Introduction to Imperative C

Readings: CP:AMA 2.4, 3.1, 4.2–4.5, 5.2, 10

- the ordering of topics is different in the text
- some portions of the above sections have not been covered yet
- some previously listed sections have now been covered in more detail

The primary goal of this section is to be able to write programs that use I/O and mutation.
Functional vs. imperative programming

In CS 135 the focus is on functional programming, where functions behave very “mathematically”. The only purpose of a function is to return a value, and the value depends only on the argument value(s).

The functional programming paradigm is to only use constant values that never change. Functions return new values rather than changing existing ones.

A programming paradigm can also be thought of as a programming “approach”, “philosophy” or “style”.
example: functional programming paradigm

```scheme
(define n 5)
(add1 n) ; => 6
n ; => 5
```

With functional programming, `(add1 n)` returns a **new** value, but it does not actually *change* `n`. Once `n` is *defined*, it is a **constant** and always has the same value.

```scheme
(define lon '(15 23 4 42 8 16))
(sort lon <) ; => '(4 8 15 16 23 42)
lon ; => '(15 23 4 42 8 16)
```

Similarly, `(sort lon)` returns a **new** list that is sorted, but the original list `lon` does not change.
In this course, our focus is on the *imperative programming paradigm*. 

In the English language, an imperative is an instruction or command: “Give me your money!”

In imperative programming we use a **sequence of statements** (or “instructions”).

To highlight the difference, we consider an imperative example in Racket (which will seem bizarre).

Many modern languages are “**multi-paradigm**”. Racket was primarily designed as a functional language but also supports imperative language features.
Functions with multiple expressions

We have only seen Racket functions with a *single* expression:

\[
\text{(define (f n)} \\
\quad (\ast \ n \ 4))
\]

\[
(f \ 10) \\
\Rightarrow \ 40
\]

In full Racket, functions can contain *multiple* expressions:

\[
\text{(define (mystery n)} \\
\quad (/ \ n \ 2) \\
\quad (+ \ n \ 3) \\
\quad (\ast \ n \ 4))
\]

\[
\text{(mystery 10)} \\
\Rightarrow \ ???
\]
Racket (implicitly) always uses the `begin` special form:

```
(define (mystery n)
  (begin ; explicitly showing begin
    (/ n 2)
    (+ n 3)
    (* n 4)))
```

The behaviour of `begin` is as follows:

- each expression is evaluated **in order**
- the value of each expression is *discarded* (or “ignored”), except for the last expression
- `begin` produces the value of the **last** expression

In the *functional paradigm*, discarded expressions are **useless**.
Side effects

In the *imperative paradigm*, an expression can generate a *side effect* (in addition to producing a value).

A *side effect* changes the *state* of the program (or “the world”).

Functions and programs can also have side effects.

We borrow terminology from medicine, where a pill to cure an illness (e.g., a headache) might also have a side effect (e.g., it causes all of your hair to fall out).
example: side effects

Imagine that the following function exists:

```
(define (scary n)
  (begin
    (turn-the-lights 'off)
    (play-mp3 "scary.mp3")
    (shout "Boo!"))
  (* n 4))
```

(scary 10)

=> 40

(scary 10) still returns 40 as expected, but it also does more: it changes the state of the “world” by turning out the lights, playing scary music and shouting Boo!
(define (scary n)
  (begin
    (/ n 2) ;; not useful
    (turn-the-lights 'off)
    (play-mp3 "scary.mp3")
    (shout "Boo!")
    (* n 4)))

Adding an extra mathematical expression such as (/ n 2) is not useful because it does not have a side effect.

**begin** is an **imperative** special form that only makes sense if the discarded expressions have side effects.
In functional programming there are no side effects.

This is one of the significant differences between imperative and functional programming.

Some purists insist that a function with a side effect is no longer a “function” in the mathematical sense and call it a “procedure” (or a “routine”).

We are more relaxed: a function can have side effects.

