A Functional Introduction to C

Readings: CP:AMA 2.2–2.4, 2.6–2.8, 3.1, 4.1, 5.1, 9.1, 9.2, 10, 15.2

- the ordering of topics is different in the text
- some portions of the above sections have not been covered yet

A brief history of C

C was developed by Dennis Ritchie in 1969–73 to make the Unix operating system more portable.

It was named “C” because it was a successor to “B”, which was a smaller version of the language BCPL.

C was specifically designed to give programmers “low-level” access to memory (discussed in Section 05 and Section 06).

It was also designed to be easily translatable into “machine code” (discussed in Section 13).

Today, thousands of popular programs, and portions of all of the popular operating systems (Linux, Windows, Mac OSX, iOS, Android) are written in C.

There are a few different versions of the C standard. In this course, the C99 standard is used.

From Racket to C

To ease the transition from Racket, we will first learn to write some simple C functions using a functional paradigm.

This allows us to become familiar with the C ***syntax*** without introducing too many new concepts.

in Section 04 imperative programming concepts are introduced.

Read your assignments carefully: you may not be able to "jump ahead" and start programming with imperative style (*e.g.*, with mutable variables or loops).

Comments

```
; Racket comment

#| Racket multi-line
   comment |#
```

```
// C comment

/* C multi-line
   comment */
```

In C, any text on a line after `// is a comment.`

Any text between `/* and */` is also a comment, and can extend over multiple lines. This is useful for commenting out a large section of code.

C's multi-line comment cannot be "nested":

```
/* this /* nested comment is an */ error */
```

Defining a variable (constant)

```
; Racket Constant:
(define my-number 42)
```

```
// C Constant:
const int my_number = 42;
```

In C, a semicolon (`;`) is used to indicate the end of a variable definition.

The `const` keyword indicates the variable is immutable (`const`ant).

The `int` keyword **declares** that `my_number` is an `int`eger.

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” (introduced in Section 04) and “constants”.

In this Section, all variables are `const`ants.

C identifiers ("names") are more limited than in Racket. They must start with a letter, and can only contain letters, underscores and numbers (`my_number` instead of `my-number`).

We use `underscore_style`, but `camelCaseStyle` is a popular alternative. Consistency is more important than the choice of style.

C identifiers can start with an underscore (`_name`) but their use is restricted. It's best to avoid them.

Typing

Racket and C handle types very differently.

- Racket uses ***dynamic typing***: the type of an identifier is determined **while** the program is running.

```
; dtype can be a number or a string
(define dtype (cond [(>= x 0) 42]
                    [else "invalid"]))
```

- C uses ***static typing***: the type of an identifier must be known **before** the program is run. The type is *declared* in the definition and cannot change.

```
// stype is always a const int
const int stype = 42;
```

C Types

To start, we will only work with **`int`egers** in C.

More types will be introduced soon.

Because C uses static typing, there are no functions equivalent to the Racket type-checking functions (*e.g.,* `integer?` and `string?`).

Initialization

```
// C Constant:
const int my_number = 42;
```

The "= 42" portion of the above definition is called ***initialization***.

Constants **must** be initialized.

Expressions

C uses the more familiar *infix* algebraic notation (3 + 3) instead of the *prefix* notation used in Racket (+ 3 3).

```
; Racket Expressions:
(define six (+ 3 3))
(define seven (+ 1 (* 3 2)))
(define eight (* (+ 1 3) 2))
```

```
// C Expressions:
const int six = 3 + 3;
const int seven = 1 + 3 * 2;
const int eight = (1 + 3) * 2;
```

With infix notation, parentheses are often necessary to control the **order of operations**.

Use parentheses to clarify the order of operations in an expression.

C operators

C distinguishes between ***operators*** (*e.g.,* `+`, `-`, `*`, `/`) and functions.

The C order of operations (*"operator precedence rules"*) are consistent with mathematics: multiplicative operators (`*`, `/`) have higher precedence than additive operators (`+`, `-`).

In C there are also *non-mathematical operators* (*e.g.,* for working with data) and almost 50 operators in total.

As the course progresses more operators are introduced.

The full order of operations is quite complicated (see CP:AMA Appendix A).

In C, each operator is either *left* or *right* associative to further clarify any ambiguity (see CP:AMA 4.1).

The multiplication operators are *left*-associative:

`4 * 5 / 2` becomes `(4 * 5) / 2`.

The distinction in this particular example is important in C.

