Week 8 Lecture 2: Testing!

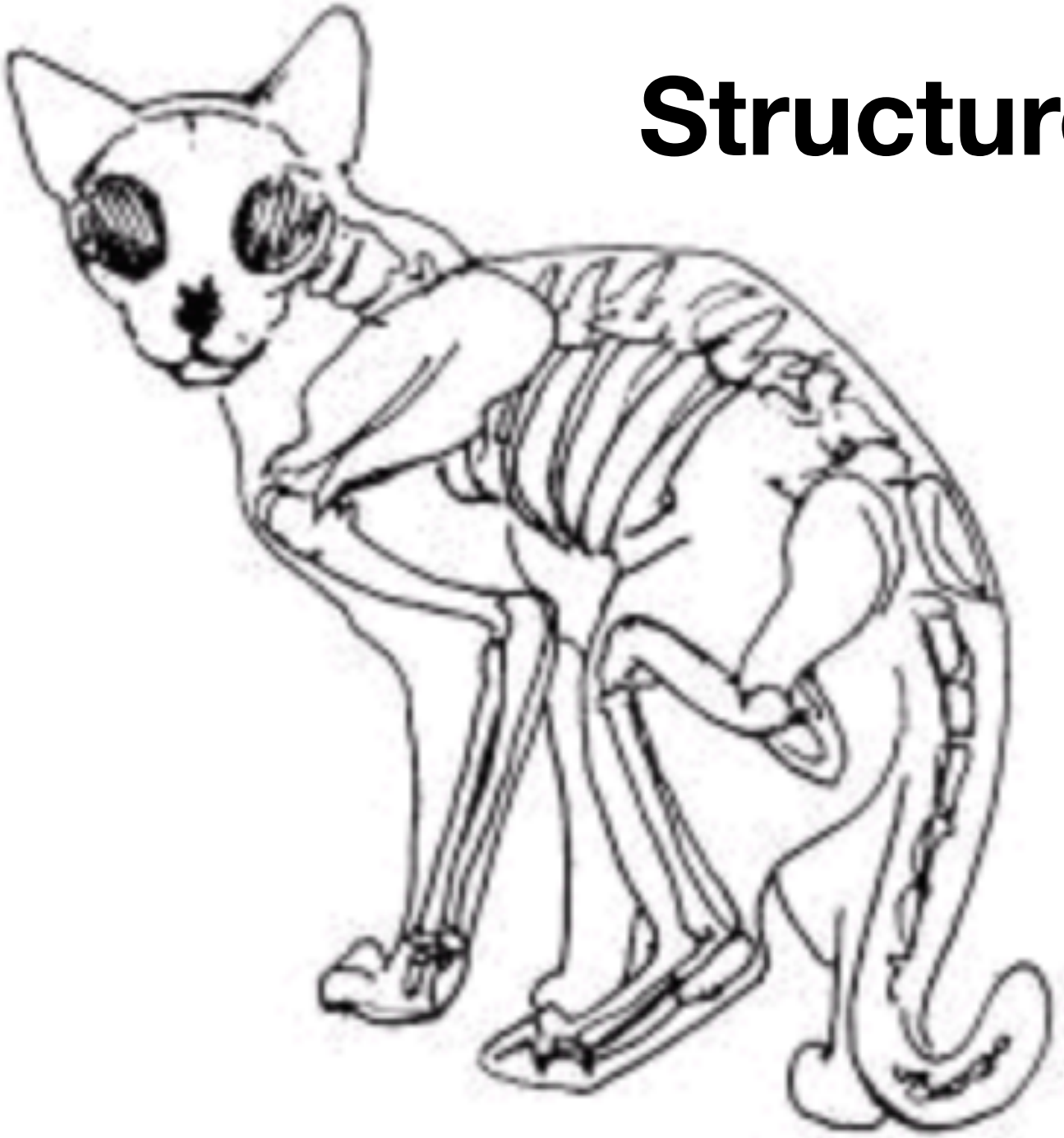
Review and Preview

- Structures
- Assignment #3

- Structures and Functions
- Testing

K&R: Chapters 1-5
Sections 4.5, 4.11
Sections 5.1-5.5
Sections 6.1-6.3

Structures in C



Structures

- A **structure** is a *user-defined* datatype in C.
- It allows a combination of data of different types unlike an array that contains only data of the same type.

```
struct tag {
    member variable
    member variable
};
struct tag strName;
```

The *tag* names this structure and can be used subsequently as a shorthand for the part of the declaration inside the braces.

