Introduction to Imperative C

Optional Textbook 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 programming

In CS 135 we used the ***functional programming paradigm***:

- functions are "pure" (*a.k.a.* "mathematical"):
  - functions **only return values**
  - return values **only depend** on argument values
- only **constants** are used

In the ***imperative programming paradigm*** functions may be "impure" and we will use **variables**.

> A programming *paradigm* can also be thought of as a programming "approach", "philosophy" or "style".

Compound statement

In English, an imperative is an instruction: *"Give me your money!"*

In imperative programming, a **sequence of instructions** (or "statements") are *executed*. We have already seen this:

```
int main(void) {
  trace_int(1 + 1);     // do this first
  assert(3 > 2);        // then do this
  return 0;             // and then do this
}
```

A block `{}` is formally known as a ***compound statement***, which is simply a **sequence of statements**† (to be executed *in order*).

† Blocks can also contain local variable *definitions*.

Imperative programming

A program is mostly a sequence of statements to be executed.

Control flow is used to change the order of statements (Section 04).

The most significant difference between the functional and imperative paradigms is:

Imperative programming uses *side effects*
(functional programming does not).

In this section we explore *side effects*, beginning with *output*.

I/O

I/O (**I**nput/**O**utput) is the term used to describe how programs *interact* with the "real world".

A program ("app") on your phone may interact with you in many different ways:

Input: touch screen (onscreen keyboard), voice, camera

Output: screen (display), sounds, vibrations

It may also interact with non-human entities:
files, printers, GPS, other computers on the internet

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

Text I/O

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

```c
// My first program with I/O

#include "cs136.h"

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

```
Hello, World
```

`printf` is different than the *tracing tools* we use to debug and informally test our code (more on this distinction later).

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

```
Hello, WorldC is fun!
```

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

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

```
Hello, World
C is
fun!
```

The first parameter of `printf` is a `"string"`.

To output values, use a ***format specifier*** (the **f** in `printf`) within the string and provide an **additional argument**.

For an integer in "**d**ecimal format" the format specifier is `"%d"`.

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

```
2 plus 2 is: 4
```

In the output, the format specifier is **replaced** by the additional argument value.

Strings are introduced in Section 09.

There can be multiple format specifiers, each requiring an additional argument.

```
printf("%d plus %d is: %d\n", 2, 10 / 5, 2 + 2);
```

```
2 plus 2 is: 4
```

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

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

```
I am 100% sure you should watch yourspacing!
```

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 format specifier syntax as C.

The full C `printf` format specifier syntax controls the format and alignment of 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 sufficent.

Functions with side effects

Consider the two functions below:

```
int sqr(int n) {
  return n * n;
}

int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!\n", n);
  return n * n;
}
```

Both functions `return` the same value.

However, `noisy_sqr` does **more** than `return` a value.

In addition to returning a value, it **also produces output**.

`noisy_sqr` has a ***side effect***.

Side effects and state (introduction)

In general, a programming *side effect* is when the ***state*** of something "changes".

State refers to the value of some data (or "information") **at a moment in time**.

Consider the following "real world" example: You have a blank piece of paper, and then you write your name on that paper.

You have *changed the state* of that paper: at one moment it was blank, and in the next it was "autographed".

In other words, the *side effect* of writing your name was that you *changed the state* of the paper.

Documenting side effects

The `printf` function has a side effect: it changes the output (or "display").

By calling `printf`, the function `noisy_sqr` also has a side effect.

Clearly communicate if a function has a side effect.

```
// noisy_sqr(n) computes n^2
// effects: produces output

int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!\n", n);
  return n * n;
}
```

Add an **`effects:`** section to document any side effects.

If a side effect occurs conditionally, add the word "may":

```c
// noisy_abs(n) computes |n|
// effects: may produce output

int noisy_abs(int n) {
  if (n < 0) {
    printf("Yo! I'm changin' the sign!\n");
    return -n;
  } else {
    return n;
  }
}
```

In this course, there is no need to provide any detail in the "effects:" section (`// effects: produces output` is sufficient).

Occasionally, you may want to describe the output in the purpose (*e.g.*, understanding the output is essential to understanding the core behaviour of the function).

Debugging tools

Statements used for debugging and informal testing (*e.g.*, `assert`, `trace_int`) are **not** considered side effects.

Do not document `assert`s or tracing in the "effects:" section.

Feel free to leave your tracing code in your assignments: they will not affect your test results.

> Large software projects often have thousands of `assert` and tracing statements to aid debugging.
>
> They are usually disabled (or "turned off") when a project is finalized to improve performance, but they remain in the code.

I/O terminology

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

