

# Computer Science @ The University of Waterloo

## CS 115, CS 116, CS 135, & CS 136

### Style Guide

Last Updated: 2015.05.04

## 1 Introduction

The code you submit for assignments, as with all code you write, can be made more readable and useful by paying attention to style. This includes the placement of comments, whitespace, indentation, and choice of variable and function names. None of these things affect the execution of a program, but they affect its readability and extensibility.

Just as in writing English prose, the goal is communication, and you need to think of the needs of the reader. This is especially important when the reader is assigning you a grade.

The first sections of this guide detail the style expected for Racket programs, which is common to CS 115, CS 116, CS 135, and CS 136. Changes to expectations in Python (for CS 116) and C (for CS 136) follow in later sections.

### 1.1 A Warning on Examples From Other Sources

The examples in the presentation slides, handouts and tutorials/labs are often condensed to fit them into a few lines; you should not imitate their condensed style for assignments, because you do not have the same space restrictions.

Similarly, you should not directly follow the style used in the HtDP textbook (*e.g.*, the HtDP design recipe). After years of experience teaching at Waterloo, we have modified the HtDP style to improve its usefulness for an introductory Racket course and beyond. However, the spirit of HtDP style remains and the examples in the textbook are good, particularly those illustrating the design recipe, such as Figure 3 in Section 2.5.

For your assignments, use the coding style in this guide, not the style from the HtDP textbook, or the condensed style from the presentation slides.

## 1.2 A Warning on DrRacket

DrRacket(.rkt) files can store rich content that can include images, extended formatting, comment boxes, and special symbols. Using this rich content may make your assignment unmarkable. Unfortunately, some of the content in the Interactions window may be “rich” (*e.g.*, rational numbers), and so you should avoid copy-and-pasting from your interactions window into your definitions window. In addition, code that appears in a .pdf document (*e.g.*, presentation slides and assignments) may contain hidden or unusual symbols, so you should not copy-and-paste from those sources either.

For your assignments, save in **plain text** format. In DrRacket (version 6.1) you can set the format of your file using the Edit > Modes menu item, and you can ensure your .rkt file is saved using plain text format by using the menu items: File > Save Other > Save Definitions as Text.

Never include Comment Boxes or images in your submissions. Do not copy-and-paste from the Interactions window or from .pdf documents into the Definitions window, or your assignment may become unmarkable.

## 2 Assignment Formatting

### 2.1 Comments

In Racket, anything after a semicolon (;) on a line is treated as a comment and ignored. Always follow a semicolon by a space to improve readability.

- For full-line comments use two semicolons
- For in-line comments use one semicolon

```
;; This is a full-line comment: Double semicolons and a space  
;; Below is an example of an in-line comment:  
(define hst-rate 0.13) ; HST Rate in Ontario effective July 1, 2010
```

Use in-line comments sparingly. If you are using standard design recipes and templates, and following the rest of the guidelines here, you should not need many additional comments. Any such comment can either be put on its own line, or tacked onto the end of a line of code, providing it fits.

## 2.2 Header

Your file should start with a header to identify yourself, the term, the assignment and the problem. There is no specifically required format, but it should be clear and assist the reader. The following is a good example.

```
;;  
;; *****  
;;   Arthur Dent (12345678)  
;;   CS 135 Fall 2014  
;;   Assignment 03, Problem 4  
;; *****  
;;
```

## 2.3 Whitespace, Indentation and Layout

Insert two consecutive blank lines between functions or “function blocks”, which include the documentation (*e.g.*, design recipe) for the function. Insert one blank line before and after the function definition: do not use more than one consecutive blank line within a function block. If you have many functions and/or large function blocks, you may want to insert a row of symbols (such as the \*’s used in the file header above) to separate your functions. If the question asks you to write more than one function, the file should contain them in the order specified by the assignment. Helper functions should be put with the assignment function they are helping. Typically, helper functions are placed before the main function, but that is a judgment and they can be placed afterward depending on the situation. Remember that the goal is to make it easier for the reader to determine where your functions are located.

Indentation plays a big part in readability. It is used to indicate level of nesting, to align related subexpressions (*e.g.*, arguments of a function), and to make keywords more visible. DrRacket’s built-in editor will help with these. If you start an expression (`my-fn` and then hit enter or return, the next line will automatically be indented a few spaces. However, DrRacket will never break up a line for you, and you can override its indentation simply by putting in more spaces or erasing them. DrRacket also provides a menu item for reindenting a selected block of code (Racket > Reindent) and even a keyboard shortcut reindenting an entire file (Ctrl+I in Windows).

When to start a new line (hit enter or return) is a matter of judgment. Try not to let your lines get longer than about 70 characters, and definitely no longer than 80 characters. You do not want your code to look too horizontal, or too vertical. Occasionally, examples and tests may exceed the line length guidelines, when there is no obvious place to break the line. You should still strive to keep the lines within the suggested limits though.

Style marks may be deducted if you exceed 80 characters on any line of your assignment.

Indentation examples:

```
;; GOOD
(define (squaresum x y)
  (+ (* x x)
     (* y y)))

;; BAD (too horizontal)
(define (squaresum x y) (+ (* x x) (* y y)))

;; BAD (too vertical)
(define
  (squaresum x y)
  (+
    (*
      x
      x)
    (*
      y
      y)))
```

If indentation is used properly to indicate level of nesting, then closing parentheses can just be added on the same line as appropriate, and you will see this throughout the textbook and presentations. Some styles for other programming languages expect that you place closing brackets on separate lines that are vertically lined up with the opening brackets. However, in Racket this tends to negatively effect the readability of the code and is considered “poor style”.

For conditional expressions, placing the keyword `cond` on a line by itself, and aligning not only the questions but the answers as well can improve readability (provided that they are short). However, this recommendation is not strictly required. Each question must appear on a separate line, and long questions/answers

should be placed on separate lines. Marks may be deducted if a `cond` is too dense or confusing.

