CS113 Assignment 2 — Functions, Pointers, Arrays, Spring 2008
Due Friday February 8, 2008, 11:59:59PM

Solve <u>two</u> of the following three problems. You may work with a partner if you wish.

1 Go Fish!

Go fish! is a simple children's game played with a deck of 52 playing cards. Each card has a number between 1 and 13, and the deck consists of four cards of each number. At the start of the game the deck is shuffled and then seven cards are dealt to each player. The remaining cards are placed face down on the table.

A game of Go fish! is a sequence of *turns* alternating between players A and B. Player A's turn comes first, and proceeds as follows:

1. If player A has no cards in hand, then he or she draws a single card from the deck.
2. Player A chooses a number n appearing on one of the cards in his or her hand, and then asks the other player for cards of that number. For example, if player A has a 5, he or she could ask, "Do you have any 5's?"
   - If player B has any cards with number n, he or she must give all of them to player A. Player A continues the turn at step 2.
   - If player B has no cards with number n, he or she says "Go fish!". Player A then draws a single card from the deck and proceeds to step 3.
3. Player A puts any *books* in his or her hand face up on the table. A *book* is a a set of four cards with the same number – four 5's, for example.

Player A's turn is then finished, and it is player B's turn to perform the three steps above. Play alternates back and forth between the two players until all 52 cards have been formed into books — that is, when the deck and both players' hands are empty. The player who has the most books at the end of the game wins.

Write a C program that implements Go Fish! The program should allow the user (player A) to play against the computer (player B). Here's an example of how your program might work:

```
Welcome to Go Fish!

Shuffling the deck...
Dealing cards...
Ready to play!

---- Your turn
Your hand is:  3 5 8 8 10 13 13
Your books are:
Computer's books are:

Which card would you like to ask for?  9
Error: You don't have any 9's! Try again.

Which card would you like to ask for?  8

-> The computer hands you 1 card.
-> Your hand is now:  5 8 8 8 10 13 13

Which card would you like to ask for?  10
```

```
-> The computer hands you 3 cards.
-> Your hand is now:  5 8 8 8 10 10 10 10 13 13

Which card would you like to ask for?  5

-> The computer says Go Fish!
-> You draw a 6.
-> You lay down a book of 10's.

---- Computer's turn
The computer asks you for your 4's.
You say go fish!

---- Your turn
Your hand is:  5 6 8 8 8 13 13
Your books are: 10's
Computer's books are:  3's

Which card would you like to ask for?

[ ... lots of interactions later ... ]

---- Your turn
Your hand is:  8 8 8
Your books are: 5's 6's 7's 9' 10's
Computer's books are:  1's 2's 3's 4's 11's 12's 13's

Which card would you like to ask for?  8

-> The computer hands you 1 card.
-> You lay down a book of 8's.

Game over! The computer won.
```

Your output need not match this example exactly. The computer user may use a very simple strategy: during step 1 of its turn, it should choose n at random from the cards in its hand. Submit your solution as `gofish.c`.

Hints: You will need some arrays to store the deck, each player's hand, and each player's set of books. To shuffle, you will need a random number generator; a good choice is `rand()` which is available in `stdlib.h`. If you need to sort the contents of an array, use `qsort()` (also available in `stdlib.h`).

2 Encryption

An *encryption algorithm* converts a string of characters into a new string that is unintelligible. However, given the encrypted string and knowledge of the encryption key, a user can use a *decryption algorithm* to reconstruct the original.

Here is a very simple encryption algorithm:

1. If the string has an odd number of characters, add a space to the end. So "`HELLO`" becomes "`HELLO `".
2. Then reverse the string. For example, "`HELLO `" becomes " `OLLEH`".
3. Next, rotate the string forward by $\frac{n}{2}$ characters, where n is the number of characters in the string. In other words, the character at position 0 is moved to position $\frac{n}{2}$, position 1 is moved to position $\frac{n}{2}+1$,

etc. When the end of the string is reached, characters wrap around to the beginning. For example, " `OLLEH`" becomes "`LEH OL`".

4. Finally, take the *exclusive or* (XOR) between each character's ASCII value and the key k. XOR is computed in C using the caret operator (`^`). For example if $k = 10$, then "`LEH OL`" becomes "`FOB*EF`".

An interesting property of this algorithm is that it works for both encryption and decryption. That is, running the algorithm twice with the same key produces the original, unencrpyted string.

Write a program that implements this encryption algorithm. The program should ask the user for a key and a string. It should then compute and display the encrypted string. Assume that the key is an integer in the range $[0, 31)$. For simplicity, assume that the original input string contains only uppercase letters, numbers, and spaces.

Here's a sample program execution:

```
Enter a key: 10
Enter a strint to encrypt or decrypt: THE RUTABAGA BLOOMS AT MIDNIGHT.

Encrypted string: EFH*KMKHK^_X*OB^$^BMCDNCG*^K*YGE
```

Here's another example, showing decryption of the same string:

```
Enter a key: 10
Enter a strint to encrypt or decrypt: EFH*KMKHK^_X*OB^$^BMCDNCG*^K*YGE

Encrypted string: THE RUTABAGA BLOOMS AT MIDNIGHT.
```

Submit your solution as `crypt.c`.

3 Buffer overruns

This problem is difficult and involves material beyond the scope of the class. You may need a CS or ECE background in order to solve it.

As we discussed in class, C makes no attempt to detect or prevent array bounds errors. A particularly dangerous scenario is when a program reads from an input device (like a keyboard or network socket) into a fixed-size buffer. An unexpectedly large input could cause a *buffer overrun*. For example, consider the following C code snippet:

```
char buf[1024];
// get a line of input from the keyboard
gets(buf);
```

If a user enters more than 1024 characters, the `gets()` function continues to write to memory locations beyond the end of the buffer. This bug could cause the program to crash, or it might cause other variables to change unexpectedly. Worse, it introduces a security hole: a cleverly-designed string could modify the execution stack and change the behavior of the program!

In this problem, consider the following program that performs a very simple form of password authentication:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char secret_password[] = "fishstix";

// read in a string, then check whether it is
// the secret password
```

```
//
int check_password()
{
  char buf[10];

  gets(buf);
  buf[strlen(buf)] = 0;  // remove newline character from end of string

  return !strcmp(buf, secret_password);
}

int main()
{
  int valid = check_password();

  if(valid)
    printf("Access granted!\n");
  else
    printf("Access denied!\n");

  return 0;
}
```

Obviously `secret_password` is supposed to be kept secret. However, the program's use of `gets()` into a fixed-size buffer of length 10 means that a buffer overrun is possible.

Design a clever input string that causes the program to display the secret password. You may not modify the program itself. Submit a file called `overrun.txt` that includes the string as well as a detailed description of how it works. Make sure to mention which compiler and machine architecture you are using. (It is fine if your solution only works with one particular architecture and/or compiler.)

4 What to submit

Submit any two of the following files:

1. `crypt.c`
2. `grades.c`
3. `overrun.txt`