601.220 Intermediate Programming

Summer 2022, Meeting 3 (June 10th)

Today's agenda

- Exercises 3-A, 3-B, 4 review
- "Day 5" material
  - Arrays and strings
  - Exercise 5
- "Day 6" material
  - File I/O, functions, command line arguments
  - Exercise 6

Reminders

- HW1 due Wednesday, June 15th

Exercise 3-A review

Personal repository setup:

```
git clone https://github.com/jhu-ip/2022-summer-student-JHEDID.git
cd 2022-summer-student-JHEDID
emacs README
git add README
git commit -m'added README to my personal repository'
git push
```

`git` commands that need to access the remote repository on Github (such as `git clone`, `git push`, `git pull`) will require your Github username and Personal Access Token to authenticate.

(Alternatively, you could create an ssh key and register it to your Github account.)

Exercise 3-A review

Preparing a git log and zipfile (exercise and homework submission procedure):

```
git log > gitlog.txt
zip -9r ex03.zip README gitlog.txt
cd
mv 2022-summer-student-JHEDID/ex03.zip .
```

On your local computer (enter your ugrad account password when prompted):

```
scp USERNAME@ugradx.cs.jhu.edu:ex03.zip .
```

The zipfile `ex03.zip` will be on your local machine in whatever directory your command prompt was in (typically your local home directory).

At this point you could upload the zipfile to Gradescope.

Exercise 3-B review

Create a clone of the course "public" repository:

```
git clone https://github.com/jhu-ip/cs220-summer22-students.git
```

Create a temporary directory and copy exercise starter files:

```
cd
mkdir temp
cd temp
cp ~/cs220-summer22-students/exercises/ex02/* .
```

Compile and run the program:

```
gcc -std=c99 -Wall -Wextra -pedantic hello_world.c
./a.out
```

Exercise 3-B review

Make a copy of the program, modify it, compile and run the modified version:

```
cp hello_world.c hello_me.c
emacs hello_me.c
gcc -std=c99 -Wall -Wextra -pedantic hello_me.c
./a.out
```

Exercise 4 review

Renaming your personal repository (to make it easier to access for future exercises, homework, etc.):

```
mv 2022-summer-student-JHEDID my220repo
```

Now you can refer to your personal repository as `~/my220repo`.

Exercise 4 review

Starting an exercise (this general procedure will work for future exercises):

```
cd ~/my220repo
mkdir -p exercises/ex04
cd exercises/ex04
cp ~/cs220-summer22-public/exercises/ex04/* .
```

Exercise 4 review

Writing a loop to read grade followed by number of credits:

```
char grade;
float num_credits;

while (scanf(" %c %f", &grade, &num_credits) == 2) {
  // tally grade and number of credits
  // ...
}
```

Note that

- The space before %c makes `scanf` skip whitespace characters before reading the grade character
  - Otherwise it could read a whitespace character instead of the letter grade
- If the user enters Control-D, that ends the input

Exercise 4 review

Using a switch statement to match the grade:

```
switch (grade) {
case 'A': case 'a':
  quality_points += (num_credits * 4.0);
  break;

case 'B': case 'b':
  quality_points += (num_credits * 3.0);
  break;

// ...etc...

default:
  // invalid grade
}
```

Day 5 review question

1. When we declare an array in C, what are the initial values?
2. What is the ASCII (Unicode) table?
3. What is a null terminator? What is its ASCII value?
4. Consider c-string `"ab\0cd\0"` - what is the reported string length?
5. How do we check if two C-strings are the same? In addition, are these two strings the same: `"ab\0cd\0"` and `"ab\0"`?

1. When we declare an array in C, what are the initial values?

Elements of an array are uninitialized by default. For example:

```c
int a[3];
printf("%d\n", a[0]); // undefined behavior
```

2. What is the ASCII (Unicode) table?

Text characters are represented as integer "character codes".

ASCII codes range from 0 to 127. Examples:

- "`!`" has the code 33
- "`0`" has the code 48
- "`A`" has the code 65
- "a" has the code 97

In C, a *character literal* (in single quotes) yields the ASCII code for that chracter. E.g., `'A'` is the integer value 65.

Unicode: encoding scheme for (essentially) all characters in all human languages, plus symbols, emojis, etc.

3. What is a null terminator? What is its ASCII value?

In C, a character string is

- stored in an array of `char` elements, and
- is terminated by a "NUL" character, which is the character whose integer character code is 0

The NUL character can be written as `'\0'` or just `0`.