The “imperative programming paradigm” is also closely related to the “procedural programming paradigm”.
Documentation: side effects

You should clearly communicate if a function has a side effect.

;; (scary n) multiplies n by four
;; scary: Int -> Int
;; effects: modifies the lights
;;         plays sound
;;         shouts

(define (scary n)
  (begin
    (turn-the-lights 'off)
    (play-mp3 "scary.mp3")
    (shout "Boo!")
    (* n 4)))

Add an effects: section to the function documentation if there are any side effects.
Types of side effects

In this course, we encounter two types of side effects:

- Input & Output (I/O for short)
- Mutation
  (memory modification & variables)

Both involve changing state (more on this later).

In this section we first introduce output, then we explore mutation and conclude with input.
I/O

I/O is the term used to describe how programs interact with the “real world”. For example a program (“app”) on your phone may interact with you in many different ways.

Input may include: a touch screen (onscreen keyboard), your voice or the camera.

Output may include: the screen (display), sounds or vibrations.

A program may also interact with non-human entities such as: a file, a GPS, a printer or even a different computer on the internet.
Text I/O

In this course, we only use simple text-based I/O.

To display text output in C, we use the printf function.

```c
#include "cs136.h"

int main(void) {
    printf("Hello, World");
}
```

Hello, World

For now, make sure you have the above #include.

We omit it in the following slides to save space.
C blocks

What if we want to have more than one `printf`?

We have already seen C’s equivalent of `begin`: a `block` (`{}`), also known as a `compound statement`, is a `sequence of statements`. (Unlike `begin`, a C block does not produce a value).

```c
int main(void) {
    printf("Hello, World");
    printf("C is fun!");
}
```

† Blocks can also contain local variable `definitions`, which are not statements.
int main(void) {
    printf("Hello, World");
    printf("C is fun!");
}

Hello, WorldC is fun!

The **newline** character (\n) is necessary to properly format your output to appear on multiple lines.

printf("Hello, World\\n");
printf("C is\\nfun\\n");

Hello, World
C is
fun!
The first parameter of `printf` is a "string". Until we discuss strings in Section 09, this is one of the few places you are allowed to use strings.

We can output other value types by using a `placeholder` within the string and providing an additional argument.

```
printf("2 plus 2 is: %d\n", 2 + 2);
2 plus 2 is: 4
```

The "%d" placeholder is replaced with the value of the additional argument. There can be multiple placeholders, each requiring an additional argument.

```
printf("%d plus %d is: %d\n", 2, 2, 2 + 2);
2 plus 2 is: 4
```
C uses different placeholders for each type. The placeholder we use for integers is "\%d" (which means “decimal format”).

To output a percent sign (%), use two (\%\%).

```c
printf("I am %d%% sure you should watch your", 100);
printf("spacing!\n");
```

I am 100% sure you should watch yours-spacing!

Similarly,

- to print a backslash (\), use two (\\)
- to print a quote ("), add an extra backslash (\")
Many computer languages have a `printf` function and use the same placeholder syntax as C. The placeholders are also known as `format specifiers` (the `f` in `printf`).

The full C `printf` placeholder syntax allows you to control the format and align your output.

```
printf("4 digits with zero padding: %04d\n", 42);
4 digits with zero padding: 0042
```

See CP:AMA 22.3 for more details.

In this course, simple "`%d`" formatting is usually sufficient.
Functions with side effects

Both functions below return the same value: an int \((n^2)\).

```c
// quiet_sqr(n) squares n
int quiet_sqr(int n) {
    return n * n;
}

// noisy_sqr(n) squares n
// effects: produces output
int noisy_sqr(int n) {
    printf("I'm squaring %d\n", n);
    return n * n;
}
```

However, only noisy_sqr has a side effect.
I/O terminology

In the context of I/O, be very careful with your terminology.

```c
int quiet_sqr(int n) {
    return n * n;
}
```

Informally, someone might say:

“if you input 7 into quiet_sqr, it outputs 49”.

This is **poor terminology**: quiet_sqr does not read input and does not print any output.

