Functional 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

From Racket to C

To ease the transition from Racket to C, we will first introduce C in a very Racket-like *functional* style.

In later modules, we will introduce more C language features and learn a more traditional approach to programming in C.

There are a few different versions of the C standard. In this course, we will be using the C99 standard.

Comments

```
; Racket comment                // C comment

#| Racket multi-line            /* C multi-line
   comment |#                      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 lot of code.

Typing

Racket and C handle types very differently. In C, all types have to be declared **before** the program is run. In Racket, types are determined **while** the program is running.

In Racket, there are functions to check types (e.g.: `integer?`, `string?`), but in C all types are known in advance, so there are no equivalent functions.

Racket uses *dynamic typing*, whereas C uses *static typing*. Unfortunately, C uses the word *static* in many different ways, so to avoid confusion we will not use it here.

Constant declaration

```
; Racket Constant:               // C Constant:
(define lucky-num 42)            const int lucky_num = 42;
```

In C terminology, constants are ***declared*** (in Racket they are *defined*).

In Racket, definitions and expressions begin and end with parentheses (brackets). In C, a semicolon (`;`) indicates the end of a declaration.

As previously mentioned, the *type* of the constant (`int`) is *required* in the declaration.

For now, we will only work with `int`egers in C.

```
; Racket Constant:             // C Constant:
(define lucky-num 42)          const int lucky_num = 42;
```

C identifiers (“names”) are much more limited than in Racket. They must start with a letter, and can only have letters, underscores and numbers (`lucky_num` instead of `lucky-num`).

We will use `underscore_style`, but `camelCaseStyle` is a popular alternative. Consistency is the most important.

C Identifiers can start with an underscore (`_name`), but C restricts their use, so it’s best to avoid them.

Symbols

In C, there is no equivalent to the Racket `'symbol` type. To achieve similar behaviour in C, you must declare a unique constant integer for each “symbol”. It is common to use `ALL_CAPS` for these constants.

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

(define genre1 'pop)             const int genre1 = POP;
(define genre2 'rock)            const int genre2 = ROCK;
```

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

This is an example of a C language feature that we will *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.

Expressions

```
; 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;
```

In C, there is a distinction between ***operators*** (e.g.: +, -, *) and *functions*. In Racket, all operators were functions.

C uses *infix* notation, where parentheses control the **order of operations**. When in doubt, **use parentheses for clarity**.

The full order of operations (also known the *precedence rules*) in C is quite complicated, as there are many non-mathematical operators (see CP:AMA Appendix A).

The / and % operators

Warning: the C integer division operator (`/`) will truncate (round toward zero) any intermediate values. It behaves the same as the `quotient` function in Racket.

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

```
const int a = 4 * 5 / 2;    // a => 10
const int b = 4 * (5 / 2);  // b => 8 !!
const int c = -5 / 2;       // c => -2 !!
const int d = 13 % 5;       // d => 3
```

Avoid using `%` with negative integers (CP:AMA 4.1).

Function terminology

Racket and C have different terminologies for working with functions.

In Racket terminology, we ***apply*** a function. A function ***consumes*** arguments and ***produces*** a value.

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

Function definitions

```
; Racket function:                // C function:
; my-add: Int Int -> Int          int my_add(int x, int y) {
(define (my-add x y)                return x + y;
  (+ x y))                        }
```

In Racket, the function contract types are provided in the documentation to aid communication.

In C, the function definition requires the ***return type***, and the type of each parameter. The syntax is:

```
return_type func_name(param_type param_name, ...) {...}
```

The return type of `my_add` is an `int` and it has two `int` parameters (`x` and `y`).

```
; Racket function:              // C function:
; my-add: Int Int -> Int        int my_add(int x, int y) {
(define (my-add x y)              return x + y;
  (+ x y))                      }
```

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

The `return` keyword must appear before the expression to be returned, which is followed by a semicolon (`;`).

The calling syntax in C is consistent with traditional mathematics:

`my_add(2,3)` returns 5, and

`my_add(my_add(2,3),4)` returns 9.

Function documentation

```
; my-sqr: Int -> Int
;   PRE:  true
;   POST: produces value >=0
; (my-sqr x) squares x
(define (my-sqr x)
  (* x x))
```

```
// my_sqr(x) squares x
//   PRE:  true
//   POST: return value >= 0
int my_sqr(int x) {
  return x * x;
}
```

In C, the contract types are part of the function definition, so we will re-arrange the documentation and place the purpose at the top.

Scope

```
; Racket:
(define a 2)

(define (f x)
  (define b (+ a x))
  (define c (* b b))
  (+ 1 c))
```

```
// C:
const int a = 2;

int f(int x) {
  const int b = a + x;
  const int c = b * b;
  return 1 + c;
}
```

The ***scope*** behaviour for C identifiers is consistent with the behaviour in Racket. Above, the constant `a` has ***global scope***. The parameter `x` and the constants `b` and `c` are ***local*** to the function `f`.

