The primary 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 05) and be easily translatable into “machine code” (Section 13).

Thousands of popular programs and portions of all of the popular operating systems are written in C.

C versions

There are a few different versions of the C standard.

In this course, we use the C99 standard (from 1999).

The C11 standard (2011) added some new features to the language, but those features are not needed in this course.

The C18 standard (2018) only fixes a few bugs from C11.

The C2x standard (202?) is currently in development.
**Introductory C**

In this section we learn to write simple functions and programs 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).

---

**Comments**

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

Any text between /* and */ is also a comment.

/* ... */ can extend over multiple lines and can comment out large sections of code.

// C comment (one-line only)

/* This is a multi-line comment */

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

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

---

**Expressions**

C expressions use traditional *infix* algebraic notation: (e.g., 3 + 3).

Use parentheses to specify the **order of operations** (normal arithmetic rules apply).

\[
1 + 3 \times 2 \quad \Rightarrow \quad 7 \\
(1 + 3) \times 2 \quad \Rightarrow \quad 8
\]

Racket uses *prefix* notation: (+ 3 3)

Languages that use prefix (or postfix) notation do not require parenthesis to specify the order of operations.
Operators

In addition to the traditional mathematical operators (e.g., +, -, *), C also has non-mathematical operators (e.g., data operators).

With over 40 operators in total, the order of operations is complicated (see CP:AMA Appendix A).

C does not have an exponentiation operator (e.g., \(x^n\)).

Confusingly, the “bitwise exclusive or” operator (\(^\)) looks like an exponentiation operator. Bitwise operators are beyond the scope of this course.

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.

\[
(4 * 5) / 2 \Rightarrow 10 \\
4 * (5 / 2) \Rightarrow 8 \\
-5 / 2 \Rightarrow -2
\]

Remember, use parentheses to clarify the order of operations.

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

The C modulo operator (%) produces the remainder after integer division.

9 % 2 ⇒ 1
9 % 3 ⇒ 0
9 % 5 ⇒ 4

The value of (a % b) is equal to: a - (a / b) * b.

It is often best to avoid using % with negative integers.

(i % j) has the same sign as i (see CP:AMA 4.1).

C identifiers

Every function, variable and structure requires an identifier (or “name”).

C identifiers must start with a letter, and can only contain letters, underscores and numbers.

In this course, we use underscore style (or snake case) for identifiers with compound words.

For example:

hst_rate
trace_int
quick_sort
bst_insert

underscore_style is the most popular style for C projects.

In other languages (e.g., Java) camelCaseStyle is popular.

In practice, it is best to use the recommended style for the language and/or follow the project (or corporate) style guide.

Above all else, use a consistent style in your code.

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.
Anatomy of a function definition

```c
int my_add(int a, int b) {
    return a + b;
}
```

- braces `()` indicate the beginning/end of a function block.
- `return` keyword, followed by an expression, followed by a semicolon `;`
- parameters `a, b` are separated by a comma.
- the function and parameter types are specified (i.e., `int`).

Note the placement of the braces `()` and the use of whitespace and indentation (more on this later).

Static type system

C uses a **static type system**: all types must be known before the program is run and the type of an identifier cannot change.

For now, we will only use C **integers** (more types in Section 04).

```c
int my_add(int a, int b) {
    return a + b;
}
```

The return type of `my_add` is an `int` (the first `int`).

The parameters `a` and `b` are also both `int`s.

Racket uses a **dynamic** type system.

If you omit the type in a function definition:

```c
int my_add(int a, int b) { // properly typed
    return a + b;
}
```

```c
bad_add(a, b) { // missing types
    return a + b;
}
```

C assumes a missing type is an `int` and may display a warning such as:

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

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

```
(my-add "hello" 3) ; Racket runtime error
```

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

```
my_add("hello", 3) // does not run in C
```

### Function terminology

We **call** a function by passing it arguments.

A function **returns** a value.

```
my_add(1, 2) ⇒ 3
```

We **call** `my_add` and **pass** it the arguments 1 and 2.

`my_add` **returns** 3.

In “functional” language terminology (e.g., Racket) we **apply** a function, which **consumes** arguments and it **produces** a value.

