# 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

# Imperative programming

In English, an **imperative** statement is a *command* to do something: “Make Kraft Dinner!”.

Similarly, an ***imperative program*** is a *sequence of statements* that explain **how** to achieve an outcome.

The imperative programming *paradigm* is to use *control flow* to manipulate ***state***.

In our C execution model, the *state* is the combination of:

- the current *program location*, and
- the current contents of the *memory*.

To properly interpret a program, we must keep track of both the location and all of the memory.

This becomes difficult because the location **changes** during the execution of a program, and the memory also changes.

In direct contrast to the imperative paradigm is the ***declarative paradigm***.

A purely *declarative* program describes **what** outcome is desired, without describing exactly *how* it should be done. A purely declarative program does not specify any control flow, and there is no state manipulation.

The ***functional*** approach we have learned so far is considered *declarative*, but it does not fully achieve the declarative goal.

Functional programming does not use control flow to manipulate state, but it still describes “how” to achieve an outcome.

Consider an *imperative* vs. a *functional* approach for making **Kraft Dinner** *(this example should not be taken too seriously)*.

**Imperative:**

1. Boil 6 cups of water
2. Add pasta to water
3. Cook 7 minutes
4. Drain, place in bowl
5. Add 1 Tbsp. margarine
6. Add 1/3 cup milk
7. Add cheese sauce mix
8. Mix

**Functional:**

```
(mix
 (bowl
  (combine
   (tbsp 1 'margarine)
   (cups 1/3 'milk)
   'cheese-sauce-mix
   (drain
    (cook 7
     (combine
      'pasta
      (boil (cups 6 water)))))))))
```

# begin

Racket is a *multi-paradigm* language that supports functional and imperative features.

For example, the `begin` special form evaluates a *sequence* of expressions. `begin` **produces the value** of the **last** expression:

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

```
(mystery)
4
```

`begin` evaluates each expression in sequence, "ignoring" all of the values except the last one.

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

**1)** In full Racket there is an *implicit* `begin` (similar to implicit `local`):

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

**2)** It's rarely "useful" to evaluate expressions and "ignore" the results.

However, in the *imperative* paradigm, expressions (or functions) can also have ***side effects***.

A side effect is when an expression or function does *more* than produce a value: it also changes the *state* of the program (or "the world").

# Side effects

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

The side effect of `printf` is that it changes the *state* of the output.

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

This is the significant difference between imperative and functional programming: **A purely functional program has no side effects.**

> Some insist that a function with a side effect is no longer a function and use the term "procedure" or "routine" instead.
>
> We will be more relaxed and still call them functions.

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

In Racket, the `"~a"` placeholder will work 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. The combination of `begin` and `printf` is useful when debugging:

```
(define (noisy-add x y)
  (define r (+ x y))
  (printf "noisy-add: ~a + ~a = ~a\n" x y r)
  r)
```

# Postconditions: side effects

Contract *postconditions* **must also list any side effects.**

```
;; take-headache-pills: Int -> String
;;   PRE:  qty > 0
;;   POST: displays "Nausea" if qty > 3
;;         produces "Headache gone!"

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

For interfaces, you should describe the side effects in sufficient detail so that clients can use your module. You should not disclose any implementation details when describing the side effects. Additional *implementation* documentation is often helpful.

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

For example, Racket's `printf` produces `#<void>`, which is a special Racket value to represent "nothing"
(in contracts, use `-> Void`).

In C, `void` is used to declare a function with no parameters. It's also used to declare functions that produce "nothing". In a `void` function, the `return` is optional.

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

# C statements

C's `printf` is not a `void` function. It returns an `int` representing the number of characters printed.

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

When we put a semicolon (`;`) at the end of it:

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

It becomes an ***expression statement.***

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

Blocks can also contain local *declarations*, which are not statements.

A C block (`{}`) is similar to Racket's `begin`: every expression statement is evaluated in sequence, and the value of each expression is "ignored".

Like `begin`, a sequence of expression statements with no side effects is not very useful.

Unlike `begin`, even the last expression statement value in a block is ignored, which is why `return` is needed.

The `return` statement is a special kind of statement known as a ***control flow statement***.

All statements are either expression statements or control flow statements.

## C terminology so far

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

int add1(int x);                    // function declaration

int add1(int x) {                   // function definition
                                    // and start of a block

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

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

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

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

# if statement

The C ***if statement*** is another control flow statement. The syntax is:

```
if (expression) statement
```

where the `statement` is only executed `if` the `expression` is true (non-zero).