The / operator

When working with integers, the C division operator (`/`) truncates (rounds toward zero) any intermediate values, and behaves the same as the Racket `quotient` function.

```
const int a = (4 * 5) / 2;  // 10

const int b = 4 * (5 / 2);  // 8 !!

const int c = -5 / 2;       // -2 !!
```

Remember, use parentheses to clarify the order of operations.

The % operator

The C remainder operator (%) (also known as the **modulo** operator) behaves the same as the `remainder` function in Racket.

```
const int a = 21 % 2;  // 1

const int b = 21 % 3;  // 0

const int c = 13 % 5;  // 3
```

In this course, avoid using % with negative integers
(see CP:AMA 4.1 for more details).

Function terminology

In this course, we use a function terminology that is more common amongst imperative programmers.

In our "functional" CS 135 terminology, we ***apply*** a function. A function ***consumes*** arguments and ***produces*** a value.

In our new terminology, we ***call*** a function. A function is ***passed*** arguments and ***returns*** a value.

Function definitions

```racket
; Racket function:
; my-sqr: Int -> Int
(define (my-sqr x)
  (* x x))
```

```c
// C function:
int my_sqr(const int x) {
  return x * x;
}
```

In C, braces (`{}`) indicate the beginning and end of the function body (or the function ***block***).

The `return` keyword is placed before the expression to be returned.

Because C is *statically typed*, the function ***return type*** and the type of each parameter is **required**.

The return type of `my_sqr` is an `int`, and it has a `const int` parameter `x`.

In the previous example, the parameter was a `const int`, but the *return type* was just an `int`.

```
int my_sqr(const int x) {
  return x * x;
}
```

Most versions of C will allow a `const int` return type:

```
const int my_sqr(const int x) { ... }
```

However, the `const` in the return type is **ignored**, so we will not use it.

We will discuss using `const` parameters and return types in more detail later.

```c
int my_add(const int x, const int y) {
  return x + y;
}

int my_num(void) {
  return my_add(40, 2);
}
```

Parameters are separated by a comma (`,`) in the function definition and calling syntax.

The `void` keyword is used to indicate a function has no parameters.

If you omit the `void` in a parameterless function definition:

```
int my_num() {
  // ...
}
```

C will allow it. This is because `( )` is used in a special syntax to indicate an "unknown" or "arbitrary" number of parameters (that is beyond the scope of this course).

It is better style to use `void` to clearly communicate and enforce that there are no parameters.

```
int my_num(void) {
  // ...
}
```

Function documentation

```
; (my-sqr x) squares x
; my-sqr: Int -> Int

(define (my-sqr x)
  (* x x))
```

```
// my_sqr(x) squares x

int my_sqr(const int x) {
  return x * x;
}
```

In C, the contract *types* are part of the function definition. No additional contract documentation is necessary. However, you should still add a **requires** section if appropriate.

```
// some_function(n) ....
// requires: n > 0

int some_function(const int n) {
  // ...
}
```

In Racket, the contract types are only documentation. It is possible to violate the contract, which may cause a “type” runtime error.

```
;; Racket:
(my-sqr "hello")    ; => runtime error !!!
```

In statically typed languages like C, it is impossible to violate the contract *type*, and “type” *runtime* errors do not exist.

```
// C:
my_sqr("hello")     // => will not run !!!
```

Getting started

At this point you are probably eager to write your own functions in C.

Unfortunately, we do not have an environment similar to `DrRacket`'s interactions window to evaluate expressions and informally test functions.

Next, we will demonstrate how to run and test a simple C program that can display information.

Entry point

Typically, a program is "run" (or "launched") by an Operating System (OS) through a shell or another program such as `DrRacket`.

The OS needs to know where to **start** running the program. This is known as the ***entry point***.

In many interpreted languages (including Racket), the entry point is simply the **top** of the file you are "running".

In C, the entry point is a special function named `main`.

Every C program must have one (and only one) `main` function.

main

`main` has no parameters[†] and an `int` return type.

```
int main(void) {
  //...
}
```

The return value communicates to the OS if the program is successful (zero) or if an error occurred (non-zero).

The return value is *optional*, and defaults to zero.

`main` does **not** require any documentation (*i.e.,* a purpose).

[†] `main` has *optional* parameters (discussed in Section 13).

