Module 2: Imperative C

Programming Paradigms

There are several different ways of thinking about computer programming. You will see several during your studies.

1. In CS135, you focused on the *functional* paradigm: our programs consisted of *pure functions*, that is, functions with no *side effects*. (We never changed a variable; we instead created a new value.)
2. in CS136 we focus on the *procedural* paradigm: we will write "procedures" that are a collection of instructions, and these instructions will often have *side effects*, such as changing what is in memory.

Last module was working in C, but our code was (mostly) in the functional paradigm.

Now we will start working more with *side effects*.

Defining variables

We create a new variable in much the same way that we did it in Racket.

Only the syntax is different: `<type> <name> = <value> ;`

```
(define value 17)
(define (calc value)
  (cond [(even? value)
         (local [(define value 42)]
           value)]
        [else
         value]))
```

```
int value = 17;
int calc(int value) {
  if (0 == value % 2) {
    int value = 42;
    return value;
  } else {
    return value;
  }
}
```

- Outside any function, we create a **global variable**.
- Inside a function, we create a **parameter**.
- Inside a `{}` block (`local` block in Racket), we create a **local variable**.

Defining variables

Note: it is possible to define a variable without specifying its value. But then the variable will still have a value; we just won't know what it is.

```
int foo;
printf("foo is: %d\n", foo);
```

Don't do this. Always initialize your variables.

Mutation

Unlike in the Racket we used in CS135, we can *change* the value of a variable.

We must start with a defined variable:

```
int x = 5;  // define and initialize x.
```

Then to mutate it, we can:

- Use the **assignment operator**, `=`, even outside a definition.

```
x = 6;      // assign a new value to x.
x = x + 10; // calculate a new value, and assign to x.
```

- Use **mutation-assignment** operators, a "shorthand" for the assignment operator:

```
x *= 2;     // assignment-mutation operator; like x = x * 2
x += 8;     // assignment-mutation operator; like x = x + 8
```

- Use **increment** and **decrement** operators `++` and `--`.

```
++x;        // increment operator; like x = x + 1
x++;        // increment operator; also like x = x + 1
```

Mutation

```c
void make_cookies(void) {
  int total = 0;
  int shortening = 190;
  total = total + shortening;
  printf("Adding %d g shortening; total is %d g\n", shortening, total);

  int sugar = 130;
  total = total + sugar;
  printf("Adding %d g sugar; total is %d g\n", sugar, total);

  int flour = 360;
  total += flour;
  printf("Adding %d g flour; total is %d g\n", flour, total);

  int vanilla = 20;
  total += vanilla;
  printf("Adding %d g vanilla; total is %d g\n", vanilla, total);
}
```

Mutation

In C, an expression involving mutation operators has a value, so we *can* use that value in another expression.

```
int x = 42;
int y = 10;
int z = 6;
z *= (y *= y + (x %= z - 2) + z);
```

What are the values of `x`, `y`, and `z` after executing this line?

I don't actually care. **Don't write such code.** Instead write:

```
x %= z - 2;
y *= y + x + z;
z *= y;
```

> **!** Each statement should only "do one thing". Write for clarity.

++x and x++

If you use ++x or x++ alone, their behaviour is indistinguishable:

```
x = 5;
x++;
// x is now 6.
```

```
x = 5;
++x;
// x is now 6.
```

++x and x++

Technically, if we start with `x = 5;`:

- In `y = x++`, we assig `x` to `y`, and afterward increment `x`. We end with `y == 5`.
- In `y = ++x`, we increment `x` first, and afterwards assign this to `y`. We end with `y == 6`.

Exercise

Discuss with your neighbour; figure out what the following code should display. Once you have written down your best guess, run it.

```
void squish(int lo, int hi) {
  if (lo < hi) {
    printf("%d %d\n", lo, hi);
    return squish(++lo, hi--);
  }
}

int main(void) {
  squish(4, 8);
}
```

```c
void squish(int lo, int hi) {
  if (lo < hi) {
    printf("%d %d\n", lo, hi);
    return squish(++lo, hi--);
  }
}

int main(void) {
  squish(4, 8);
}
```

Here one line does 3 things: mutate `lo`, mutate `hi`, and make the recursive call. Don't do that.

Instead write:

```c
++lo;
hi--;
return squish(lo, hi);
```

Or better:

```c
return squish(lo + 1, hi - 1);
```

! Each statement should only "do one thing". Write for clarity.

Exercise

Without running it, figure out what the following code displays.

```
int whatever(int n, int acc) {
  if (n = 1) {
    return acc;
  } else {
    return whatever(n - 1, n + acc);
  }
}

int main(void) {
  printf("%d\n", whatever(3, 0));
}
```

The expression `n = 1` has a value of `1`, which is a true value. So we always run the first `if`.

What would happen if we wrote `if (n = 0)` instead?

This is why I always write `if (0 == n)`. `if (0 = n)` is a syntax error; the compiler will notice.

Constants

Sometimes, we have “variables” that will not change while our program is running.

We should add **`const`** in our type to indicate that the value will not change.