```
if (n < 0) printf("n is less than zero\n");
```

The `if` statement does not produce a value. It only controls the flow of execution. This is significantly different than C's ? operator and Racket's `if` and `cond` functions.

The `if` statement only affects whether the *next* statement is executed. To execute more than one statement when the expression is true, braces (`{}`) are used to insert a *compound statement* (a sequence of statements) in place of a single statement.

```
if (n <= 0) {
  printf("n is zero\n");
  printf("or less than zero\n");
}
```

This style is **strongly recommended** *even if there is only one statement*, as it makes the code much easier to follow.

The `if` statement also supports an `else` statement, where a second statement (or block) will be executed when the expression is false.

```
if (expression) {
  statement(s)
} else {
  statement(s)
}
```

`else`s can be combined with more `if`s for multiple conditions:

```
if (expression) {
  statement(s)
} else if (expression) {
  statement(s)
} else if (expression) {
  statement(s)
} else {
  statement(s)
}
```

Instead of having many `else if` statements, some programmers prefer to use the `switch` control flow statement. The behaviour of the `switch` statement is unintuitive and is a source of many bugs. We will not use `switch` in this course (see CP:AMA 5.3 for more details).

Braces are sometimes necessary to avoid a "dangling" `else`:

```
if (y > 0)
  if (y != 5)
    printf("you lose");
else
  printf("you win!");  // when does this print?
```

# Example: if

As we introduce more control flow statements, there may be more than one `return` in a function:

```
int sum(int k) {
  if (k <= 1) {
    return 1;
  } else {
    return k + sum(k-1);
  }
}
```

Note that this is equivalent to:

```
int sum(int k) {
  if (k <= 1) {
    return 1;
  }
  return k + sum(k-1);
}
```

In this course, we will not be using the `goto` control flow statement (CP:AMA 6.4). The `goto` statement is one of the most hated language features in this history of computer science, mostly because it can make *"spaghetti code"* that is hard to understand. Modern opinions have tempered slightly and agree it is useful in *very rare circumstances*.

To use `goto`s, you must also have *labels* (code locations).

```
if (k < 0) {
  goto mylabel;
} else {
  //...
mylabel:
  //...
}
```

# Constants

Up to this point, we have only been working with *constants*:

```
(define x 42)
```

```
const int x = 42;
```

If you've read either textbook (HtDP or CP:AMA), you may have noticed that they refer to `x` as a ***variable***, which seems absurd because something that is variable is the *opposite* of something that is constant.

> It is common practice to refer to `x` as a "constant variable", which sounds nonsensical.

# Mutation

In practice, it is often possible to **change** the value of a *variable*.

This is called ***mutation*** (a fancy word for “change”).

In Racket, the `set!` special form is used to change a variable, and it can even change the *type* of the variable.

```
(define x 6)          ; x => 6
(set! x 28)           ; x => 28
(set! x '(1 2 3))     ; x => '(1 2 3)
(set! x "crazy!")     ; x => "crazy!"
```

The `!` in `set!` is like a giant caution symbol from the Racket authors that `set!` should be avoided and only used in very special circumstances.

For most imperative programmers, mutation is second nature and not given a special name. The term *“Mutation”* is rarely ever used (it does not appear in the CP:AMA text).

It is actually more common to refer to constant data as “immutable data”, especially in object oriented programming.

The term “mutation” was popularized by functional programmers because it accurately reflects how data is changed, and it is also the opposite of immutable.

Functional programmers also use the term because it has a slightly negative connotation.

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