E.g.:

```
char s[4] = "foo";
assert(s[0] == 'f');
assert(s[1] == 'o');
assert(s[2] == 'o');
assert(s[3] == 0);
```

4. Consider c-string "ab\0cd\0" - what is the reported string length?

Note that \0 in a string literal means a literal NUL character.

The `strlen` function determines the length of a string, which is the number of characters preceding the NUL terminator marking the end of the string.

So:

```
assert(strlen("ab\0cd\0") == 2);
```

5. How do we check if two C-strings are the same? In addition, are these two strings the same: `"ab\0cd\0"` and `"ab\0"`?

The `strcmp` function returns 0 if the two strings passed as arguments consist of the same sequence of characters.

So:

```
assert(strcmp("ab", "ab\0cd") == 0);
assert(strcmp("ab", "abc") != 0);
```

Exercise 5

- Enhanced `.bashrc` and `.bash_profile` startup scripts
- Working with character arrays and strings

Breakout rooms 1–10 are "social", breakout rooms 11+ are for individual or small group work.

Day 6 recap questions

1. Is `fprintf(stdout, "xxx")` the same as `printf("xxx")`?
2. When should we use assertions instead of an *if* statement?
3. What will happen if you pass an `int` variable to a function that takes a `double` as its parameter? What will happen if a `double` is passed to an `int` parameter?
4. What is "pass by value"?
5. How do you change the *main* function so that it can accept command-line arguments?

1. Is `fprintf(stdout, "xxx")` the same as `printf("xxx")`?

    Yes.

2. When should we use assertions instead of an *if* statement?

An assertion (use of the `assert` macro) means "this condition must be true, or else we have proved that there is a bug in the program."

Assertions are useful for checking *invariants*. They are also useful for *unit testing*. A unit test is an automated test for a small "unit" of the program, typically a single function. Assertions are used in a unit test to verify that the code being tested behaved correctly.

Assertions should *never* be used to check for conditions that could be true in the normal operation of the program. For example, it would be incorrect to use an assertion to check whether a file was opened successfully.

3. What will happen if you pass an `int` variable to a function that takes a `double` as its parameter? What will happen if a `double` is passed to an `int` parameter?

An `int` value can be freely converted to a `double` with no loss of information. The `double` value will be numerically the same as the original `int` value.

If a `double` is converted to an `int`, it is *truncated*, i.e., the fractional part is discarded.

Conversions

```c
// conversions.c:
#include <stdio.h>
#include <assert.h>

int as_int(int x) { return x; }
double as_double(double x) { return x; }

int main(void) {
  assert(as_double(3) == 3.0);
  assert(as_int(6.79) == 6);
  printf("tests passed\n");
  return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic conversions.c
$ ./a.out
tests passed
```

4. What is “pass by value”?

“Pass by value” means that a parameter is a variable that is distinct from any other variable in the program. Changing the value of a parameter does not change the value of any other variable in the program.

C uses pass-by-value for all parameters except for array parameters.

Pass by value example

```c
// pbv.c:
#include <stdio.h>

void f(int x) {
  x = 42;
  printf("x=%d\n", x);
}

int main(void) {
  int y = 17;
  f(y);
  printf("y=%d\n", y);
  return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic pbv.c
$ ./a.out
x=42
y=17
```

5. How do you change the *main* function so that it can accept command-line arguments?

Declare it like this:

```
int main(int argc, char *argv[]) {
  // ...
}
```

`argc` is one greater than the number of command line arguments.

`argv` is an array of strings, where the strings are the command line arguments. (Note that `argv[0]` is always the name of the program.)

Command line arguments example

```c
// cmdargs.c:
#include <stdio.h>

int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; i++) {
    printf("argv[%d] is '%s'\n", i, argv[i]);
  }
  return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic cmdargs.c
$ ./a.out C is a "fun language"
argv[0] is './a.out'
argv[1] is 'C'
argv[2] is 'is'
argv[3] is 'a'
argv[4] is 'fun language'
```

A quick synopsis of C file I/O

- The `FILE*` type represents an open file (for reading or writing)
- Opening a file for reading:
  `FILE *in; in = fopen(`*filename*`, "r");`
- Opening a file for writing:
  `FILE *out; out = fopen(`*filename*`, "w");`
- If `fopen` returns `NULL`, it means the file wasn't opened successfully
- Use `fscanf` to read from a file, `fprintf` to write to a file
- When the program is done with a file, use `fclose` to close it

Exercise 6

Compound interest ($\leftarrow$ click for formulas to calculate)

Breakout rooms 1–10 are "social", breakout rooms 11+ are for individual or small group work.

Notes

Notes

Notes

Notes

Notes