Instead, you should say:

“if you **pass** 7 to quiet_sqr, it **returns** 49”.
int noisy_sqr(int n) {
    printf("I'm squaring %d\n", n);
    return n * n;
}

For noisy_sqr, you should say:

"if you pass 7 to noisy_sqr, it outputs a message and returns 49".

It is common for beginners to confuse output (e.g., via printf) and the return value.

Ensure you understand the correct terminology and read your assignments very carefully.
Testing I/O

We can use asserts to test that quiet_sqr and noisy_sqr return the correct values:

```c
int main(void) {
    assert(quiet_sqr(3) == 9);
    assert(quiet_sqr(7) == 49);
    assert(noisy_sqr(3) == 9);
    assert(noisy_sqr(7) == 49);
}
```

But how do we test that noisy_sqr produced the correct output?

Testing functions with side effects is much more challenging.
Seashell: [RUN] vs. [I/O TEST]

In Seashell there are two ways of “running” your program.

The only difference is the I/O behaviour:

- With the [RUN] button, input is read from the **keyboard**. Any output is displayed in the console (“screen”).

- With the [I/O TEST] button, input is read from **input file(s)** (e.g., `testfile.in`) instead of the keyboard.

If a corresponding **output test file** exists (e.g., `testfile.expect`), Seashell checks the output against the expected output test file to see if they match.
We expect the output of this program to be:

I'm squaring 3
I'm squaring 7

So we can create an `.expect` file with that expected text. Even though this program does not read any input, we must also create a corresponding `.in` file for the I/O test to work correctly.
Tracing tools

In this course we have provided *tracing tools* to help you “see” what is happening in your program and to help **debug** your code.

- `trace_msg(msg);` displays a tracing message
- `trace_int(expression);` evaluates an *(int)* expression

The tracing tools do **not** interfere with your I/O testing.

For your assignments, you should never `printf` any unnecessary output as it may affect your correctness results.

You should always use our tracing tools to help debug your code.
```c
int quiet_sqr(int n) {
    trace_msg("quiet_sqr was called");
    trace_int(n);
    return n * n;
}

int main(void) {
    trace_msg("main started");
    assert(quiet_sqr(3) == 9);
    trace_int(quiet_sqr(7));
}
```

"main started"
"quiet_sqr was called"
n => 3
"quiet_sqr was called"
n => 7
quiet_sqr(7) => 49
Our tracing tools print to a different I/O stream.

By default, `printf` outputs to the `stdout` (standard output) stream.

Our tracing tools output to the `stderr` (standard error) stream, which is used for errors and other diagnostic messages.

In Marmoset, and when Seashell performs an [I/O TEST], only the `stdout` stream is tested.

When your [RUN] your code, the two streams may appear mixed together in the screen (console) output.
void functions

A function may be designed to only generate a side effect, and not return a value.

We have already used `void` to define functions with no parameters.

`void` is also used to define a function that returns “nothing”.

```
// say_hello() displays a friendly message
// effects: produces output

void say_hello(void) {
    printf("hello!\n");
    return;               // this is optional
}
```

In a `void` function, `return` has no expression and it is optional.
Expression statements

It might surprise you that C’s `printf` is not a `void` function.

`printf` returns an `int` corresponding to the number of characters printed.

`printf("hello!\n")` is an `expression` with the value of 7. (note that the newline (`\n`) is a single character).

An `expression statement` is an expression followed by a semicolon (`;`).

`printf("hello!\n");`
In the above code, there is a single block (compound statement) that contains a sequence of five expression statements with the values of 9, 10, 11, 12 and 13, respectively.

What happens to those values?
The value of an expression statement is discarded. (just like with Racket’s begin).

The five values from the previous example are never used.

The only purpose of an expression statement is to generate side effects.

