I/O & Testing

Readings: CP:AMA 2.5

I/O

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.

Output

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

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`).

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

Writing to **text files** 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.

```
const bool TRACE = true;  // set to false to turn off tracing
//..
if (TRACE) printf("The value of i is: %d\n",i);
```

In practice, tracing is commonly implemented with *macros* (`#define`) 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 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.

`EOF` is a special value to indicate that there is no more input.

In our `Seashell` environment, a `Ctrl-D` ("Control D") keyboard sequence sends an `EOF`.

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.

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

Just as with `printf`, you can read more than one value in a single call to `scanf` with multiple placeholders.

However, in this course **only read in one value per `scanf`**.

This will help you debug your code and facilitate our testing,

`scanf` requires a **pointer** to a variable to **store the input value**.

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

The return value of `scanf` is the number (count) of values successfully read, so in this course a value of one is “success”.

`scanf("%d",&i)` will ignore whitespace (spaces and newlines) and read in the next integer. However, if the next input to be read is not a valid integer (e.g., a letter), it will stop reading and return zero.

The return value can also be the special constant value `EOF`. You should **not assume that** `EOF` is zero (or `false`).

`scanf` can be used to read in characters (`"%c"`).

You may or may not want to ignore whitespace when reading in a `char`.

```
// reads in next character (may be whitespace character)
count = scanf("%c",&c);

// reads in next character, ignoring whitespace
count = scanf(" %c",&c);
```

The extra leading space in the second example indicates that whitespace should be ignored.

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, Noun, Adverb & Adjective :\n")
  (define verb (read))
  (define noun (read))
  (define adverb (read))
  (define adj (read))
  (printf "The two ~as were too ~a to ~a ~a.\n"
          noun adj verb adverb))

(madlib)
```

example 1: interactive C

```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);
}
```

example 2: interactive C

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

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

example: interactive testing in C

```c
#include "addsqr.h"

int main(void) {
  char func;
  int x,y;
  while (1) {
    if (scanf(" %c", &func) != 1) break;
    if (func == 'x') break;
    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));
    }
  }
}
```

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 advantage 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 `Seashell` environment has support to *automate* interactive testing.

Seashell testing Demo

(to be done in class)

Testing in C

Here are some additional tips for 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 a *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.

When you test the interaction **between** modules and **groups** of modules at once, it is known as ***integration testing***. Whenever a module is changed, a full integration test should be run.

Goals of this Section

At the end of this section, you should be able to:

- use the I/O terminology introduced
- use the input functions `read` in Racket and `scanf` in C to make interactive programs
- use the `Seashell` testing environment effectively