```
int x = 5;
x = 28;
x = 3;
```

Note that `x` is a *variable* because the declaration did not use `const`.

In C, the `const` keyword is used to identify constants.

Although rarely necessary, it is still considered **good style** to use `const` when a "variable" will not change:

- it clearly communicates to humans what your intent is,
- it prevents 'accidental' or unintended mutation, and
- it may allow the compiler to optimize your code.

# Assignment Operator

The assignment operator (=) **is not equals** (==).

Also, the = in a *declaration* is **not** the assignment operator.

In variable declarations, using = is known as ***initialization*** because it is only the *"initial"* value. Some syntaxes used in initialization (declarations) are not valid with the assignment operator:

```
struct posn p = {3,4};

p = {5,7};                  // INVALID

p.x = 5;                    // VALID
p.y = 7;                    // VALID
```

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 not “equals”, and to highlight the nature of assignment.

The assignment operator clearly has a **side effect**:

it changes the *state* of the program by **changing memory**.

The statement:

```
x = x + 1;
```

"reads" the contents of x from memory, and then "overwrites" the contents of that memory location (the *address* of x) with the new value of x.

Note that the x on the "right hand side" is treated differently than the x on the "left hand side".

The *value* of x is used on the RHS.

The *address* of x is used on the LHS.

Because assignment changes memory, the expression on the left hand side of the assignment operator must have an *address*.

```
x = 5;       // VALID
5 = x;       // INVALID

x = x + 1;   // VALID
x + 1 = x;   // INAVLID
```

An expression that has an address, and can appear on the "left" of the assignment operator is called an `lvalue`.

A expression without an address is called an `rvalue`, and is often a temporary result from a calculation.

# Pointer assignment

Consider the following code:

```
int i = 5;
int j = 6;

int *p = &i;
int *q = &j;

p = q;
```

The statement `p = q;` is a *pointer assignment*. It means "`p` now points at what `q` points at". It changes the *value* of `p` to be the value of `q`. In this example, it assigns the *address* of `j` to `p`.

It does not change the value of `i`.

Using the same initial values,

```
int i = 5;
int j = 6;

int *p = &i;
int *q = &j;
```

Now consider the statment:

```
*p = *q;
```

This does **not** change the value of `p`: it changes the value *of what `p` points at*. In this example, it assigns the value of 6 to `i`, *even though `i` was not used in the statement*.

This is an example of ***aliasing***, which is when the same memory address can be accessed from more than one variable.

# Example: Aliasing

```
int i = 2;
struct posn p = {3,4};

int *pi = &i;
int *pp = &p;

printf("i = %d, p = (%d,%d)\n", i, p.x, p.y);

*pi = 7;
pp->x = 8;              // (*pp).x = 8;
pp->y = 9;

printf("i = %d, p = (%d,%d)\n", i, p.x, p.y);
```

```
i = 2, p = (3,4)
i = 7, p = (8,9)
```

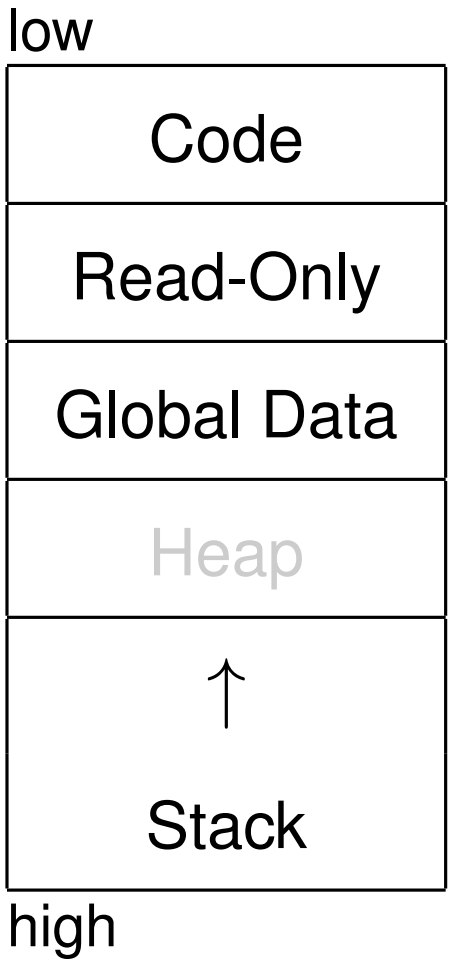
# Global data section

When a variable is declared at the top-level (outside of a function) it is called a ***global variable***, and all global variables are stored in the fourth section of memory known as ***global data***. Like the stack, global data can *change* during execution.

While global *constants* are encouraged, global *variables* should be avoided and only used when absolutely necessary.

`static` global variables (with *modular scope*) are also stored in global data.

Modular scope variables are also discouraged, but are much better than having global variables shared across several modules.


low
Code
Read-Only
Global Data
Heap
Stack
high