Any time we write code in the functional paradigm, we should use **`const`**:

```
// fact(n) Calculate n!.
const int fact(const int n) {
  if (0 == n) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

Technically, we should have been doing this since day 1.

Constants

It is good style to use `const` when appropriate, as it:

- communicates the intended use of the variable,
- prevents “accidental” or unintended mutation, and
- may help to optimize (speed up) your code

It is almost always bad style to change the value of global variables.

> **!** In this course, unless otherwise specified, **always** use `const` for global variables.

Side Effects

If we write something down, erase something, or otherwise change the state of the universe, it is a side effect.

In computer programming, there are two main types of side effects:

1. I/O: Input and output (interacting with things outside of the program)
2. Mutation: changing the contents of memory

Merely changing the value of a variable, inside the function where it is defined, is mutation. But it is not a side effect of the function.

To count as a side effect, it must affect the "outside".

For now, the only "mutation" side effect we could have is to change the value of a global variable... but it's almost always a bad idea to do that, and we forbid it.

At this point, the only "side effect" that a function can have is I/O.

Side Effects

I/O is the term used to describe how programs interact with the “real world”.
An app on your phone might:

- Write output:
  - display something on the screen
  - make a sound
  - write to local data or upload data to a server
- Read input:
  - get user input (touch screen, keyboard, microphone)
  - read from local data or a server
  - get sensor input (camera, accelerometer, GPS)

Side effect: output using `printf`

We have seen the function `printf`: it takes a string and possibly additional values, and prints to "output".

```
printf("One thing is %d and the other is %d.\n", 2 + 3, 84 / 2);
```

We see:

```
One thing is 5 and the other is 42.
```

We have seen that in a `printf` format string, `%d` is replaced with an integer.

In a `printf` format string, `%` followed by anything is treated specially. To print a single `%`, write two in the format string:

```
printf("I want you to give %d%% today!\n", 110);
```

We see:

```
I want you to give 110% today!
```

Side effect: output from `printf`

We always want to be able to carefully test our code.

`assert` only checks that an expression has a true value.
It cannot "see" anything that we printed.

We can look at the output to verify that it looks correct: use the "Run Code" button in edX.

Better: our edX environment has been set up to let us create I/O tests.

For each test, we need to create two files:

1. one called `<something>.in` that will used as "input" to our program
2. one called `<something>.expect` that will be the "expected output"

> **!** To test output, we need to create a "input" file, even if it is empty.

Side effect: output from `printf`

A function that has side effects might not return a value.
We indicate this by using **`void`** as the “return type”.

Exercise

Write a simple recursive function **`void`** `countdown(`**`int`** `n)` that prints all the numbers from `n` down to 1, one per line.
Call `countdown` from your `main` function, and create the necessary files to test it.

Side effect: output from `printf`

As part of the documentation of our functions, we should indicate any side-effects.

For any function that uses `printf`, add:

```
// effects: produces output
```

Ex. Complete the documentation of your function `countdown`.

We do not consider use of `assert` or tracing tools such as `trace_int` to be a side-effect. We imagine that these tools are used only in development, and will be removed in production.

Reading input with `read_int`

In the `cs136` module we define a function `int read_int(void)`.

- If it can read an integer from input, it returns that integer.
- If it cannot read an integer, it returns `READ_INT_FAIL`, a constant defined in the module.

Reading from input is another side effect.
If we read from input, we should document it with:

```
// effects: reads from input
```

> **!** It turns out that `READ_INT_FAIL` is `-2147483648`, but don't depend on this! Use the constant to get this value.

Reading input with `read_int`

Here is a program that reads all the integers from input, and doubles them all:

```
// twice_all() Reads as many ints as
//     there are, and prints each * 2.
// effects: reads input
//          produces output
void twice_all(void) {
  const int val = read_int();
  if (READ_INT_FAIL != val) {
    printf("%d\n", 2 * val);
    twice_all();
  }
}

int main(void) {
  twice_all();
}
```

- We can run it by putting something in the "stdin" frame
- or test it by creating a pair of test files:
  - with input in `foo.in`,
  - correct output in `foo.expect`

Reading input with `read_int`

Exercise

Write a recursive function `void countdown_all(void)` that reads all integers from input, and counts down from each of these numbers.
For example, if the input is `3 1`, it should print:

```
3
2
1
1
```

Create a pair of files to test this set of inputs.
Then create a second pair of files to test with different inputs.

As a reminder, here is the function from the last slide:

```
void twice_all(void) {
  const int val = read_int();
  if (READ_INT_FAIL != val) {
    printf("%d\n", 2 * val);
    twice_all();
  }
}
```

Reading input with `read_int`

A function that reads input (repeatedly), then calls a function to be tested is a kind of "testing harness". Your function `countdown_all` doesn't really do anything; all it does is test your function `void countdown(int n)`.

Exercise

The function `fact2` is written very compactly using the ternary operator `?:`
Write a testing harness for `fact2`.

```
// fact2(n) Calculate n!.
// requires: n >= 0
int fact2(int n) {
  assert(n >= 0);
  return n ? n * fact2(n - 1) : 1;
}
```

The value of `X ? Y : Z` will be `Y` if `X` is non-zero, and `Z` if `X` is zero.
It usually makes code more confusing; I would avoid it.

Exercise

Write a function `void downup(int n)` that prints values counting down from `n` to 0, then back up to `n`. **Do not write a second function or add a parameter.**

For example, `downup(2)` should print:

```
2
1
0
1
2
```

Exercise

Write a function that reads as many integers as are available, printing the partial sum after reading each new value. The function should return the total.

For example, if it reads `2 4 6 0 1`, it should print `2 6 12 12 13`, one value per line.

Module summary

- Create variables, with and without `const`
- Change the value of variables using the assignment operator `=`, as well as assignment-mutation operators such as `+=`, `/=`, and `%=`, and increment operators `++` and `--`.
- Use `printf` to display output, and `read_int` to read an integer from input.
- Use these tools to create a I/O based testing harness.