An Introduction to C

Readings: CP:AMA 2.2, 2.3, 2.7, 4.1, 5.1, 9.1

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

The goal of this section is to be able to write and test simple functions in C.
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 04 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 OS X, 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

First, we learn to write simple functions in C. This allows us to become familiar with the C syntax without introducing too many new concepts.

In Section 03 we introduce new programming concepts (imperative programming).

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

// C comment ; Racket comment

/* C multi-line comment */ #| Racket 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 */
C uses the more familiar *infix* algebraic notation \((3 + 3)\) instead of the *prefix* notation used in Racket \((+ 3 3)\).

```plaintext
// C Expressions:  
3 + 3 
1 + 3 * 2 
(1 + 3) * 2 

; Racket Expressions:  
(+ 3 3) 
(+ 1 (* 3 2)) 
(* (+ 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 is equivalent to (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.

\[(4 \times 5) / 2 \Rightarrow 10\]

\[4 \times (5 / 2) \Rightarrow 8\]

\[-5 / 2 \Rightarrow -2\]

Remember, use parentheses to clarify the order of operations.

† C99 standardized the “(round toward zero)” behaviour.
The % operator

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

\[
\begin{align*}
9 \ % \ 2 & \Rightarrow 1 \\
9 \ % \ 3 & \Rightarrow 0 \\
9 \ % \ 5 & \Rightarrow 4
\end{align*}
\]

In this course, avoid using % with negative integers.

In C99, \((i \ % \ j)\) has the same sign as \(i\).
(see CP:AMA 4.1 for more details).
Function definitions

// C function:
int my_sqr(int n) {
    return n * n;
}

; Racket function:
(define (my-sqr n) (* n n))

There are many subtle and important differences between defining a function in Racket and in C.

We explore those differences over the next few slides.
Identifiers

C identifiers (“names”) must start with a letter, and can only contain letters, underscores and numbers.

For example, use my_sqr instead of my-sqr.

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

C identifiers can start with a leading underscore (_name) but they may interfere with reserved keywords. Avoid them in this course as they may interfere with marmoset tests.
Function block and return

int my_sqr(int n) {
    return n * n;
}

; my-sqr: Int -> Int
(define (my-sqr n)
    (* n n))

Braces ({}) indicate the beginning and end of the function body, known in C as the function **block**.

The expression is preceded by the **return** keyword and followed by a semicolon (;).

Note the placement of the braces ({}) and the use of whitespace/indentation (more on this later).
Dynamic typing in Racket

Racket uses *dynamic typing*: types are determined **while** the program is running.

```
;; dtype: (anyof Int Str) -> (anyof Bool Sym)
(define (dtype param)
  (cond [(integer? param) true]
        [(string? param) 'dynamic]))
```

The types of the parameter and the produced ("returned") value are both *dynamic*.

We communicate the types in the *contract* as a *comment*. 
C uses **static typing**: all types must be known **before** the program is run and they cannot change.

The return type of `my_sqr` is an int (short for integer).

Parameter `n` is also an int.
C Types

For now, we only use **integers** 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?). In Racket, a contract violation may cause a “type” runtime error.

```racket
(my-sqr "hello") ; Racket runtime error
```

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

```c
my_sqr("hello") // Will not run in C
```
If you omit the type in a function definition:

```c
int my_sqr(int n) { // properly typed
    return n * n;
}

my_bad_sqr(n) { // missing types
    return n * n;
}
```

C will assume the type is an int and will often display a warning such as:

```
type specifier missing, defaults to 'int'
```

This is very bad style: you should specify every type.
Function terminology

In this course, we use more common terminology:

We *call* a function by *passing* it arguments and it *returns* a value.

apply $\Rightarrow$ call

consume $\Rightarrow$ pass

produce $\Rightarrow$ return

In “functional” CS 135 terminology, we *apply* a function, which *consumes* arguments and it *produces* a value.
More function syntax

Multiple arguments and parameters are separated by commas (,).

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

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

Use the `void` keyword to indicate a function has no parameters.

Also, in C you cannot have *local* (or “nested”) functions defined inside of other functions.
If you omit the `void` in a parameterless function definition:

```c
int my_num() {
    // ...
}
```
C will allow it. This is because `()` is used in an older C syntax to indicate an “unknown” or “arbitrary” number of parameters (beyond the scope of this course).

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

```c
int my_num(void) {
    // ...
}
```
Function documentation

Purpose statements are nearly identical:

```c
int my_sqr(int n) {
    return n * n;
}
```

In C, the contract *types* are part of the function definition so no type contract documentation is necessary. However, you should still add a `requires` comment if appropriate.

```c
int my_divide(int x, int y) {
    return x / y;
}
```
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.

We will now demonstrate how to run and test a simple C program.
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.

```c
int main(void) {
    //...
    return 0;     // success!
}
```

The return value communicates to the OS the “error code” (also known as the “error number” or just errno).

A successful program returns zero (no error code).

† main has *optional* parameters (discussed in Section 13).
main is a special function and does not require an explicit return value. The default value is success (zero).

```c
// main() does not need a purpose statement
int main(void) {
    //...
    return 0;  // this is optional
}
```

In this course, main does **not** require a purpose statement, but in general it is good style to document the purpose of a program.

In this course, your main function should never return a non-zero value, as it will cause your marmoset tests to fail.
Top-level expressions

In DrRacket, the final values of *top-level expressions* (code outside of a function) are displayed in the “interactions window”.