### Functions without parameters

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

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

To call a parameterless function, put nothing between the parentheses (you do not pass `void`).

```
my_num() ⇒ 42
```
If you omit the `void` in a parameterless function definition:

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

C allows 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**

Provide a purpose for every function that shows an example of it being called, followed by a brief description of **what** the function does (not **how** it does it).

No contract types are necessary (they are part of the definition).

However, you should still add a **requires** comment if appropriate.

```c
// my_divide(a, b) evaluates a/b using integer division
// requires: b is not 0
int my_divide(int a, int b) {
    return a / b;
}
```

---

**Whitespace**

C mostly ignores whitespace.

```c
// The following three functions are equivalent
int my_add(int a, int b) { // GOOD
    return a + b;
}

int my_add(int a,int b){return a+b;} // BAD

int my_add(int a, int b){return a+ b ;} // RIDICULOUSLY BAD
```

You should follow the course style. The course staff and markers may not understand your code if it is poorly formatted.
**CS 136 style**

```c
int my_add(int a, int b) {
    return a + b;
}
```

- a block start (open brace {) appears at the end of a line
- a block end (close brace }) is aligned with the line that started it, and appears on a line by itself
- indent a block 2 (recommended), 3 or 4 spaces: **be consistent**
- add a space after commas and around arithmetic operators

Typing Ctrl-I in Seashell will auto-indent your code for you.

When you have a large number of parameters, a large expression or a long purpose you should continue (indented) on the following line.

```c
// my_super_long_function(a, b, c, d, e, f, g) does some
// amazing things with those parameters...
int my_super_long_function(int a, int b, int c, int d,
    int e, int f, int g) {
    return a * b + b * c + c * d + d * e + e * f +
        f * g + g * a;
}
```

The “best” way to style your code (e.g., block formatting) is a matter of taste and is often a topic of debate.

The style we have chosen is the most widely accepted style for C (and C++) projects (e.g., it conforms to the Google style guide).

---

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

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

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

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 “exit code”, “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 causes your marmoset tests to fail.
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

---

**Top-level expressions**

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

```c
// 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 does not affect your results (no need to comment it out).
We're now ready to run our first program.

```c
// 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 is explained later).

**Function ordering**

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

```c
int main(void) {
    trace_int(1 + 1);
    trace_int(my_sqr(7));
}
int my_sqr(int n) { // now below main
    return n * n;
}
```

You get an error such as:

```c
implicit declaration of function 'my_sqr' is invalid
```

For now, always place function definitions **above** any other
functions that reference them (so `main` is at the bottom).

**No nested functions**

In C, you **cannot** “nest” (define) a function *inside* of another function
(a.k.a. local functions).

```c
int main(void) {
    int my_sqr(int n) { // INVALID
        return n * n;
    } // ...
}
```

The GNU C environment (gcc) has introduced a *language extension* for nested C functions, but it is not part of the C standard.
**Boolean expressions**

To facilitate testing, we need Boolean expressions.

In C, a Boolean expression does not produce “true” or “false”.

It will produce 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).

---

**Equality operator**

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

\[(3 == 3) \Rightarrow 1 \text{ (true)}\]
\[(2 == 3) \Rightarrow 0 \text{ (false)}\]

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

\[(2 != 3) \Rightarrow 1 \text{ (true)}\]

---

**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 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 ||.

\[
! (3 == 3) \Rightarrow 0 \\
(3 == 3) && (2 == 3) \Rightarrow 0 \\
(3 == 3) && !(2 == 3) \Rightarrow 1 \\
(3 == 3) || (2 == 3) \Rightarrow 1
\]

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

\[
(a \neq 0) && (b / a == 2)
\]
does not generate an error if \(a\) is 0.

---

A common mistake is to use a single & or | instead of && or ||.

---

All non-zero values are true

Operators that *produce* a Boolean value (*e.g.*, ==) will always produce 0 or 1.

Operators (or functions) that *expect* a Boolean value (*e.g.*, &&) will consider any non-zero value to be “true”.

