Week 7 - Monday

COMP 2400

Last time

- What did we talk about last time?
- Pointers to pointers
- Returning pointers

Questions?

Project 3

Project 4

Quotes

Don't worry if it doesn't work right. If everything did, you'd be out of a job.

Mosher's Law of Software Engineering

Input with `scanf()`

scanf()

- So far, we have only talked about using **`getchar()`** (and command line arguments) for input
- As some of you have discovered, there is a function that parallels **`printf()`** called **`scanf()`**
- **`scanf()`** can read strings, **`int`** values, **`double`** values, characters, and anything else you can specify with a `%` formatting string

```
int number;
scanf("%d", &number);
```

Why didn't I teach you `scanf()` before?

- In the first place, you have to use pointers (or at least the reference operator **`&`**)
- I wanted you to understand character by character input (with **`getchar()`**) because sometimes that's the best way to solve problems
  - Indeed, **`scanf()`** is built on character by character input
- Crazy things can happen if **`scanf()`** is used carelessly

Format specifiers

- These are mostly what you would expect, from your experience with **`printf()`**

Specifier	Type
`%d`	`int`
`%u`	`unsigned int`
`%o %x`	`unsigned int` (in octal for **o** or hex for **x**)
`%hd`	`short`
`%c`	`char`
`%s`	null-terminated string
`%f`	`float`
`%lf`	`double`
`%Lf`	`long double`

scanf() examples

```c
#include <stdio.h>

int main ()
{
  char name[80];
  int age;
  int number;

  printf("Enter your name: ");
  scanf("%s",name);
  printf("Enter your age: ");
  scanf("%d",&age);
  printf("%s, you are %d years old.\n", name, age);
  printf("Enter a hexadecimal number: ");
  scanf("%x",&number);
  printf("You have entered 0x%08X (%d)\n", number, number);

  return 0;
}
```

Return value for `scanf()`

- **`scanf()`** returns the number of items successfully read
- Typically, **`scanf()`** is used to read in a single variable, making this value either **0** or **1**
- But it can also be used to read in multiple values

```
int value1, value2, value3;
int count = 0;

do {
  printf("Enter three integers: ");
  count = scanf("%d%d%d",&value1, &value2, &value3);
} while( count != 3 );
```

`scanf()` practice

- Write a program that asks a user how many strings they want to enter
  - Read this number with **`scanf()`**
- Then, read in each string with **`scanf()`**
- Print out the string that comes earliest in the dictionary
- **Hint:** We don't need to store all the strings, only the current one and the earliest one we've found
- We can assume that the strings will be no longer than 100 characters (not including the null character)

Dynamic Memory Allocation

malloc()

- Memory can be allocated dynamically using a function called **malloc()**
  - Similar to using **new** in Java or C++
  - **#include <stdlib.h>** to use **malloc()**
- Dynamically allocated memory is on the heap
  - It doesn't disappear when a function returns
- To allocate memory, call **malloc()** with the number of bytes you want
- It returns a pointer to that memory, which you cast to the appropriate type

```
int* data = (int*)malloc(sizeof(int));
```

Allocating single values

- Any single variable can be allocated this way

```
int* number = (int*)malloc (sizeof(int));
double* value = (double*)malloc (sizeof(double));
char* c = (char*)malloc (sizeof(char));
*number = 14;
*value = 3.14;
*c = '?';
```

- But why would someone do that when they could declare the variable locally?

Allocating arrays

- It's much more common to allocate an array of values dynamically
- The syntax is exactly the same, but you multiply the size of the type by the number of elements you want

```
int i = 0;
int* array = (int*)malloc (sizeof(int)*100);
for (i = 0; i < 100; i++) // Initialize for fun
  array[i] = i + 1;
```

Returning allocated memory

- Dynamically allocated memory sits on the heap
- So you **can** write a function that allocates memory and returns a pointer to it

```
int* makeIntArray(int size)
{
  int* array = (int*)malloc (sizeof(int)*size);
  return array;
}
```

`strdup()` example

- **`strdup()`** is a function that
  - Takes a string (a **`char*`**)
  - Allocates a new array to hold the characters in it
  - Copies them over
  - Returns the duplicated string
- Let's write our own with the following prototype

```
char* new_strdup(char* source);
```

free()

- C is not garbage collected like Java
- If you allocate something on the stack, it disappears when the function returns
- If you allocate something on the heap, you have to deallocate it with **free()**
- **free()** does not set the pointer to be **NULL**
  - But you can (and should) afterwards

```
char* things = (char*)malloc (100);
free(things);
things = NULL;
```

Who is responsible?

- Who is supposed to call **`free()`**?
- You should feel fear in your gut every time you type **`malloc()`**
  - That fear should only dissipate when you write a matching **`free()`**
- You need to be aware of functions like **`strdup()`** that call **`malloc()`** internally
  - Their return values will need to be freed eventually
- Read documentation closely
  - And create good documentation for any functions you write that allocate memory

Double freeing

- If you try to free something that has already been freed, your program will probably crash
- If you use data that's already been freed, your program *might* crash
- If you try to free a **`NULL`** pointer, it's fine

- Life is hard.

Upcoming

Next time…

- Practice with dynamic allocation
- Dynamically allocating multi-dimensional arrays

Reminders

- Keep reading K&R chapter 5
- Work on Project 3