# Introduction to Imperative C

**Readings:** CP:AMA 2.1, 4.2–4.5, 5.2, 6, 9.4

- 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

# 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 produce a value, and the value depends **only on the parameter(s)**.

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

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

## example: functional programming paradigm

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

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

```
(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)` produces a **new** list that is sorted, but the original list `lon` does not change.

In this course, the focus is on *imperative* programming and in this section we introduce imperative concepts.

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

Similarly, in imperative programming we use a **sequence of statements** (or "commands") to give *instructions* to the computer.

To highlight the difference, we will consider an imperative special form within Racket (which will seem odd).

Many modern languages are "**multi-paradigm**". Racket was primarily designed as a functional language but also supports imperative language features.

# begin

the `begin` special form evaluates a *sequence* of expressions.

`begin` evaluates each expression but "ignores" all of the values except the last one. `begin` **produces the value** of the **last** expression.

```
(define (mystery)
  (begin
    "four"
    'four
    (+ 2 2)))
```

```
(mystery)
4
```

There are two reasons you rarely see `begin` used in practice...

The first reason is a bit sneaky: In full Racket there is an *implicit* `begin` (similar to implicit `local`).

```
(define (mystery)
  (begin
    "four"
    'four
    (+ 2 2)))
```

```
;; this is equivalent
(define (mystery)
  "four"
  'four
  (+ 2 2))
```

The more obvious reason is that it's rarely "useful" to evaluate an expression and "ignore" the result.

# Side effects

In the *imperative* paradigm, an expression can also have a ***side effect***.

An expression with a side effect does *more* than produce a value: it also changes the *state* of the program (or "the world").

Functions (or programs) can also have side effects.

We have already seen a C function with a side effect: `printf`.

The side effect of `printf` is that it displays "output". In other words, `printf` changes the *state* of the output.

```
printf("Hello, World!\n");
```

The existence of side effects is a significant difference between imperative and functional programming.

**In functional programming there are no side effects.**

Some purists insist that a function with a side effect is no longer a "function" and call it a "procedure" (or a "routine").

The "imperative programming paradigm" is also known as the "procedural programming paradigm".

In this course we are more relaxed and a function can have side effects.

Racket also has a `printf` function very similar to C's `printf`.

In Racket's `printf` the `~a` placeholder works for **a**ll types.

```
(printf "There are ~a lights!\n" "four")
(printf "There are ~a lights!\n" 'four)
(printf "There are ~a ~a!\n" (+ 2 2) "lights")
```

```
There are four lights!
There are four lights!
There are 4 lights!
```

With side effects, `begin` makes more sense.

In fact, the combination of `begin` and `printf` can be quite useful to trace or debug your code.

```
(define (noisy-add x y)
  (printf "HEY! I'M ADDING ~a PLUS ~a!\n" x y)
  (+ x y))
```

```
(define (collatz n)
  (printf "~a " n)
  (cond [(= n 1)   1]
        [(even? n) (collatz (/ n 2))]
        [else      (collatz (+ 1 (* 3 n)))]))
```

# Documentation: side effects

Add an **effects:** section to a contract if there are any side effects.

```
;; (take-headache-pills qty) simulates taking qty pills
;; take-headache-pills: Nat -> Str
;; requires: qty > 0
;; effects: may display a message

(define (take-headache-pills qty)
  (cond [(> qty 3) (printf "Nausea\n")])
  "Headache gone!")
```

For *interfaces*, you should only describe side effects in enough detail so that a client can use your module. Avoid disclosing any implementation-specific side effects.

Some functions are designed to *only* cause a side effect, and not produce a value.

For example, Racket's `printf` produces `#<void>`, which is a special Racket value to represent "nothing"

(in contracts, use `-> Void`).

```
;; (say-hello) displays a friendly message
;; say-hello: Void -> Void
;; effects: displays a message

(define (say-hello)
  (printf "Hello\n"))
```

In C, we used `void` to declare that a function has no parameters.

It is also used to declare that a function returns "nothing".

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

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

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

As mentioned earlier, `main` is the only non-`void` function where the `return` is optional (`main` returns an `int`).

# Expression statements

C's `printf` is not a `void` function.

`printf` returns an `int` representing the number of characters printed.

`printf("hello!\n")` is an **expression** with the value of 7.

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

```
printf("hello!\n");
```

In an expression statement, the **value** of the expression is **ignored**.

