CSE 374 Lecture 10

(C continued - I/O stuff)
I/O : Printf, scanf

Printf and scanf are two I/O functions, prototyped in stdio.h

→ Printf (print-format)
→ int printf(const char *format, ...)
→ ‘Format’ is a string that can contain format tags
→ + additional arguments to match tags
→ Number of arguments better match number of %
→ Corresponding arguments better have the right types (%d, int; %f, float; %e, float (prints scientific); %s, \0- terminated char*; … Compiler might check, but not guaranteed
  ◆ best case scenario: you crash
→ printf("%s: %d %g\n", p, y+9, 3.0)

→ scanf (gets input, formatted)
→ int scanf(const char *format, ...)
→ ‘Format’ is a string that can contain format tags
→ + additional arguments to match tags - should be pointers to the right data type so input can be stored in them
→ scanf("%d %s", &n, str);
→ scanf("%s %d", &a);
  ◆ %s ignores string until space, then reads in an integer

Demo: fopen, fgets
Formatting Tricks

%[flags][width][.precision][length]specifier

Where the specifier character at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

<table>
<thead>
<tr>
<th>specifier</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>d or i</td>
<td>Signed decimal integer</td>
<td>392</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
<td>7325</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal</td>
<td>619</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal integer</td>
<td>7fa</td>
</tr>
<tr>
<td>X</td>
<td>Unsigned hexadecimal (uppercase)</td>
<td>7FA</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point, lowercase</td>
<td>392.65</td>
</tr>
<tr>
<td>F</td>
<td>Decimal floating point, uppercase</td>
<td>392.65</td>
</tr>
<tr>
<td>e</td>
<td>Scientific notation (mantissa/exponent), lowercase</td>
<td>3.9265e+2</td>
</tr>
<tr>
<td>E</td>
<td>Scientific notation (mantissa/exponent), uppercase</td>
<td>3.9265E+2</td>
</tr>
<tr>
<td>g</td>
<td>Use the shortest representation: %e or %f</td>
<td>392.65</td>
</tr>
<tr>
<td>G</td>
<td>Use the shortest representation: %E or %f</td>
<td>392.65</td>
</tr>
<tr>
<td>a</td>
<td>Hexadecimal floating point, lowercase</td>
<td>-0x.c90fep-2</td>
</tr>
<tr>
<td>A</td>
<td>Hexadecimal floating point, uppercase</td>
<td>-0X.C90FEP-2</td>
</tr>
<tr>
<td>c</td>
<td>Character</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>String of characters</td>
<td>sample</td>
</tr>
<tr>
<td>p</td>
<td>Pointer address</td>
<td>08000000</td>
</tr>
<tr>
<td>n</td>
<td>Nothing printed.</td>
<td>.</td>
</tr>
<tr>
<td>%</td>
<td>The corresponding argument must be a pointer to a signed int.</td>
<td>%n</td>
</tr>
<tr>
<td>s%</td>
<td>A % followed by another % character will write a single % to the stream.</td>
<td>s%</td>
</tr>
</tbody>
</table>

From cplusplus.com
Learning about iostreams

https://cplusplus.com/reference/cstdio/

fprintf

int fprintf (FILE * stream, const char * format, ...);

Write formatted data to stream

type

FILE

Object containing information to control a stream

Object type that identifies a stream and contains the information needed to control it, including a pointer to its buffer, its position indicator and all its state indicators.

FILE objects are usually created by a call to either fopen or tmpfile, which both return a pointer to one of these objects.
(f)scanf:

- **Whitespace character**: The function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters). A single whitespace in the format string validates any quantity of whitespace characters extracted from the stream (including none).

- **Non-whitespace character, except format specifier (%)**: Any character that is not either a whitespace character (blank, newline or tab) or part of a format specifier (which begin with %) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character; if the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.

