Lecture 5 – Introduction to C (pt 3)

C Memory Management

Apple’s iPad, day 2

After the dust has settled, what do we have? Name causes chuckles & lawsuits (Fujitsu). “Haters” say nothing new, closed system.

Pointers (1/4)

• Sometimes you want to have a procedure increment a variable?

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

int y = 5;
AddOne(y);
printf("y = %d\n", y);
```

Pointers (2/4)

• Solved by passing in a pointer to our subroutine.

```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?

```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

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

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

Dynamic Memory Allocation (1/4)

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

```c
int ar[3]; // Or: int ar[] = {54, 47, 99}
ssizeof(ar) ⇒ 12
```

• Assume size of objects can be misleading and is bad style, so use sizeof(type)

```c
• 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}
ssizeof(ar) ⇒ 12
```

• as well for arrays whose size is determined at run-time:

```c
int a = 3;
int ar[a]; // Or: int ar[fun_that_returns_3()];
ssizeof(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 `free`s 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;
    p = (int *) malloc (sizeof(int));
    q = &x;
    *p = 1; // p[0] would also work here
    printf("p:%u, p:%u, &p:%u\n", *p, p, &p);
    *q = 2; // q[0] would also work here
    printf("q:%u, q:%u, &q:%u\n", *q, q, &q);
    *a = 3; // a[0] would also work here
    printf("a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```

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/boys 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)

“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.
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){
  printf("(%d,%d)\n", p.x, p.y);
}
```

```c
struct point p1 = {0,10}; /* x=0, y=10 */
PrintPoint(p1);
```

As always in C, the argument is passed by "value" — a copy is made.

C structures: Pointers to them

• Usually, more efficient to pass a pointer to the struct.
  • The C arrow operator (->) dereferences and extracts a structure field with a single operator.
  
```c
struct point *p;
/* code to assign to pointer */
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

How big are structs?

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

```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;
};
```

```c
typedef struct Node NodeStruct;
```

```c
typedef NodeStruct *List;
```

```c
typedef char *String;
```

```c
/* 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;
}
```

```c
String s1 = "abc", s2 = "cde";
List theList = NULL;
theList = cons(s2, theList);
theList = cons(s1, cons(s2, NULL));
```
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:
s:
"abc"

... NULL

node:
list:
s:
"abc"

... NULL

node:
list:
s:
"abc"

... NULL

node:
list:
s:
"abc"

... NULL