Hello, World

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

```c
// hello.c
// My first C program

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

This may not work: we need to "require" the module that contains the `printf` function.

`Seashell` may be kind and require it for us, but it will give a warning message.

`printf` is part of the C `stdio` (**st**an**d**ard **i**/**o**) module. We will introduce C modules later, but for now we will just use the C version of Racket's `require` without any discussion:

```
// hello2.c
// My second C program

#include <stdio.h>   // <-- require stdio module

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

```
Hello, World
```

In the slides, we typically omit `#includes` to save space.

```c
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.

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

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

The first parameter of `printf` must be a `"string"`. Until we discuss strings in Section 08, this is the only place you are allowed to use strings.

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

```
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 parameter. There can be multiple placeholders, each requiring an additional parameter.

```
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 "**d**ecimal format").

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!
```

We will discuss I/O in more detail in Section 07

Many computer languages have a `printf` function and use the same placeholder syntax as C. The placeholders are also known as **f**ormat 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 sufficent.

Boolean expressions

In C, “false” is represented by zero (`0`) and “true” is represented by one (`1`). Any *non-zero* value is also considered “true”.

The ***equality*** **operator** in C is == (note the **double** equals).

The ***not equal*** **operator** is `!=`.

The value of `(3 == 3)` is `1` (“true”).

The value of `(2 == 3)` is `0` (“false”).

The value of `(2 != 3)` is `1` (“true”).

Always use a *double* == for equality, not a *single* =.

The accidental use of a *single* = instead of a *double* == for equality is one of the most common programming mistakes in C.

This can be a serious bug (we revisit this in Section 04).

It is such a serious concern that it warrants an extra slide as a reminder.

The ***not***, ***and*** and ***or*** operators are respectively `!`, `&&` and `||`.

The value of `!(3 == 3)` is `0`.

The value of `(3 == 3) && (2 == 3)` is `0`.

The value of `(3 == 3) && !(2 == 3)` is `1`.

The value of `(3 == 3) || (2 == 3)` is `1`.

The value of `(2 && 3 || 0)` is `1`.

Similar to Racket, C ***short-circuits*** and stops evaluating an expression when the value is known.

`(a != 0) && (b / a == 2)` does not produce an error if `a` is `0`.

A common mistake is to use a single `&` or `|` instead of `&&` or `||`. These operators have a different meaning.

Comparison operators

The operators <, <=, > and >= behave exactly as you would expect.

The value of `(2 < 3)` is `1`.

The value of `(2 >= 3)` is `0`.

The value of `!(a < b)` is equivalent to `(a >= b)`.

It is always a good idea to add parentheses to make your expressions clear.

> has higher precedence than ==, so the expression `1 == 3 > 0` is equivalent to `1 == (3 > 0)`, but it could easily confuse some readers.

Conditionals

There is no direct C equivalent to Racket's `cond` special form.

We can use C's `if` *statement* to write a function that has conditional behaviour.

```
int my_abs(const int n) {
  if (n < 0) {
    return -n;
  } else {
    return n;
  }
}
```

There can be more than one `return` in a function.

The `cond` special form consumes a sequence of question and answer pairs (questions are Boolean expressions).

Racket functions that have the following `cond` behaviour can be re-written in C using `if`, `else if` and `else`:

```
(define (my-function ...)
  (cond
    [q1   a1]
    [q2   a2]
    [else a3]))
```

```
int my_function(...) {
  if (q1) {
    return a1;
  } else if (q2) {
    return a2;
  } else {
    return a3;
  }
}
```

example: collatz

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

```
int collatz(const int n) {

  if (n == 1) {
    return 1;
  } else if (n % 2 == 0) {
    return collatz(n / 2);
  } else {
    return collatz(3*n + 1);
  }
}
```

Recursion in C behaves the same as in Racket.

`cond` produces a value and can be used inside of an expression:

```
(+ y (cond [(< x 0) -x]
           [else     x]))
```

C's `if` *statement* does not produce a value: it only controls the "flow of execution" and cannot be similarly used within an expression.