`cond` examples:

```
;; Looks good for short cond questions and answers
(cond
  [(< bar 0) (neg-foo bar)]
  [else      (foo bar)])

;; acceptable
(cond [(< bar 0) (neg-foo n)]
      [else (foo n)])

;; BAD (place each question on a separate line)
(cond [(< bar 0) (neg-foo n)] [else (foo n)])

;; place long questions/answers on separate lines
(cond
  [(and (>= bar 0) (<= bar 100))
   (really-long-function-or-expression (/ bar 100))]
  [else
   (some-other-really-long-function-or-expression bar)])
```

## 2.4 Constant (Variable) and Function Identifiers

Try to choose names for functions and parameters that are descriptive, not so short as to be cryptic, but not so long as to be awkward. The detail required in a name will depend on context: if a function consumes a single number, calling that number `n` is probably fine. It is a Racket convention to use lower-case letters and hyphens, as in the identifier `top-bracket-amount`. (DrRacket distinguishes upper-case and lower-case letters by default, but not all Racket implementations do.) In other languages, one might write this as `TopBracketAmount` or `top_bracket_amount`, but try to avoid these styles when writing Racket code.

You will notice some conventions in naming functions: predicates that return a Boolean value usually end in a question mark (e.g., `zero?`), and functions that do conversion use a hyphen and greater-than sign to make a right arrow (e.g., `string->number`). This second convention is also used in contracts. In some course materials you may see an actual arrow produced (e.g., `string→number`), but you should always type them as two characters (`->`).

Constants should be used to improve your code in the following ways:

- To improve the readability of your code by avoiding “magic” numbers. For example, the purpose of the following code using the constant `cost-per-minute`

```
(define cost-per-minute 0.75)  
... (* cost-per-minute total-minutes) ...
```

is much clearer than when a constant is not used:

```
(* 0.75 total-minutes)
```

- To improve flexibility and allow easier updating of special values. If the value of `cost-per-minute` changes, it is much easier to make one change to the definition of the constant than to search through an entire program for the value `0.75`. When this value is found, it is not obvious if it refers to the cost per minute, or whether it serves a different purpose and should not be changed.
- To define values for testing and examples. As values used in testing and examples become more complicated (*e.g.*, lists, structures, lists of structures), it can be very helpful to define named constants to be used in multiple tests and examples.

## 2.5 Summary

- Use two semicolons (;;) for full-line comments.
- Use one semicolon (;) for in-line comments, and use them sparingly inside the body of functions.
- Provide a file header for your assignments.
- Use one blank line before and after your function definition.
- Make it clear where function blocks begin and end (*i.e.*, two blank lines between function blocks).
- Order your functions appropriately.
- Indent to indicate level of nesting and align related subexpressions.
- Avoid overly horizontal or vertical code layout.
- Use reasonable line lengths.
- Format conditional expressions appropriately.
- Choose meaningful identifier names and follow our naming conventions.

Style marks may be deducted if you have poor headers, identifier names, whitespace, indentation or layout.

### 3 The Design Recipe: Functions

**Warning!** The format of the Design Recipe starting in Fall 2014 is different than the one used in the HtDP textbook and in previous course offerings. This style guide will be used for assessment (*i.e.*, assignments and exams).

We hope you will use the design recipe as part of the process of working out your solutions to assignment questions. If you hand in only code, even if it works perfectly, you will earn only a fraction of the marks available. Elements of the design recipe help us to understand your code.

Not everything in this section will make sense on first reading, and some of the following details will only appear at the end of the course. We suggest that you review it before each assignment.

Develop your purpose, contract and examples before you write the code for the function.

#### 3.1 Design Recipe Sample

```
;; (sum-of-squares p1 p2) produces the sum of      <--- Purpose
;;   the squares of p1 and p2
;; sum-of-squares: Num Num -> Num                <--- Contract
;; Examples:                                     <--- Examples
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

(define (sum-of-squares p1 p2)                    ; <--- Function Header
  (+ (* p1 p1)                                     ; <--- Function Body
     (* p2 p2)))

;; Tests:                                         <--- Tests
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
```



## 3.2 Purpose

```
;; (sum-of-squares p1 p2) produces the sum of  
;;   the squares of p1 and p2
```

The purpose statement has two parts: an illustration of how the function is applied, and a brief description of what the function does. The description does not have to explain how the computation is done; the code itself addresses that question.

- The purpose starts with an example of how the function is applied, which uses the same parameter names used in the function header.
- Do not write the word “purpose”.
- The description must include the names of the parameters in the purpose to make it clear what they mean and how they relate to what the function does (choosing meaningful parameter names helps also). Do not include parameter types and requirements in your purpose statement - the contract already contains that information.
- If the description requires more than one line, “indent” the next line of the purpose 2 or 3 spaces. If your purpose requires more than two or three lines, you should probably consider re-writing it to make it more condensed.

## 3.3 Contract

```
;; sum-of-squares: Num Num -> Num
```

The contract contains the name of the function, the types of the arguments it consumes, and the type of the value it produces.

```
;; function-name: Type1 Type2 ... TypeN -> Type
```

See Section 4 below for a description of the valid types.

### 3.3.1 Additional Contract Requirements

If there are important constraints on the parameters that are not fully described in the contract, add an additional **requires** section after the contract. For example, this can be used to indicate a Num must be in a specific range, a Str must be of a specific length, or that a (listof ...) cannot be empty. Indent multiple requirements. For example:

```
;; (crazy-function n1 n2 n3 s lon) does something crazy...
;; crazy-function: Num Num Num Str (listof Num) -> Bool
;; requires: 0 < n1 < n2
;;           n3 must be non-zero
;;           s must be of length 3
;;           lon must be non-empty
;;           elements of lon must be unique and sorted in ascending order
```

## 3.4 Examples

```
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

The examples should be chosen to illustrate ‘typical’ uses of the function and to illuminate some of the difficulties to be faced in writing it. The examples do not have to cover all the cases that the code addresses; that is the job of the tests, which are designed after the code is written. It is very useful to write your examples before you start writing your code. These examples will help you to organize your thoughts about what *exactly* you expect your function to do. You might be surprised by how much of a difference this makes.

Typically, for recursive data, your examples should include at least one base case and at least one non-base case.

## 3.5 Function Header and Body

```
(define (sum-of-squares p1 p2)
  (+ (* p1 p1)
     (* p2 p2)))
```

You’d be surprised how many students have lost marks in the past because we asked for a function `my-fn` and they wrote a function `my-fun`. Their code failed all of our tests, of course, because they did not provide the function we asked for.

To avoid this situation, be sure to use the provided basic tests when you submit your assignments. If your code does not pass these tests, then you should carefully check your submission for spelling errors and the like.

## 3.6 Tests