The *member variables* can be of any type including other structures!

Declaration of a variable of type *tag*

Structure Example

```
struct student {
    char firstName[30];
    char lastName[50];
    unsigned int idNum;
    unsigned int semesterLevel;
};
struct student singleStudent;

singleStudent.idNum = 16377809;
singleStudent.semesterLevel = 1;
strcpy (singleStudent.firstName, “Charles”);
strcpy (singleStudent.lastName, “Brown”);
```

Create the new “type"

Declare a variable of type “student”

Assign values to parts of singleStudent

```
#define MAXLENGTH 20
struct popNames {
    int year;
    int rank[200];
    char maleName[200][MAXLENGTH];
    int maleNumber[200];
    char femaleName[200][MAXLENGTH];
    int femaleNumber[200];
};
struct popNames popular;
```

Reading Information into a Structure

```
if ( (f1 = fopen(argv[1], "r")) != NULL ) {
  index = 0;
  while ( fgets(string, 100, f1) != NULL ) {
     sscanf (string, "%d %s %s %s %s",
          &popular.rank[index],
          popular.maleName[index], maleSNumber,
          popular.femaleName[index], femaleSNumber);
     index++;
   }
}
```

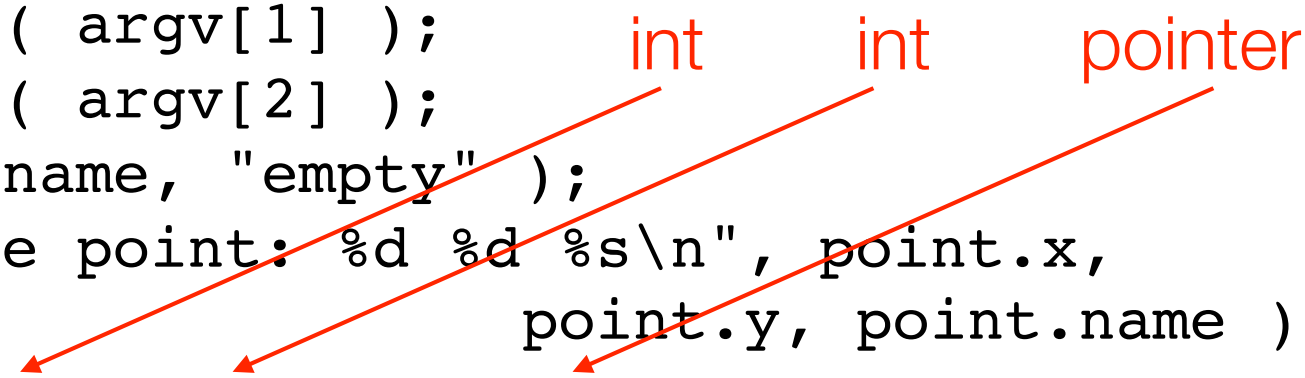
```
for ( i=0; i<10; i++ ) {
    printf ("%d (%s %d) (%s %d)\n",
    popular.rank[i],
    popular.maleName[i], popular.maleNumber[i],
    popular.femaleName[i], popular.femaleNumber[i]);
}
```

```
./example 1880Names.txt
1 (John 89950) (Mary 91668)
2 (William 84881) (Anna 38159)
3 (James 54056) (Emma 25404)
4 (George 47651) (Elizabeth 25006)
5 (Charles 46656) (Margaret 21799)
6 (Frank 30967) (Minnie 21724)
7 (Joseph 26292) (Ida 18283)
8 (Henry 24139) (Bertha 18263)
9 (Robert 24074) (Clara 17717)
10 (Thomas 23750) (Alice 17142)
```

