I/O & Testing

Readings: CP:AMA 2.5

I/O streams

Input & Output (***I/O*** for short) is the term used to describe how programs *interact* with the "real world".

A program may interact with a human by receiving data from an input device (like a keyboard, mouse or touch screen) and sending data to an output device (like a screen or printer).

A program can also interact with non-human entities, such as a file on a hard drive or even a different computer.

A popular programming abstraction is to represent I/O data as a ***stream*** of data that moves (or "flows") from a **source** to a **destination**.

A program can be both a destination (receives input) and a source (produces output).

As mentioned previously, the source/destination of a stream could be a device, a file, another program or another computer. The stream programming *interface* is the same, regardless of what the source/destination is.

Some programs connect to specific streams, but many programs use the *"standard"* input & output streams known as `stdin` & `stdout`.

For example, `printf` outputs to the `stdout` stream.

The default source for `stdin` is the keyboard, and the default destination for `stdout` is the "output window".

The user (or the operating system) can change (or "redirect") the standard streams to come from any source or go to any destination.

For example, a program may always `printf` a message to `stdout`, but the user may have redirected the `stdout` to a file instead of the output window.

In this course, our programs use the standard streams, but (as we will see later) RunC may use redirection on our behalf to use input & output files.

You can often “chain” programs together so the output stream of program A is the input stream of program B, and so on.

This is a very “functional programming” approach, and is similar to the way data can flow from function to function.

For those familiar with the command line:

```
$ A < input.txt | B | C > output.txt
```

The “flow” of the above commands would be:

`input.txt` $\rightarrow$ `A` $\rightarrow$ `B` $\rightarrow$ `C` $\rightarrow$ `output.txt`

A stream is typically characterized as either a ***text stream*** or a ***binary stream***, determined by the type of the data it contains.

In a *text stream*, the data is human-readable and composed of printable ASCII characters (and a few specialized ASCII control characters such as `\n`).

In a *binary stream*, the data is machine-readable and is similar to how data is stored in memory.

For example, the number `1000000000` in a text stream would be 10 characters long, but in a binary stream it would be the 4-byte representation of the number in binary.

Similarly, there are text files (such as `hello.txt` or `program.c`) and binary files (such as `song.mp3`).

Output

We have already seen the `printf` function (in both Racket and C) that prints formatted output (via placeholders) to the `stdout` stream.

In C, we have seen the placeholders `%d`(ecimal integer), `%c`(haracter), `%f`(loat) and `%p`(ointer / address).

In Racket, we have seen `~a`(ny). The `~v`(alue) placeholder is useful when debugging as it shows extra type information (such as the quote for a `'symbol`).

`printf` is only useful for outputting to *text* streams.

In this course, we only use text streams, and so `printf` is the only output function we need.

Writing to **text file** streams directly is almost as straightforward as using `printf`. The `fprintf` function (`file printf`) has an additional parameter that is a file pointer (`FILE *`). The `fopen` function opens (creates) a file and return a pointer to that file.

```
#include <stdio.h>

int main(void) {
  FILE *file_ptr;
  file_ptr = fopen("hello.txt","w");   // w for write
  fprintf(file_ptr, "Hello World!\n");
  fclose(file_ptr);
}
```

See CP:AMA 22.2 for more details.

Debugging output

Output can be very useful to help ***debug*** our programs.

We can use `printf` to output intermediate results and ensure that the program is behaving as expected. This is known as ***tracing*** a program. *Tracing* is especially useful when there is mutation.

A global variable can be used to turn tracing on or off.

```
if (TRACE) printf("The value of i is: %d\n",i);
```

> In practice, tracing is commonly implemented with *macros* that can be turned on & off (CP:AMA 14).

You can even use different **tracing levels** to indicate how much detail you want in your tracing output. Once you have debugged your code, you can simply set the level to zero and turn off all tracing.

```
int TRACELEVEL = 2;

int main(void) {
  if (TRACELEVEL >= 1) printf("starting main\n");
  int sum = 0;
  if (TRACELEVEL >= 2) printf("before loop: sum = %d\n",sum);
  for (int i=0; i < 10; i++) {
    if (TRACELEVEL >= 3) printf("loop iteration: i = %d\n",i);
    sum += i;
    if (TRACELEVEL >= 3) printf("sum has changed: sum = %d\n",sum);
  }
  if (TRACELEVEL >= 2) printf("after loop: sum = %d\n",sum);
  if (TRACELEVEL >= 1) printf("leaving main\n");
}
```

Input

The Racket `read` function attempts to read (or "get") a value from the `stdin` stream (by default, the keyboard). If there is no value available, `read` **pauses** the program and waits until there is.

```
(define my-value (read))
```

`read` may produce a special value (`#<eof>`) to indicate that the **E**nd **O**f **F**ile (EOF) has been reached. In our `RunC` environment, a `Ctrl-D` ("Control D") keyboard sequence sends an `EOF`.

> A better term would be **E**nd **O**f **S**tream (EOS), but in practice streams are often files.

The `read` function is quite complicated, so we present a *simplified* overview that is sufficient for our needs:

`read` interprets the input as if a single quote `'` has been inserted before each “value” (again, not really but close enough).

If your value begins with an open parenthesis `(`, Racket reads until a corresponding closing parenthesis `)` is reached, interpreting the input as one value (a list).

Text is interpreted as symbols, not a string (unless it starts with a double-quote `"`). The `symbol->string` function is often quite handy when working with `read`.

Example: read

```
(define (read-to-eof)
  (define r (read))
  (printf "~v\n" r)
  (cond [(not (eof-object? r)) (read-to-eof)]))
```

```
1
two
"three"
(1 two "three")
Ctrl-D

1
'two
"three"
'(1 two "three")
#<eof>
```

scanf

In C, the `scanf` function is the counterpart to the `printf` function.