```
;; Tests:  
(check-expect (sum-of-squares 0 0) 0)  
(check-expect (sum-of-squares -2 7) 53)
```

You do not have to repeat your examples in your tests section. Your examples also count as tests. Make sure that your tests are actually testing every part of the code. For example, if a conditional expression has three possible outcomes, you have tests that check each of the possible outcomes. Furthermore, your tests should be directed: each one should aim at a particular case, or section of code. Some people write tests that use a large amount of data; this is not necessarily the best idea, because if they fail, it is difficult to figure out why. Others write lots of tests, but have several tests in a row that do essentially the same thing. It's not a question of quantity, but of quality. You should design a small, comprehensive test suite.

Never figure out the answers to your tests by running your own code. Work out the correct answers by hand.

### 3.6.1 Testing Tips

Parameter type	Consider trying these values
Num	positive, negative, 0, non-integer values, specific boundaries
Int	positive, negative, 0
Bool	true, false
Str	empty string (""), length 1, length > 1, extra whitespace, different character types, etc.
(anyof ...)	values for each possible type
(listof T)	empty, length 1, length > 1, duplicate values in list, special situations, etc.
User-Defined	special values for each field (structures), and for each possibility (mixed types)

## 3.7 Additional Design Recipe Considerations

### 3.7.1 Helper Functions

Do not use the word “helper” in your function name: use a descriptive function name. The full design recipe is required for helper functions, but this should not discourage you from writing helper functions. Marks may be deducted if you are not using helper functions when appropriate.

### 3.7.2 Mutually Recursive Functions

For mutually recursive functions, only one group of examples and tests are required. If the required function is one of the mutually recursive functions, include the examples and tests with that function. In the following example, the tests and examples are included with the function `is-odd?`.

```
;; (is-even? x) produces true if x is even, and false otherwise
;; is-even?: Nat -> Bool
;; Examples and tests: see mutually recursive function is-odd?

(define (is-even? x)
  (cond [(= x 0) true]
        [else (is-odd? (sub1 x))]))

;; (is-odd? x) produces true is x is odd and false otherwise
;; is-odd?: Nat -> Bool
;; Examples:
(check-expect (is-odd? 0) false)
(check-expect (is-odd? 45) true)

(define (is-odd? x)
  (cond [(= x 0) false]
        [else (is-even? (sub1 x))]))

;; Tests
(check-expect (is-odd? 1) true)
(check-expect (is-odd? 10) false)
```

### 3.7.3 Wrapper Functions

When using wrapper functions, only one of the functions requires examples and tests. If the required function is the wrapper function, then include the examples and tests with it. Otherwise, the choice of which function is a judgment. In the

following example, the tests and examples are included with the wrapper function `remove-from`.

```
;; (remove-from-list loc c) produces a new list, like loc, but with
;;   all occurrences of c removed
;; remove-from-list: (listof Char) Char -> (listof Char)
;; Examples and tests: see wrapper function remove-from

(define (remove-from-list loc c)
  (cond [(empty? loc) empty]
        [(char=? (first loc) c) (remove-from-list (rest loc) c)]
        [else (cons (first loc) (remove-from-list (rest loc) c))]))

;; (remove-from s) produces a new string like s, but with all
;;   A and a characters removed
;; remove-from: Str -> Str
;; Examples:
(check-expect (remove-from "" #\X) "")
(check-expect (remove-from "ababA" #\a) "bbA")
(check-expect (remove-from "Waterloo" #\x) "Waterloo")

(define (remove-from s c)
  (list->string (remove-from-list (string->list s) c)))

;; Tests:
(check-expect (remove-from "X" #\X) "")
(check-expect (remove-from "A" #\y) "A")
(check-expect (remove-from "Waterloo" #\o) "Waterl")
(check-expect (remove-from "00000" #\0) "")
```

### 3.7.4 Local Helper Functions

For functions defined with a `local` block, no tests or examples are necessary, as illustrated in the following example.

```
;; (remove-short los len) produces the list of strings in los
;;   which are longer than len.
;; remove-short: (listof Str) Nat -> (listof Str)
;; Examples:
(check-expect (remove-short empty 4) empty)
(check-expect (remove-short (list "ab" "1234" "hello" "bye") 3)
              (list "1234" "hello"))

(define (remove-short los len)
  (local
    [;; (long? s) produces true if s has more
    ;; long?: Str -> Bool
    (define (long? s)
      (> (string-length s) len))])
```

```

(filter long? los)))

;; Tests
(check-expect (remove-short (list "abc") 4) empty)
(check-expect (remove-short (list "abcdef") 2) (list "abcdef"))
(check-expect (remove-short (list "ab" "1234" "hello" "bye") 1)
              (list "ab" "1234" "hello" "bye"))
(check-expect (remove-short (list "ab" "1234" "hello" "bye") 20)
              empty)\newpage

```

## 4 Racket Types

The following table lists the valid types:

Num	Any number, including rational numbers
Int	Integers: ...-2, -1, 0, 1, 2...
Nat	Natural Numbers (non-negative Integers): 0, 1, 2...
Bool	Boolean values (true and false)
Sym	Symbol ( <i>e.g.</i> , 'Red, 'Rock)
Str	String ( <i>e.g.</i> , "Hello There", "a string")
Char	Character ( <i>e.g.</i> , \#a, \#A, \#newline)
Any	Any value is acceptable
(anyof T1 T2...)	Mixed data types. For example, (anyof Int Str) can be either an Int or a Str. If false is used to indicate an unsuccessful result, use false instead of Bool to improve communication: (anyof Int false).
(listof T)	A list of <b>arbitrary length</b> with elements of type T, where T can be any valid type For example: (listof Any), (listof Int), (listof (anyof Int Str)).
(list T1 T2...)	A list of <b>fixed length</b> with elements of type T1, T2, etc. For example: (list Int Str) always has two elements: an Int (first) and a Str (second).
User-Defined	For structures and user-defined types (see section below). Capitalize your user defined types.
X, Y, ...	Matching types to indicate parameters must be of the same type. For example, in the following contract, the X can be any type, but all of the X's must be the same type: my-fn: X (listof X) -> X

## 5 User-Defined Types and Template Functions

A user-defined type can be a structure (defined with `define-struct`) or simply a descriptive definition.

- With the exception of the Racket built-in types (*e.g.*, `Posn`), every user-defined type used in your assignment must be defined at the top of your file (unless an assignment instructs you otherwise).
- User-Defined types that appear in contracts and other type definitions should be capitalized (use `My-Type`, not `my-type`).

### 5.1 Structures

When you define a structure, it should be followed by a type definition, which specifies the type for each of the fields. For example:

```
(define-struct date (year month day))  
;; A Date is a (make-date Nat Nat Nat)
```

If there are any additional requirements on the fields not specified in the type definition, a **requires** section can be added. For example:

```
(define-struct date (year month day))  
;; A Date is a (make-date Nat Nat Nat)  
;; requires: fields correspond to a valid Gregorian Calendar date  
;;           year >= 1900  
;;           1 <= month <= 12  
;;           1 <= day <= 31
```

### 5.2 Descriptive Definitions

A descriptive definition can be used to define a new user-defined type to improve the readability of contracts and other type definitions

```
;; A StudentID is a Nat  
;; requires: the value is between 1000000 and 99999999  
  
;; A Direction is an (anyof 'up 'down 'left 'right)
```

A descriptive definition can also be a mixed type, with more than one possible type:

```
;; A CampusID is one of:
;; * a StudentID
;; * a StaffID
;; * a FacultyID
;; * 'guest
```

### 5.3 Template Functions

For any non-trivial user-defined type, there is a corresponding template function. Template functions are recommended, but are not required unless otherwise specified.

For structures, the template function would have placeholders for each field:

```
(define-struct date (year month day))
;; A Date is a (make-date Nat Nat Nat)

;; my-date-fn: Date -> Any
(define (my-date-fn d)
  ( ... (date-year d) ...
        ... (date-month d) ...
        ... (date-day d) ...))
```

For mixed user-defined types, there is a corresponding `cond` question for each possible type:

```
;; A CampusID is one of:
;; * a StudentID
;; * a StaffID
;; * a FacultyID
;; * 'guest

;; my-campusid-fn: CampusID -> Any
(define (my-campusid-fn cid)
  (cond
    [(studentid? cid)      ...]
    [(staffid? cid)       ...]
    [(facultyid? cid)     ...]
    [(symbol=? 'guest cid) ...]))
```

As you combine user-defined types, their templates can also be combined. See the course handouts for more examples.



## 6 A Sample Submission

Problem: Write a Racket function `earlier?` that consumes two times and will produce `true` if the first time occurs earlier in the day than the second time, and `false` otherwise.

Note how the named constants makes the examples and testing easier, and how the introduction of the `time->seconds` helper function makes the implementation of `earlier?` much more straightforward.

```
;;
;; *****
;;   Arthur Dent (12345678)
;;   CS 135 Fall 2014
;;   Assignment 03, Problem 1
;; *****
;;

(define-struct time (hour minute second))
;; A Time is a (make-time Nat Nat Nat)
;; requires: 0 <= hour < 24
;;           0 <= minute, second < 60

;; my-time-fn: Time -> Any
(define (my-time-fn t)
  ( ... (time-hour t) ...
        ... (time-minute t) ...
        ... (time-second t) ... ))

;; useful converters
(define seconds-per-minute 60)
(define minutes-per-hour 60)
(define seconds-per-hour (* seconds-per-minute minutes-per-hour))

;; useful constants for examples and testing
(define midnight (make-time 0 0 0))
(define just-before-midnight (make-time 23 59 59))
(define noon (make-time 12 0 0))
(define eight-thirty (make-time 8 30 0))
(define eight-thirty-and-one (make-time 8 30 1))
```

```

;; (time->seconds t) Produces the number of seconds since midnight
;;   for the time t
;; time->seconds: Time -> Nat
;; Examples:
(check-expect (time->seconds midnight) 0)
(check-expect (time->seconds just-before-midnight) 86399)

(define (time->seconds t)
  (+ (* seconds-per-hour (time-hour t))
     (* seconds-per-minute (time-minute t))
     (time-second t)))

;; Tests:
(check-expect (time->seconds noon) 43200)
(check-expect (time->seconds eight-thirty) 30600)
(check-expect (time->seconds eight-thirty-and-one) 30601)

;; (earlier? time1 time2) Determines if time1 occurs before time2
;; earlier?: Time Time -> Bool
;; Examples:
(check-expect (earlier? noon just-before-midnight) true)
(check-expect (earlier? just-before-midnight noon) false)

(define (earlier? time1 time2)
  (< (time->seconds time1) (time->seconds time2)))

;; Tests:
(check-expect (earlier? midnight eight-thirty) true)
(check-expect (earlier? eight-thirty midnight) false)
(check-expect (earlier? eight-thirty eight-thirty-and-one) true)
(check-expect (earlier? eight-thirty-and-one eight-thirty) false)
(check-expect (earlier? eight-thirty-and-one eight-thirty-and-one) false)

```

## 7 Mutation (CS 116 and CS 136)

Mutation allows for a function to do more than manipulate its arguments. With mutation, a function can change the value of a global or state variable or one of its arguments, or have other effects. Mutation is explored in some depth in Section 36.4 of *How to Design Programs*. While allowing programs to do even more, the introduction of mutation requires changes to some steps of our design recipe, along with the introduction of a new step.

A sample submission involving mutation is included in a later section.

## 7.1 Purpose

Without mutation, the purpose of a function is simply what value the function produces. With mutation, it is possible that a function does not produce a new value, but instead changes another variable in some way. Some functions will produce a value and also mutate other variables. When a variable (either a state variable or one of the function arguments) is mutated, this should be included in the purpose statement. The purpose statement does not need to provide all the details of how the mutation is performed, but should describe (using parameter names as needed) the change itself.

## 7.2 Effects

To ensure that it is very clear that a variable has been changed by a function, a new design recipe step is included: Effects. In this step, you must explicitly state what state variable or parameter is changed when the function is called. The description of the change itself should be included, as noted above, in the purpose statement for the function.

## 7.3 Contract

When a function performs only a mutation, but does not produce a value, a new type `Void` is used as the produced type in a contract. Also, with the introduction of mutation, it is possible that a function consumes no parameters. In this case, write the consumed type as `Void` as well.

## 7.4 Examples and Tests

When a function mutates a variable, examples and tests must be revised to account for the passage of time: variables may have a different value before and after the function is called. Examples and tests will still be written using `check-expect` or `check-within`, as required by the types involved, but must also include the use of a `begin` expression to handle the “before” and “after”.

An example or test for a function involving mutation has the following format:

```
;; Initialization: define variables needed for testing
(check-expect
  (begin
    ;; Step 1: set values specific to this test
    ;; Step 2: call the function
    ;; Step 3: determine result of the test
```

```
)  
;; include expected value of begin expression  
)
```

The result of the test may simply be the variable which has been changed. In this case, the expected value is the expected new value due to the mutation. However, it is possible that the function may produce a value and mutate several variable. In this case, create a list containing all the information that should be verified by your test.

## 7.5 Sample Submission: Mutation of a State Variable

```
;;
;; *****
;;   Arthur Dent (12345678)
;;   CS 115 Fall 2014
;;   Assignment 03, Problem 1
;; *****
;;

;; State variables
(define x 100)
(define y 0)

;; (change-x-and-y ch) increases x by 1 if ch>0,
;;   decreases x by 1 if ch<0, and updates y to
;;   be the current value of x if ch=0.
;; Effects: x is changed if ch is not 0, and
;;   y is changed if ch is 0.
;; change-x-and-y: Num -> Void
;; Example:
(check-expect (begin (set! x 5)
                     (set! y 12)
                     (change-x-and-y -3)
                     (list x y))
              (list 4 12))

(define (change-x-and-y ch)
  (cond [(< ch 0) (set! x (- x 1))]
        [(> ch 0) (set! x (+ x 1))]
        [else     (set! y x)]))

;; Test with ch < 0
(check-expect (begin (set! x 33)
                     (set! y -12)
                     (change-x-and-y 7)
                     (list x y))
              (list 34 -12))

;; Test with ch = 0
(check-expect (begin (set! x -5)
                     (set! y 2)
                     (change-x-and-y 0)
                     (list x y))
              (list -5 -5))
```

## 7.6 Sample Submission: Mutation of a Parameter (a Structure)

```
;;
;; *****
;;   Arthur Dent (12345678)
;;   CS 115 Fall 2014
;;   Assignment 03, Problem 1
;; *****
;;

;; Create a posn, test-p, to use for testing
(define test-p (make-posn 0 0))
;; Create a variable, answer, to store the answer of the test calls
(define answer 0)

;; (scale-and-max p mult) updates the x and y fields of p by
;;   multiplying each by mult, and then produces the larger of
;;   the new values of the two fields
;; Effects: Mutates the x and y fields of p
;; scale-and-max: Posn Num -> Num
;; Examples:
(check-expect
 (begin
  (set! test-p (make-posn 4 -5))
  (set! answer (scale-and-max test-p 2))
  (list answer test-p))
 (list 8 (make-posn 8 -10)))

(define (scale-and-max p mult)
  (begin
   (set-posn-x! p (* mult (posn-x p)))
   (set-posn-y! p (* mult (posn-y p)))
   (max (posn-x p) (posn-y p))))

;; Testing - create variables for testing

;; mult = 0
(check-expect
 (begin
  (set! test-p (make-posn 1.5 -2.25))
  (set! answer (scale-and-max test-p 0))
  (list answer test-p))
 (list 0 (make-posn 0 0)))
;; mult < 0
(check-expect
 (begin
  (set! test-p (make-posn -3 9))
  (set! answer (scale-and-max test-p -10))
  (list answer test-p))
 (list 30 (make-posn 30 -90)))
```

```
;; posn is origin
(check-expect
 (begin
  (set! test-p (make-posn 0 0))
  (set! answer (scale-and-max test-p 123456.7890))
  (list answer test-p))
 (list 0 (make-posn 0 0)))
```

## 8 Changes for Python (CS 116)

### 8.1 Purpose

As before, your purpose statements should briefly explain what the function does using parameter names to show the relationship between the input and the function's actions. The actions may include producing a value, mutations, the use of `input` and `print`, as well as any file operations.

### 8.2 Effects

When a function's role involves anything in addition to, or instead of, producing a value, it must be noted in an effects statement. This includes: mutations, the use of `input` and `print`, as well as any file operations.

### 8.3 Contract

Most of the types that were used in contracts for Racket will carry over to Python, with a few notable exceptions. The following table lists the valid Python types.

Float	Any non-integer value
Int	Integers: ...-2, -1, 0, 1, 2...
Nat	Natural Numbers (non-negative Integers): 0, 1, 2...
Bool	Boolean values (True and False)
Str	String ( <i>e.g.</i> , "Hello There", "a string")
Any	Any value is acceptable
(anyof T1 T2...)	Mixed data types. For example, (anyof Int Str) can be either an Int or a Str. If False is used to indicate an unsuccessful result, use False instead of Bool to improve communication: (anyof Int False).
(listof T)	A list of <b>arbitrary length</b> with elements of type T, where T can be any valid type For example: (listof Any), (listof Int), (listof (anyof Int Str)).
(list T1 T2...)	A list of <b>fixed length</b> with elements of type T1, T2, etc. For example: (list Int Str) always has two elements: an Int (first) and a Str (second).
User_Defined	For structures and user-defined types. Capitalize your user defined types.
X, Y, ...	Matching types to indicate parameters must be of the same type. For example, in the following contract, the X can be any type, but all of the X's must be the same type: my-fn: X (listof X) -> X
(dictof T1 T2)	A dictionary with keys of type T1 and associated values of type T2.
File	A File object.
None	The None value designates when a function does not include a return statement.

There are a few significant omissions from the Racket types. Note that:

- Python does not have a symbol type. You should use strings or numbers in the role of symbolic constants as needed.
- Python does not have a character type. Instead, use strings of length 1.
- Python does not use the `Num` type. If a value can be either an integer or non-integer value, use `(anyof Int Float)`.

If there are additional restrictions on the consumed types, continue to use a requirements statement following the contract. If there are any requirements on



data read in from a file or from standard input, these should be included in the requirements statement as well.

## 8.4 Examples

Unlike in Racket, examples in Python cannot be written as code using the provided `check` module. Unfortunately, the function calls in the `check` functions cannot come before the actual function definitions. Therefore, instead of writing examples as code, we must write them as comments. The format of the example depends on whether or not the function has any effects.

- If the function produces a value, but has no effects, then the example can be written as a function call, with its expected value, e.g.

```
## Example:  
## combine-str-num("hey", 3) => "hey3"
```

- If the function involves mutation, the example needs to reflect what is true before and after the function is called.

