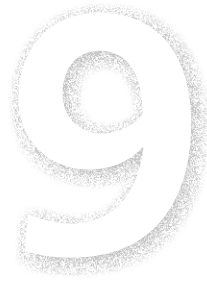

Writing Tested Code

Everyone likes to think that they write perfect code the first time. Rust even makes it easier to write correct code since its strong type system eliminates whole classes of errors such as buffer overflows and use-after-free issues. However, there are always logical errors that can creep into code written by the best-intentioned developer. Who has not inverted the logic of a test or returned the wrong value out of a function? While Rust cannot provide implicit protections for those types of bugs, Rust does integrate a testing framework into the toolchain that makes it easy for developers to write test functions.

If you are only writing code for your own use, you can probably get away with just writing unit tests to make sure that individual functions are written correctly. However, if you are writing code that is being shared with others (perhaps you are publishing a crate on `crates.io`), then you may want to provide more types of tests such as integration tests that test the boundaries between your modules, documentation tests that verify that the examples you provide in your documentation is correct, or benchmark tests that assist you in performance testing. As we will explore in this chapter, Rust's test framework supports developers in writing all of these types of tests.

To demonstrate how Rust assists you in writing tests, we will first walk through an easy-to-understand data structure API for a minimum binary heap that lives in a fictional `binheap` crate and use that as a structure on which we can hang our tests.

A MINIMUM BINARY HEAP

In computer science, a minimum heap is a tree-like data structure that satisfies the heap property – given a tree, the parent of the tree has a strictly smaller value than the children of that parent. But what does the heap property actually mean? Well, take a minimum binary heap, for example. A minimum binary heap is a binary tree structure where a parent can have 0, 1, or 2 children and the value of each child node is strictly less than the value of the parent node. However, a minimal binary heap is only partially sorted — while a binary search tree is absolutely ordered – meaning that the order of insertion into the tree does not affect the order of the tree and a depth-first traversal of the tree will produce the contents of the tree in order, the same does not hold for a minimal binary heap. Only upon extraction from the heap will you produce an absolute ordering of the contents of the heap.

Why would you want to use a heap, then? What good is a partially-ordered tree? Typically, a heap is used to implement a priority queue. In a priority queue, the absolute order of the queue does not matter. Instead, we always want to extract the item with the lowest priority from the queue. So, why would we want to take the extra cycles keeping the tree absolutely ordered when all we care about is that at the root of the tree is the minimum value? A minimum binary heap gives us just this property. For example, take a look at Figure 9-1.

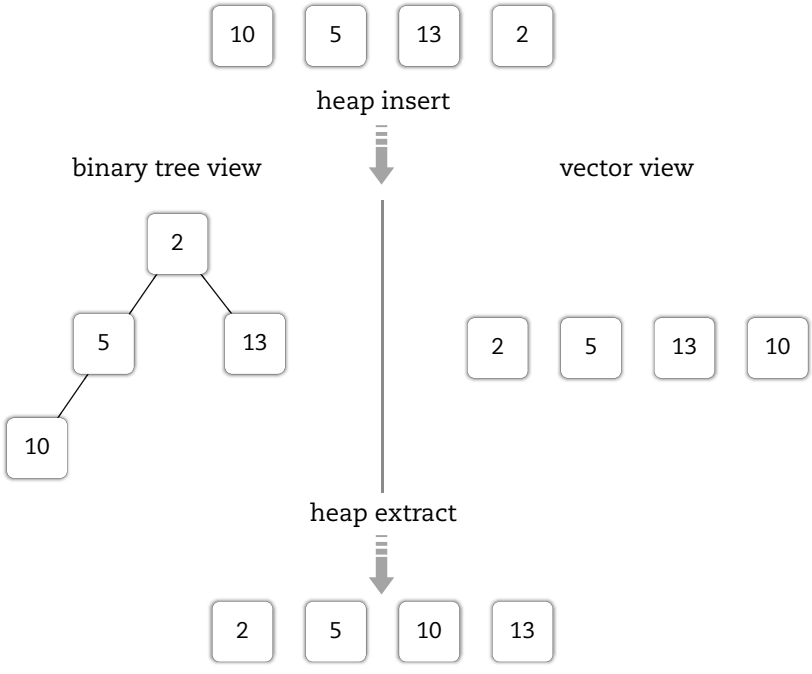

Figure 9-1: Insertion into and extraction from a binary heap

At the top of the figure, we have a stream of data that we want to insert into our priority queue (our heap). We insert them into the heap from left to right and produce the two views of our heap – one as a binary tree with a very explicit node structure and another laid out as a vector in memory. Finally, at the bottom of the figure, we can extract each element from the heap, producing the ordered view of the values from least to greatest. From the figure, we can gleam another property from our heap – the absolute root of the tree must be the smallest value in the heap.

