Week 7 - Monday

CS222

Last time

- What did we talk about last time?
- Exam 1!
- Before that:
  - Pointers to pointers
  - Lab 6

Questions?

Project 3

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

`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

```
#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 );
```

Returning pointers

- Functions can return pointers
- If you get a pointer back, you can update the value that it points to
- Pointers can also be used to give you a different view into an array

```c
char* moveForward(char* string) {
  return string + 1;
}
```

```c
char* word = "pig feet";
while( *word ) {
  printf("%s\n", word);
  word = moveForward( word );
}
```

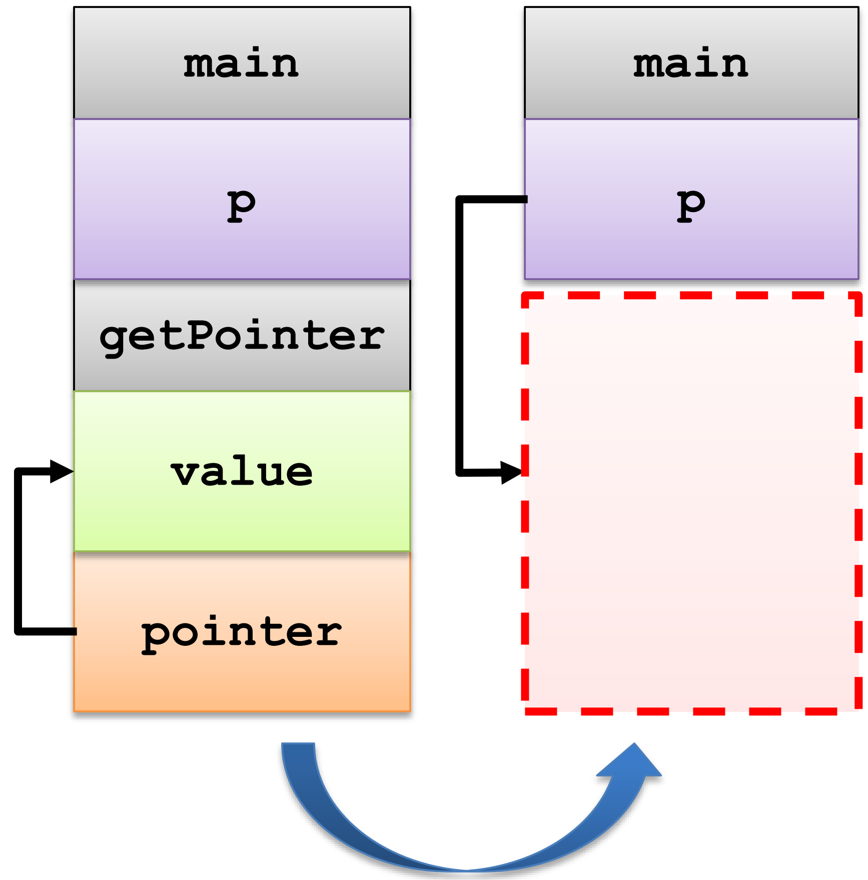
Pointer return problems

- Unfortunately, you can't return a pointer to a local variable
  - Well, you can, but it would be crazy
- It would be pointing to a value that is no longer on the stack
- Maybe it's still there…
- But the next time a method is called, it could be blown away

Stack visualization

```
int* getPointer()
{
  int value = 5;
  int* pointer = &value;
  return pointer;
}
```

```
int* p = getPointer();
```



After **return**

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 is 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++ )
  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* makeArray( 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 afterwards

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

Who is responsible?

- Who is supposed to call **`free()`**?
- You should feel fear in your gut every time you write a **`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 try to free a **`NULL`** pointer, it doesn't do anything

- Life is hard

Memory leaks

- Everything gets freed at the end of your program
- So, you can just hope you don't run out of space
- However, if you are constantly allocating things and never freeing them, you will run out of space

Using dynamic allocation

- Prompt the user for an integer giving the size of a list of numbers
- Dynamically allocate an array of the appropriate size
- Read each of the numbers into the array
- Sort the array
- Print it out
- Free the memory

Upcoming

Next time…

- Dynamic memory allocation examples
- Dynamically allocating multi-dimensional arrays

Reminders

- Keep reading K&R chapter 5
- Keep working on Project 3
  - Due Friday