CSE 451: Operating Systems
Spring 2003

Lecture 2
C and Pointers

Steve Gribble
gribble@cs.washington.edu
323B Sieg Hall
Today’s agenda

• Administrivia
  – programming assignment
    • get started early…
    • tomorrow’s lab sections are a good opportunity to get help
  – office hours
    • Doug: Thursday at 11am in the TA offices (226a)
    • Valentin: Friday at 2pm, in 232

• Continuing through the trickier aspects of C
Typecasting

• A mistake from last time:

```c
int x = 0x87654321;
char y;

y = (char) x;    printf("%d\n", (int) y);
```

• ANSI C defines:
  – if converting an integer to a signed type, the result is implementation-defined if the value cannot be represented in the new type
  – if converting an integer to an unsigned type, a complicated rule basically gives left-truncation of the bits
  – regardless, don’t do this...!
Memory management in Java

• the Java runtime manages memory on your behalf
  – you never allocate memory directly
    • instead, you instantiate objects using “new”

  \[
  \text{String } x = \text{new String(“hello world”);} \\
  \]

  – the garbage collector frees memory for you
    • figures out when an object can be reclaimed (i.e., no more references exist to that object)
Memory management in C

• some memory is managed on your behalf
  – the instructions which implement your functions
    • compiler, linker, and OS collude to allocate memory for this
  – the memory that backs global and “static” variables
    • compiler, linker, and OS collude to allocate memory for this
  – the memory that backs local variables within functions
    • compiler allocates this out of the “stack” when function is called
    • compiler frees this from the stack when function exits
Memory management in C

• you need to manage some memory on your own
  – allocate memory to hold your data structures
    • hash tables, linked lists, …etc.
    • allocated out of the “heap”
  – you must free this memory when you are done with it!
    • this is hard: elaborate bookkeeping to keep track of what
      memory you have allocated and when it is safe to free

```c
char *x;     // a pointer – we’ll cover this soon
x = (char *) malloc(12);  // allocate 12 bytes
if (x == NULL) exit(-1);  // out of memory?
free(x);                  // free the allocated memory
```
Memory

- memory is an array of bytes
  - potential addresses from 0 to $2^N - 1$
  - for Intel x86, $N=32$ (32-bit architecture)

- each Unix program uses three memory zones
  - the heap
    - things you allocate with malloc
  - the stack
    - local variables within functions, and other bookkeeping in “stack frames”
    - done automatically for you
  - the ‘text segment’
    - code, global and static variables
    - OS sets this up for you when program is loaded
      - “linker” provides the loader a recipe to fill in values
Pointers

• a pointer contains a memory address
  – a pointer “points” to a location in memory

unsigned short    x;
unsigned short    *y;
unsigned short    **z;

x = 1;
y = 4;   *y = 100;   y++;  
z = 0;   **z = 101;   z++;
A brain-teaser: what gets printed out?

```c
unsigned char *p;
unsigned char y = 0x4E;

p = (unsigned char *) 0x00000002;
*p = 0x05;
*(p + 1) = 0x11;
*(p - 1) = 0x3F;
*(p - 2) = y;

printf("%08x\n", (unsigned int) *(p-2));

printf("%08x\n",
    *((unsigned int *) (p-2)));
```
Strings: arrays of characters

- Strings in C are just NULL-terminated arrays of chars

```c
char *my_string = "Hi!";
char another_string[4] = {'H', 'i', '!', '\0'};
char *final_string;

final_string = (char *) malloc(4*sizeof(char));
if (final_string == NULL) exit(-1);
final_string[0] = 'H';
final_string[1] = 'i';
final_string[2] = '!' ;
final_string[3] = '\0';

printf("%s %s %s\n", my_string, another_string, final_string);
```
Pointers and addresses

- \& = "address of"

```c
int main(void) {
    int x=1, *z;

    z = &x;
    printf("%d %08x %08x\n", *z, z, &z);

    z = (int *) malloc(2 * sizeof(int));

    *z = 100;
    *(z+1) = 101;
    *(z+2) = 102;   // whoops!

    return 0;       // same as exit(0)
}
```
Brain teasers...
#1: predict the printout

#include <stdio.h>

void main(void) {

    int i = 6, j = 3;

    *(int *) ( (i<j) ? &i : &j ) = 2;

    printf("%d", i+j);

}


#2: spot the bug

typedef struct ll_st {
    struct ll_st *next;
    int value;
} linked_list_element;

...

void free_linked_list(linked_list_element *head) {

    free(head);
    free_linked_list(head->next);
}

...
#3: spot the bug

typedef struct {
    char test_string[5];
} embedded_string;

char *extract_string(embedded_string extract_from_me) {
    return extract_from_me.test_string;
}

void main() {
    char *x;
    embedded_string y;

    ...
    x = extract_string(y);
    *x = "hi!";
    ...
}
#4: predict the output

```c
#include <stdio.h>

void main(void) {
    char input[256];

    gets(input);
    printf("User inputted: " %s \n", input);

    return;
}
```
#5: spot the bugs

```c
void foo(int print, int value) {
    char *string;

    string = (char *) malloc(10*sizeof(char));

    if (input > 1) {
        sprintf(string, "value: %d", value);
        printf(string);
        free(string);
    }

    return;
}
```
#6: spot the bug (subtle)

unsigned short x, *x_ptr;
unsigned int y;
unsigned char *c_ptr;

// assign some values
y = 0; x=0xFFFF;

// point x_ptr into the "middle" of y
  c_ptr = (char *) (&y);
  x_ptr = (unsigned short *) (c_ptr+1);
  *y_ptr = x;