```
3 + 4;
7;
printf("hello!\n");
```

The three values of 7 in the above expression statements are never used.

The purpose of an expression statement is to produce a **side effect**.

`Seashell` may give you a warning if you have an expression statement without a side effect (*e.g.*, `3 + 4;`).

# Block statements

A *block* (`{}`) is also known as a ***compound statement***, and contains a **sequence of statements**[*].

A C block (`{}`) is similar to Racket's `begin`: statements are evaluated **in sequence**.

Unlike `begin`, a block (`{}`) does not "produce" a value. This is one reason why `return` is needed.

[*] Blocks can also contain local scope *definitions*, which are not statements.

Earlier, we stated that in imperative programming we use a *sequence of statements*.

We have seen two types of C statements:

- **compound statements** (a sequence of statements)
- **expression statements** (for producing side effects)

The only other type are ***control flow statements***.

# Control flow statements

As the name suggests, *control flow statements* change the "flow" of a program and the order in which other statements are executed.

We have already seen two examples:

- the `return` statement ends the execution of a function to `return` a value.
- the `if` (and `else`) statements execute statements *conditionally*.

We will discuss control flow in more detail and introduce more examples later.

# C terminology (so far)

```c
#include <stdio.h>                  // preprocessor directive

int add1(const int x);              // function declaration

int add1(const int x) {             // function definition
                                    // and block statement

  const int y = x + 1;              // local definition

  printf("add1 called\n");          // expression statement
                                    // (with a side effect)

  2 + 2 == 5;                       // expression statement
                                    // (useless: no side effect)

  return y;                         // control flow statement
}
```

# State

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

We described how a side effect changes the *state* of a program (or "the world"). For example, `printf` changes the state of the output.

The defining characteristic of the *imperative programming paradigm* is to **manipulate state**.

However, we have not yet discussed state.

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

For an example of state, 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 05.

In a program, each variable is in a specific state (corresponding to it's *value*).

In functional programming, each variable has only one possible state.

In imperative programming, each variable can be in one of many possible states.

The value of the variable can change during the execution of the program (hence, the name "variable").

## example: changing state

```
int main(void) {

  int n = 5;

  printf("the value of n is: %d\n", n);

  n = 6;

  printf("the value of n is: %d\n", n);
}
```

```
the value of n is: 5
the value of n is: 6
```

Note that `n` is **not** a `const int`.

# 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).

Ironically, imperative programmers often use the oxymoronic terms "immutable variable" or "constant variable" instead of simply "constant".

# Mutable variables

The `const` keyword is explicitly required to define a constant.

Without it, a variable is *mutable*.

```
const int c = 42;          // constant
int m = 23;                // mutable variable
```

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 allow the compiler to optimize (speed up) your code.

# Assignment Operator

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

```
int m = 5;          // initialization
m = 28;             // assignment operator
```

The assignment operator can be used on `struct`s.

```
struct posn p = {1,2};
struct posn q = {3,4};
p = q;
p.x = 23;
```

The = in an *initialization* is **not** the assignment operator.

Some initialization syntaxes are **invalid** with the assignment operator.

```
struct posn p = {3,4};    // VALID INITIALIZATION

p = {5,7};                // INVALID ASSIGNMENT
p = {.x = 5};             // INVALID ASSIGNMENT
```

This is especially important when we introduce arrays and strings in Section 08.

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 assignment and not symmetric.

# Side effects

Clearly, an assignment operator has a side effect.

A function that mutates a global variable also has a side effect.

```
int count = 0;

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

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

```
1
2
3
```

Even if a function does not have a side effect, its behaviour may depend on other mutable global variables.

```
int n = 10;

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

int main(void) {
  printf("addn(5) = %d\n", addn(5));
  n = 100;
  printf("addn(5) = %d\n", addn(5));
}
```

```
addn(5) = 15
addn(5) = 105
```

In the functional programming paradigm, a function cannot have any side effects and the value it produces depends *only on the parameter(s)*.

In the imperative programming paradigm, a function may have side effects and its behaviour may depend on the *state* of the program.

Testing functions in an imperative program can be challenging.

Racket supports mutation as well. The `set!` special form (pronounced "set bang") "re-defines" the value of an existing identifier. `set!` can even change the **type** of an identifier.

```
(define n 5)
n                ; => 5
(set! n "six")
n                ; => "six"
```

The `!` in `set!` is a Racket convention used to express "**caution**" that the functional paradigm is not being followed.

Racket structures can become mutable by adding the `#:mutable` option to the structure definition.

For each field of a mutable structure, a `set-structname-fieldname!` function is created.

```
(struct posn (x y) #:mutable #:transparent)

(define p (posn 3 4))

(set-posn-x! p 23)
(set-posn-y! p 42)
```

# More assignment operators

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.

```
printf("y is %d\n", y = 5 + (x = 3)); // don't do this!
z = 1 + (z = z + 1);                  // or this!
```

The value produced by the assignment operator is why accidentally using a single = instead of double == for equality is so dangerous!

```
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 programmers get in the habit of writing `(1 == x)` instead of `(x == 1)`. If they accidentally use a single = it causes an error.

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;         // equivalent to x = x + 1;
y += z;         // 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 ***increment*** and ***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 C++ is a pun: one bigger (better) than C.

The *prefix* increment operator (++x) and the *postfix* increment operator (x++) both increment x, they just 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
```

++x is preferred in most circumstances to improve clarity and efficiency.

# Goals of this Section

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

- explain what a side effect
- document a side effect in a contract with *effects* section
- use the new terminology introduced, including: expression statements, control flow statements, compound statements (`{}`)
- use the assignment operators