Local declarations have ***block scope*** and are only valid within the block they are declared in.

```
const int a = 2;
const int b = 3;

int f(int b) {
  const int a = 4;
  {
    const int a = 5;
    return a + b;
  }
}
```

While you should generally *avoid* re-using the same identifiers (names), C will use the *innermost* (or most local) context.

In C, each block (`{}`) creates a new local environment (similar to `local` in Racket).

In the above example, `f(10)` returns `15`.

Additional C restrictions

In C, you cannot define a “local” function within another function. All functions are *global*.

In C, you cannot have any *top-level* expressions. You can only have top-level declarations and definitions. As we have seen, you can have an expression *within* a top-level declaration, but it cannot contain a function call.

```
// C top level:
const int a = 2 + 3;          // VALID
const int b = my_add(2,3);    // INVALID
2 + 3;                        // INVALID
my_add(2,3);                  // INVALID
```

Booleans

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) and 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”).

It is a common mistake to use a *single* = for equality instead of a *double* ==.

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 will revisit this in a future module).

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 will ***short-circuit*** and stop evaluating an expression when the value is known.

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

Using a single & or | is also a common mistake.

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.

The expression: `1 == 3 > 0` is equivalent to `1 == (3 > 0)`, but it could easily confuse some readers.

The ?: operator

There is no `cond` function in C, but there is an operator that behaves the same as the Racket `if` function.

```
; Racket if function:
(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), `b` otherwise.

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

Just as the following usage of `cond`:

```
(cond [q1 a1]
      [q2 a2]
      [else a3])
```

can be replaced with: `(if q1 a1 (if q2 a2 a3))`,

we can “nest” the C `?:` operator:

`q1 ? a1 : (q2 ? a2 : a3)`.

Later, we will see an alternative strategy for working with conditionals in C.

You should avoid using the `?:` excessively (it makes code hard to follow).

Recursion

```
; Recursion in Racket
(define (sum k)
  (if (<= k 1) 1
    (+ k (sum (- k 1)))))
```

```
// Recursion in C
int sum(int k) {
  return (k <= 1) ? 1 :
         k + sum(k-1);
}
```

Recursion in C behaves as you would expect.

In the next module, we will have a better understanding of how recursion works in C.

Example: accumulative recursion

```racket
; Accumulative Racket
(define (accsum k acc)
  (if (<= k 0) acc
    (accsum (- k 1)
            (+ k acc))))

(define (sum k)
  (accsum k 0))
```

```c
// Accumulative C
int accsum(int k, int acc) {
  return (k <= 0) ? acc :
         accsum(k-1, k + acc);
}

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

In this C example, the `accsum` definition must appear *before* `sum`.

C needs to know the return type and the parameter types of a function before it can appear in the code.

To avoid this problem, a function can be *declared* before it is *defined*.

Functions: declarations vs. definitions

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

The syntax for a function *declaration* is the same as a definition, but with the function body (code block) replaced by a semicolon (`;`).

```
// this is a function *declaration* (no {}'s)
int accsum(int k, int acc);

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

// this is the *definition* (with code body)
int accsum(int k, int acc) {
  return (k <= 0) ? acc :
         accsum(k-1, k + acc);
}
```

A function declaration is only required if the definition appears *below* the first appearance of the function in the code.

For *mutual recursion*, at least one of the functions **must** have a declaration.

Function declarations are important for module *interfaces*.

C does not require the parameter *names* in the declaration:

```
int accsum(int, int);
```

But it is good practice to include them to aid communication.

Scope review

Before introducing *modules* in C, we will briefly review our *scope* terminology. Functions and constants at the top-level have *global scope*, and are “visible” anywhere*. Parameters and local constants have *block scope*, and are only “visible” within their block*.

```
const int g = 42;              // g has global scope

int f(int p) {                 // function f has global scope
  const int b = g + p + 2;     // b and p have block scope (local)
  //...
}
```

* of course, identifiers are only “visible” *after* (or *below*) their declaration.

Global scope & extern

By default, functions and constants with *global scope* **are also available for use in other modules** (`.c` files).

If module B *declares* a function that is in A, it becomes “visible” in B.

For constants, the declaration in B requires the additional `extern` keyword, which means “this constant is declared somewhere else”.

```
// sum.c (module A)

const int g = 42;

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

```
// client.c (module B)

extern const int g;

int sum(int k);

int my_sum(int n) {
  return g * sum(n);
}
```

Module scope

The `static` keyword changes the scope of a function (or top-level constant) to ***module scope***, meaning that it is **only available** or "visible" within the *current module* (`.c` file).

