CSE 303 – Autumn 2005
Midterm Key

Grades on questions involving programming we based on my estimate of how far from a working solution the answer was.

1. [4 points]
   Suppose your shell has a current working directory of `/tmp`.
   (a) Give a single command that will list all files in `/tmp` whose names contain '303'.

   *ls *303**

   (b) Give a single command that will list all files in `/tmp/courses` whose contents contain '303'.

   *The expected answer: grep 303 courses/**
   (That prints more than the file names, though. grep –l 303 courses/ will print just the file names. There was no reason to know that switch, though.)*

   (c) Give a single command that will list the names of all non-hidden files in your home directory.

   ls ~

   (d) Give a single command that will print the full path name of your home directory.

   echo ~

2. [6 points]
   Most commands issued to the shell cause it to launch the program whose name is the first token of the command line. The shell uses `$PATH` to look for an executable with that name.

   (a) Suppose I want to add the directory `/uns/bin` to my `$PATH`. Give a command that will do that.

   PATH="$PATH:/uns/bin"

   (b) Typing that command into a shell adds the directory to `$PATH` only for that one shell – any other shells running at that time, and any new shells started later, will not have `/uns/bin` in their `$PATH`. Why?

   Each process has its own set of environment variables.

3. **(c)** Suppose I want all shells I launch from now on to have `/uns/bin` in their `$PATH`. What should I do to make that happen?

   Edit ~/.bash_profile, adding the line that is the solution to (a).
   (bash executes the commands in that file when starting.)
   (A number of people answered using 'export'. That affects whether ror not the environment variable will be inherited by forked subprocesses, which isn't what we're interested in here.)

4. [8 points]
   In HTML files (web pages), the characters '<' and '>' are special. If you want either of them to be displayed, you must write '<' and '>' respectively.

(a) Write a C program that:
* reads stdin and writes stdout
* replaces each '<' with '<', and each '>' with '>'
* passes through all other characters unchanged

Note: `getc(stdin)` will read a single character, including '\n'. The printf format '%c' writes a single character.

```
#include <stdio.h>
int main( int argc, char* argv[] ) {
   char c;
  while ( (c=getc(stdin)) != EOF ) {
      if ( c == '>' ) printf( "&gt;" );
     else if ( c == '<' ) printf( "&lt;" );
     else printf( "%c", c );
  }
 return 0;
}
```

(b) Suppose your program for (a) works, and has been compiled into an executable named `trans` in a directory that is on your $PATH. Give a command that will take the file `./foo.c` and produce `./foo.c.html`, the same contents but with all '<' and '>' characters translated.

```
./trans <foo.c >foo.c.html
```

(c) Write a shell script that will let you enter commands that cause more than one file to be translated. For example, if the script were called `dotrans`, you could say

```
$ ./dotrans foo.c sub.c pilot.c
```

and files `foo.c.html, sub.c.html`, and `pilot.c.html` would be produced.

```
#!/bin/bash
for f in "$@"; do
   ./trans <$f >$f.html
done
```

5. [7 points]
The sample mean, m, of a list of N integers, x_1, x_2, …, x_N is $(x_1+ x_2+…+x_N)/N$. The sample variance is $((x_1-m)^2+ (x_2-m)^2+…+(x_N-m)^2)/(N-1)$. Here is some C code whose goal is to compute the sample mean and variance, given a list of integers:

```
#include <stdio.h>
int main( int argc, char* argv[] ) {
     int     argIndex;
     int     nSamples = argc - 1;
     double  mean = 0.0;
     double  var = 0.0;
     int     sample[10];

     for (argIndex=1; argIndex<argc; argIndex++) {
        sample[argIndex] = atoi( argv[argIndex] );
        mean += sample[argIndex];
     }
     mean = mean / nSamples;

     for ( argIndex=1; argIndex<argc; argIndex++) {
         var += (sample[argIndex]-mean)*(sample[argIndex]-mean);
     }
     if (nSamples >1 ) var = var / (nSamples -1);
     printf( "Sample mean = %lf\nSample variance = %lf\n",
                  mean, var );
     return 0;
}
```

(a) Suppose this code has been compiled into executable `./stats`. Give a command that will cause the sample mean and variance of the three integers 1,2,3 to be printed.

./stats 1 2 3

(b) Assume now that the execution of this code printed the correct results (mean = 2, variance = 1), which in fact it does. There *is* a bug in the code, though. What is it?

An array of 10 elements has been created to hold the arguments. If the user types more than 10 integers, we're write to memory using out of bounds array indices.

(c) The bug can be fixed by changing one line of existing code and adding two new lines. Give the changed line and the two new lines. (I should be able to figure out where they go, so don't worry about telling me that.)

What I was expecting:
```
int* sample;
sample = (int*)malloc(sizeof(int)*argc);
free(sample);
```

Some people answered:
```
int sample[nSamples];
```
That was okay, except there's an off-by-one error (should be 'int sample[argc];').