How we implement the heap is not relevant for the purposes of this chapter, but we do need to take a look at the interface that the heap provides so that we can test it appropriately.

```
use std::fmt::Debug;
#[derive(Debug, Clone)]
pub struct MinBinaryHeap<T: Debug + Ord> {
    buf: Vec<T>,
}

impl<T: Debug + Ord> MinBinaryHeap<T> {
    pub fn new() -> MinBinaryHeap<T> {
        MinBinaryHeap { buf: Vec::new() }
    }

    fn left(&self, idx: usize) -> Option<usize> {
       ...
    }
    fn right(&self, idx: usize) -> Option<usize> {
       ...
    }
    fn parent(&self, idx: usize) -> usize {
       ...
    }
    pub fn root(&self) -> Option<&T> {
        self.buf.get(0)
    }
    pub fn insert(&mut self, value: T) {
       ...
    }
    pub fn extract(&mut self) -> Option<T> {
       ...
    }
}
```

From this interface, we can determine a few things about the implementation that might be useful for testing purposes. First, we are backing the heap with a simple vector instead of building out a full tree structure. There are non-public methods that let us get at the index of a parent node or, given a parent node, the left and right indices of the children nodes. Since the child nodes may not exist, we return those indices as `Options`. Our public interface consists of a method `root` that allows us to peek at the root, or minimal, value of the heap. Additionally, we have public methods – `insert` and `extract` – that allow us to add and remove values from the heap, respectively.

Given those methods, what are some simple tests that we can add to verify that our heap code is correct? For that matter, how do we even *add* tests to our Rust code?

SIMPLE UNIT TESTING

The simplest kind of tests that we can add to Rust code is the unit test. A unit test is a test of the smallest possible unit of code. In the case of Rust code, our units are typically individual functions or methods that are implemented on a data structure. To support unit tests, Rust has special markers that you can put on functions to declare that those functions are tests that can be extracted out and run by Cargo when you run the `cargo test` command. When you execute the `cargo test` command, you will see output similar to the following:

```
$ cargo test
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running /Users/jonathan/Documents/Programming Rust/src/chapter_09/
binheap/target/debug/deps/binheap-148856f953ef21b1

running 4 tests
test tests::new_heap_empty ... ok
test tests::insert_adds_value ... ok
test tests::extract_pulls_min_value ... ok
test tests::new_heap_empty_unwrap ... ok
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured
```

This example test run lists how many test functions were found in the crate, the name of each of the test functions with their pass state, and a summary line at the end. In this test run, our crate has four test functions, and they all passed. Now that we know how to run the unit tests that we want to create, we can demonstrate creating one of those simple unit tests:

```
❶ #[cfg(test)]
  mod tests {
     use super::*;

❷    #[test]
     fn new_heap_empty() {
        let mut heap: MinBinaryHeap<usize> = MinBinaryHeap::new();
        assert_eq!(heap.buf.len(), 0);
        assert_eq!(heap.extract(), None);
     }

     #[test]
     #[should_panic]
     fn new_heap_empty_unwrap() {
         let mut heap: MinBinaryHeap<usize> = MinBinaryHeap::new();
         heap.extract().unwrap();
     }
}
```

The above code example should live in the same source file as the implementation for the heap. The first interesting bit of the example is marked ❶ – the `#[cfg(test)]` line. When Cargo builds your test programs, it enables the `test` feature that can be detected by the `#[cfg(test)]` line. In this case, the `tests` module is only compiled in the source file when the `test` feature is enabled so that test code does not exist in your actual module. Inside of the `tests` module, we are importing all of the symbols that are defined in the parent module (the rest of the source file). This will let us easily call the functions that we want to test.

The other interesting bit in the example is marked ❷ – the `#[test]` line. This line tells the Rust compiler that the following function is a test function and should be run as part of the test framework. Inside of that test, we simply create a new `MinBinaryHeap` for `usize` values and make sure that it is empty. Since we know that the heap is backed by a vector, we can just assert that the vector has a 0 length. Additionally, for an empty heap, we want to make sure that the `extract` method correctly returns `None`, so we use another assertion. In the test, the `assert_eq!` macro is used to check whether two expressions are equal to each other. If they are not, the macro panics. Rust assumes that a panic corresponds to a failing test. If the test function runs to completion, then the test is considered passing.