We revisit `if` in Section 05 after we understand how "statements" differ from expressions. For now, only use `if` as we have demonstrated:

```
if (q1) {
  return a1;
} else if (q2) {
  return a2;
} else {
  return a3;
}
```

Unlike C's `if` *statement*, the C `?:` *operator* does produce a value and behaves the same as Racket's `if` special form.

```
;; Racket's if special form:
(define c (if q a b))
(define abs-v (if (>= v 0) v (- v)))
(define max-ab (if (> a b) a b))
```

The value of (`q ? a : b`) is `a` if `q` is true (non-zero), and `b` otherwise.

```
// C's ?: operator
const int c = q ? a : b;
const int abs_v = (v >= 0) ? v : -v;
const int max_ab = (a > b) ? a : b;
```

Function placement

```
int accsum(const int k, const int acc) {
  if (k == 0) {
    return acc;
  } else {
    return accsum(k - 1, k + acc);
  }
}

int sum(const int k) {
  return accsum(k, 0);
}
```

This example illustrates the importance of ordering (placing) C functions in a file. Because `sum` calls `accsum`, we placed `accsum` *before* (or *"above"*) `sum` in the code.

In C, a function (or any *identifier*) must be **declared** before (or *"above"*) any expression it appears in.

A ***declaration*** communicates to C the **type** of an identifier. Function declarations include both the return type and the parameter type(s).

In C, there is a subtle difference between a **definition** and a **declaration**.

Declaration *vs.* definition

- A declaration only specifies the *type* of an identifier.
- A definition instructs C to *"create"* the identifier.

However, a definition must also specify the type of the identifier, so

a definition also includes a declaration.

An identifier can be declared multiple times, but only defined once.

Unfortunately, not all computer languages and reference manuals use these terms consistently.

A function declaration is a header without a body, with just a semicolon (`;`) instead of a code block.

example: function declaration

```
int accsum(const int k, const int acc);     // DECLARATION

int sum(const int k) {
  return accsum(k, 0);                      // this is now ok
}

int accsum(const int k, const int acc) {    // DEFINITION
  if (k == 0) {
    return acc;
  } else {
    return accsum(k-1, k + acc);
  }
}
```

C ignores the parameter names in a function declaration (it is only interested in the parameter *types*).

The parameter names can be different from the definition or not present at all.

```
int accsum(const int, const int ignored);
```

It is good practice to include the correct parameter names in the declaration to aid communication.

A variable declaration starts with the `extern` keyword and is not initialized.

example: variable declaration

```
extern const int fun_number;        // variable DECLARATION

int is_fun(const int n) {
  if (n == fun_number) {            // this is now ok
    return 1;
  } else {
    return 0;
  }
}

const int fun_number = 4010;        // variable DEFINITION
```

Variable declarations are uncommon.

C modules

Unlike Racket, there are no “built-in” functions in C.

Fortunately, C provides several ***standard modules*** (also known as *libraries*) with many useful functions.

We have already seen how the `stdio` standard module provides the `printf` function.

We will use several *standard modules* throughout this course, including `assert`, `stdbool`, `limits`, `string` and `stdlib`.

To **require** the `stdio` module, we wrote:

```
#include <stdio.h>
```

The angle brackets (<>) specify that the module is one of the *standard modules*.

We will see the meaning of the `.h` shortly.

To **require** a "regular" module (*i.e.,* one you have written) the syntax is slightly different. Quotes (`""`) are used instead of angle brackets:

```
#include "mymodule.h"
```

The different notations (`<stdio.h>` *vs.* `"mymodule.h"`) tell C *where* to look for the module interface.

Creating a module in C

To create a module in C, we place the **interface** and the **implementation** into separate files.

In the ***interface (.h) file*** we place **declarations** for the functions (and variables) that the module **provides**. We also place the documentation for the client in the interface file.

In the ***implementation (.c) file*** we place all of the **definitions**.

```c
// mymodule.h [INTERFACE]

extern const int my_constant;

// my_sqr(x) squares x
int my_sqr (const int x);
```

```c
// mymodule.c [IMPLEMENTATION]

const int my_constant = 7;

// see mymodule.h for details
int my_sqr(const int x) {
  return x * x;
}
```

```c
////////////////////////////////////////////////////////////
// client.c [CLIENT]

#include <stdio.h>
#include "mymodule.h"

int main(void) {
  printf("my_constant squared is %d\n", my_sqr(my_constant));
}
```

```
my_constant squared is 49
```

#include

The behaviour of C's `#include` is *very* different than Racket's `require` special form.

`#include` is what is known as a preprocessor ***directive***. Directives can modify a source file just before it is run.

`#include` *"inserts"* the contents of the interface (`.h`) file directly into the client source file. This makes all of the provided declarations available to the client.

In Appendix A.4 we discuss `#include` and C modules in more detail.

One use of `#include` that may not be intuitive is that a module **implementation (`.c`)** file can `#include` **its own interface**.