While you can read more than one value in a single call to `scanf`, do **not** do this and **only read one value at a time**.

The return value of `scanf` is the number (count) of values read in. `scanf` requires a **pointer** to a variable to **store the input value**.

```
scanf("%d",&i); //read in an integer, store in i
```

A `scanf` return value of 1 is “success” (one value was read in).

The return value can also be the special constant value `EOF`.

While `scanf` can be used to read in characters (`"%c"`), it is more common (and more efficient) to use the function `getchar`, which returns either a character (byte) or `EOF`.

```
c = getchar();
```

When reading in ASCII text, it is okay to use a `char` to store the result of `getchar`.

However, the return value of `getchar` is actually an `int`, because when reading in *binary* input, There are 257 possible values: any of the 256 character values plus the special `EOF` sentinal value.

User interaction

With the combination of input & output, we can make ***interactive*** programs that change their behaviour based on the input.

```
(define (get-name)
  (printf "Please enter your first name:\n")
  (define name (read))
  (printf "Welcome, to our program, ~a!\n" name)
  name)
```

Example: interactive Racket

```
(define (madlib)
  (printf "Let's play Mad Libs! Enter 4 words :\n")
  (printf "a Verb, Adverb, Noun & Adjective :\n")
  (define verb (read))
  (define adverb (read))
  (define noun (read))
  (define adj (read))
  (printf "The two ~as were too ~a to ~a ~a.\n"
          noun adj verb adverb))

(madlib)
```

Example: interactive C

```
int main(void) {
  int count = 0;
  int i = 0;
  int sum = 0;

  printf("how many numbers should I sum? ");
  scanf("%d",&count);
  for (int j=0; j < count; j++) {
    printf("enter #%d: ", j+1);
    scanf("%d", &i);
    sum += i;
  }
  printf("the sum of the %d numbers is: %d\n", count, sum);
}
```

```c
int main(void) {
  int count = 0;
  int i = 0;
  int sum = 0;

  printf("keep entering numbers, press Ctrl-D when done.\n");
  do {
    printf("enter #%d: ", count+1);
    if (scanf("%d", &i) == 1) {
      sum += i;
      count++;
    } else {
      break;
    }
  } while (1);

  printf("\n");
  printf("the sum of the %d numbers is: %d\n", count, sum);
}
```

Interactive testing

In DrRacket, the *interactions window* was quite a useful tool for debugging our programs.

In RunC, we can create **interactive testing modules**.

Consider an example with a simple arithmetic module:

```
// addsqr.h

// sqr(x) returns x*x
int sqr(int x);

// add(x,y) returns x+y
int add(int x, int y);
```

```c
#include "addsqr.h"

int main(void) {
  char func;
  int x,y;
  do {
    func = getchar();
    if (func == 'a') {
      scanf("%d", &x);
      scanf("%d", &y);
      printf("add %d %d = %d\n", x, y, add(x,y));
    } else if (func == 's') {
      scanf("%d", &x);
      printf("sqr %d = %d\n", x, sqr(x));
    }
  } while (func != 'x');
}
```

With this *interactive* testing module, tests are entered via the keyboard:

Input:

```
a 3 4
a -1 0
a 999 -1000
s 5
s -5
s 0
x
```

Output:

```
add 3 4 = 7
add -1 0 = -1
add 999 -1000 = -1
sqr 5 = 25
sqr -5 = 25
sqr 0 = 0
```

One big advantages of this interactive testing approach is that we can experiment with our module without having to program (code) each possible test.

It's also possible that someone could test the code without even knowing how to program.

A disadvantage of this approach is that it can become become quite tedious to rely on human input at the keyboard.

Fortunately, the RunC environment has support to *automate* interactive testing.

RunC testing

In RunC, you can provide `.in` files that are used as inputs to your program (using *redirection*), instead of the keyboard.

If your "`main`" program is `filename.c` or `filename.rkt`, RunC looks for a corresponding `filename.in.1` file.

For each `.in` file (`.1`, `.2`, etc.) you provide, RunC **runs** your program and generates a corresponding `.out` file.

In the previous example, with a testing module `"test-addsqr.c"`, if an input file `"test-addsqr.in.1"` is placed in the same directory, a `"test-addsqr.out.1"` file is generated.

RunC also validates your input. If you provide an `.expect` file in addition to an `.in` file, RunC compares your `.expect` file against the `.out` file. If the `.out` and `.expect` files are different, it generates a `.check` file that shows any discrepancies.

This approach allows you to automatically test your code without human interaction and without the inconvenience of coding each individual test case.

Additional tips for testing within `RunC`:

- If you prefer to use a non-interactive testing strategy (all of your tests are coded directly), you can have an empty `.in` file, and you can still use a `.expect` file to test any output
- for complicated or lengthy tests, you can write a separate program that generates your test inputs

Testing in C

Here are some additional tips to look for when testing in C:

- check for “off by one” errors in loops
- consider the case that the initial loop condition is not met
- make sure every control flow path is tested
- consider large argument values (`INT_MAX` or `INT_MIN`)
- test for special argument values (`-1`, `0`, `1`, `NULL`)

Module testing

When testing an entire *module* that has side effects, testing each individual function may not be sufficient.

You may also have to test the interaction between functions, by testing *sequences* of function calls.

Goals of this module

You should be comfortable with the terminology for I/O, streams and tracing.

You should have a basic understanding that there are different streams, including the standard streams, and how streams are either text or binary streams.

You should be comfortable using the input functions `read` in Racket, and `scanf` & `getchar` in C to make interactive programs.

You should be comfortable using the `RunC` testing environment.