```c
const int r = 42;
int g = 15;

int main(void) {

  const int s = 23;

  printf("the address of main is:   %p\n", main); // CODE
  printf("the address of    r is:   %p\n", &r);   // READ-ONLY
  printf("the address of    g is:   %p\n", &g);   // GLOBAL DATA
  printf("the address of    s is:  %p\n",  &s);   // STACK
}
```

```
the address of main is:   0x804d2e0
the address of    r is:   0x805b8c0
the address of    g is:   0x8062080
the address of    s is:  0xbf999600
```

# Static local variables

The `static` keyword is used to give functions and *global* variables *modular* scope.

When the `static` keyword is used on a *local* variable, it makes the variable ***persistent***. A *persistent* local variable keeps its value between calls to the function. Persistent variables are only initialized once (at the start of the program).

```
int add_count(int x, int y) {
  static int count = 0;
  count++;
  printf("add_count has been called %d times\n", count);
  return x+y;
}
```

`static` local variables achieve persistence because they are stored in the *global data* section instead of the stack.

Like global variables and global constants, they are given a fixed address that exists for the entire duration of the program. They do not disappear with a stack frame after a `return`.

```
// running_total(x) returns the sum of all
//   arguments ever passed to running_total
int running_total(int x) {
  static int sum = 0;
  sum += x;
  return sum;
}
```

The *global data* section is sometimes called “global / static” area.

# Uninitialized data

In C, you can *declare* a variable, without *initializing* it.

```
int i;
```

This is considered poor style.

Almost all variables should be initialized.

C will auto-initialize all **global** variables to zero.

Regardless, it is good style to explicitly initialize a global variable even if it is initialized to zero.

```
int g = 0;
```

A *stack* variable that is uninitialized will have an **arbitrary** initial value.

```
void mystery(void) {
  int k;
  printf("the value of k is: %d\n", k);
}
```

RunC will give you a warning if you access an uninitialized variable.

In the example above, the value of `k` will be whatever value happens to be left over from a previous stack frame.

# More assignment

The assignment operator (=) is an operator, so in addition to the mutation side effect, it also produces the value of the expression on the right hand side.

This is occasionally used to perform multiple assignments:

```
x = y = z = 0;
```

Using the value of an assignment operator inside of a larger expression is considered poor style, and should be avoided.

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

The value of 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");
}
```

The above code assigns 1 to `x` and always prints `disaster!`

Some programmers get in the habit of writing `(1 == x)` instead of `(x == 1)`, because if they accidentally use a single =, it's not a valid assignment and will not compile.

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:

`-=`, `*=`, `/=`, `%=`.

Like 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 increases and decrease the value of a variable by one:

```
x++;
x--;
```

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

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
```

# Mutation within functions

Consider the following C program:

```
void inc(int i) {
  i++;
}

int main(void) {
  int x = 5;
  inc(x);
  printf("%d\n", x);   // 5 or 6 ?
}
```

It is important to remember that when `inc` is called, a **copy** of `x` is placed in the stack frame of `inc`, so `x` is unchanged.

The `inc` function is free to change it's own copy of a variable without changing the original.

C uses the ‘’pass by value” convention, where the *value* of the argument is passed to the function. In practice, a **copy** of the value is passed to the function.

The alternative convention is “pass by reference”, where a variable passed to a function can be changed by the function. Some languages support both conventions.

What if we want a C function to change a variable?

In C we can *emulate* “pass by reference” by passing **a pointer** to the variable we want the function to change. This is still considered “pass by value” because we pass the **value** of the pointer.

By passing the *pointer* to `x`, we can change the *value* of `x`:

