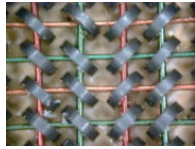


Memory management



Lecture 7
CS 113 – Spring 2008

Announcements

- Assignment 2 posted, due Friday
 - Do two of the three problems
- Assignment 1 graded
 - see grades on CMS

2

Safe user input

- If you use `scanf()`, include a maximum width in the format string

```
char string[1024];
scanf("%1023s", string);
```

- Never use `gets()`!
 - There's no way to use it safely
 - Use `fgets()` instead

```
char string[1024];
if(!fgets(string, 1024, stdin))
    goto error;

// remove newline character
if(string[strlen(string)-1] == '\n')
    string[strlen(string)-1] = 0;
```

3

Function pointers

- In C, you can also take the address of a function
 - I.e. the memory location of the function's machine code
- Example of declaring a function pointer:

```
int (*fctnptr)(int, int);
```

 - This is a pointer to a function that takes two parameters of type `int` and returns an integer
- Use the `&` operator to take the address of a function, and the `*` operator to dereference a function pointer

4

Uses of function pointers

- Systems programming
 - e.g. setting up interrupt vector tables
- Writing generic code
 - A function can take a function as a parameter
 - Example: a generic function to compute integrals

```
double compute_integral(double a, double b, double (*f)(double))
{ /* lots of code */ }

double x_squared(double x) { return x * x; }
double x_cubed(double x) { return x * x * x; }

int main() {
    compute_integral(0, 1, x_squared);
    compute_integral(5, 9, x_cubed);
}
```

5

Function pointer example

```
#include <stdio.h>

void change_array( int *a, int size, int (*f)(int))
{
    int j;
    for( j=0; j < size; j++)
        a[j] = f(a[j]);
}

int add_one(int x) { return x + 1; }
int square(int x) { return x * x; }

int main()
{
    int a[5] = { 0, 1, 2, 3, 4 };

    change_array(a, 5, add_one); // increment every element
    change_array(a, 5, square); // square every element
    return 0;
}
```

Other examples

- `qsort()`, `heapsort()`, `mergesort()` are standard library functions for generic sorting
 - defined in `stdlib.h`
 - They take a comparison function as a parameter
 - They can sort any type of array, as long as an appropriate comparison function is given
- Also available:
 - `lsearch()` : linear search
 - `bsearch()` : binary search through a sorted array

7

String headaches

- Remember that you are responsible for allocating enough space for strings!

- This code crashes: `s1` isn't big enough to hold `s1 + s2`

```
char s1[] = "Any person, ";
char s2[] = "any study."
strcat(s1, s2); // crash
```

- This code works:

```
char s1[1024] = "Any person, ";
char s2[] = "any study."
strcat(s1, s2); // OK
```

8

More string headaches

- Idea: what if we create a wrapper function for `strcat`?
 - What is wrong here?

```
char *my_strcat(char *s1, char *s2) {
    char s3[1024];
    strcpy(s3, s1);
    strcat(s3, s2);
    return s3;
}

int main() {
    char s1[] = "hello", s2[] = "world";
    char *result = my_strcat(s1, s2);
    printf("%s\n", result);
    return 0;
}
```

9

Local variables

- Recall that local variables are stored on the stack
 - Memory for local variables is deallocated when function returns
 - Returning a pointer to a local variable is almost always a bug!

```
char *my_strcat(char *s1, char *s2) {
    char s3[1024];
    strcpy(s3, s1);
    strcat(s3, s2);
    return s3; // BUG! returns a pointer to a deallocated
               // memory region.
}
```

- C requires that the size of variables on the stack be known at compile time, so dynamically-sized arrays aren't possible

10

The Heap

- C can use space in another part of memory: the *heap*
 - The heap is separate from the execution stack
 - Heap regions are not deallocated when a function returns
 - Note: this is completely unrelated to the *Heap data structure*
- The programmer requests storage space on the heap
 - C never puts variables on the heap automatically
 - But local variables might point to locations on the heap
 - Heap space must be explicitly allocated and deallocated by the programmer

11

malloc

- Library function in `stdlib.h`
 - Stands for memory allocate
- Requests a memory region of a specified size
 - Syntax: `void *malloc(int size)`

address of memory region size in bytes

- `void *` is a generic pointer type
 - It cannot be dereferenced
 - But it can be cast to any other pointer type

```
int size = 4;
char *char_array = (char *) malloc(size);
// char_array now points to a memory region
// of 4 bytes allocated to me
```

12

Using heap regions

- You can use the allocated memory however you'd like
 - e.g. treat it as an array, use pointer notation, etc.
 - Example:

```
int main()
{
    int size = 4;
    char *buf = (char *) malloc(size);
    buf[0] = 7;
    scanf("%c", buf+1);
    return 0;
}
```

13

Creating arrays using malloc

- `malloc()`'s parameter is the size in bytes
 - *Not* the number of elements in the array!
 - e.g. an array with 100 integers typically requires 400 bytes
 - Use the `sizeof` operator to find the size of a type, e.g.:

```
float *array =
    (float *) malloc(element_count * sizeof(float));
```

- `malloc()` might fail
 - if there isn't enough memory available to satisfy the request
 - `malloc()` returns `NULL` in this case