```
static const int module_var = 1;

static int module_function(int p) {...}
```

Any `static` functions (or constants) in module A cannot be used in module B.

The `static` keyword was a terrible choice.

A *much* better choice would have been `private`.

The `static` keyword is also useful to avoid “name collision”. If two modules have functions (or top-level constants) with the same name, C generates an error. To avoid this problem, use `static` to give the functions (or constants) *module scope*.

The scope terminology we are using (block, module, global) is a *simplified* version of scope and ***linkage*** in C (CP:AMA 18.1,18.2).

C interface files

In C, we can create a module ***interface file*** (`.h` file) that is separate from the module *implementation* (`.c` file).

The *interface file* contains the interface documentation and the *declarations* for the interface functions (and constants).

```
// sum.c (IMPLEMENTATION)

const int g = 42;

// see sum.h for details
int sum(int k) {
  return (k <= 1) ? 1 :
          k + sum(k-1);
}
```

```
// sum.h (INTERFACE)

extern const int g;

// sum(k) finds the sum of all
//     integers from 1..k
//   PRE:  k >= 1
//   POST: returns int >= 1
int sum(int k);
```

To use a module (module A), the client (module B) needs to have the interface declarations appear at the top of the client module.

```
// client.c (module B)

// declarations copied from sum.h (module A)
extern const int g;
int sum(int k);

int my_sum(int n) {
  return g * sum(n);
}
```

Of course, we could just *“copy”* the declarations from the module’s interface file (`.h`) and *“paste”* them into the client file, but that would be very inefficient and would remove one of the advantages of modularization (*maintainability*).

#include

Fortunately, C has a built-in *“copy & paste”* feature:

`#include "module.h"`

literally pastes the contents of the file `module.h` into the current file.

```
// client.c -- ORIGINAL

#include "sum.h"

int my_sum(int n) {
  return g * sum(n);
}
```

```
// client.c -- AFTER PASTE

extern const int g;

// sum(k) finds the sum of all
//     integers from 1..k
//   PRE:  k >= 1
//   POST: returns int >= 1
int sum(int k);

int my_sum(int n) {
  return g * sum(n);
}
```

Each implementation (`module.c`) should *also* `#include` its own interface file (`module.h`).

This will help detect any discrepancies between the interface and the implementation.

This also reduces the number of declarations that must appear at the top of the implementation.

Later, we will see additional reasons to include the interface file.

RunC environment

If a client “requires” a module, `#include`-ing the interface file (`module.h`) makes the interface functions (and constants) available.

Conceptually, `#include` is very similar to Racket’s `require`, but the behaviour is very different: `require` “runs” the *implementation*, whereas `#include` “copy & pastes” the *interface*.

The `RunC` environment will *automatically* “bundle” into your program the *implementation* of each module you `#include`.

This is one of the many tasks that `RunC` manages for you when you “run” your program.

C standard modules

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

Fortunately, C provides several standard modules (called “libraries”) with many convenient functions.

When you `#include` a standard module, the syntax is a little different (<> instead of `""`).

```
// For standard C modules:
#include <module.h>

// For your own modules:
#include "mymodule.h"
```

stdbool

One of the C standard modules is `stdbool`, which makes working with Boolean values more convenient.

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

```
#include <stdbool.h>

const bool is_cool = true;

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

assert

Another C standard module is `assert`, which provides a testing mechanism similar to Racket's `check-expect`.

`assert(e)` will *halt* the program and display a message if the expression `e` is false (nothing happens if the expression is true).

`assert` is especially useful for verifying pre-conditions. In this course, you are expected to `assert` all of your pre-conditions.

```
#include <assert.h>

// PRE: k >= 1
int sum(int k) {
  assert(k >= 1);
  return (k == 1) ? 1 : k + sum(k-1);
}
```

Preprocessor directives

`#include` is a ***preprocessor directive***.

When you "run" a C program, there are several "stages" that occur. The very first stage is the *preprocessor*, which scans your program looking for ***directives***.

Directives start with # and are "special instructions" that can modify your program before continuing to the next stage.

You will typically never "see" the effects of the preprocessor. (you will not see the `#include` directive "paste" into your file).

After #include, #define is the most common preprocessor directive. In its simplest form it will perform a *search & replace*.

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

bool is_lucky(int n) {
  return n == LUCKY_NUM;
}
```

In C99, it is better style to declare a `const int`, but you will still see #define in “real world” C.

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

main

When you “run” a Racket program, it starts at the top of the file and then for each top-level expression it *evaluates* the expression and then *displays* the final value.

In C, top-level expressions are not allowed (at the top-level, there are only declarations, definitions and directives).

So what happens when you “run” a program in C?

Every C program **must** have a function named `main`.

When you “run” a program in C, you *call* `main`.

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

The `void` keyword is used to declare that there are no parameters.

The `main` function is special for several reasons:

- The *Operating System* (OS) calls `main`.
- Even though the *return type* of `main` is an `int`, `main` does not require a `return` value.
- The purpose of the `main` return value is to indicate to the OS if the program was successful (zero) or if there was an error (non-zero). No return value also means “success”.

Testing module

For each C program, there can only be one `main` function.

There should be a "main" client module that has the `main` function.

In the `RunC` environment, this is the file that should be "run".

Submodules (modules that are not the "main" client) should not have a `main` function.

For each submodule you create, you should also create a client *testing module* that has a `main`.

