C Programming for CS 5600

This document gives a set of rules, suggestions, and recipes for C programming in this class. Rules and suggestions are just that – violating rules will result in points being taken off, while suggestions are meant to be helpful but may be ignored. Recipes are patterns for particular tasks you will run into in the homeworks, which you are free to use verbatim if you want to.

Rule 1 – All Pointers Must Be Initialized

When you declare a pointer variable, you must initialize it to a valid value, or NULL:

```
/* initialize to NULL – always OK:
 */
char *ptr = NULL;

/* initialize char* to a constant string:
 */
char *str = “a string”;

/* initialize to allocated memory (remember to free it).
 * note that sizeof(*ptr) is more reliable than sizeof(struct foo)
 */
struct foo foo_ptr = malloc(sizeof(*foo_ptr));

/* initialize to a pointer to a local buffer (works best with char):
 */
char buf[1024];
char *buf_ptr = buf;
```

If I see any declarations like this, I will **deduct points**:

```
struct foo *foo_ptr;
char *buf_ptr;
```

Both of these are accidents waiting to happen. The problem is that the stack locations allocated for `foo_ptr` and `buf_ptr` often contain valid pointers, and if you forget to assign values to these pointers, your program will happily use the pre-existing ones, and scribble over memory that it shouldn't. The end result is that it crashes long afterwards, or it only crashes when it runs on the grader's machine.

Rule 2 – Function names must match what they do

Typically what happens is that in question 1, a student writes a function like this:

```
void set_up_thingie(...) {
      allocate thingie
      initialize it
      …
}
```

and then later on they start adding more functions to it:

```
void set_up_thingie(...) {
      allocate it
      initialize it
      start it running
      do something else entirely
}
```

Then several questions down they forget that they tacked extra stuff on, and totally forget that it's doing something besides allocating and initializing something. (and it's quite confusing for the grader, as well)

You don't need perfectly named functions. However, if they're wildly misleading (as in this case) I will deduct points.

Rule 3 – Indent your code

Take some pride in your code, and format it so that it's readable and not too ugly. As with rule 2, I'm not asking for perfection; however, if you make your code sufficiently difficult to read I will deduct points. In other words, don't do this:

```
function()
{
line not indented;
another;
if() {
not indented;
}
   if (spurious indent) {
indented the wrong way;
}
   }
```

(and yes, I've seen submissions that are almost this bad.)

Suggestion 1 – Don't use unsigned integers

Unsigned integers are good if you really need all 32 or 64 bits, or if you're doing modulo arithmetic. However, most of the time that you think you want to use them, you don't, because they behave very oddly around zero – that's why they were deliberately omitted from Java.

As an example:

```
int i;
unsigned int len = 0;
for (i = 0; i <= len-1; i++)
      …
```

You would expect this code to loop '`len`' times, and it does if `len` is >0. However, in the case here (len=0) it loops forever, because any unsigned int is <= 0xFFFFFFFF, which is the unsigned value of (0-1).

Suggestion 2 – Use valgrind

There is a program called 'valgrind' which will check your code for memory errors. Unfortunately it doesn't work well for the second half of Homework 1, but other than that I highly suggest that you use it. To use it you just pass a command and arguments to the 'valgrind' command:

```
valgrind ./myprog argument1 argument2
```

If I compile and run this program:

```
int main(int argc, char **argv) {
      char buf[10];
      char *ptr = buf;
      printf("%s\n", ptr[20]);
      ptr[20] = 0;
      return 0;
}
```

it will happily run and print a value, '-1' in my case. However, if I run it under valgrind:

```
pjd@bubbles:/tmp$ valgrind ./a.out
==23662== Memcheck, a memory error detector
==23662== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==23662== Using Valgrind-3.5.0-Debian and LibVEX; ...
==23662== Command: ./a.out
==23662==
==23662== Use of uninitialised value of size 4
==23662==    at 0x4085186: _itoa_word (_itoa.c:195)
==23662==    by 0x4088AD1: vfprintf (vfprintf.c:1613)
==23662==    by 0x408FFFF: printf (printf.c:35)
==23662==    by 0x804840D: main (myprog.c:4)
==23662==
==23662== Conditional jump or move depends on uninitialised value(s)
==23662==    at 0x408518E: _itoa_word (_itoa.c:195)
==23662==    by 0x4088AD1: vfprintf (vfprintf.c:1613)
==23662==    by 0x408FFFF: printf (printf.c:35)
==23662==    by 0x804840D: main (myprog.c:4)
==23662==
```

and lots of other warnings about what we're doing wrong.

Suggestion 3 – Use the man pages

The 'man' (manual) command in Unix/Linux is a source of a large amount of detailed, disconnected, and hard-to-find information. It is divided into “chapters”, where chapter 1 is for normal commands (e.g. ls), chapter 2 is for system calls (read, fork, etc.) and 3 is for library functions like fprintf.

It searches chapter 1 first, so if you just type 'man printf' you will get documentation on an obscure shell command for printing formatted output, instead of the function you wanted to find out about. (And 'man read' gives you the shell manpage, because the shell has a 'read' command.) In that case you have to specify the chapter: 'man 3 printf' and 'man 2 read' give you the documentation you want.

This is why you will sometimes see references to things like printf(3) and _exit(2) – the number is an obscure indicator as to whether it's a library function or a system call.

Recipe 1 – Simple way to read a file into memory

This function takes a filename, a pointer to a buffer, and the length of the buffer. (note that ptr/len pairs form a common C pattern) If successful it returns the number of bytes read; in the case of an error it prints a message and returns 0.

```
int readfile(char *file, char *buf, int buflen)
{
    FILE *fp = fopen(file, "r");
    if (fp == NULL) {
        perror("can't open file");
        return 0;
    }

    int i, c;
    for (i = 0; i < buflen; i++) {
        c = getc(fp);     /* 'c' *has* to be an 'int', not a 'char' */
        if (c == EOF)
            break;
        buf[i] = c;
    }
    fclose(fp);
    return i;
}
```

Note that it's very important that 'c' be an 'int', not a 'char', since it has to represent all 256 legal byte values plus the EOF value.

Recipe 2 – Reading lines of input

This recipe works quite well for reading lines from standard input or from a file:

```
FILE *fp = stdin;
if (... input file is 'file' ...) {
    fp = fopen(file, "r");
    if (fp == NULL) {
        perror("can't open file");
        exit(1);
}
char line[128];
while (fgets(line, sizeof(line), fp) {
    ... do something with 'line' ...
}
```

Recipe 3 – Splitting a line into words

This is complicated in C, because (a) you have to do memory management yourself, and (b) the standard library functions for doing it are terrible.

The non-standard way uses the following custom function:

```
/* find the next word starting at 's', delimited by characters
 * in the string 'delim', and store up to 'len' bytes into *buf
 * returns pointer to immediately after the word, or NULL if done.
 */
char *strwrd(char *s, char *buf, size_t len, char *delim)
{
    s += strspn(s, delim);
    int n = strcspn(s, delim);  /* count the span (spn) of bytes in */
    if (len-1 < n)              /* the complement (c) of *delim */
        n = len-1;
    memcpy(buf, s, n);
    buf[n] = 0;
    s += n;
    return (*s == 0) ? NULL : s;
}
```

like this:

```
char line[some_length];
char argv[10][20];
int argc;
for (argc = 0; argc < 10; argc++) {
    line = strwrd(line, argv[argc], sizeof(argv[argc]), " \t");
    if (line == NULL)
        break;
}
```

Now the words in the line are stored into argv[0], … argv[argc-1].

For the standard way to do this, look at the library (3) man pages for 'strtok' and 'strsep'.