Only zero (0) is “false”.

\[
(2 && 3) \Rightarrow 1 \\
(0 || -2) \Rightarrow 1 \\
!5 \Rightarrow 0
\]

† The value NULL (Section 05) is also considered false.

---

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

\[
1 == 3 > 0 \text{ is equivalent to } 1 == (3 > 0), \text{ but it could easily confuse some readers.}
\]
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>Symbol</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>

**Assertions**

The `assert` function can be used to test your code:

```c
assert(my_sqr(7) == 49);
```

`assert(exp)` **stops** the program and displays a message if the expression `exp` is false (zero).

If `exp` is true (non-zero), it does “nothing” and continues to the next line of code.

**assert is very similar to Racket's check-expect:**

```racket
(check-expect (my-sqr 7) 49)
```

```c
// 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(2) == 4);
    assert(my_sqr(32) == 1024);
    assert(my_sqr(-32) == 1024);
}
```

We discuss additional testing methods later. For now, test your code with `asserts` in your `main` function as above.
Function requirements

The assert function is also very useful for verifying function requirements.

```c
// my_divide(x, y) ....
// requires: y is not 0

int my_divide(int x, int y) {
    assert(y != 0); // assert(y) also works
    return x / y;
}
```

In the slides, we often omit asserts to save space.

You should assert any feasible function requirements.

Some requirements are infeasible to assert, or (as we will discuss in Section 08) they would be inefficient to assert.

In practice, it would be good style to communicate (i.e., document) that the requirement is not asserted:

```c
// my_function(n) ....
// requires: n is a prime number [not asserted]
```

However, we will not require that in this course.

If you have a function with more than one requirement:

```c
// my_function(x, y, z) ....
// requires: x is positive
//         y < z

int my_function(int x, int y, int z) {
    assert((x > 0) && (y < z)); // POOR
    assert(x > 0); // GOOD
    assert(y < z); // GOOD
    //...
}
```

It is better to have several small assert statements. That way it is easier to determine which assertion failed (and which requirement was not met).
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

C's if statement allows us to have functions with 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, but only one value is returned.

The function "exits" when the first return is reached.

```c
example: recursion in C

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

int sum_first(int n) {
    assert(n >= 0);
    if (n <= 0) {
        return 0;
    } else {
        return n + sum_first(n - 1);
    }
}
```
else if

If there are more than two possible results, use `else if`.

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

bool in_between(int x, int lo, int hi) {
  assert(lo <= hi);
  if (x < lo) {
    return false;
  } else if (x > hi) {
    return false;
  } else {
    return true;
  }
}
```

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`:

```racket
(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;
  }
}
```

C’s `if` statement does not produce a value: it only controls the “flow of execution” and cannot be used inside of 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.

Given the examples we have seen so far, it might appear that Racket’s `cond` and C’s `if` are “the same”.

Fundamentally, they are quite different. Unlike `if`, `cond` produces a value and can be used inside of an expression:

```racket
(+ y (cond \[(< x 0) -x\]
      \[else x\]))
```
Unlike C's if statement, the C ternary conditional operator (?:) does produce a value.

The value of the expression:

\[ q ? a : b \]

is a if q is true (non-zero), and b otherwise.

For example:

\[
(v >= 0) ? v : -v // abs(v) \\
(a > b) ? a : b // max(a, b)
\]

You may use the ?: operator in this course, but use it sparingly. Overuse of the ?: operator can make your code hard to follow.

When working with integer values in C, do not add a leading (preceding) zero (0) to the value. For example, do not write 017 if you want to represent the number 17.

A leading zero may seem harmless, but it is not:

\[
\text{trace\_int}(17); \quad 17 \Rightarrow 17 \\
\text{trace\_int}(017); \quad 017 \Rightarrow 15 \\
\text{trace\_int}(\text{my\_sqr}(010)); \quad \text{my\_sqr}(010) \Rightarrow 64
\]

In C, integer values that start with a zero are evaluated in octal (base 8), so 010 is equivalent to 8.

Integer values that start with 0x are evaluated in hexadecimal, so 0x10 is equivalent to 16.

**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
- Write a simple function in C
- 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
- use assert for testing and to verify requirements
- provide the required documentation for C functions