This is actually very good practice as it ensures there are no discrepancies between the interface and the implementation.

```
// mymodule.c [IMPLEMENTATION]

#include "mymodule.h"                  // good practice

const int my_constant = 7;

// see mymodule.h for details
int my_sqr(const int x) {
  return x * x;
}
```

The CP:AMA textbook frequently uses the `#define` directive.

In its simplest form it performs a *search & replace*.

```
// replace every occurrence of MY_NUMBER with 42
#define MY_NUMBER 42

int my_add(const int n) {
  return n + MY_NUMBER;
}
```

In C99, it is better style to define a variable (constant), but you will still see `#define` in the "real world".

`#define` can also be used to define *macros*, which are hard to debug, considered poor style, and should be avoided.

Scope

Local scope behaviour is the same in both C and Racket. Each C block (`{}`) creates a new local environment similar to `local` in Racket. If an identifier *shadows* another with the same name, the innermost ("most local") instance is used.

However, Racket and C have different global scoping behaviour.

In Racket, each global identifier has **module** scope, unless it is explicitly `provide`d.

In C, each global identifier has **program** scope by default.

To declare that a C global function or variable has **module** scope, the `static` keyword is used. `static` indicates that the identifier is "restricted" to the current file.

```
int pfunction (void) { ... }          // program Scope
static int mfunction (void) {...}     // module Scope

const int pvariable = 42;             // program Scope
static const int mvariable = 23;      // module Scope
```

In Appendix A.4 we discuss this in more detail.

`private` would have been a better choice than `static`.

The C keyword `static` is **not** related to *static typing*.

One more scope-related difference between the languages is that in C you cannot have any **top-level expressions**.

The initialization of a global variable can contain a simple expression, but it cannot contain a function call.

```
// C top level:
const int a = 3 * 3;          // VALID

const int b = my_sqr(3);      // INVALID
3 * 3;                        // INVALID
my_sqr(3);                    // INVALID
```

Also, in C99 you cannot define a local function within another function.

assert standard module

A useful standard module is `assert`, which provides the `assert` function.

`assert(e)` **stops** the program and displays a message if the expression `e` is false. If `e` is true, nothing happens.

`assert` is especially useful for verifying function requirements. You should `assert` your requirements **when feasible**.

```
#include <assert.h>

// requires: n > 0
int some_function(const int n) {
  assert(n > 0);
  //...
}
```

example: test client

As mentioned previously, it is good practice to create a test client for each module you create. `assert` can be used in a manner similar to Racket's `check-expect`.

```
// test-mymodule.c: testing client for mymodule

#include <assert.h>
#include "mymodule.h"

int main(void) {
  assert(my_sqr(0) == 0);
  assert(my_sqr(-1) == 1);
  assert(my_sqr(1) == 1);
  assert(my_sqr(2) == 4);
  assert(my_sqr(3) == 9);
}
```

bool type

Boolean types are not “built-in” to C, but are available through the `stdbool` standard module.

`stdbool` provides a new `bool` *type* (can only be 0 or 1) and defines the constants `true` (1) and `false` (0).

```
#include <stdbool.h>

const bool is_cool = true;

bool is_even(const int n) {
  return (n % 2) == 0;
}
```

Symbol type

In C, there is no equivalent to the Racket `'symbol` type. To achieve similar behaviour in C, you can define a unique integer for each "symbol". It is common to use an alternative naming convention (such as `ALL_CAPS`).

```
; Racket Symbols:

(define genre1 'pop)
(define genre2 'rock)
```

```
// C's alternative:
const int POP = 1;
const int ROCK = 2;

const int genre1 = POP;
const int genre2 = ROCK;
```

In C, there are ***enumerations*** (`enum`, CP:AMA 16.5) which allow you to create your own `enum` types and help to facilitate defining constants with unique integer values.

Enumerations are an example of a C language feature that we do *not* introduce in this course.

After this course, we would expect you to be able to read about `enum`s in a C reference and understand how to use them.

If you would like to learn more about C or use it professionally, we recommend reading through all of CP:AMA *after* this course is over.

Floating point types

The C `float` (floating point) type can represent real (non-integer) values.

```
const float pi = 3.14159;
const float avagadro = 6.022e23;   // 6.022*10^23
```

Unfortunately, `float`s are susceptible to precision errors.