Structures and Functions

- There are three different ways to pass structures or parts of structures to functions.
  - Pass individual member variables of the structure by value to the function.
  - Pass individual member variables of the structure by reference (pointer) to the function.
  - Pass the entire structure by reference (pointer) to the function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int nameMe ( int, int, char * );
int main ( int argc, char *argv[] ) {
    struct stuff {
        int x;
        int y;
        char name[10];
    };
    struct stuff point;
    point.x = atoi ( argv[1] );
    point.y = atoi ( argv[2] );
    strcpy ( point.name, "empty" );
    printf ( "Before point: %d %d %s\n", point.x,
                                    point.y, point.name );
     nameMe ( point.x, point.y, point.name );
    printf ( "After point: %d %d %s\n", point.x,
                                    point.y, point.name );
    return ( 0 );
}
```

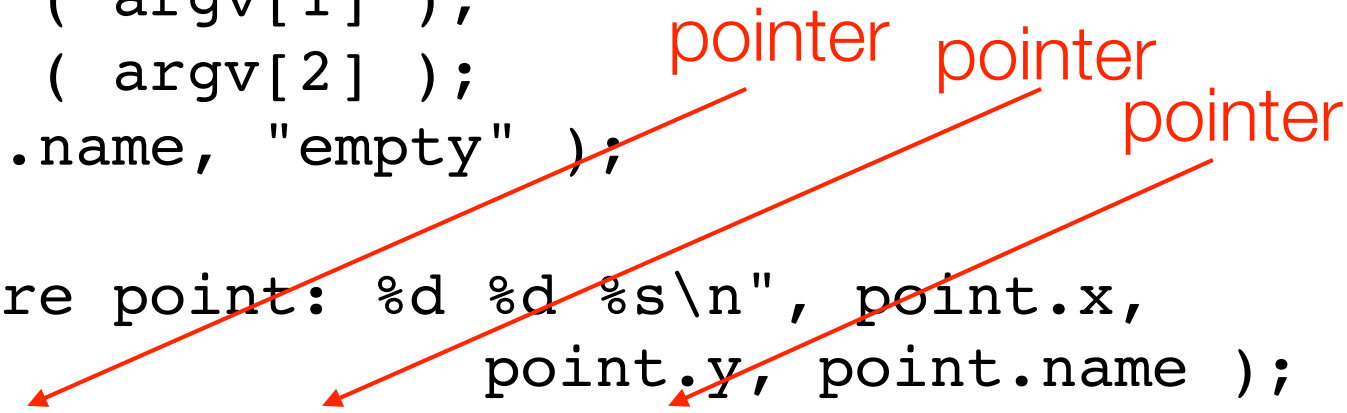


```c
#include <string.h>

int nameMe ( int a, int b, char *name ) {
    if ( a < 20 && b > 10 ) {
        strcpy ( name, "Bob" );
    } else {
        strcpy ( name, "Robert" );
    }
    return ( 0 );
}
```

```
$ gcc -c -ansi -Wall nameMe.c
$ gcc -ansi -Wall -o testSF
                    testSF.c nameMe.o
```

```
$ ./testSF 10 20
Before point: 10 20 empty
After point: 10 20 Bob

$ ./testSF 20 10
Before point: 20 10 empty
After point: 20 10 Robert
```

```
int nameMe2 ( int *, int *, char * );

int main ( int argc, char *argv[] ) {
   struct stuff {
      int x;
      int y;
      char name[10];
   };
   struct stuff point;

   point.x = atoi ( argv[1] );
   point.y = atoi ( argv[2] );
   strcpy ( point.name, "empty" );

   printf ( "Before point: %d %d %s\n", point.x,
                                  point.y, point.name );
   nameMe2 ( &point.x, &point.y, point.name );
   printf ( "After point: %d %d %s\n", point.x,
                                  point.y, point.name );
   return ( 0 );
}
```



```c
#include <string.h>

int nameMe2 ( int *a, int *b, char *name ) {
    if ( *a < 20 && *b > 10 ) {
        strcpy ( name, "Bob" );
        *a = 1;
        *b = 2;
    } else {
        strcpy ( name, "Robert" );
        *a = 3;
        *b = 4;
    }
    return ( 0 );
}
```

Dereferencing the pointer

```
$ gcc -c -ansi -Wall nameMe2.c
$ gcc -ansi -Wall -o testSF2
          testSF2.c nameMe2.o
```

```
$ ./testSF2 10 20
Before point: 10 20 empty
After point: 1 2 Bob

$ ./testSF2 20 10
Before point: 20 10 empty
After point: 3 4 Robert
```

```
struct stuff {
    int x;
    int y;
    char name[10];
};
int nameMe3 ( struct stuff * );

int main ( int argc, char *argv[] ) {
    struct stuff point;
    point.x = atoi ( argv[1] );
    point.y = atoi ( argv[2] );
    strcpy ( point.name, "empty" );
    printf ( "Before point: %d %d %s\n", point.x, point.y,
                                         point.name );
    nameMe3 ( &point );
    printf ( "After point: %d %d %s\n", point.x, point.y,
                                        point.name );
    return ( 0 );
}
```

Structure pointer

Pointer to the structure