- **Format specifiers**: A sequence formed by an initial percentage sign (%) indicates a format specifier which is used to specify the type and format of the data to be retrieved from the stream and the locations pointed by the additional arguments.

```c
int main()
{
    FILE* ptr = fopen("abc.txt", "r");
    if (ptr == NULL)
    {
        printf("no such file.");
        return 0;
    }

    /* Assuming that abc.txt has content in below format
    NAME AGE CITY
    abc 12 hyderabad
    bef 25 delhi
    cce 65 bangalore */
    char buf[100];
    while (fscanf(ptr, "%s %s %s %s", buf) == 1) {
        printf("%s\n", buf);
    }
    return 0;
}
```

*From geeksforgeeks.org*
Pointer Review

Pointers point to an address in memory &x returns the address

Declare a pointer to a pointer type and it has a specific type/size of memory:

T *x; or T* x; or T * x; or T*x
(T is a type, x is a variable)

An expression to dereference a pointer
*x (more generally *expression)
    Dereference - get the value at the address

Arrays have an implicit pointer type
T = x[n] implies x is of type T*

```c
int var = 349;
int *varptr = &var;
```
Pointers to pointers

Levels of pointers make sense:
I.e.: argv, *argv, **argv
Or: argv, argv[0],
argv[0][0]
But
&(&p) doesn’t make sense

void f(int x) {
    int*p = &x;
    int**q = &p;
    // x, p, *p, q, *q, **q
}

Integer, pointer to integer, pointer to
pointer to integer

&p is the address of ‘p’,

&(&p) would be the address of the
address of p, but that value isn’t stored
separately anywhere and doesn’t have an
address

Try using printf (“The address
of x is %p\n”, &x);
Storage

- Variables need a place to live in memory
- Get ‘allocated’ a physical space in memory (with an address)
- Size of memory allocation depends on datatype
- Get ‘deallocated’ to release the space in memory
Scope

Variables may be accessed by the caller only at certain times - this is scope

Scope and storage are related, but not the same thing

- **Global variables**
  - Scope: entire program
  - Not desirable (violate encapsulation) But can be OK for truly global data like conversion tables, physical constants, etc.

- **Static global variables**
  - Scope: containing file
  - Static functions cannot be called from other files

- **Static local variables**
  - Scope: that function, rarely used

- **Local variables (automatic)**
  - Scope: that block – With recursion, multiple copies of same variable (one per stack frame/function activation)

allocated before main, deallocated after main - memory in ‘global’ block

allocated “when reached” deallocated “after block” - memory in frame on stack
// includes for functions & types
defined elsewhere
#include <stdio.h>
#include "localstuff.h"

// symbolic constants
#define MAGIC 42

// global variables (if any)
static int days_per_month[ ] = { 31, 28, 31, 30, …};

// function prototypes
// (to handle “declare before use”)  
void some_later_function(char, int);

// function definitions
void do_this( ) { … }  
char *return_that(char s[ ], int n) { … }
int main(int argc, char ** argv) { … }
**The stack**

Stack stores active functions & **local** variables

Frames deleted when function returns  
Local variables do not persist

Local variables must have defined size  
Can not make run-time adjustments  
(Arrays must have length)
Local variable initialization

Memory allocation and initialization are not the same thing

Unlike Java, you MUST provide a value to initialize a bit of memory

It is possible to access un-initialized bits unlike Java with sets defaults and checks for initialization

best case scenario: you crash
Arguments

Demo

Storage allocation and variable scope is like local variables (i.e. space is part of the function frame added to the stack, and the variable may be used in the function).

All arguments passed by value. (i.e. a copy of the value is made and assigned to the variable.)
Stack (main)

```c
int main (int argc, char **argv) {
    int mainint = 3;
    int *mainintptr = &mainint;  // hold address of mainint
    int returnint;
}
```

`mainint = 3;`  
`mainintptr =`  
`returnint (random val)`
Stack (demoint)

```c
int demoint (int d) {
    d = d*100;
    return d;
}

// demo functions here

// test passing just the integer
returnint = demoint(mainint);
```

```
d = 3;
mainint = 3;
mainintptr = returnint (random val)
```
Stack (dempointer)

```c
int demopointer (int *p) {
    // p is a pointer
    *p = *p * 10;
    return (*p + 5); // return an int
}
```

```c
// test passing a pointer
returnint = demopointer(mainintptr);
```
Stack (morepointer)

```c
int morepointer (int *p) {
    int d = *p; // value located at p
    int *q = &d; // q is pointer to d
    *p = *q; // value in p equals value of q
    printf ("Addresses %p (p) and %p (q).\n", p, q);
    printf ("Values of \%d (p) and \%d (q).\n", *p, *q);
    *p = *q + 1;
    return *q + 2;
}
```

