Miscellaneous Topics

Control Flow

- There is a goto in C but it is not recommended for several reasons
  - Optimization
  - Performance
  - Clarity
- Plus it will only jump within a single function
- Consider these two example code statements

```
while (a < 5) {
```

```
A: if (a >= 5) goto B;
```

- How does control reach each statement (as in what possible lines executed before each of these statements)?
  - For the while, control must come from the statement preceding the while() or it must come back from the end if the while loop
  - For the labelled statement, control can come from ***anywhere*** in the function! You must search for all the goto statements to see which ones reference A!

Control Flow

- Even worse…
- There is a C library construct that will allow jumping all over the place.
- USE IT SPARINGLY!!!!!
- (This is the underpinning of the Java exception-throwing facility, exposed for all sorts of uses and abuses.)

setjmp() … longjmp()

```
#include <setjmp.h>
jmp_buf x;        // this will hold location x
…
t = setjmp(x);    // mark a location as x
                  // t will be 0 for normal
                  // ctrl flow, non-zero if
                  // coming from longjmp()
…
longjmp(x, 3); // transfer back to location
               // x, and setjmp will have
               // the return value 3 (an
               // integer)
```

Note: Read Wikipedia Article!!!!

setjmp() … longjmp()

- A call to setjmp() saves various information about the calling environment (typically, the stack pointer, the program counter, possibly the values of other registers and the signal mask) in the jmp_buf and returns 0
- longjmp() must be called later in the function OR in a function that's called by the function that did the setjmp(). In other words, the stack frame of the function that called setjmp() must still be on the stack (somewhere) when longjmp() is called
- Longjmp() restores the saved CPU state from jmp_buf while setting the return value to the value in its second argument
- This makes it appear as if setjmp() returns for a SECOND time without being called again, distinguished only by its return value!

Variadic Functions

- We have to have type-conformance of all of our arguments and parameters, right?
- So how the heck does printf() get away with calls like
  - printf(“Hello world!\n”);
  - printf(“Count %d, average %f\n”, ct, avg);
  - printf(“%d %d %d %d\n”, a[0], a[3], a[2], a[1]);

Enter stdarg.h

```
#include <stdarg.h>
void printargs(int arg1, ...);

int main(void) {
    printargs(5, 2, 14, 84, 97, 15, -1, 48, -1);
    printargs(84, 51, -1);
    printargs(-1);
    printargs(1, -1);
    return 0;
}
```

Printargs()

```
void printargs(int arg1, ...)
{
  va_list ap;
  int i;

  va_start(ap, arg1);
  for (i = arg1; i >= 0; i = va_arg(ap, int))
    printf("%d ", i);

  va_end(ap);
  putchar('\n');
}
```

And the result...

```
$ ./a.out
5 2 14 84 97 15
84 51

1
$
```

```
int main(void)
{
   printargs(5, 2, 14, 84, 97,
             15, -1, 48, -1);
   printargs(84, 51, -1);
   printargs(-1);
   printargs(1, -1);
   return 0;
}
```

Question

Which of these statements are true about C functions with variable numbers of arguments?

A. C sets the first function argument to an integer count of the number of arguments passed

B. There must always be at least one argument in the call and in the function prototype

C. It is the responsibility of the called function to figure out how many arguments it was passed

D. B and C

E. All of the above

Today's number: 11111

10

The printf() prototype

- And by the way, the prototype for printf() is simply

```
void printf(char *format, ...);
```

- And note that printf() and its friends use the format string to determine how many arguments are present. Gcc even goes out of its way to check that for you.

Main() Return Value

- 0 means okay
- Anything else means a problem
- Main() can return an 8-bit value
- Any function can exit and set return value

```
exit(99);
```

- You can see the value on the command line with:

```
echo $?
```

Idioms for Opening Files

- In the C environment, there are three files opened for you on pre-defined “streams”, typically connected to your keyboard/display
  - stdin – Standard input
  - stdout – Standard output – printf(…) defaults to stdout
  - stderr – Standard error output – use fprintf(stderr, …)
- Any other files must be opened and closed by calling a Standard IO routines

```
FILE *infile;
if( (infile = fopen(“f.txt”,”r”)) == NULL){
        // Handle error
}

FILE *infile;
if( !(infile = fopen(“f.txt”,”r”))){
        // Handle error
}
```

Unformatted (Binary) I/O

```
#include <stdio.h>

size_t fread(void *ptr,          // Read size*nmemb bytes
             size_t size,        // from stream
             size_t nmemb,
             FILE *stream );

size_t fwrite(const void *ptr, // Write size*nmemb bytes
              size_t size,      // to stream
              size_t nmemb,
              FILE *stream);
```