We could have re-written those assertions using the general `assert!` macro like the following:

```
assert!(heap.extract().is_none(), "got unexpected value");
```

The general `assert!` macro takes any expression as the first argument and, if it evaluates to be true, succeeds. Otherwise, the `assert!` macro panics and provides the optional second argument as the panic message.

In the second test function we added, there is another directive passed to the compiler, the `#[should_panic]` directive. That directive inverts Rust's assumption that a panic corresponds to a failing test. If you specify that the test should panic, then a panic corresponds to a passing test and, if the function does not panic, then the test failed. If you are writing tests where you are wanting to make sure that your functions correctly return an error state, consider putting `#[should_panic]` on the test and simply unwrapping the return. It can make your test code much more concise and easier to understand.

For brevity, we did not show an implementation for unit tests for the `insert` and `extract` functions. However, that leads to an interesting problem. How can we test, in general, that those methods are doing the right thing? In a unit test, we can create several contrived test cases that will push at the boundary conditions of our functions, but how can we be sure that we are really covering all of our bases? What if, instead of writing test cases, we simply tested that our functions generally met the requirements they are supposed to implement? We can do that with property-based unit testing.

PROPERTY-BASED UNIT TESTING

The main idea behind property-based testing is to make general assertions about your code by writing property functions that will succeed when your code is valid. Then, you use a property-testing library to automatically generate test cases and run them against your properties to ensure that they hold. The most well-known library for property-based testing is QuickCheck — a Haskell library for automatically generating test cases. QuickCheck has been ported to many programming languages, including Rust. We will use the `quickcheck` crate to generate heap test cases by implementing the `Arbitrary` trait for our data structure.

```
extern crate quickcheck;
use quickcheck::{Arbitrary, Gen};
impl<T: Arbitrary + Debug + Ord> Arbitrary for MinBinaryHeap<T> {
    fn arbitrary<G: Gen>(g: &mut G) -> Self {
        let mut src: Vec<T> = Vec::arbitrary(g);
        let mut arb = Self::new();
        for value in src.drain(..) {
            arb.insert(value);
        }
        arb
    }
}
```

The example code above would be included in our `tests` modules in the crate source file. In it, we are pulling in the `Arbitrary` and `Gen` traits. The `Arbitrary` trait is used to specify that the data structure can be automatically generated (that we can produce an arbitrary instance of the data structure). The `Gen` trait represents a generator — a trait that produces random values and can be used to help generate our `Arbitrary` values. In addition to these two traits, the `quickcheck` crate provides implementations

of the `Arbitrary` trait for many types in the standard library, including the primitives, vectors, strings, maps, tuples, and even `Option` and `Result` types. That makes our implementation for `MinBinaryHeap<T>` simple, since we can create an `Arbitrary Vec<T>` and populate our `MinBinaryHeap` from that. Now that we have a way to generate arbitrary heaps, what can we do with them?

The `quickcheck` crate provides a handy macro, called `quickcheck!`, to let us easily define properties that we want to test. But, which properties should we choose? To answer that, let us go back to our definition of the minimum binary heap. One of the properties of a minimum binary heap is that, for every heap, the root value of the heap should be the absolute minimum value of the data structure. So, let us encode that property as a function.

```
quickcheck! {
  fn prop_root_is_minimum_value(xs: MinBinaryHeap<usize>) -> bool {
    let min = xs.buf.iter().min();
    match (min, xs.root()) {
      (Some(min), Some(root)) => min == root,
      (_, None) => true,
      (None, _) => unreachable!(),
    }
  }
}
```

There are a few pertinent features worth exploring here. First, note that our property test is simply a normal function, declared inside of the `quickcheck!` macro. The function takes a `MinBinaryHeap<usize>`, which is a type that has `Arbitrary` implemented on it (since `MinBinaryHeap<T>` implements `Arbitrary` as does the primitive `usize`), and returns `bool` (whether or not the property is met for the passed in data structure). When the tests are run, the `quickcheck` library will execute the property function on up to 100 arbitrary instances of the heap and verify that the property passes. If the property fails, the library will attempt to shrink the generated data structure to find the minimum failing test case and will print that out for you.

Looking further at the example, the property is fairly simple. We peek at the interior buffer and get the minimum value of the buffer using the `min` iterator consumer. Then, we peek at the root value using our public `root` method. If the root value matches the minimum value, we succeed.