Seashell may give you a warning if you have an expression statement without an obvious side effect (10 + 3;) but when there is a function call (quiet_sqr(3);) there is no warning because it assumes it has a side effect (even if there is none).
Statements

There are only three types of C statements:

- **compound statements (blocks)**
  a sequence of statements

- **expression statements**
  for generating side effects

- **control flow statements**
  control the order in which other statements are executed
  (*e.g.*, `return`, `if` and `else`)

We discuss control flow in more detail in Section 04.
State

The biggest difference between the imperative and functional paradigms is the existence of *side effects*.

We have seen how a side effect changes the *state* of a program (or “the world”). For example, `printf` changes the state of the output.

Another way of defining the *imperative programming paradigm* is that it *manipulates state*.

*State* refers to the value of some data (or “information”) *at a moment in time*. 
Consider a simple light switch: at any moment in time it is either in the “on” state or the “off” state.

For another example, consider your bank account balance.

At any moment in time, your bank account has a specific balance.

In other words, your account is in a specific “state”.

When you withdraw money from your account, the balance changes and the account is in a new “state”.

State is related to memory, which is discussed in Section 04.
Variables

We use **variables** to store state information (values).

To define a variable in C, we need (in order):

- the **type** (*e.g.*, `int`)
- the **identifier** (“name”)
- the **initial value**

```c
int my_variable = 7;
```

The equal sign (`=`) and semicolon (`;`) complete the syntax.
Variable scope

Variables can have *global scope* or *local scope*.

```c
int my_global_variable = 7;

int main(void) {
  int my_local_variable = 11;
  //...
}
```

Global variables are defined *outside* of functions (at the “top level”).

Local variables are defined *inside* of functions.

For now, always make sure you define your variables *above* any code that references them.
Local variables are also known as **block variables** because their scope is restricted to the *block* they are defined in.

Variables with the same name can *shadow* other variables from outer scopes, but this is obviously very poor style. The following code defines three **different** variables named *n*.

```c
int n = 1;

int main(void) {
    trace_int(n);    // n => 1
    int n = 2;
    trace_int(n);    // n => 2
    {
        int n = 3;
        trace_int(n);    // n => 3
    }
}
```
Initialization

C allows you to define variables without initializing them, but it is very bad style.

```c
int my_variable = 7; // initialized

int another_variable; // uninitialized (BAD!)
```

Always initialize your variables.

In Section 04 we discuss the behaviour of uninitialized variables.
Mutation

When the value of a variable is changed, it is known as *mutation*.

For most imperative programmers, mutation is second nature and not given a special name. They rarely use the term “*mutation*” (the word does not appear in the CP:AMA text).
```c
int main(void) {
    int m = 5;
    trace_int(m);
    m = 6; // mutation!
    trace_int(m);
} 

m => 5
m => 6
```
Assignment Operator

In C, mutation is achieved with the assignment operator \( = \).

\[
m = m + 1;
\]

- The “left hand side” (LHS) of the assignment operator \textbf{must} be the name of a variable (for now).
- The RHS must be an expression with the same \textit{type} as the LHS.
- The variable on the LHS is changed (mutated) to store the \textbf{value} of the RHS. In other words, the RHS value is \textbf{assigned} to the variable.
- This is a \textit{side effect}: the state of the variable has changed.
The assignment operator is not symmetric.

\[ x = y; \]

is **not the same** as

\[ y = x; \]

Some languages use

\[ x := y \]

or

\[ x <- y \]

to make it clearer that it is an asymmetric assignment.
In addition to the mutation side effect, the assignment operator (=) also produces the value of the expression on the right hand side. This is occasionally used to perform multiple assignments.

\[ x = y = z = 0; \]

Avoid having more than one side effect per expression statement.

```c
printf("y is %d\n", y = 5); // never do this!
printf("y is %d\n", y = 5 + (x = 3)); // this is even worse!
z = 1 + (z = z + 1); // really bad style!
```
Remember, **always use a double == for equality**, not a single = (which we now know is the assignment operator).

```c
x = 0;
if (x = 1) {
    printf("disaster!\n");
}
```

(x = 1) assigns 1 to x and produces the value 1, so the if expression is always true, and it always prints disaster!