6. [3 points]
Consider this code:

```
#include <stdio.h>
int* getInitArray( int N ) {
  int  index;
  int  result[N];
  for ( index=0; index<N; index++ ) {
    result[index] = index;
  }
  return result;
}
int main( int argc, char* argv[] ) {
  int index;
  int* p = getInitArray(5);
  for ( index=0; index<5; index++ ) {
    printf( "%d\n", p[index] );
  }
}
```

I can compile it and there are no errors. When I run it, I see this, though:

```
$ ./a
0
1628316064
2289352
1628223944
2289988
```

Why?

getInitArray() returns a pointer to a stack allocated variable. (So, when it returns, that stack space is released. Any routines subsequently called (in this case, printf()) will overwrite the values at the memory locations pointed at by the returned pointer.)

7. [4 points]

(a) Suppose that I compile and see the following:

```
$ gcc -Wall Q5.c
Q5.c: In function `main':
Q5.c:9: warning: implicit declaration of function `printf'
```

What is most likely wrong?

Forgot #include <stdio.h> in Q5.c

(b) Suppose instead I see (just) this printed after issuing the `gcc` command:

```
$ gcc -Wall Q5.c
/tmp/cc8ffnJc.o(.text+0xb6): In function `main':
Q5.c: undefined reference to `sub'
collect2: ld returned 1 exit status
```

What is most likely wrong?

Q5.c includes a call to a routine, sub, but the code for sub is not in Q5.c. (This is a linker error. It is not caused by failing to #include something – that would result in a compiler error, at most.)

8. [4 points]

The following files are very slightly modified versions of those in sample code linked from the syllabus.

structs.h

```
#ifndef STRUCTS_H
#define STRUCTS_H

struct demostruct {
 char  stringField[80];
 char* pStrField;
};
#endif // STRUCTS_H
```

structs.c

```
#include <stdio.h>
#include <string.h>
#include "structs.h"

void printStruct( struct demostruct s)
{
  printf( "stringField = '%s'\n", s.stringField );
  printf( "pStrField = '%s'\n", s.pStrField );

  strcpy( s.stringField, "printStruct was here" );
  strcpy( s.pStrField, "PS was here" );
}

int main( int argc, char* argv[] )
{
  struct demostruct  myStruct;

  strcpy( myStruct.stringField, "literal" );
  myStruct.pStrField = strdup( "malloc'ed memory" );

  printStruct( myStruct );
  printStruct( myStruct );
}
```

What is printed when `structs.c` is compiled and run?

stringField = 'literal'
pStrField = 'malloc'ed memory'
stringField = 'literal'
pStrField = 'PS was here'

[In C, arguments are always passed call-by-value. That means the struct is passed by value. The array it contains is copied. The pointer it contains is copied. Overwriting the array doesn't matter, because the copy is destroyed on return. The second strcpy() writes to where the pointer points, though, and so affects the memory (also) pointed at by the pointer in myStruct.]

9. [4 points]
Give **two** reasons why you might use "conditional compilation" (i.e., `#ifdef` and the like) in a C program.

1. To avoid infinite loops in the preprocessor if there is a cycle of #include's in .h files.
2. To allow multiple versions of the program to be built from a single version of the source: e.g., debug versus release versions.

10. [10 points]
Here is a purposefully vague description of a C subroutine I'd like you to write. Part of this problem is to make the decisions required to go from vague to an actual implementation, given that you're implementing in C.

The subroutine takes an array of strings. It changes each string in the array by prepending it with the value of that string's index in the array. For instance, if the string with index 3 is "monkey business" it becomes something like " 3: monkey business".

To write this routine, you're going to have to make some decisions and assumptions. **State what they are.** (Probably doing that at the end, once you're done, will be easiest, but you can litter these comments about if that helps you. I need to be able to distinguish comments from, say, notes to yourself, though, so try to be reasonably clear if there's anything I should disregard.)

Note: `sprintf( pStr, "%8d", myInt)` will convert the `int` value of `myInt` into a string of exactly 8 characters (i.e., up to 8 decimal digits, padding with spaces if needed) and will write the characters in memory at the location pointed at by pointer `pStr`.

Here's the version I expected. Other versions are possible, by making other assumptions.

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

void sub( char* orig[], int n ) {
   int  index;
  char*  pNew;

 for ( index=0; index<n; index++ ) {
    pNew = (char*)malloc( sizeof(char) * ( strlen(orig[index] + 11 ) );
    sprintf( pNew,"%8d",  index );
    strcat( pNew, ": " );
    strcat( pNew, orig[index] );
    free( orig[index] );
   orig[index] = pNew;
 }
}
```

Assumptions:
1. orig[0..n-1] contains pointers to malloc'ed memory
*2. n < 10**8*
(3. The caller is responsible for eventually freeing the memory pointed at by orig[i], 0<=i<n)

Other approaches:
- don't pass an indication of the array length; instead, the array ends with an element == NULL
- don't modify orig at all, instead pass back an entirely new array
- assume that the elements of orig[] point at malloc'ed memory at least 10 characters larger than the strings they currently hold, and write the result into that memory