// test more complicated pointer
returnint = morepointer(mainintptr);

mainint = 30;
mainintptr =
returnint = 35;
Stack (morepointer)

\[ p = \quad d = 30 \quad q = \]

\[
\text{mainint} = 30; \\
\text{mainintptr} = \\
\text{returnint (random val)}
\]

```c
int morepointer (int *p) {
    int d = *p; // value located at p
    int *q = &d; // q is pointer to d
    *p = *q; // value in p equals value of q
    printf("Addresses %p (p) and %p (q).\n", p, q);
    printf("Values of %d (p) and %d (q).\n", *p, *q);
    *p = *q + 1;
    return *q + 2;
}
```
Dangling pointers

Pointers referring to memory that has been released (Demo)

Garbage collecting languages (like Java) only delete memory that is unreachable to avoid this problem.
Dangling Pointers (line 4)

```c
int* f(int x) {
    int *p;
    if (x) {
        int y = 3;
        p = &y; /* ok */
        printf("y=%d and *p=%d\n", y, *p);
    } /* ok, but p now dangling */
    printf("*p=%d\n", *p);
    /* y = 4 does not compile */

    *p = 7; /* could CRASH but probably not */
    printf("*p=%d\n", *p);
    return p; /* uh-oh, but no crash yet */
}

void g(int *p) {
    *p = 123;
    f(123); // now just to demonstrate the stack
    printf("in g: *p=%d\n", *p);
}

int main(int argc, char **argv) {
    g(f(7)); /* HOPEFULLY YOU CRASH (but maybe not) */
}
```
Dangling Pointers (line 7)

```c
int f(int x) {
    int *p;
    if (x) {
        int y = 3;
        p = &y; /* ok */
        printf("y=%d and *p=%d\n", y, *p);
    } /* ok, but p now dangling */
    printf("*p=%d\n", *p);
    /* y = 4 does not compile */
    *p = 7; /* could CRASH but probably not */
    printf("*p=%d\n", *p);
    return p; /* uh-oh, but no crash yet */
}

void g(int *p) {
    *p = 123;
    f(123); // now just to demonstrate the stack
    printf("in g: *p=%d\n", *p);
}

int main(int argc, char **argv) {
    g(f(7)); /* HOPEFULLY YOU CRASH (but maybe not) */
    }
Dangling Pointers (line 13)

```c
int* f(int x) {
    int *p;
    if (x) {
        int y = 3;
        p = &y; /* ok */
        printf("y=%d and *p=%d\n", y, *p);
    } /* ok, but p now dangling */
    printf("*p=%d\n", *p);
    /* y = 4 does not compile */
    *p = 7; /* could CRASH but probably not */
    printf("*p=%d\n", *p);
    return p; /* uh-oh, but no crash yet */
}

void g(int *p) {
    *p = 123;
    f(123); // now just to demonstrate the stack
    printf("in g: *p=%d\n", *p);
}

int main(int argc, char **argv) {
    g(f(7)); /* HOPEFULLY YOU CRASH (but maybe not) */
}
```
Arrays again

“A reference to an object of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T.”

http://c-faq.com/aryptr/aryptrequiv.html

Right: x is the array, which decays to a pointer to an int and &x returns a pointer to the entire array.

```c
void f1(int* p) {   // takes a pointer
    *p = 5;
}
int* f2() {
    int x[3];      // x on stack, is pointer
    x[0] = 5;
    (&x)[0] = 5;   // address of x, points to
                   // same place but different T
    *x = 5;        // put value at location x
    *(x+0) = 5;    // Also put value at x
    f1(x);
    f1(&x);       // wrong – watch types!
    x = &x[2];    // No! X isn’t really a pointer
    int *p = &x[2];
    return x;    // correct type, but is a
                  // dangling pointer
}
```
Pointer arithmetic

- If $p$ has type $T*$ or $T[]$ and $*p$ has type $T$
- If $p$ points to one item of type $T$, $p+1$ points to a place in memory for the next item of type $T$
  - So, $p[0]$ is one item of type $T$, $p+i = p[i]$
- $T[]$ always has type $T*$, even if it is declared as $T[]$
  - Implicit array promotion

Result: Arrays are always passed by reference, not by value. (The information passed is the address of where the values are stored.)