```
int sqr(int n) {
  return n * n;
}
```

Informally, someone might say:

"if you input 7 into `sqr`, it outputs 49".

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

Instead, say:

*"if 7 is **passed** to `sqr`, it **returns** 49".*

```c
int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!\n", n);
  return n * n;
}
```

For `noisy_sqr`, say:

*"if 7 is **passed** 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 carefully.**

Testing I/O

`assert`s can **test** that `sqr` and `noisy_sqr` **return** the correct values:

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

But `assert` **cannot** be used to **test** that `noisy_sqr` produces the **correct output**.

We will need a new method to test output...

Seashell: [RUN] vs. [I/O TEST]

In `Seashell` there are two ways of "running" a 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.

example: I/O test

```c
int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!\n", n);
  return n * n;
}

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

`test1.expect`:

```
Yo! I'm squaring -3!
Yo! I'm squaring 7!
```

A blank `test1.in` would also be necessary (I/O tests always require an `.in` file).

Our tracing tools print to a different output **stream** than `printf` (like writing on two different pieces of paper).

By default, `printf` outputs to the `stdout` (**st**an**d**ard **out**put) stream.

Our tracing tools output to the `stderr` (**st**an**d**ard **err**or) 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 you **[RUN]** your code, the two streams may appear mixed together in the screen (console) output.

void functions

`noisy_sqr` has a side effect *and* returns a value.

A function may *only have a side effect*, and **no return a value**.

The `void` keyword is used to indicate a function returns “nothing”.

```
// display_score(score, max) displays the player score
// effects: produces output

void display_score(int score, int max) {
  printf("your score is %d out of %d.\n", score, max);
  return;  // optional
}
```

In a `void` function, the `return` is **optional** and has no expression (when the end of a `void` function is reached, it `return`s automatically).

printf return value

Surprisingly, `printf` is **not** a `void` function (it `return`s an `int`).

`printf` returns the number of characters printed.

`printf("hello!\n")` returns 7 (`\n` is a single character).

In the following code, where does the 7 “go”?

```
int main(void) {
  printf("hello!\n");
}
```

Let’s revisit what a *statement* is...

Expression statements

The following expressions all have a value of `11`:

```
11
10 + 1
sqr(6) - sqr(5)
printf("expression\n")
printf("five\n") + 6
```

An ***expression statement*** is an expression with a semicolon (`;`).

```
int main(void) {
  11;
  10 + 1;
  sqr(6) - sqr(5);
  printf("expression\n");
  printf("five\n") + 6;
}
```

What happens to all of those elevens?

The **value** of an expression statement is **discarded** after it is executed.

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

Imperative programming is “programming by side effects”.

`Seashell` will give a warning if an expression statement *obviously* has no side effect: `8 + 1;`.

If an expression contains a function call there is no warning because side effects are “assumed” (even if there are none).

Statements

There are only three types of C statements:

- **compound statements (blocks)** `{}`

  a sequence of statements (to be executed in order)

- **expression statements**

  for generating side effects (values are discarded)

- **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.

More side effects

In addition to:

- Output (*e.g.*, `printf`)

we will encounter two more types of side effects (in this section):

- Input
- Mutation (modifying variables)

All side effects involve *changing state*.

Variables

Variables store values.

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

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

```
int my_variable = 7;     // definition
```

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

Definitions are not *statements*.

They are different “things” (syntactic units).

Mutation

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

```
int main(void) {
  int m = 5;           // definition (with initialization)
  trace_int(m);
  m = 6;               // mutation!
  trace_int(m);
  m = -1;              // more mutation!
  trace_int(m);
}
```

```
m => 5
m => 6
m => -1
```

At every moment in time, a variable must have a value.

When mutation occurs, the “state” (value) of the variable changes.

Mutation is a **side effect**.

For a “real world” example, consider your bank account balance.

At any moment in time, your account balance has a specific value (or “state”).

When you withdraw money from your account, the balance *changes* to a new value.

Withdrawing money has a *side effect*: it changes the value your bank balance.