```c
#include <string.h>
struct stuff {
    int x;
    int y;
    char name[10];
};
int nameMe3 ( struct stuff *p ) {
    if ( p->x < 20 && p->y > 10 ) {
        strcpy ( p->name, "Bob" );
        p->x = 1;
        p->y = 2;
    } else {
        strcpy ( p->name, "Robert" );
        p->x = 3;
        p->y = 4;
    }
    return ( 0 );
}
```

```
./testSF3 10 20
Before point: 10 20 empty
After point: 1 2 Bob
$ ./testSF3 20 10
Before point: 20 10 empty
After point: 3 4 Robert
```

```
$ gcc -c -ansi -Wall nameMe3.c
$ gcc -ansi -Wall -o testSF3 testSF3.c nameMe3.o
```

```
struct stuff {
    int x;
    int y;
    char name[10];
};
struct stuff * nameMe4 ( struct stuff *p );

int main ( int argc, char *argv[] ) {

    struct stuff point, *pt;

    point.x = atoi ( argv[1] );
    point.y = atoi ( argv[2] );
    strcpy ( point.name, "empty" );

    printf ("Before: %d %d %s\n", point.x,point.y,point.name);
    pt = nameMe4 ( &point );
    printf ("After: %d %d %s\n", pt->x, pt->y, pt->name );
    printf ("After: %d %d %s\n", point.x,point.y,point.name);

    return ( 0 );
}
```

Returns a pointer to a structure

Pointer to a structure

Pointer to the structure

```
#include <string.h>
struct stuff {
    int x;
    int y;
    char name[10];
};

struct stuff * nameMe4 ( struct stuff *p ) {
    p->x = 100;
    p->y = 200;
    strcpy ( p->name, "Harriet" );

    return ( p );
}
```


DESTROYED THE HOUSE
The Ultimate Tester
Testing
TIME FOR A NAP
com

Introduction to Testing

..testing can demonstrate the presence of bugs, but not their absence....Edsger Dijkstra

- Debugging and testing are not the same thing!
- Testing is a systematic attempt to break a program.
  - Bug-free programs by construction are the goal but not possible so we must test!

Introduction to Testing

- It is not unusual for developers to spend 40% of the total project time on testing.
    - For life-critical software (*e.g.* flight control, reactor monitoring), testing can cost 3 to 5 times as much as all other activities combined.
- Since testing is basically destructive in nature, it requires that the tester discard preconceived notions about the software being tested.

Software Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an error.
- A successful test is one that uncovers an as yet undiscovered error.
- But, testing cannot show the absence of defect -- it can only show that software defects are present.

Software Testing Objectives

- Testing should systematically uncover different classes of errors in a minimum amount of time and with a minimum amount of effort.

- A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specification.

The Nature of Software Defects

- Typographical errors are random.
- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- General processing tends to be well understood while special case processing tends to be prone to errors.

Steps to Testing Nirvana

- **Think** about potential problems **as you design** and implement.
    - Make a note of them and develop tests that will exercise these problem areas.
- **GIGO** - what happens when garbage goes in?
- What happens when something is not there?
    - *e.g.* command line arguments

Steps to Testing Nirvana

- **Document** all loops and their boundary conditions, all arrays and their boundary conditions, all variables and their range of permissible values.

- Pay particular attention to **parameters** from the command line and into functions and what are their valid and invalid values.

Steps to Testing Nirvana

- Test **systematically**, starting with easy tests and working up to more elaborate ones.
- **Test as you write** -- the earlier that errors are detected, the easier they are to locate and fix.
- **Document** your tests so that you can repeat them constantly as the code grows and changes.
    - This is called **regression testing**.

Tricks of the Trade

- **Test boundary conditions**
  - loops and conditional statements should be checked to ensure that loops are executed the correct number of times and that branching is correct
  - if code is going to failure, it usually fails at a boundary
  - check for off-by-one errors, empty input, empty output

Tricks of the Trade

- **Test pre- and post-conditions**
  - before and after a critical piece of code, check to see if the necessary pre- and post-conditions (value of variables, etc) exist
  - remember a famous divide by zero error

On the USS Yorktown (guided-missile cruiser), a crew member entered a zero, which resulted in a divide by zero, the error cascaded and shut down the propulsion system and the ship drifted for a couple of hours.

Tricks of the Trade

- **Program defensively**
  - Test for cases which “can't occur” just in case!
  - Test for extreme values - negative values, huge values, etc

Tricks of the Trade