```
void inc(int *i) {
  *i += 1;
}

int main(void) {
  int x = 5;
  inc(&x);              // Note the &
  printf("%d\n", x);    // NOW it's 6
}
```

Instead of: `*i += 1;` we could have written: `(*i)++;`

The brackets are necessary, otherwise the ++ would have incremented `i`, not `*i`. C is a minefield of these kinds of bugs: the best strategy is to use straightforward code.

A classic example of mutation within a function:

```
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(void) {
  int x = 3;
  int y = 4;
  printf("x = %d, y = %d\n", x, y);
  swap(&x, &y);                          // Note the &
  printf("x = %d, y = %d\n", x, y);
}
```

```
x = 3, y = 4
x = 4, y = 3
```

Like Racket, C functions can only return a single value. Pointer parameters can be used to emulate "returning" more than one value. This is sometimes used to return errors.

```
int do_something_dangerous(int p, bool *error);

int main(void) {
  bool err;
  int result = do_something_dangerous(42, &err);
  if (err) //...
}
```

This is an example where initializing a variable (`err`) is unnecessary.

C library functions often use "invalid" sentinel values such as `-1` or `NULL` to indicate when an error has occurred.

We have seen how structure pointers are often passed to functions for convenience and to reduce unnecessary copying overhead.

However, a structure pointer also allows a function to change the fields of the structure.

```
void scale(struct posn *p, int f) {
  p->x *= f;
  p->y *= f;
}
```

The `const` keyword should be used in the function declaration to indicate that the pointer is for convenience, and that the structure contents will not change.

```
int sqr_distance(const struct posn *p1, const struct posn *p2);
```

The syntax for working with pointers and `const` is awkward:

```
int *p;                         // p can point at any int

const int *p;                   // p can point at any const int

int * const p = &i;             // p always points at i, but
                                // i is not constant

const int * const p = &i;       // p is constant, i is constant
```

The rule is actually: "`const` applies to the type to the left of it, unless it's first, and then it applies to the type to the right of it".

Note: the following are equivalent and a matter of style:

```
const int i = 42;
int const i = 42;
```

# Looping: control flow with mutation

With mutation, we can control flow with a method known as ***looping***.

The first example we will see is:

```
while (expression) statement
```

`while` is similar to `if`: the `statement` will only be executed `if` the `expression` is true. The difference is, `while` will **repeatedly** *"loop back"* and execute the `statement` until the `expression` is false.

This example illustrates how `while` can be used:

```
int i = 100;                    // this will
while (i >= 0) {                // print the
  printf("%d\n", i);            // numbers from
  i--;                          // 100 ... 0
}
```

Like with `if`, you should always use braces (`{}`) for a *compound statement*, even if there is only a single statement.

A simple mistake can cause an “endless loop” or “infinite loop”.

Each of the following is an endless loop:

```
while (i >= 0)                  // missing {}
  printf("%d\n", i);
  i--;

while (i >= 0); {               // extra ;
  printf("%d\n", i);
  i--;
}

while (i = 100) { ... }         // assignment typo

while (1) { ... }               // constant true expression
```

Loops can be “nested” within each other:

```
int i = 5;
while (i >= 0) {
  int j = i;
  while (j >= 0) {
    printf("*");
    j--;
  }
  printf("\n");
  i--;
}
```

```
******
*****
****
***
**
*
```

The `break;` and `continue;` statements can be used within `while` loops for additional control flow.

`break` will immediately terminate the current (innermost) loop.

`continue` will skip over the rest of the statements in the current block (`{}`) and "continue" with the loop.

```
int i = 0; // looking for a magic & lucky number
while (1) {
  i++;
  if (!is_magic(i)) continue;
  if (is_lucky(i)) break;
}
```

`break` and `continue` can always be avoided by restructuring the code. Some C programmers consider them bad style.

The `do` control flow statement is very similar to `while`:

```
do statement while (expression);
```

The difference is that `statement` is always executed *at least* once, and the `expression` is checked at the *end* of the loop.