Pro Tip: some defensive programmers get in the habit of writing (1 == x) instead of (x == 1). This causes an error if they accidentally use a single =.
Initialization is not assignment

The equal sign (\(=\)) used in *initialization* is **not** the assignment operator.

Both initialization and assignment use the equal sign (=), but they have different semantics.

```c
int n = 5;    // initialization syntax
n = 6;        // assignment operator
```

The distinction is not too important now, but the subtle difference becomes important later.
Mutating local variables

When documenting the side effects of a function, we are only concerned in documenting state changes that are external to (or visible outside of) the function.

```c
int f(int n) {
    int k = 3;
    k = k + 1;
    return n * k;
}
```

The function f mutates the local variable k, but it does not modify any global variables or use any I/O.

As a result, we say that f itself has no side effects.
Mutating parameters

As we will explore in Section 04, C makes a copy of the argument values passed to a function (“pass by value” convention).

This means that parameters are nearly indistinguishable from local variables, and can be mutated.

```c
int g(int n) {
    n = n * 4;
    return n;
}
```

As with the previous example, the function g itself does not have any side effects.
example: mutating a parameter

In the following example, the variable $x$ in `main` is never changed because a **copy** of $x$ is passed to `g`.

```c
int g(int n) {
    n = n * 4;
    return n;
}

int main(void) {
    int x = 10;
    trace_int(x);
    trace_int(g(x));
    trace_int(g(x));
    trace_int(x);
}
```

- $x$ => 10
- $g(x)$ => 40
- $x$ => 10
Mutating global variables

A function that mutates a global variable does have a side effect.

```c
int count = 0;

int increment(void) {
    count = count + 1;
    return count;
}

int main(void) {
    trace_int(increment());
    trace_int(increment());
    trace_int(increment());
}
```

increment() => 1
increment() => 2
increment() => 3
Even if a function does not have a side effect, its behaviour may depend on other mutable global variables.

```c
int n = 10;

int addn(int k) {
    return k + n;
}

int main(void) {
    trace_int(addn(5));
    n = 100;
    trace_int(addn(5));
}

addn(5) => 15
addn(5) => 105
```
Functions: different paradigms

Functional programming paradigm:

- functions have no side effects
- returned values \textit{only depend on the argument value(s)}

Imperative programming paradigm:

- functions may have side effects
- in addition to the argument value(s), returned values may also depend on the \textit{state} of the program
More assignment operators

The following statement forms are so common

\[
x = x + 1;
y = y + z;
\]

that C has an addition assignment operator (+=) that combines the
addition and assignment operator.

\[
x += 1; \quad // \text{equivalent to } x = x + 1;
y += z; \quad // \text{equivalent to } y = y + z;
\]

There are also assignment operators for other operations.

\[-=, *=, /=, %=\].

As with the simple assignment operator, do not use these operators
within larger expressions.
As if the simplification from \( x = x + 1 \) to \( x += 1 \) was not enough, there are also \textit{increment} and \textit{decrement} operators that increase and decrease the value of a variable by one.

\[
++x; \\
--x; \\
// or, alternatively \\
x++; \\
x--; \\
\]

It is best not to use these operators within a larger expression, and only use them in simple statements as above.

The difference between \( x++ \) and \( ++x \) and the relationship between their values and their side effects is tricky (see following slide).

The language \texttt{C++} is a pun: one bigger (better) than \texttt{C}.
The *prefix* increment operator (\texttt{++x}) and the *postfix* increment operator (\texttt{x++}) both increment \texttt{x}, they just have different *precedences* within the *order of operations*.

\texttt{x++} produces the “old” value of \texttt{x} and then increments \texttt{x}.

\texttt{++x} increments \texttt{x} and then produces the “new” value of \texttt{x}.

```c
x = 5;
j = x++; // j = 5, x = 6

x = 5
j = ++x; // j = 6, x = 6
```

\texttt{++x} is preferred in most circumstances to improve clarity and efficiency.
Constants

A *constant* is a “variable” that is *immutable* (not mutable).