```c
// test-sum.c: testing module for sum.h

#include <assert.h>

#include "sum.h"

int main(void) {
  assert(sum(1) == 1);
  assert(sum(2) == 3);
  assert(sum(3) == 6);
  assert(sum(10) == 55);
  assert(sum(99) == 4950);
}
```

Interactive testing

In Racket, it was convenient to use top-level expressions to “display” values to the screen, or to use the “interactions window” in DrRacket to interactively test our code.

To achieve some of this convenience in C, we will “skip ahead” and introduce a method for displaying output.

This will require us to stray from our purely “functional C” introduction.

We already strayed a little when we introduced `assert`.

printf

The standard C module `stdio` provides functions for performing *Input & Output (IO or I/O)*.

One of the output functions is `printf`, which is for printing "**f**ormatted" output.

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

```
#include <stdio.h>

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

```
Hello, World!printf is fun
```

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

```
printf("Hello, World\n");
printf("printf\nis fun\n");
```

```
Hello, World!
printf
is fun
```

You can also display **values** with `printf`:

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

```
two plus two is: 4
```

`"%d"` is a ***placeholder***, which indicates *where* in the output you wish to display the value of the argument.

We will learn more placeholders, but for now `"%d"` (which means “display in **d**ecimal”) is the only placeholder you should use.

You can have more than one placeholder in the string. For each placeholder, you must pass an additional argument to `printf`:

```
printf("I am 99%% sure that %d + %d = %d\n", 2, 2, 5);
```

```
I am 99% sure that 2 + 2 = 5
```

More examples: printf

```
printf("you can combine more than ");
printf("%d", 2);
printf(" printfs into %d line\n", 1);
printf("watch your");
printf("spacing!\n");
printf("powers of ten: %d%d%d\n", 1, 10, 100);
printf("powers of two: %d %d %d\n", 1, 2, 4);
```

```
you can combine more than 2 printfs into 1 line
watch yourspacing!
powers of ten: 110100
powers of two: 1 2 4
```

Most computer languages have a `printf` function (or an equivalent) and many use the same placeholder syntax as C.

The full C `printf` placeholder syntax allows for a lot of control to format and align your output. For example:

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

```
Four digits with zero padding: 0042
```

See CP:AMA 22.3 for more details.

In this course, simple `"%d"` formatting is suffficent.

Structures in C

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

```
(struct posn (x y)                struct posn {
  #:transparent)                    int x;
                                    int y;
                                  };

(define p (posn 3 4))             const struct posn p = {3,4};

(define a (posn-x p))             const int a = p.x;
(define b (posn-y p))             const int b = p.y;
```

In C no *functions* are generated when you define a structure.

```
(struct posn (x y)                struct posn {
  #:transparent)                    int x;
                                    int y;
                                  };

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

The C ***structure definition*** requires the *type* of each field.

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

> Do not forget the semicolon (`;`) in the structure definition.

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

(define a (posn-x p))        const int a = p.x;
(define b (posn-y p))        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 declare structures:

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

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

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

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

Also, you cannot directly `printf` structures. You have to print each field 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 declaration syntax**, and should not be used inside of an expression.

```
const struct posn p = i > 0 ? {1,1} : {-1,-1}; // INVALID

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

This can be avoided by declaring separate constants as required:

```
const struct a = {1,1};
const struct b = {-1,-1};
const struct posn p = i > 0 ? a : b;

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

If a module wants to make a structure accessible to clients, the definition is placed in the *interface file* (`.h`).

```c
// posn.c

#include "posn.h"

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

```c
// posn.h

struct posn {
  int x;
  int y;
};

int f(struct posn p);
```

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

Goals of this module

You should understand the C syntax and terminology introduced and be comfortable declaring constants and writing expressions in C.

You should be comfortable re-writing a simple Racket function in C.

You should understand the C operators introduced in this module (including `% ? == != >= && || .`).

You should understand the difference between a function declaration and a function definition.

You should understand the differences between block, module and global scope and how `static` and `extern` are used.

You should understand how to modularize in C with implementation and interface files, and how to use `#include`.

You should understand the importance of the `main` function in C.

You should understand how to perform basic testing and I/O in C using `assert` and `printf`.

You should understand how to use structures in C.

You should understand the documentation we will require for C code in this course.