```
## Example:  
## If lst1 is [1,-2,3,4], after calling mult_by(lst1, 5),  
##     lst1 = [5,-10,15,20]
```

- If the function involves some other effects (reading from keyboard or a file, or writing to screen or a file), then this needs to be explained, in words, in the example as well.

```
## Example:  
## If the user enters Waterloo and Ontario when prompted by  
##     enter_hometown(), the following is written to the screen:  
## Waterloo, Ontario
```

- If the function produced a value and has effects, you will need to use a combination of the above situations.

```
## Example: If the user enters Smith when prompted,  
##     enter_new_last("Li", "Ha") => "Li Smith"  
##     and the following is printed to "NameChanges.txt":  
## Ha, Li  
## Smith, Li
```

## 8.5 Testing

Python doesn't present us with a function like `check-expect` in Racket for testing our programs. To emulate this functionality, you can download the `check.py` module from the course website. This module contains several functions designed to make the process of testing Python code straightforward, so that you can focus on choosing the best set of test cases. You must save the module in the same folder as your program, and include the line `import check` at the beginning of each Python file that uses the module. You do not need to submit `check.py` when you submit your assignment.

Our tests for most functions will consist of up to seven parts; you only need to include the parts that are relevant to the function you are testing. More detail on these parts can be found following the summary. Note that steps 2-5 must come before steps 6-7 for the test to work properly.

More detail on these parts can be found following the summary. Note that steps 2-5 must come before steps 6-7 for the test to work properly.

1. Write a brief description of the test as a comment (including a description of file input when appropriate).
2. If there are any state variables to consider, set them to specific values.
3. If you expect user input from the keyboard, call `check.set_input(lst)`.
4. If you expect your function to print anything to the screen, call `check.set_screen`.
5. If your function writes to any files, call `check.set_file_exact(actual, expected)` or `check.set_file(actual, expected)`, as needed.
6. Call either `check` function with your function and the expected value (which may be `None`).
7. If the effects for the function include mutating state variables or parameters, call the appropriate `check` function once for each mutated value.

Additional information about testing:

- Step 1: If your function reads from a file, you will need to create the file (using a text editor like Wing IDE) and save it in the same directory as your `|aXXqY.pyl` files. You do not need to submit these files when you submit your code, but any test that reads from a file should include a comment with a description of what is contained in the files read in that test.

- Step 2: You should always set the values of every state variable on every test, in case your function inadvertently mutates their values.
- Step 3: If the value to test is a call to a function with screen output, you need to use `check.set_screen` (which consumes a string describing what you expect your function to print) before running the test. Screen output has no effect on whether the test is passed or failed. When you call `check.set_screen`, the next test you run will print both the output of the function you are testing, and the expected output you gave to `check.set_screen`. You need to visually compare the output to make sure it is correct. As you are the one doing the visual comparison, you also the one to determine the format of the string passed to `check.set_screen`.
- Step 4: If your function uses keyboard input (such as `input`), you will need to use `check.set_input` before running the test. This function consumes a list of strings, which it will used as input the next time you run your function. You do not need to do any typing for `input` when you run your tests. You will get an error if the list you provide to `check.set_input` is the wrong length.
- Step 5: If your function writes to a file, you will need to use either `check.set_file` or `check.set_file_exact` before running the test. Each of these functions consumes two strings: the first is the name of the file that will be produced by the function call in step 6, and the second is the name of a file identical to the one you expect to be produced by the test. You will need to create the second file yourself using a text editor.

The next call to `check.expect` or `check.within` will compare the two files. If the files are the same, the test will print nothing; if they differ, the test will print which lines don't match, and will print the first pair of differing lines for you to compare. If you use `check.set_file`, the two files are compared with all white space removed from both files. If you use `check.set_file_exact`, the files must be character for character identical in order to be considered the same.

- Steps 6 and 7: The two main functions included in `check` are `check.expect` and `check.within`; these functions will handle the actual testing of your code. You should only use `check.within` if you expect your code to produce a floating point number (or if one part of the produced value is a floating point value); in every other case, you should use `check.expect`. When

testing a function that produces nothing, you should use `check.expect` with `None` as the expected value.

- `check.expect` consumes three values: a string (a label for the test, such as “Question 1 Test 6”), a value to test, and an expected value. You will pass the test if the value to test equals the expected value; otherwise, it will print a message that includes both the value to test and the expected value, so that you can compare the results.
- `check.within` consumes four values: a string (a label of the test, such as “Question 1 Test 6”), a value to test, an expected value, and a tolerance. You will pass the test if the value to test and the expected value are close to each other (to be specific, if the absolute value of their difference is less than the tolerance); otherwise, it will print a message that includes both the value to test and the expected value, so that you can compare the results. In the case of lists or objects containing floating point values, the test will fail if any one of these components is not within the specified tolerance.

Note that both `check.expect` and `check.within` only print a message if your test fails: if nothing gets printed, the test passed.

## 8.6 Additional changes

There are a few other changes to be aware of:

- The dash (“-”) character cannot be used in an identifier in Python. You should instead use an underscore (“\_”) where a dash would have been used (e.g. `tax_rate`).
- Our Racket programs were generally short enough that the design recipe steps provided sufficient documentation about the program. It is more common to add comments in the function body when using imperative languages like Python. While there will not usually be marks assigned for internal comments, it can be helpful to add comments so that the marker can understand your intentions more clearly. In particular, comments can be useful to explain the role of local variables, if statements, and loops.
- Helper functions should not be defined locally in your Python programs (they can be, but the course style is to avoid local function definitions). However, tests and examples are not required for any helper functions in CS 116.

## 8.7 Sample Python Submissions

### 8.7.1 Mutation

```
##
## *****
## Arthur Dent (12345678)
## CS 115 Fall 2014
## Assignment 03, Problem 4
## *****
##

import check

# add_one_to_evens(lst) mutates lst by adding 1 to each even value
# add_one_to_evens: (listof Int) -> None
# Effects: Mutates lst
# Examples: if L = [], add_one_to_evens(L) => None,
#           then L = [].
#           if L = [3,5,-18,1,0], add_one_to_evens(L) => None,
#           then L = [3,5,-17,1,1]

def add_one_to_evens(lst):
    for i in range(len(lst)):
        if lst[i]%2==0:
            lst[i] = lst[i] + 1

# Test 1: Empty list
L = []
check.expect("Q1T1", add_one_to_evens(L), None)
check.expect("Q1T1 (L)", L, [])

# Test 2: List of one even number
L = [2]
check.expect("Q1T2", add_one_to_evens(L), None)
check.expect("Q1T2 (L)", L, [3])