For most imperative programmers, mutation is second nature and not given a special name (they rarely use the term *"mutation"*).

The word "mutation" does not appear in the CP:AMA textbook.

Assignment Operator

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

```
m = m + 1;
```

- The “right hand side” (RHS) must be an *expression* that produces a **value** with the same *type* as the LHS.
- The LHS **must** be the name of a variable (for now).
- The LHS variable is changed (mutated) to store the **value** of the RHS expression. In other words, the RHS value is *assigned* to the variable.
- This is a *side effect*: the state of the variable has changed.

The use of the equal sign (=) can be misleading.

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 the assignment more obvious.

In addition to the mutation side effect, the assignment operator (=) also produces the right hand side value.

This is occasionally used to perform multiple assignments.

```
x = y = z = 0;              // (x = (y = (z = 0)));
```

> Avoid having more than one side effect per expression statement.

```
printf("%d\n", y = 5);                  // never do this!

printf("%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).

```
if (i = 13) {
  printf("disaster!\n");
}
```

`(i = 13)` assigns 13 to `i` and produces the value 13, so the `if` expression is always true, and it always prints `disaster!`

Pro Tip: some defensive programmers get in the habit of writing `(13 == i)` instead of `(i == 13)`. This causes an error if they accidentally use a single =.

Initialization

C allows a variable definition *without* initialization, but it is bad style.

```
int my_variable = 7;          // initialized

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

Always initialize variables.

In Section 04 we discuss the behaviour of uninitialized variables.

Initialization is not assignment

The `=` used in *initialization* is **not** the assignment operator.

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

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

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

This distinction is especially important in C++.

C allows you to define more than one variable at once.

```
int x = 0, y = 2, z = 3;
```

Most modern style guides discourage this (bad style).

In the following example, x is *uninitialized*.

```
int x, y = 0;
```

More assignment operators

The *compound* addition assignment operator (`+=`) combines the addition and assignment operator (for convenience).

```
x += 2;              // x = x + 2;
```

Additional compound operators include: `-=`, `*=`, `/=`, `%=`.

There are also *increment* and *decrement* operators that increase or decrease a variable by one (either *prefix* or *postfix*).

```
++x;              x++;              // x += 1;
--x;              x--;              // x -= 1;
```

If you follow our *"one side effect per expression"* rule it does not matter if you use prefix or postfix (see next slide).

The language C++ is a pun: one bigger (better) than C.

The *prefix* (++x) and *postfix* (x++) increment operators have different *precedences* within the *order of operations*.

x++ produces the "old" value of x and then increments x.

++x increments x and then produces the "new" value of x.

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

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

Prefix (++x) is usually preferred 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.

```
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.

Global and local variables

Variables are either ***global*** or ***local***.

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

Local variables are defined *inside* of functions.

```
int my_global_variable = 7;

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

Variable Scope

The ***scope*** of a variable is the region of code where it is “accessible” or “visible”.

For global variables, the scope is anywhere *below* its definition.

```
                                // g OUT of scope
int g = 7;
                                // g IN scope
int main(void) {
                                // g IN scope
  printf("%d\n", g);
}
```

We will revisit global scope in Section 06.

Block (local) scope

Local variables have ***block scope***. Their *scope* extends from their definition to the *end of the block* they are defined in.

```
void f(int n) {
                    // b OUT of scope
  if (n > 0) {
                    // b OUT of scope
    int b = 19;
                    // b IN scope
    // ...
  }
                    // b OUT of scope
  // ...
}
```

Variables with the same name can *shadow* other variables from outer scopes, but this is obviously poor style.

The following code defines three **different** variables named n.

```
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
  }
  trace_int(n);   // n => 2
}
```

In older versions of C, all the local variable definitions had to be at the start of the function block (before any statements).

In C99, you may define a local variable anywhere in a block.

Modern programming guides recommend that you define a variable:

- in the narrowest scope possible
- as close to its first use as possible

This improves readability and ensures that when a variable is first used its type and initial value are accessible.

“Impure” functions

Recall that the functional paradigm requires “pure” functions:

- functions **only return values** (no side effects)
- return values **only depend** on argument values

For example, the `noisy_sqr` function is “impure” because it has a side effect (produces output).

```
int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!\n", n);
  return n * n;
}
```

“Impure” functions are sometimes called “procedures” or “routines” to distinguish their behaviour from “pure” functions.

