601.220 Intermediate Programming

Summer 2022, Meeting 4 (June 13th)

Today's agenda

- Exercises 5 and 6 review
- "Day 7" material
  - Function declarations, passing arrays to functions, recursion
  - Exercise 7
- "Day 8" material
  - Separate compilation, Makefiles, header guards
  - Exercise 8

Reminders

- HW1 due Wednesday, June 15th
- HW3: will be posted very soon

Exercise 5 review

- Copying `.bashrc` and `.bash_profile` from the public repo
- These are "shell startup scripts"
  - Make `emacs` the default editor
  - Add `gccc` and `g+++` aliases for running `gcc` and `g++` with the recommended compiler options

Exercise 5 review

`count1.c`: iterate backwards over original sequence, build complement sequence:

```
for (int i = dna_len - 1; i >= 0; i--) {
  char complement = '?';
  switch (dna[i]) {
  case 'A': complement = 'T'; break;
  case 'T': complement = 'A'; break;
  case 'C': complement = 'G'; break;
  case 'G': complement = 'C'; break;
  default: /* bad data */; break;
  }
  rev_comp[rci] = complement;
  rci++;
}
```

Exercise 5 review

Set NUL terminator at end of complement sequence:

```
rev_comp[rci] = 0;
```

Exercise 5 review

`count2.c`: classify characters using `<ctype.h>` functions:

```
for (int i = 0; i < text_len; i++) {
  char c = text[i];
  if (isalpha(c)) { num_alpha++; }
  if (isdigit(c)) { num_digits++; }
  if (isspace(c)) { num_space++; }
}
```

Exercise 5 review

Could also use knowledge of how characters are encoded in ASCII:

```
for (int i = 0; i < text_len; i++) {
  char c = text[i];
  if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
    { num_alpha++; }
  if (c >= '0' && c <= '9')
    { num_digits++; }
  // ...etc...
}
```

Using the <ctype.h> functions is simpler, and makes the program more readable.

Exercise 5 review

`count3.c`: initializing array of per-character counts:

```
int ascii_count[256] = {0};
```

Note that when an array initializer has fewer initial values than the array has elements, the remaining elements are initialized to zero.

So, all of the elements of `ascii_count` are set to 0 initially.

Exercise 5 review

Tabulating occurrence counts for each character value:

```
for (int i = 0; i < text_len; i++) {
  int c = text[i];
  assert(c >= 0);
  ascii_count[c]++;
}
```

Note `char` values can be negative, but using a negative array index would result in an invalid access. Hence, the use of `assert`. It is a precondition that all of the character values must be non-negative.

Also note that `gcc` will complain if you try to use a `char` value as an array index.

Exercise 5 review

Finding most frequent and second most frequent characters:

```
for (int i = 0; i < 256; i++) {
  if (ascii_count[i] > top_freq) {
    top_char = (char) i;
    top_freq = ascii_count[i];
  } else if (ascii_count[i] > next_freq) {
    next_char = (char) i;
    next_freq = ascii_count[i];
  }
}
```

Exercise 6 review

Opening input and output files:

```
FILE *in = fopen(filename, "r");
if (in == NULL) {
  fprintf(stderr, "Could not open '%s' for reading\n", filename);
  return 1;
}

FILE *out = fopen("output.txt", "w");
if (out == NULL) {
  fprintf(stderr, "Could not open 'output.txt' for writing\n");
  return 1;
}
```

Exercise 6 review

Reading principal and APR from input file:

```
parse = fscanf(in, "%f %f", &p, &r);
```

This needs to go before main loop starts, and also at end of body of main loop.

Exercise 6 review

Computing accumulated principal (in `compound_interest` function):

```
const float t = 1.0;
if (n > 0) {
  return p * pow(1.0 + r/n, n*t);
} else {
  return p * exp(r*t);
}
```

Exercise 6 review

Checking for errors after main loop terminates:

```
if (parse != EOF) {
  fprintf(stderr, "Error reading input\n");
  return 1;
}
if (ferror(in)) {
  fprintf(stderr, "Input error indicator was set\n");
  return 1;
}
if (ferror(out)) {
  fprintf(stderr, "Output error indicator was set\n");
  return 1;
}
```

Exercise 6 review

Closing input and output files:

```
fclose(in);
fclose(out);
```

Day 7 recap questions

1. How do you get the number of elements of an integer array?
2. Is the size of a string array the same as the string length?
3. What is the difference between a function declaration and a function definition?
4. Can you have two functions with the same function name in a program?
5. How does passing an integer array to a function differ from passing a single integer variable into a function?
6. How can you make an array that is passed into a function not modifiable?
7. What is the down-side to recursion?

1. How do you get the number of elements of an integer array?

```
int arr[10];

// ...

size_t num_elts = sizeof(arr) / sizeof(int);
printf("%lu\n", num_elts); // prints 10
```

Note that this will **not** work if `arr` is a function parameter. (Array parameters are not actually arrays. We'll see what they are soon.)

2. Is the size of a string array the same as the string length?

No.

If an array of `char` elements will be used to store a string value, its number of elements must be *at least* the string length plus 1. (Enough room to store the characters in the string, and the NUL terminator.)

It is totally fine for a `char` array to have more room than necessary. In this case, the elements after the NUL terminator aren't used.