# Test 3: List of one odd number
L = [7]
check.expect("Q1T3", add_one_to_evens(L), None)
check.expect("Q1T3 (L)", L, [7])

# Test 4: General case
L = [1,4,5,2,4,6,7,12]
check.expect("Q1T4", add_one_to_evens(L), None)
check.expect("Q1T4 (L)", L, [1,5,5,3,5,7,7,13])
```

### 8.7.2 Keyboard Input and Screen Output

```
##
```

```

## *****
## Arthur Dent (12345678)
## CS 115 Fall 2014
## Assignment 03, Problem 4
## *****
##

import check

## mixed_fn(n, action) produces 2*n if action is "double",
## n/2 if action is "half", and produced None otherwise.
## If action is "string", the user is prompted to enter a string
## which is then n copies of that string is printed on one line.
## For any other action, "Invalid action" is printed to the screen.
## Effects: If action is "string", a string is read from standard input,
## and copies of the string is printed. For any other action besides
## "string", "half", or "double", "Invalid action" is printed.
## Examples:
## mixed_fn(2,"double") => 4
## mixed_fn(11, "half") => 5.5
## mixed_fn(6,"oops") => None and prints "Invalid action"
## mixed_fn(3,"string") => None, if the user inputs "a"
## then "aaaaa" is printed and nothing is produced

def mixed_fn(n,action):
    if action=="double":
        return 2*n
    elif action=="half":
        return n/2
    elif action=="string":
        s = input("enter a non-empty string: ")
        print (s*n)
    else:
        print ("Invalid action")

# Testing
# Test 1: action == "double"
check.expect("Q2T1", mixed_fn(2, "double"), 4)
# Test 2: action=="half", odd number
check.within("Q2T2", mixed_fn(11, "half"), 5.5, .001)
# Test 3: action=="half", even number
check.within("Q2T3", mixed_fn(20, "half"), 10.0, .001)
# Test 4: action=="string"
check.set_input(["hello"])
check.set_screen("hellohellohello")
check.expect("Q2T4", mixed_fn(3, "string"), None)
# Test 5: invalid action
check.set_screen("Invalid action")
check.expect("Q2T5", mixed_fn(2, "DOUBLE"), None)

```

### 8.7.3 File Input and Output

```

##
## *****
## Arthur Dent (12345678)
## CS 115 Fall 2014
## Assignment 03, Problem 4
## *****
##

## file_filter(fname, minimum) reads opens the file fname,
## reads in each integer, and writes each integer > minimum
## to a new file, "summary.txt".
## Effects: Opens, reads, and closes file called fname
## Opens, writes, and closes file called "summary.txt"
## file_filter: String Int -> None
## requires: 0 <= minimum <= 100
## Examples:
## If "ex1.txt" is empty, then file_filter("ex1.txt", 1)
## will create an empty file named summary.txt
## If "ex2.txt" contains 35, 75, 50, 90 (one per line) then
## file_filter("ex2.txt", 50) will create a file
## named "summary.txt" containing 75, 90 (one per line)

def file_filter(fname, minimum):
    infile = open(fname, "r")
    lst = infile.readlines()
    infile.close()
    outfile = open("summary.txt", "w")
    for line in lst:
        if int(line.strip()) > minimum:
            outfile.write(line)
    outfile.close()

# Testing
# Test 1: empty file (example 1)
check.set_file("summary.txt", "empty.txt")
check.expect("t1", file_filter("empty.txt", 40), None)
# Test 2: example 2
check.set_file("summary.txt", "eg2-summary.txt")
check.expect("t2", file_filter("eg2.txt", 50), None)
# Test 3: file contains one value, it is > minimum
check.set_file("summary.txt", "one-value.txt")
check.expect("t3", file_filter("one-value.txt", 20), None)
# Test 4: file contains one value, is it < minimum
check.set_file("summary.txt", "empty.txt")
check.expect("t4", file_filter("one-value.txt", 80), None)
# Test 5: file contains one value, is it minimum
check.set_file("summary.txt", "empty.txt")
check.expect("t5", file_filter("one-value.txt", 50), None)
# test 6: file contains 1-30 on separate lines
check.set_file("summary.txt", "sixteen-thirty.txt")
check.expect("Q3T4", file_filter("thirty.txt", 15), None)

```

## 8.7.4 Class

```
##
## *****
## Arthur Dent (12345678)
## CS 115 Fall 2014
## Assignment 03, Problem 4
## *****
##

import check

class time:
    ## A Time value is a time(Nat Nat Nat)
    ## requires: 0 <= hour < 24
    ##           0 <= minute, second < 60

    ## Constructor: Create a Time object by calling time(h,m,s),
    ## requires: 0 <= hour < 24, and 0 <= minute, second < 60
    def __init__(self, h, m, s):
        self.hour = h
        self.minute = m
        self.second = s

    ## String Representation: Implicitly called by print(t), where t is of type Time
    def __repr__(self):
        return "{0:.2}:{1:.2}:{2:.2}".format(self.hour, self.minute, self.second)

    ## Equality test: Implicitly called when calling t1 == t2 or t1 != t2, where
    ## t1 is a Time value, and t2 is of type Any.
    def __eq__(self, other):
        return instance(other, time) and \
            self.hour == other.hour and \
            self.minute == other.minute and \
            self.second == other.second

    ## useful converters
    seconds_per_minute = 60
    minutes_per_hour = 60
    seconds_per_hour = seconds_per_minute * minutes_per_hour

    ## useful constants for examples and testing
    midnight = time( 0, 0, 0)
    just_before_midnight = time (23, 59, 59)
    noon = time (12, 0, 0)
    eighty_thirty = time( 8, 30, 0)
    eighty_thirty_and_one = time (8, 30, 1)

    ##;; time_to_seconds (t) produces the number of seconds since midnight
    ##;; for the time t
    ## time_to_seconds: Time -> Nat
```



```

##;; Examples:
## time_to_seconds (midnight) => 0
## time_to_seconds (just-before-midnight) => 86399

def time_to_seconds (t):
    return (seconds_per_hour * t.hour) + seconds_per_minute * t.minute + t.second