14

Example: using malloc

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int item_count = 0;
    float *array;

    printf("How many items do you have? ");
    scanf("%d", &item_count);

    array = (float *) malloc(sizeof(float) * item_count);
    assert(array != NULL);

    for(j = 0; j < item_count; j++)
        scanf("%f", array+j);

    return 0;
}
```

15

free

- `malloc()`'ed memory is *not* freed automatically
 - Even after all pointers to it have been destroyed!
 - Must call `free()` to deallocate memory when done using it

```
void f()
{
    char *p;
    p = (char *) malloc(1000);
}

int main()
{
    while(1)
        f();
}
```

16

free

- `malloc()`'ed memory is *not* freed automatically
 - Even after all pointers to it have been destroyed!
 - Must call `free()` to deallocate memory when done using it

```
void f()
{
    char *p;
    p = (char *) malloc(1000);
    /* some code */
    free(p);
}

int main()
{
    while(1)
        f();
}
```

17

Memory leaks

- You need a call to `free()` for every call to `malloc()`
- Forgetting to call `free()` causes a *memory leak*
 - The memory is not being used but still allocated
 - Memory leaks are very common, but often difficult to find
- Other bad memory errors
 - Calling `free()` more than once
 - Calling `free()` on a pointer not returned by `malloc()`

18

strcat() wrapper

- Idea: what if we create a wrapper function for strcat?
 - This version didn't work:

```
char *my_strcat(char *s1, char *s2) {
    char s3[1024];
    strcpy(s3, s1);
    strcat(s3, s2);
    return s3;
}

int main() {
    char s1[] = "hello", s2[] = "world";
    char *result = my_strcat(s1, s2);
    printf("%s\n", result);
    return 0;
}
```

19

strcat() wrapper

- Solution: allocate the result on the *heap*
 - But the caller is responsible for free'ing it

```
char *my_strcat(char *s1, char *s2) {
    int length = strlen(s1) + strlen(s2) + 1;
    char *s3 = (char *) malloc(length);

    strcpy(s3, s1);
    strcat(s3, s2);
    return s3;
}

int main() {
    char s1[] = "hello", s2[] = "world";
    char *result = my_strcat(s1, s2);
    printf("%s\n", result);
    free(result);
    return 0;
}
```

20

An aside: Memory management in Java

- Java has similar memory management concepts
 - Primitive types and references are stored on the stack
 - You can allocate space on the heap using *new*

```
String getString() {
    return new String();
}

void main() {
    String s = getString();
    /* do stuff with the string */
}
```

- But you don't have to worry about deallocating memory. Why?

21

Garbage collection

- Java uses a technique called *garbage collection*
 - The JVM occasionally scans for heap space no longer in use
 - It frees objects not pointed to by any reference
 - This *garbage collector* is run as a background thread
- Java was specifically designed for garbage collection
 - The JVM can figure out whether or not an object is in use
 - This isn't possible in C because of pointer arithmetic

22

Garbage collection in C

- Pointers make garbage collection difficult or impossible
 - It's very difficult to determine whether memory is still being used

```
char *s = (char *) malloc(1024);
s = s - 10000;
// nothing points to the allocated memory
// region. Could it be garbage collected?
s = s + 10000;
// no, because now something points to it again!
```

- Java's *references* are a restricted form of pointers that don't allow arithmetic, just because of this issue
- There are garbage collecting libraries for C, but they aren't guaranteed to work with any program

23

Memory management: Java vs. C

Java	C
- Pro - Easier to program - Memory leaks impossible - Easier to write secure programs - Con - Slower - No control over GC - Not suitable for systems programming	- Pro - Programmer can control, optimize memory use - Faster - Con - More difficult to learn - Takes discipline to write safe, secure code

24

Multidimensional arrays

- C lets you create arrays of any dimensionality
 - e.g. to create a 3-dimensional array on the stack

```
int array[10][20][30];
```

- You can also initialize multidimensional arrays, e.g.

```
int array[][] = { { 1, 2, 3 }, { 4, 5, 6 } }
```

creates a 2-D array that looks like:

1	2	3
4	5	6

25

Accessing array elements

- Access array elements using brackets []

```
int i, j, k, array[10][20][30];
for(i=0; i<10; i++)
    for(j=0; j<20; j++)
        for(k=0; k<30; k++)
            array[i][j][k] = i+j+k;
```

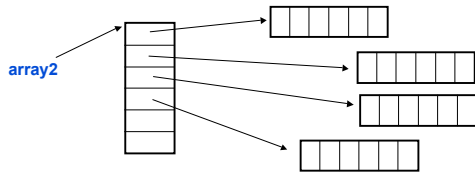
26

Multidimensional arrays and pointers

- Recall: a 1-D array is just a pointer
- A 2-D array is just an array of arrays

```
int array2[10][20]; // type of array is int **
```

- *array2* is a pointer to a pointer to an integer
- e.g. **array2* or *array2[0]* has type *int **, and is a pointer to the beginning of the first row of the array

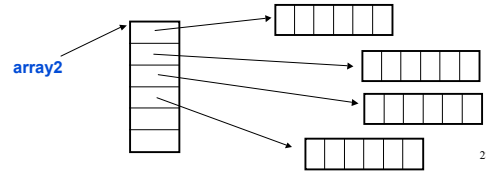


27

Multidimensional arrays on the heap

- Creating a multi-D array on the heap is more involved
 - Setup the array of row pointers and each row array explicitly

```
// create array with 10 rows, 20 columns
int **array2 = (int **) malloc(10 * sizeof(int *));
for(i=0; i<10; i++)
    array2[i] = (int *) malloc(20 * sizeof(int));
```



28