```
if ( grade < 0 || grade > 100 ) /* impossible grades! */

        letter = '?';

else if ( grade >= 80 )

        letter = 'A';

else   …
```

Always check for NULL pointers, out of range subscripts, divide by zero.

Tricks of the Trade

- **Check returns**
  - Always look at the values returned from library functions and system calls.
  - Remember to check for output errors and failures as well as input failures.

Strategies

- **Test incrementally.**
  - Write, test, add more code, test again, repeat. And repeat. And repeat.

http://3.bp.blogspot.com/-uUuUxmCT-K0/UFThsJghKII/AAAAAAAAAww/Lo0Z_OUj1G4/s1600/repeater6.jpg

Strategies

- **Test simple parts first**
  - Focus on testing the simplest features and only move on when these are working.
  - Enumerate the possible scenarios for the code and design tests for all of them.
  - Test individual functions (another good reason to modularize) using testing code to rigorously and automatically test a function or piece of code.

```c
#include <stdio.h>
#include <string.h>
#include "textProc.h"

#define MAXSIZE  500  /* max size of input line */

int main ( int argc, char *argv[] ) {
   char line[MAXSIZE];
   int ret;

   while ( fgets(line, MAXSIZE, stdin) != NULL ) {
      ret = replaceDigits ( line );
      printf ( "%d - %s", ret, line );
   }
   return ( 0 );
}
```

Test 1 - Replace digits with spaces and check return value and text

---

replaceDigitsTestFile.txt

This is test #19 in test
suite 17 on this day in
2019 and for 2019 and 2020.
Test: 0123456789

```
$ cat replaceDigitsTestFile.txt | ./testReplaceDigits
```

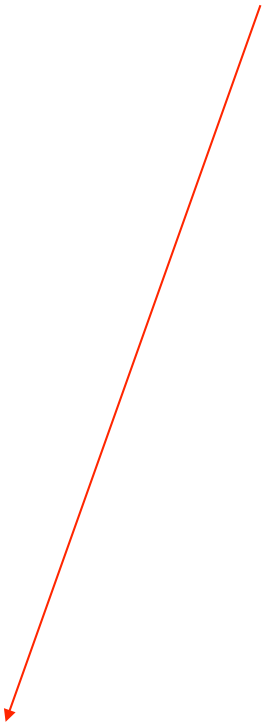
2 - This is test # in test
2 - suite on this day in
12 - and for and .
10 - Test:

```c
#include <stdio.h>
#include <string.h>
#include "textProc.h"

#define MAXSIZE  500     /* max size of input line */

int main ( int argc, char *argv[] ) {
   char line[MAXSIZE];
   int ret = 0;

   while ( fgets(line, MAXSIZE, stdin) != NULL ) {
      ret = replaceDigits ( line );
      printf ( "%s", line );
   }

   return ( 0 );
}
```

Test 2: Check that the digits are replaced with spaces

```
$ cat replaceDigitsTestFile.txt | ./test2ReplaceDigits > testfile.txt
```

```
$ wc replaceDigitsTestFile.txt testfile.txt
  4  20  94 replaceDigitsTestFile.txt
  4  16  94 testfile.txt
  8  36 188 total
```



> is called **redirection**

The output on stdout is put into the file testfile.txt

Test 2: Check that the digits are replaced with spaces

```
$ od -c testfile.txt
0000000  T   h   i   s       i   s       t   e   s   t       #
0000020      i   n       t   e   s   t  \n   s   u   i   t   e
0000040          o   n       t   h   i   s       d   a   y       i   n
0000060 \n                   a   n   d       f   o   r
0000100              a   n   d                   .  \n   T   e   s
0000120  t   :                                   \n
0000136
```

od (octal dump) is a UNIX command that “dumps” the contents of a file using different interpretations of the bytes. In this case, the -c flag means to print bytes as printable ASCII characters (1 byte at a time).

Testing Strategies

- **Know what output to expect**
  - It is obvious that you cannot know if your program is correct unless you know what output it should produce in all situations.
  - As programs become bigger and more complicated, this becomes harder and harder to do.

Testing Strategies - Use Tools

- Use programs like **`cmp`** (compare files for identity) and **`diff`** (report differences) to compare against known results.

- Get to know shell commands, shell scripting, and the "little languages" that can help you construct test scripts.