Mutating global variables

A function that mutates a global variable has a *mutation side effect* (which makes it “impure”).

```
int counter = 0;    // global variable

// increment() returns the number of times it has been called
// effects: modifies counter
int increment(void) {
  counter += 1;
  return counter;
}

int main(void) {
  assert(increment() == 1);
  assert(increment() == 2);
}
```

Document any functions with mutation side effects.

Mutating local variables

Mutating a **local** variable does **not** give a function a side effect.

It does not affect state *outside* of the function (global state).

```
int add1(int n) {
  int k = 0;
  k += 1;
  return n + k;
}

int main(void) {
  assert(add1(3) == 4);
}
```

The statement "`k += 1;`" has a side effect (mutation), but it only affects state *inside* of the function (local state).

`add1` has no side effects and is still a "pure" function.

Mutating parameters

Parameters are nearly *indistinguishable* from local variables, and can also be mutated.

```
int add1(int n) {
  n += 1;
  return n;
}

int main(void) {
  int j = 3;
  assert(add1(j) == 4);
  assert(add1(j) == 4);
}
```

This version of `add1` is also a “pure” function (no side effects).

We model how parameters work in Section 04.

Global dependency

A "pure" function only depends on its argument values.

A function that depends on a global *mutable* variable is "impure" even if it has **no** side effects.

```
int n = 10;

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

int main(void) {
  assert(addn(5) == 15);
  n = 100;
  assert(addn(5) == 105);
}
```

Avoiding global mutable variables

Global *mutable* variables are almost always poor style and should be avoided.

Unless otherwise specified, you are **not allowed** to use global *mutable* variables on your assignments.

On the other hand, global *constants* are great style and strongly encouraged.

This topic is revisited in Section 05 in more detail.

Static local variables have the *scope* of a local variable, but the *duration* of a global variable (discussed in Section 04). Their value *persists* between function calls.

```
int increment(void) {
  static int counter = 0;
  counter += 1;
  return counter;
}
```

Like global mutable variables, they are almost always poor style and should be avoided.

They are not allowed in this course.

Text input

Earlier, we learned how to **output** text with `printf`.

We will now learn how to **input** text.

The converse of `printf` is `scanf`, but we are not quite ready to use it (we introduce `scanf` in Section 05).

For now, we will use an alternative method for reading input...

read helper functions

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

```
// 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
```

Note how the side effect of reading input is documented:

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

example: reading input (recursively)

```
// 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]** (**E**nd **O**f **F**ile) button, or type **Ctrl-D**.

example: reading input (continued)

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
```

example: reading input (continued)

One of the great features of the **[I/O TEST]** in `Seashell` is that you can add **multiple** test files.

`test2.in`

```
1 3 3 7
```

`test2.expect`

```
0
```

`test3.in`

```
6 6 6
```

`test3.expect`

```
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

Be careful when reading input in our `Seashell` environment: once you read in a value, **it can no longer be read again**.

Typically, it is best to *store* the read value (*e.g.,* returned by `read_int`) in a variable so it can be referenced multiple times.

For example, consider this **incorrect** partial implementation:

```
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`.

Testing harness

To summarize our function testing strategies:

- **return values**: use `assert`ions (*e.g.*, in `main`)
- **input and output**: [I/O TEST] (`.in` and `.expect` files)

There is an *alternate* approach for testing return values.

To test a function `f` we can write a *dedicated test function* that reads in argument values from input, passes those values to `f`, and then prints out the corresponding return values.

This strategy is known as a ***testing harness***.

example: testing harness for sqr

```
// test_sqr() is an I/O testing harness for sqr
//   it continuously reads in argument values (e.g., n)
//   and then prints out sqr(n)
// effects: reads input
//          produces output

void test_sqr(void) {
  int n = read_int();
  if (n != READ_INT_FAIL) {
    printf("%d\n", sqr(n));
    test_sqr();                // recurse
  }
}

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

example: files for sqr test harness

sqr-test-1.in	sqr-test-1.expect
-3	9
7	49
0	0

This strategy is useful for testing both “pure” and “impure” functions.

On some assignment questions, we may build a testing harness for you (later, you may be expected to build your own).

> For this harness, new tests can be added by editing text files.
>
> In the “real world”, this strategy allows non-coders (*e.g.*, “end users) to develop tests.

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 (`{}`)