Computer Science 35L: Week 6

Low-level Programming

Rats and Ratatouille

This worksheet will guide you through writing a simple C program. It should help you with the following aspects of Assignment 5:

1. Writing C code.
2. Using the `write()` system call.
3. Writing Makefiles.
4. Refactoring a single C file into multiple C and header files.

We included some hints in these footnotes 🤫. But try the problems yourself before you look down there!

Writing C Code

1. Copy and paste the following code into a file named `ratatouille.c`.

```c
#include <stdio.h>

void ratatouille() {
    printf("The rat is named REMY not RATATOUILLE\n");
}

int main() {
    ratatouille();
    return 0;
}
```

2. Compile and run this program.
    a. What does the `ratatouille()` function do? How does it's implementation differ from something you'd write in C++?

    It prints the string `"The rat is named REMY not RATATOUILLE\n"` to standard output. Interestingly, the above code is 100% compatible with C++! However, you might be more used to using `cout` instead of `printf()` to print text in C++.

3. Write a function called `ratatouille_syscall()` that does the same thing as `ratatouille()`, but uses the `write()` system call instead of `printf()`.

    Solution:

```c
#include <unistd.h>
#include <string.h>

void ratatouille_syscall() {
  char *truth = "The rat is named REMY not RATATOUILLE\n";
  write(STDOUT_FILENO, truth, strlen(truth));

  // or just:
  //  write(1, "The rat is named REMY not RATATOUILLE\n", 38);
}
```

4. Write a function called `ratatouille_N()` that takes an integer argument `N`. It should do the same thing as `ratatouille_syscall()`, but write the string in chunks of `N` bytes.

Solution:

```c
#include <unistd.h>
#include <string.h>

void ratatouille_N(int N) {
  char *truth = "The rat is named REMY not RATATOUILLE\n";
  int length = strlen(truth);
  for (int i = 0; i < length; i += N) {
    if (i + N >= length) {
        // The last chunk might be smaller than N.
        write(STDOUT_FILENO, &truth[i], length - i);
    } else {
        write(STDOUT_FILENO, &truth[i], N);
    }
  }
}
```

5. *Bonus:* It is possible for the `write()` system call to fail (see this section of the man page). Try to write code that checks for and reports these errors.

Solution for `ratatouille_syscall()`:

```c
#include <errno.h>   // errno
#include <stdlib.h>  // exit()
#include <stdio.h>   // fprintf()
#include <string.h>  // strlen() and strerror()
#include <unistd.h>  // write()

void ratatouille_syscall() {
  char *truth = "The rat is named REMY not RATATOUILLE\n";
  int ret = write(STDOUT_FILENO, truth, strlen(truth));
  if (ret == -1) {
    fprintf(stderr, "write error: %s", strerror(errno));
    exit(1); // Exit with status 1
  }
}
```

Makefile Magic

1. Copy and paste the following code into a file named `Makefile` (the filename here is important):

```
CC = gcc
CFLAGS = -Wall -Wextra

ratatouille:
    echo "Implement me!"
```

2. **A Makefile rule has three parts (see this link): a *target*, a list of *prerequisites*, and a *recipe*. Consider the rule in the above Makefile.**
    a. What is its target?

    The target is `ratatouille`.

    b. What are its prerequisites?

    There are no prerequisites.

    c. What is its recipe?

    It's recipe is `echo "Implement me!"`.

    d. How can you run this rule, and what happens when you run it?

    You can run this rule with the command `make ratatouille`, which will print `"Implement me!"` to standard output.

3. Using the `CC` and `CFLAGS` variables, fill in the `ratatouille` rule's recipe. When you run the command `make ratatouille`, it should output an executable called `ratatouille`.

    Solution:

```
# $@ is replaced with the target name.
# You can also just type “ratatouille” out.
ratatouille:
    $(CC) $(CFLAGS) -o $@ $@.c
```

4. Run `make ratatouille` to test your recipe.

a. Note that we currently have a small problem. If you make a change to `ratatouille.c`, running `make ratatouille` again will not actually recompile our program (Make will think that the `ratatouille` binary is "up to date").

5. Fill in the `ratatouille` rule's prerequisite list.

   Solution:

   ```
   ratatouille: ratatouille.c
       $(CC) $(CFLAGS) -o $@ $@.c
   ```

   a. How does this solve the problem in #4?

      Here, we are telling Make that `ratatouille.c` is a prerequisite of `ratatouille`. This means that when `ratatouille.c` is changed, Make will know to rebuild `ratatouille`.

Modularizing our Program

1. Move all three `ratatouille_*()` functions into a file `chef.c`.

2. Write the corresponding header file `chef.h`. Include this header in your `ratatouille.c` file, and make sure your program compiles.

   Solution:

   ```
   void ratatouille();
   void ratatouille_syscall();
   void ratatouille_N(int N);
   ```

3. Update your `Makefile` to reflect these changes.

   Solution:

   ```
   src-files = ratatouille.c chef.c
   header-files = chef.h
   ratatouille: $(src-files) $(header-files)
       $(CC) $(CFLAGS) -o $@ $(src-files)
   ```