C's `float` type is similar to **inexact numbers** in Racket (which appear with an `#i` prefix in the teaching languages):

```
(sqrt 2)          ; => #i1.4142135623730951
(sqr (sqrt 2))    ; => #i2.0000000000000004
```

example 1: inexact floats

```
const float penny = 0.01;

float add_pennies(const int n) {
  if (n == 0) {
    return 0;
  } else {
    return penny + add_pennies(n-1);
  }
}

int main(void) {
  const float dollar = add_pennies(100);
  printf("the value of one dollar is: %f\n", dollar);
}
```

```
the value of one dollar is: 0.999999
```

The `printf` placeholder to display a `float` is `"%f"`.

example 2: inexact floats

```c
int main(void) {
  const float bil = 1000000000;
  const float bil_and_one = bil + 1;

  printf("a float billion is:     %f\n", bil);
  printf("a float billion + 1 is: %f\n", bil_and_one);
}
```

```
a float billion is:     1000000000.000000
a float billion + 1 is: 1000000000.000000
```

In the previous two examples, we highlighted the precision errors that can occur with the `float` type.

C also has a `double` type that is still inexact but has significantly better precision.

Just as we use `check-within` with inexact numbers in Racket, we can use a similar technique for testing in floating point numbers C.

Assuming that the precision of a `double` is perfect or "good enough" can be a serious mistake and introduce errors.

Unless you are explicitly told to use a `float` or `double`, you should not use them in this course.

Structures

Structures (*compound data*) in C are similar to structures in Racket:

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

(define p (posn 3 4))
```

```
struct posn {
  int x;
  int y;
};

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

Racket generates functions (*e.g.,* `posn-x`) when you *define* a structure, but C does not.

In C, structures are **declared** (not *defined*).

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

(define p (posn 3 4))
```

```
struct posn {
  int x;
  int y;
};

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

Because C is statically typed, structure declarations require the *type* of each field.

The structure *type* includes the keyword "`struct`". For example, the type is "`struct posn`", not just "`posn`". This can be seen in the definition of `p` above.

Do not forget the last semicolon (`;`) in the structure declaration.

```
(define p (posn 3 4))
```

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

```
(define a (posn-x p))
(define b (posn-y p))
```

```
const int a = p.x;
const int b = p.y;
```

Instead of *selector functions*, C has a ***structure operator*** (`.`) which "selects" the value of the requested field.

C99 supports an alternative way to initialize structures:

```
const struct posn p = { .y = 4, .x = 3};
```

This prevents you from having to remember the “order” of the fields in the initialization.

Any omitted fields are automatically zero, which can be useful if there are many fields:

```
const struct posn p = {.x = 3}; //.y = 0
```

The *equality* operator (==) **does not work with structures**. You have to define your own equality function.

```
bool posn_equal (const struct posn a, const struct posn b) {
  return (a.x == b.x) && (a.y == b.y);
}
```

Also, `printf` only works with elementary types. You have to print each field of a structure individually:

```
const struct posn p = {3,4};
printf("The value of p is (%d,%d)\n", p.x, p.y);
```

```
The value of p is (3,4)
```

The braces ({}) are **part of the initialization syntax** and should not be used inside of an expression.

```
struct posn scale(const struct posn p, const int f) {
  return {p.x * f, p.y * f};  // INVALID
}
```

In the past, students more familiar with Racket and functional-style structures have found this restriction frustrating.

To avoid this, simply define new constants as required.

```
struct posn scale(const struct posn p, const int f) {
  const struct posn r = {p.x * f, p.y * f};
  return r;
}
```

To make a structure accessible (and "transparent") to clients, place the structure declaration in the *interface file* (`.h`).

```c
// posn.c

#include "posn.h"

int f(const struct posn p) {
  //...
}
```

```c
// posn.h

struct posn {
  int x;
  int y;
};

int f(const struct posn p);
```

This is another reason why every implementation file (`module.c`) should include its own interface file (`module.h`).

We discuss opaque C structures later.

Goals of this Section

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

- demonstrate the use of the C syntax and terminology introduced
- define constants and write expressions in C
- re-write a simple Racket function in C (and vice-versa)
- use the C operators introduced in this module
  (including `%` `==` `!=` `>=` `&&` `||` `.`)
- explain the difference between a declaration and a definition
- explain the differences between local, module and program scope and demonstrate how `static` and `extern` are used

- write modules in C with implementation and interface files
- explain the significance of the the `main` function in C
- perform basic testing and I/O in C using `assert` and `printf`
- use structures in C
- provide the required documentation for C code