CS 136 Fall 2021

Tutorial 02 – Control and Information Flow

September 21, 2021
Today’s Topics

In no particular order:
• Section 3
• Recursion: flow of information
Section 3

Who has read it?

Any questions?

What I think is important:

Side effects: imperative programming builds upon a state machine. The state consists of memory (e.g., variables) and files (e.g., input, output) (p. 12)

Quick reminder: assertion-based testing for returned values, IO-testing for output and returned values (with testing harness) (pp. 18, 64)

Always initialize variables: now useful as sanity check (e.g., INT_MIN), later useful to avoid SEGMENTATION FAULTS.

Global variables are bad: avoid if necessary. Side-effects are a necessary evil; control and minimize them!

Avoid shadowing identifiers: few advantages, a lot of confusion.
**Recursion: Control flow**

```c
void read(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return;
    } else {
        read();
        return;
    }
}
```

![Recursion Diagram]

- read
- base: return
- rec: read
- return

- read
- base: return
- rec: read
- return

- read
- base: return
- rec: read
- return

- read
- base: return
- rec: read
- return

...
Recursion: Control flow

```c
void read(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return;
    } else {
        read();
        return;
    }
}
```
Recursion: Information flow “upstream” 
(This is another way to think about accumulative recursion)

```c
void read_wrk(int count) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        printf("BC reached after %d numbers.\n", count);
    } else {
        read_wrk(count + 1);
    }
}

void read(void) {
    read_wrk(0);
}
```
Recursion: Information flow “downstream”
(This is another way to think about simple recursion)

```
int read(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return 0;
    } else {
        return 1 + read_wrk();
    }
}

void client(void) {
    printf("There were a total of %d numbers.\n", read());
}
```
Recursion: Information flow

What are the advantages / disadvantages of sending information “upstream”?

- Information cannot leave recursive chain
+ Multiple information can be sent upstream (as parameters)

read_wrk(n-1)ₙ
...
read_wrk(2)₃
read_wrk(1)₂
read_wrk(0)₁

What are the advantages / disadvantages of sending information “downstream”?

+ Information can be transferred to caller
- Only one information can be sent downstream (as return value)

read()ₙ => 0
...
read()₃ => n-3
read()₂ => n-2
read()₁ => n-1
Recursion: Information flow

What are the advantages / disadvantages of sending information “upstream”?  

- Information cannot leave recursive chain  
+ Multiple information can be sent upstream (as parameters)

What are the advantages / disadvantages of sending information “downstream”?  

+ Information can be transferred to caller  
- Only one information can be sent downstream (as return value)

Later, we will see how to get around this limitation using  
• compound data (sending structures “downstream”)  
• pointers (sending storage locations “upstream”)
Recursion: Information manipulation

Compare

```c
int count(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return 0;
    } else {
        return 1 + count();
    }
}
```

```c
int count(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return 0;
    } else {
        int rec_val = count();
        return 1 + rec_val;
    }
}
```

What is the advantage of the implementation on the right?

• Decoupling of recursive call from returned value.
Recursion: Information manipulation

Compare

```c
int count(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return 0;
    } else {
        return 1 + count();
    }
}
```

```c
int count(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return 0;
    } else {
        // do something here
        int rec_val = count();
        // do something here
        return 1 + rec_val;
    }
}
```

What is the advantage of the implementation on the right?

- Decoupling of recursive call from returned value.
Recursion: Information manipulation

Compare

```c
int count(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return 0;
    } else {
        return 1 + count();
    }
}
```

```c
int count(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return 0;
    } else {
        // do something here
        int rec_val = count();
        // do something here
        return 1 + rec_val;
    }
}
```

What is the advantage of the implementation on the right?

- Decoupling of recursive call from returned value (“Upstream” in green, “downstream” in orange).
Recursion: Information manipulation

Modify the program below so that it prints the following:

```c
int count(void) {
    int input = read_int();
    if (input == READ_INT_FAIL) {
        return INT_MIN;
    } else {
        return count();
    }
}

int main(void) {
    printf("There were a total of %d numbers.\n", count());
}
```

Tips

- Think about what information is available at which moment. In other words: think about what to print when.
- Finding the pattern in the output might help you.

There have been 0 numbers so far...
There have been 1 numbers so far...
There have been 2 numbers so far...
There have been 3 numbers so far...
There have been 4 numbers so far...
There have been 5 numbers so far...
Base condition reached after 6 numbers.
...there have been another 0 numbers.
...there have been another 1 numbers.
...there have been another 2 numbers.
...there have been another 3 numbers.
...there have been another 4 numbers.
...there have been another 5 numbers.
There were a total of 6 numbers.