##; Tests:
check.expect ("tts1", time_to_seconds (noon), 43200)
check.expect ("tts2", time_to_seconds (eight_thirty), 30600)
check.expect ("tts3", time_to_seconds (eight_thirty_and_one), 30601)

## earlier (time1, time2) determines if time1 occurs before time2
## earlier?: Time Time -> Bool
## Examples:
## earlier( noon, just-before-midnight) => true
## earlier(just-before-midnight, noon) => false

def earlier(time1, time2):
    return time_to_seconds (time1) < time_to_seconds (time2)

## Tests:
check.expect ("e1", earlier( midnight, eight_thirty) ,True)
check.expect ("e2", earlier( eight_thirty, midnight) ,False)
check.expect ("e3", earlier( eight_thirty, eight_thirty_and_one), True)
check.expect ("e4", earlier( eight_thirty_and_one, eight_thirty), False)
check.expect ("e5", earlier( eight_thirty_and_one, eight_thirty_and_one),False)

```

## 9 Changes for C (CS 136)

Note: This style has been changed for Winter 2015. An official Piazza post will be made to address any difficiencies in this guide. Please make a separate piazza post to ask questions or alert the course staff to any problems you encounter.

The general guidelines at the beginning of this guide regarding whitespace, line lengths, having meaningful paramter names, etc. also apply to C.

### 9.1 Comments

In the C99 standard, there are two types of comment syntaxes:

- Block comments, between “/\*” and “\*/”, allow comments to span multiple lines.

```
/* This symbol denotes the start of a block comment.  
   Anything inside these two symbols will be commented out.  
   The following symbol denotes the end of a block comment:  
*/  
  
int n = 0; /* they can also be used on one line */
```

- Inline comments, following “//”, allow for comments to extend to the end of one line.

```
// Comment on a line by itself.  
  
int n = 0; // after some code
```

## 9.2 Headers

As with Racket, your files should start with a header to identify yourself, the term, the assignment and the problem (or module). There is no specifically required format, but it should be clear and assist the reader. You can use a block comment or inline comments.

```
/******  
  Arthur Dent (12345678)  
  CS 136 Fall 2014  
  Assignment 03, Problem 4  
*****/  
  
// =====  
//   Arthur Dent (12345678)  
//   CS 136 Fall 2014  
//   Assignment 03, Problem 4  
// =====
```

## 9.3 Function Documentation

Every function you write should have a purpose statement. The format in C is nearly identical to the Racket format.

```
;; RACKET:  
;; (sum n) produces the sum of the numbers from 1..n  
  
// C:  
// sum(n) produces the sum of the numbers from 1..n
```

Unlike in Racket, in C **you do not have to indicate the contract *types***. However, the following components of a contract must be provided if appropriate.

- **requires:** indicates additional restrictions on the parameters, and possibly any additional state requirements.

```
// requires: n > 0  
//           init_function() has been previously called
```

- **effects:** indicates any side effects, which include I/O (printing or scanning) or the modification/mutation of any global variables or parameters. The explanation does not have to be excessively detailed, but if a function has a side effect, there should be an *effects* section.

```
// effects: prints a message
//           modifies the global variable count
```

- **time:** indicates the efficiency of the function.

```
// time: O(n), where n is the length of the list
```

NOTE: the *effects* and *time* components of the contract are only required once they have been introduced in the lectures.

// CODING EXAMPLE:

```
// negate_array(a, n) negates the elements of array a (of length n)
// requires: 0 <= n <= length of a
// effects: modifies the contents of a
// time: O(n)
```

```
void negate_array(int a[], int n) {
    assert(n >= 0);

    for (int i = 0; i < n; ++i) {
        a[i] = -a[i];
    }
}
```

NOTE: whenever feasible, `assert` any conditions that appear in the *requires* section.

## 9.4 Indentation and Whitespace

There are a variety of different opinions on good C style. In CS136, you may adopt any style but you should adhere to consistent usage of indentation, whitespaces, and code layout with the goal of making your code more readable. Some of the key ideas are as follows.

- No more than one C statement per line.
- Add a space after commas in function calls and semicolons in `for` loops.
- Add spaces around operators in expressions to make them easier to read.
- Break long lines of code (more than 80 columns) onto 2 or more lines.
- Your code should be presented so that a matching pair of braces is easy to identify and all lines of code within a set of braces should be indented consistently.
- Indentation should be consistently 2 or 3 spaces (not tab characters).

```
// The previous negate_array used a popular style of indentation.  
// The following is an ALTERNATE indentation style:
```

```
void negate_array(int a[], int n)  
{  
    assert(n >= 0);  
  
    for (int i = 0; i < n; ++i)  
    {  
        a[i] = -a[i];  
    }  
}
```

Both styles are acceptable. NOTE that in C (unlike Racket) it is **not** good style to place all ending braces on one line (}}}).

## 9.5 Naming Conventions for Identifiers

In Scheme, the words that made up the identifier often contain hyphens or punctuation such as “?” or “!”. In C, the underscore character is often used to separate words within an identifier. No separating character, or a change in case are also common conventions. An identifier in Scheme named `one-word?` might look like `one_word`, `is_one_word`, `isoneword` or `isOneWord` in C. The choice is yours on which convention you would like to use but try to be consistent with one style.

## 9.6 Code Organization

The following example helps to illustrate how your files should be organized (both `.h` interface files and `.c` implementation files).

```
// mymodule.h

/*****
  PERSONAL DETAILS
  *****/

// MODULE DESCRIPTION (FOR USERS)

// ANY #INCLUDES
#include <stdbool.h>

// GLOBAL VARIABLES (RARE)

extern int module_version;

// FUNCTION INTERFACES: DOCUMENTATION AND DECLARATION

// negate_array(a, n) negates the elements of array a (of length n)
// requires: 0 <= n <= length of a
// effects: modifies the contents of a
// time: O(n)

void negate_array(int a[], int n);
```

```

// mymodule.c

/*****
PERSONAL DETAILS
*****/

// #INCLUDES ... you should #include your own header
#include "mymodule.h"

// GLOBAL VARIABLES

int module_version = 1;

// FUNCTION IMPLEMENTATIONS:
// Documentation for provided functions appears in the .h file

// see mymodule.h for interface documentation
void negate_array(int a[], int n) {
    assert(n >= 0);

    for (int i = 0; i < n; ++i) {
        a[i] = -a[i];
    }
}

// Full purpose and contract information for local functions
// ...
int my_helper(int n) {
    //...
}

```