Buffered Output

- The Standard IO library provides buffering improvements to minimize the number of system call traps that need to be made to read() and write()
- For stdout the output is line-buffered if it's written to a terminal and block-buffered if it's written to to a disk file
  - If your program crashes, the buffer contents may be lost, especially if the file has been redirected to a disk file instead of a terminal.
  - That means you can lose your debugging output from printf if you don't end it with a newline and don't allow it to go directly to your terminal
- If you execute these printf statements just before a seg fault, you may very well not see any output(!)

```
printf( "Checkpoint 1.." );
```

…

```
printf( "Checkpoint 2.." );
```

Buffered Output

- You can force the output buffer to be flushed with fflush()
- If the program crashes, the buffer contents may be lost

```
printf( “Checkpoint 1” );
fflush(stdout);

…

printf( “Checkpoint 2” );
fflush(stdout);
```

- You can also set the buffering with a call to a function in the setvbuf() family

NOTE!

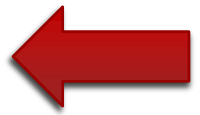
- For stderr the output is set to NOT buffered
- So output to stderr will always cause immediate I/O

```
fprintf(stderr, “Checkpoint 1”);
```

```
fprintf(stderr, “Checkpoint 2”);
```

Question

The main reason that output is buffered by default in Standard IO before sending the output to the operating system is

A. To minimize the use of operating system buffers
B. To reduce the number of I/O traps the operating system has to process 
C. To avoid using the complicated read() and write() system calls
D. To reduce the memory footprint of I/O operations

Functions may be Inlined

```
inline int myfunc(int a, int b) {
    return a+b;
}
```

Optimization

- You use the ***-O****level* option to control optimization
  - *0* – don't move code around (the default)
  - *1* or just ***-O*** – quick optimizations that don't take much compilation time
  - *2* – all optimizations that don't involve speed/space tradeoff
  - *3* – even more optimization
- The higher the optimization level, the more likely your code will not work properly if you don't follow the C rules precisely!
- For use with a debugger, use ***-O****0* or ***-O****1*.

Tricky

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"
-- Donald Knuth

- Don't write tricky code designed to be very efficient
  - You'll probably just defeat the optimizer by hiding what you're trying to accomplish
  - Until they've measured it, programmers ***rarely*** know where the CPU is spending its time in their code; this is called *premature optimization*
  - If you're really trying to be efficient in these modern times…
    - (1) measure, a.k.a. *profile*, your program to see where it spends its time
    - (2) look at the optimized assembly language output (**-S**) for your hot spots to see how the code is being generated
    - (3) Modify your C code to improve the optimized output
    - (4) Measure it again!

Tricky

- Speaking of tricky, what does this mean?

```
_ += ++_ + _--;
```

- Can you identify problems?
  - ++ and -- have side effects – it matters in what order they are executed
  - And the C standard doesn't specify an ordering for operations in an expression except in specific cases (&&, ||, ,)

What Does This Program Do?

```
#define P(a,b,c) a##b##c
#include/*++++++++/<curses.h>
int c,h, v,x,y,s, i,b; int
main () { initscr( ); P(cb,
rea, k)() ;/// x=COLS/2
P(n, oec, ho)( v=0);///
)/* */ ;for (curs_set(0); s=
; P( flu, shi, np())){ timeout(y=c=
P(c, lea, r)() ;for (P (
mva, d, dstr )(2, 3+x,
G) ; ; P( usl, eep, )(U)){//
P(m, vad, dstr )( y >>8,x,//
" "); for(i=LINES; /* */ i -->0
; mvinsch(i,0,0>(~c|i-h-H &h-i )?' '
:(i- h|h- i+H) <0?'|' :'=' ));
if(( i=( y +=v= getch( )>0?I:v+
A)>>8)>=LINES||mvinch(i*= 0<i, x)!=' '||' '
!=mvinch(i,3+x))break/*&& &*/; mvaddstr(y
>>8, x,0>v ?F:B ); i=--s
/-W; P(m, vpr, intw)(0,
COLS-9," %u/%u ",(0<i)* i,b=b<i?i:
b); refresh(); if(++ c==D){ c
-=W; h=rand()%(LINES-H-6
)+2; } } flash(); }}
```

An ASCII Version of Flappy Birds

- https://www.youtube.com/watch?v=ReWybwevcuY

International Obfuscated C Code Contest

- http://www.ioccc.org

Question

Profiling a program means

A. Looking to see where a program is fattest so it can be slimmed down in those regions

B. Instrumenting and measuring a running program to discover where the CPUs spend their time in order to find the most effective places to optimize the code

C. Optimizing a program by shaving off fuzzy portions of] its data structures

D. The inequitable practice of judging a program's correctness by relying heavily on its external appearance

Questions?