```racket
;; my racket program
(+ 1 1)        ;; <-- top level

(define (my-sqr n)
  (* n n))

(my-sqr 7)    ;; <-- top level
```

2
49
In C, *top-level expressions are not* allowed.

// my C program

1 + 1; // INVALID

int my_sqr(int n) {
    return n * n;
}

my_sqr(7); // INVALID
Tracing expressions

We have provided *tracing tools* to help you “see” what your code is doing. Here, we use `trace_int` inside of `main` to trace several expressions and display them to the screen (console):

```c
int main(void) {
    trace_int(1 + 1);
    trace_int(my_sqr(7));
}
```

1 + 1 => 2  
my_sqr(7) => 49

You can leave the *tracing* in your code. It is ignored in our tests and will not affect your results (no need to comment it out).
We’re now ready to run our first program.

// My first C program

#include "cs136.h"    // <-- more on this later

int my_sqr(int n) {
    return n * n;
}

int main(void) {
    trace_int(1 + 1);
    trace_int(my_sqr(7));
}

Note the necessary #include line at the top of the program. For now, always add this line (it will be explained later).
Function ordering

If you re-order the two functions in our program:

```
int main(void) {
    trace_int(1 + 1);
    trace_int(my_sqr(7));
}

int my_sqr(int n) { // now below main
    return n * n;
}
```

You will get an error such as:

implicit declaration of function 'my_sqr' is invalid

For now, always place function definitions **above** any other functions that reference them (so main will be at the bottom).
Boolean expressions

To facilitate testing, we will need Boolean expressions.

In C, a Boolean expression does not produce “true” or “false”. An expression produces zero (0) for “false”, and one (1) for “true”. In our environment, the constants `true` and `false` have been defined to be 1 and 0 (for convenience).

If used in a Boolean expression, any non-zero value is also considered “true”. Only zero is “false”.
Equality operator

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

\[
\begin{align*}
3 &\quad ==\quad 3 \quad \Rightarrow \quad 1 \quad (\text{true}) \\
2 &\quad ==\quad 3 \quad \Rightarrow \quad 0 \quad (\text{false})
\end{align*}
\]

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

\[
\begin{align*}
2 &\quad !=\quad 3 \quad \Rightarrow \quad 1 \quad (\text{true})
\end{align*}
\]

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 03). It is such a serious concern that it warrants an extra slide as a reminder.
The *not*, *and* and *or* operators are respectively `!`, `&&` and `||`.

\[
\begin{align*}
!(3 == 3) & \Rightarrow 0 \\
(3 == 3) && (2 == 3) & \Rightarrow 0 \\
(3 == 3) && !(2 == 3) & \Rightarrow 1 \\
(3 == 3) || (2 == 3) & \Rightarrow 1 \\
2 && 3 & \Rightarrow 1 \quad \text{(why?)}
\end{align*}
\]

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

\[
(a \neq 0) && (b / a == 2)
\]

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.

\[(2 < 3) \Rightarrow 1\]
\[(2 >= 3) \Rightarrow 0\]

\![a < b] \text{ 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.
You are not expected to “memorize” the order of operations. When in doubt (or to add clarity) **add parentheses**.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>!</td>
</tr>
<tr>
<td>multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt; &lt;= &gt;= &gt;</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
</tbody>
</table>
The assert function can be used in place of Racket’s check-expect. The equivalent of:

\[(\text{check-expect } (\text{my-sqr } 7) \ 49)\]

is:

\[\text{assert}(\text{my\_sqr}(7) == 49);\]

assert\(\text{exp}\) stops the program and displays a message if the expression exp is false (zero).

If exp is true (non-zero), nothing happens.
// My second C program  
// (now with testing!)

#include "cs136.h"

int my_sqr(int n) {  
  return n * n;  
}

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);  
}  

We will discuss additional testing methods later. For now, test your code with asserts in your main function as above.
bool type

The bool type is an integer that can only have a value of 0 or 1.

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

bool my_negate(bool v) {
    return !v;
}
```
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.

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

There can be more than one return in a function.
Racket’s cond special form consumes a sequence of question and answer pairs (where 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]))
```
```c
int my_function(...)
{
    if (q1) {
        return a1;
    } else if (q2) {
        return a2;
    } else {
        return a3;
    }
}
```
example: using if, else and else if

// in_between(x, lo, hi) determines if lo <= x <= hi
// requires: lo <= hi

bool in_between(int x, int lo, int hi) {
    if (x < lo) {
        return false;
    } else if (x > hi) {
        return false;
    } else {
        return true;
    }
}
Recursion in C behaves the same as in Racket.

```
// sum_first(n) sums the natural numbers 0..n
// requires: n >= 0

int sum_first(int n) {
    if (n <= 0) {
        return 0;
    } else {
        return n + sum_first(n-1);
    }
}
```
It might appear that Racket’s cond and C’s if are “the same” but fundamentally they are quite different.

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

\[ (+ y (\text{cond} \ [(< x 0) \ -x]\ [\text{else} \ x])) \]

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 04 after we understand how “statements” differ from expressions. For now, only use if as we have demonstrated.
Unlike C’s if statement, the C ?: operator does produce a value and behaves the same as Racket’s if special form.

```scheme
;; 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
// 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;
```
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

• re-write a simple Racket function in C (and vice-versa)

• use the C operators introduced in this module (including % == != >= && ||)

• explain the significance of the main function in C

• perform basic tracing in C using trace_int

• perform basic testing in C using assert

• provide the required documentation for C functions