CMSC 216
Introduction to Computer Systems
Lecture 8
I/O, cont. & Pointers

Jan Plane & Alan Sussman
{jplane, als}@cs.umd.edu

Administrivia

- Project 1 will be graded after late deadline
  - to return before next project due
- Project 2 posted tomorrow, due Oct. 8
  - public tests posted soon after
- Read Reek, Chapter 6: Pointers

Chapter 15, Reek

INPUT/OUTPUT

`scanf()` family

```
int fscanf(FILE *stream, char *fmt, ...);
```
  - reads formatted input from **`stream`**

```
int scanf(char *fmt, ...);
```
  - reads formatted input from **`stdin`**

```
int sscanf(char str[], char *fmt, ...);
```
  - reads formatted input from the string **`str`**

`scanf()` family

- Format strings can contain:
  - whitespace, meaning any whitespace at that point will be skipped
    - many (but not all) skip leading whitespace anyway
  - format specifiers, which cause something to be read
  - any other characters, which are then required in input
- **`scanf()`** does not check for type agreement between the format specifier and the associated variable
- If a given format specifier doesn't match, **`scanf()`** stops processing there

`scanf()` format specifiers

- **`%d`**, **`%x`**, **`%o`**: read a decimal, hex, or octal number into an **`int`**
- **`%f`**: read in a **`float`**
- **`%lf`**: read in a **`double`**
- **`%ld`**: read in a **`long`**
- **`%c`**: read in a **`char`**
- **`%s`**: read in a string (bounded by whitespace)
  - When using **`%s`**, the array parameter does not use **`&`**:

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


Reading data from strings

- Used in combination with **`fgets()`**, you can ensure that you read data line-by-line, instead of simply treating newlines as whitespace
- For what inputs would this loop stop?

```
char buf[1025];
int a, b;
while (fgets(buf, 1024, stdin) != NULL) {
  if (sscanf(buf, "%d %d", &a, &b) == 2) {
    printf("%d %d\n", a, b);
    break;
  }
}
```


The `printf()` family

```
int fprintf(FILE *stream, char *fmt, ...);
```

- sends formatted output to **`stream`**

```
int printf(char *fmt, ...);
```

- sends formatted output to **`stdout`**

```
int sprintf(char *buf, char *fmt, ...);
```

- formats output and places in **`buf`**, adding null byte

```
int snprintf(char *buf, size_t limit,
             char *fmt, ...);
```

- writes at most **`limit`** - 1 characters to **`buf`**, then null byte

Some common format specifiers

%c	print the corresponding argument to **printf** as an unsigned character
%d	print as a decimal integer
%u	print as an unsigned integer
%x	print in hexadecimal (use **%X** for capital A-F)
%f	print in floating point format
%e	print in exponential form (e.g., 6.02300e3)
%s	print as a string (null-terminated character array)
%%	print a % (so %% prints as %)

Controlling formatting

- We can supply a field width, precision, and other flags to format our output exactly as we want
  - **%04x** : format as unsigned hex number, with 4 spaces and zero padding
  - **%-10s** : format as string, allot 10 spaces, left justify (default is right justify)
  - **%6.4f** : format as floating point, allot 6 spaces, 4 digits after the decimal point
- Much more detail available in §15.10.3

Chapter 6, Reek

POINTERS

Pointers

- Pointers are variables whose value is an address
- Every variable is stored at an address in memory
- We use pointers to perform manipulation of memory, by accessing items at the address stored in the pointer

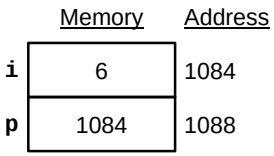
Declaration of pointers

- A pointer to an **`int`** value would be declared like this: **`int *ip;`**
- Creates a variable called **`ip`**, whose type is "pointer to **`int`**"
- We can assign the address of an **`int`** variable to be the value of this new pointer

Pointer operators

- Obtaining the address of an object (**`&`**)
  - Placed before a variable (or an object in memory)
- Accessing the value at an address (**`*`**)
  - Placed before an expression which is either a pointer or otherwise evaluates to an address
- Example:

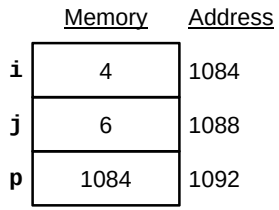
```
int i = 6;
int *p;
p = &i;
printf("%d %d\n", *p, *(&i));
```



Using a dereferenced pointer

- The **`*`** operator can be used on both the left and right sides of an assignment

```
int i = 6;
int j;
int *p;
p = &i;
j = *p;
printf("%d %d\n", i, j);
*p = 4;
printf("%d %d\n", i, j);
```



Garbage pointers

- When a pointer is declared, it points to whatever address was in the memory location allocated for the pointer (no initialization)
- Trying to dereference this random address will generally result in one of three Bad Things:
  - accessing a memory location you don't have permission to access (a "segmentation fault")
  - violating the computer's alignment policies (a "bus error")
  - silent failure: everything appears to work right... for now

`NULL` pointer

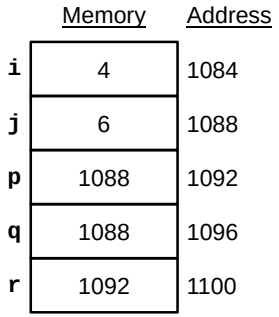
- This is a pointer that points to the address 0, where nothing is allowed to be accessed
- Defined in **`stddef.h`**, which is included by many other header files
- Analogue to Java's **`null`**
  - What happens when you try to call a method of an object which is **`null`**?
  - Very similar thing happens in C when trying to dereference a **`NULL`** pointer; it's usually a segfault
- Just like in Java, you have to check pointers to see if they're **`NULL`** **before** dereferencing them:

```
void f(int *p) {
  if (p != NULL)
    *p = 55;
}
```


Pointers to Pointers

- You can also obtain the address of a pointer variable:

```
int i = 4;
int j = 6;
int *p = &i;
int *q = &j;
int **r = &p;
printf("%d\n", **r);
*r = &j;
printf("%d\n", *p);
```



- This technique will be useful later when working with pointers as parameters