```
int i = 0;
int err;
do {
  i++;
  err = dangerous_function(i);
} while (!err);
```

# for loops

The final control flow statement we will introduce is `for`, which is often referred to as a "`for` loop".

`for` loops are a "condensed" version of a `while` loop.

The format of a `while` loop is often of the form:

```
init statement
while (expr) {
  statement(s)
  update statement
}
```

which can be re-written as a single `for` statement:

```
for (init; expr; update) { statement(s) }
```

"for" example:

```
i = 100;                    // init
while (i >= 0) {            // expr
  printf("%d\n", i);
  i--;                      // update
}
```

is equivalent to:

```
for (i = 100; i >= 0; i--) {
  printf("%d\n", i);
}

// for (init; expr; update) {...}
```

Most `for` loops follow one of these forms (or "idioms"):

```
// Counting up from 0 to n-1:
for (i = 0; i < n; i++) {...}

// Counting up from 1 to n:
for (i = 1; i <= n; i++) {...}

// Counting down from n-1 to 0:
for (i = n-1; i >= 0; i--) {...}

// Counting down from n to 1:
for (i = n; i > 0; i--) {...}
```

It is a common mistake to be "off by one" (e.g.: using < instead of <=). Sometimes re-writing as a `while` is helpful.

In C99, the “initialization” statement can be a *declaration* instead.

This is very convenient for declaring a variable that only has *block scope* within the `for` loop.

```
for (int i = 100; i >= 0; i--) {
  printf("%d\n", i);
}
```

The equivalent `while` loop would be:

```
{
  int i = 100;
  while (i >= 0) {
    printf("%d\n", i);
    i--;
  }
}
```

You can omit any of the three components of a `for` statement.

If the expression is omitted, it will always be "true".

```
for (; i < 100; i++) {...}  // i already initialized

for (;;) {...}              // endless loop
```

You can use the *comma operator* (`,`) to use more than one expression in the "init" and "update" statements of the `for` loop. See CP:AMA 6.3 for more details.

```
for (i = 1, j = 100; i < j; i++, j--) {...}
```

A `for` loop is not *always* equivalent to a `while` loop.

The only difference is when a `continue` statement is used.

In a `while` loop, `continue` jumps back to the expression. In a `for` loop, the "update" statement will be executed before jumping back to the expression.

# Iteration vs. recursion

Using a loop to solve a problem is called ***iteration***.

*Iteration* is alternative to *recursion* and is much more common in imperative programming.

```c
// recursion
int sum(int k) {
  if (k <= 1) {
    return 1;
  }
  return k + sum(k-1);
}
```

```c
// iteration
int sum(int k) {
  int s = 0;
  for (; k > 0; k--) {
    s += k;
  }
  return s;
}
```

In the beginning, it may be harder to “think iteratively”.

```c
// recursion
int gcd(int n, int m) {
  if (0 == m) {
    return n;
  }
  return gcd(m, n % m);
}
```

```c
// iteration
int gcd(int n, int m) {
  int temp;
  while (m != 0) {
    temp = n;
    n = m;
    m = temp % m;
  }
  return n;
}
```

Sometimes iteration is easier to understand, and *vice versa*.

It depends on the problem.

Even hardcore imperative programmers choose recursion when there are recursive data structures (trees, file systems), or the recursive solution is more elegant.

# Goals of this module

You should have a basic understanding of the difference between imperative and functional programming.

You should understand what a side effect is, and how to document side effects in postconditions.

You should be comfortable with the new C terminology introduced: variables, expression statements, control flow statements, compound statements (`{}`), pass by value, initialization.

You should understand mutation, and how state is changed by overwriting memory.

You should be comfortable with the assignment operators.

You should understand how pointers can be used to change a variable through aliasing and how pointers can be used as parameters to emulate pass by reference.

You should understand the global data section of memory, the persistent behaviour of `static` local variables, and the importance of initializing data.

You should be comfortable with the syntax of control flow statements (`return`, `if`, `while`, `do`, `for`, `break`, `continue`).

You should be comfortable re-writing a recursive function with iteration and *vice versa*.