

° Largest regular-season crowd in coliseum history!
int main(void) {
    int A[] = {5,10};
    int *ptr = A;

    printf("%u %d %d %d\n", ptr, *ptr, A[0], A[1]);
    ptr = ptr + 1;
    printf("%u %d %d %d\n", ptr, *ptr, A[0], A[1]);
    *ptr = *ptr + 1;
    printf("%u %d %d %d\n", ptr, *ptr, A[0], A[1]);
}

If the first `printf` outputs `100 5 5 10`, what will the other two `printf` output?

B: 104 10 5 10 then 104 11 5 11

...because `ints` in this system are 4-bytes long and the actual address increments by 4 even though it appears to only increment 1.
C structures: Overview

A struct is a data structure composed for simpler data types.

• Like a class in Java/C++ but without methods or inheritance.

```c
struct point {
    int x;
    int y;
}
void PrintPoint(point p) {
    printf("(%d,%d)\n", p.x, p.y);
}
```
C structures: Pointers to them

° The C arrow operator (\$\rightarrow\$) dereferences and extracts a structure field with a single operator.

° The following are equivalent:

```c
struct point *p;

printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```
Dynamic Memory Allocation (1/3)

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

- Assume size of objects can be misleading, so use `sizeof(type)`
  - Many years ago an `int` was 16 bits, and programs assumed it was 2 bytes

- How big is `sizeof(p)`?
  ```c
  struct p {
    char x;
    int y;
  };
  ```
  - 5 bytes? 8 bytes?
  - Compiler may word align integer `y`
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).
Once `malloc()` is called, the memory location might contain anything, 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.
Let’s look at an example of using structures, pointers, `malloc()`, and `free()` to implement a linked list of strings.

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

/* Create a new (empty) list */
List ListNew(void)
{
    return NULL;
}
```
/* add a string to an existing list */
List list_add(List list, char *string) {
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
/ * add a string to an existing list */
List list_add(List list, char *string) {
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
/* add a string to an existing list */
List list_add(List list, char *string)
{
    struct Node *node =
        (struct Node*) malloc(sizeof(struct Node));
    node->value =
        (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}

node:

“abc”

...  ...  NULL

node:
Which of these prints out 5?

A: I only
B: II only
C: III only
D: I and II
E: I and III
F: II and III
BLANK PALM: NONE
Administrivia

° HW graded on effort

° Enrollment
  • Put yourself on the waitlist of the section you want to get into
  • Section 14 still has room for 20 people!
  • It looks like we’ll be able to fit ALL the people with 61B requirements! Yay!
I: main() {
    int *a ptr; *a ptr = 5; printf(“ %d”, *a ptr); }

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

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

Which of these prints out 5?

C: III only

I fails for TWO reasons. You can’t use “−−” in names of variables and you didn’t allocate int storage!

II fails because a subroutine could be called with a pointer to a (e.g., p) which changes a.
List list_remove(List list, char *string)
{
    /* handle the first node */
    if (strcmp(list->value, string) == 0) {
        List next = list->next;
        free(list->value);
        free(list);
        return next;
    }
}

List list_remove(List list, char *string)
{
    . . . first node code from above . . .
    List prev = list;
    List node = prev->next;
    while (strcmp(node->value, string) != 0) {
        prev = node;
        node = node->next;
    }

    prev->next = node->next;
    free(node->value);
    free(node);
    return list;
}
“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.