3. What is the difference between a function declaration and a function definition?

Function declaration: just tells the compiler the important details about the function: its name, return type, and parameter types. It will use this information to check *calls* to the function. A.k.a. a "function prototype".

Function definition: defines the body (code) of the function.

Each use of a function in a program should be preceded by either a declaration or definition.

4. Can you have two functions with the same function name in a program?

No, not in C.

(C++ does allow this, and calls this possibility "function overloading.")

5. How does passing an integer array to a function differ from passing a single integer variable into a function?

Yes: functions are passed by *reference*, not by value. This means that the called function can change the values in the array.

Example

```c
// pbr.c:
#include <stdio.h>
void f(int a[]) {
  a[0] = 42;
}

int main(void) {
  int nums[3] = {1, 2, 3};
  printf("Before: nums[0]=%d\n", nums[0]);
  f(nums);
  printf("After: nums[0]=%d\n", nums[0]);
  return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic pbr.c
$ ./a.out
Before: nums[0]=1
After: nums[0]=42
```

6. How can you make an array that is passed into a function not modifiable?

Declare the element type as being `const`. Example:

```
// constelem.c:
#include <stdio.h>
void f(const int a[]) {
  a[0] = 42;
}

int main(void) {
  int nums[3] = {1, 2, 3};
  f(nums);
  return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic constelem.c
constelem.c: In function ‘f’:
constelem.c:3:8: error: assignment of read-only location ‘*a’
    3 |   a[0] = 42;
      |        ^
```

7. What is the down-side to recursion?

Each function call in C requires the creation of a run-time data structure called a *stack frame* to store parameter values, allocate storage for local variables, and keep track of other information about the function call.

The amount of memory available for stack frames is limited.

A "deep" recursion could create a large number of stack frames, which could require more memory than is available. This results in a "stack overflow" which will crash the program.

So, avoid deep recursion.

Recursion tips

- The first thing a recursive function must do is to see whether a *base case* has been reached
- If a base case hasn't been reached, find a smaller instance of the problem, solve it recursively, then extend the solution to the smaller problem so that the entire problem is solved
  - Example:

```
int sum_ints(int n) { // compute sum of integers 1..n
  if (n == 1) { return 1; }
  return sum_ints(n-1) + n;
}
```

- The subproblem solved recursively should be as *large* as possible
  - So that very little work is required to extend the solution to be a solution to the overall problem

Exercise 7

- Functions and recursion!

Day 8 recap questions

1. Why do we need header guards?
2. What is the difference between compiling and linking?
3. What compiler flag do we use to create object files and what extension do those files have?
4. What is a `target` in a Makefile?
5. What are the advantages of using Makefiles?

1. Why do we need header guards?

The `#include` directive means (essentially) "copy the contents of the named header file into the source code module that is being compiled."

Header guards prevent the contents of a header file from being copied more than once.

Some constructs which often are placed in a header file could cause an error if they appear more than once. In particular, `struct` data types can't be redefined.

2. What is the difference between compiling and linking?

Compiling: converts C source code (`.c` source file) into "object code" (`.o` "object file").

Linking: combines one or more `.o` (object) files into an executable file. Also, resolves calls to library functions (such as `printf`, `scanf`, `pow`, etc.)

3. What compiler flag do we use to create object files and what extension do those files have?

The `-c` option means "compile source code to an object file".

Object files have a `.o` file extension.

4. What is a `target` in a Makefile?

A Makefile target is a file to be created based on the contents of other files.

For a C program, typically the Makefile will have targets for each object (`.o`) file, as well as a target for the executable.

The first target in a Makefile is the *default* target. If `make` is invoked without specifying an explicit target to build, the default target is built.

5. What are the advantages of using Makefiles?

Makefiles *automate* the process of compiling and linking a program. Just run the command `make`, and (if the Makefie is written correctly) the entire program will be compiled and linked.

If each target properly lists its *dependencies*, then `make` will figure out exactly which compiler commands need to be run, and in what order. In general, if any targets are "out of date", meaning that one or more dependencies is newer, or has been modified since the target was built, `make` will know to rebuild the target.

TL;DR when you have a properly-written Makefile all you need to do to build the program is run the command `make`.

Makefile tips

Definitions (at top of the `Makefile`):

```
CC = gcc
CFLAGS = -std=c99 -Wall -Wextra -pedantic
```

Target for an executable:

```
fooprog : foo.o bar.o
	$(CC) -o fooprog foo.o bar.o
```

Target for an object file:

```
foo.o : foo.c header1.h header2.h
	$(CC) $(CFLAGS) -c foo.c
```

Makefile syntax

Note that the *commands* for a target *must* begin with a tab character.

`emacs` and `vim` will show you if any comands aren't indented with a tab character.

Exercise 8

- Implement the `concat` string-concatenation function
- Split `all_in_one.c` into separate source files as follows:
  - `run_concat.c` should contain the `main` function
  - `string_functions.h` should have a declaration of `concat`
  - `string_functions.c` should have the definition of `concat`
- Modify the `Makefile`:
  - it should have targets for `run_concat` (the executable), `run_concat.o`, and `string_functions.o`

Notes

Notes

Notes

Notes

Notes