In other words, the value of a constant cannot be changed.

```c
const int my_constant = 42;
```

To define a C *constant*, we add the `const` keyword to the type.

In this course, the term “*variable*” is used for both variable and constant identifiers.

In the few instances where the difference matters, we use the terms “*mutable variables*” and “*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.

We often omit `const` in the slides, even where it would be good style, to keep the slides uncluttered.
Text input

In this course we have provided some helper functions to make reading in input easier. For example:

```c
// read_int() returns either the next int from input
// or READ_INT_FAIL
// effects: reads input

// the constant READ_INT_FAIL is returned by read_int() when:
// * the next int could not be successfully read from input, or
// * the end of input (e.g., EOF) is encountered
```

The converse of the C output function `printf` is the input function `scanf`, but we are not quite ready to use it (we introduce `scanf` in Section 05).
example: reading input

// count_even_inputs() counts the number of even values read from input (until a read failure occurs)
// effects: reads input

int count_even_inputs(void) {
    int n = read_int();
    if (n == READ_INT_FAIL) {
        return 0;
    } else if (n % 2 == 0) {
        return 1 + count_even_inputs();
    } else {
        return count_even_inputs();
    }
}

int main(void) {
    printf("%d\n", count_even_inputs());
}
example: reading input (continued)

If we [RUN] our program in Seashell, we can interactively enter **int** values via the keyboard.

To indicate that there is no more input, press the [EOF] (**End Of File**) button, or type **Ctrl-D**.
To test our program using [I/O TEST] in Seashell, we could add the following two test files:

test1.in  
1
2
2
3
4
4
5
6
test1.expect  
5
One of the great features of the [I/O TEST] in Seashell is that you can add multiple test files.

```
  test2.in  test2.expect
  1 3 3 7   0

  test3.in  test3.expect
  6 6 6     3
```
Input formatting

When C reads in `int` values, it skips over any whitespace (newlines and spaces).

The input:

```
1
2
3
4
5
```

and:

```
1 2 3
4 5
```

are indistinguishable to a function like `read_int`. 
Reading input

When reading input in our Seashell environment you have to be careful: once you read in a value, it can no longer be read again.

Typically, you want to store the read value (e.g., returned by `read_int`) in a variable so you can refer to it multiple times.

For example, consider this incorrect partial implementation:

```c
if (read_int() == READ_INT_FAIL) {  // Bad!
    return 0;
} else if (read_int() % 2 == 0) {  // Bad!
    //...
```

The first `read_int()` reads in the first `int`, but then that value is now “lost”. The next `read_int()` reads in the second `int`, which is not likely the desired behaviour.
Invalid input

In this course, unless otherwise specified, you do not have to worry about us testing your code with invalid input files.

The behaviour of `read_int` on invalid input can be a bit tricky (see CP:AMA 3.2). For example, for the input:

```
4 23skidoo 57
```

- the first call to `read_int()` returns 4
- the second call to `read_int()` returns 23
- any additional calls to `read_int()` return `READ_INT_FAIL`. 
Documenting side effects

So far, functions can only have three possible side effects:

- produce output
- read input
- mutate a global variable

You do not have to provide specifics in the “effects” section of the contract. Simply document the existence of the side effects.
example: documenting side effects

If the side effect does not always occur, preface it with “may” in your contract.

```c
// effects: reads input
//      may produce output
//      may mutate secret

void update_secret(void) {
    int n = read_int();
    if (n == READ_INT_FAIL) {
        printf("error: could not read in number\n");
    } else {
        secret = n;
    }
}
```
Goals of this Section

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

• explain what a side effect is
• document a side effect with an *effects* section
• print output with `printf` and read input using the provided functions (e.g., `read_int`)
• define global and local mutable variables and constants
• use the C assignment operators
• use the new terminology introduced, including: mutation, expression statements and compound statements (`{ }`)