We can take a look at another example property to give you an idea of the types of properties that are useful to define. One thing we want to determine is whether or not the `insert` function is producing valid heaps. Since we are using `insert` in our definition of `Arbitrary`, we should really validate it. According to our heap definition, for every sub-tree in our heap, the parent of the sub-tree must be strictly less than either of its children. We can encode that into a property.

```
fn prop_is_heap(xs: MinBinaryHeap<usize>) -> bool {
  for idx in 0..xs.buf.len() {
    let current = &xs.buf[idx];
    // If there is a right child
    if let Some(right) = xs.right(idx) {
      // It must be larger than the current value
      if current > &xs.buf[right] {
        return false;
      }
    }
    // If there is a left child
      if let Some(left) = xs.left(idx) {
      // It must be larger than the current value
      if current > &xs.buf[left] {
        return false;
      }
    }
  }
  true
}
```

The type of this function is identical to the previous one; it is a function from a `MinBinaryHeap<usize>` to a `bool`. Once again, we exploit the fact that this is a unit test and that we know the implementation details of the data structure to walk every value in the interior vector. Each value in the vector is the parent of a sub-tree, so we check to see if the sub-tree has child nodes. If child nodes exist, we ensure that the values of the child nodes are larger than the value of the parent. If all of the children are larger than all of the parents, then the property passes. Otherwise, it fails.

The power of property-based testing is that we are thinking up logical properties that must hold for the data structure to be valid, instead of trying to think up specific test cases that might show that our properties hold. We are relying on the computer to do what computers do well — do

the brute-force computation of generating test cases while we expend programmer-time thinking at a higher level and encoding the invariants in code that we already expect our data structure to meet based on its requirements.

After spending time writing basic unit tests as well as enough property-based tests that we feel confident that our data structure is implemented correctly, we need to make sure that others can use the data structure the way we intend it to be used. We can accomplish that with integration testing.

INTEGRATION TESTING

Where unit tests are meant to test small units of code, integration tests are meant to be more general. Traditionally in Rust, a unit test will test only code that is defined in the same file that contains the unit test. But how can you test interactions between modules? How can you ensure that you made all of your public APIs and data structures actually public? You can write integration tests to make sure that your API can be used as intended.

Let us add an integration test to verify our public API for the heap data structure.

```
// this code lives in tests/lib.rs
extern crate binheap;
extern crate rand;
use rand::{thread_rng, sample};

#[test]
fn it_works() {
    let mut rng = thread_rng();
    let data = sample(&mut rng, 1..1000, 100);
    let mut heap = binheap::MinBinaryHeap::new();

    // build up our heap
    for x in &data {
        heap.insert(*x);
    }

    let mut old: Option<usize> = None;
    loop {
        // extract each value
        let new = heap.extract();
```

```
            if new.is_none() {
                break;
            }

            // and ensure older value is smaller than
            // newer one
            if let Some(old) = old {
                if let Some(new) = new {
                    assert!(old < new);
                }
            }

            old = new;
        }
    }
```

There are a few interesting bits to this example. First, note that we are not putting the test case inside of an existing source file in our crate. Rather, we are putting the integration tests inside of a file in the `tests` directory. Since the integrations tests do not live as a part of the crate, we have to import the crate using the `extern crate binheap` syntax. Since we have to import the crate, we only have access to public functions, public data structures, and public fields, just like a consumer of our library would.

In this integration test, we are verifying that we can create a heap, insert many random values into it, and remove them, in order, from smallest to largest. This workflow, while contrived for a test, is roughly how we would expect an end-user of the crate to use the API.

DOCUMENTATION TESTING

Closely related to integration testing is the idea of making sure that the examples that you have documented in your API documentation always compile and work properly. Rust has integrated support for generating documentation, but it also has built-in support for extracting source code from your documentation markup, compiling it, and running it as a documentation test.

In the following code example, we have very simply documented the `extract` method for our `MinBinaryHeap` structure. In it, we provide a simple description of the method followed by a block of example code that uses the method.

```
/// Extract the minimum value from the heap.
///
/// # Examples
///
/// '''
/// use binheap::MinBinaryHeap;
/// let mut heap = MinBinaryHeap::new();
/// for x in vec![42, 5, 13, 2] {
///     heap.insert(x);
/// }
/// let min = heap.extract().unwrap();
/// assert_eq!(min, 2);
/// '''
pub fn extract(&mut self) -> Option<T> { . . . }
```

In the documentation block, we can use standard Markdown markup to insert headings, make text bold, or in this case, add a source block between consecutive ''' markers. In the source block, simply document how the API is meant to be used. In this case, we are building up a simple heap and extracting the smaller value out of it. We are ensuring that we obtained a value by putting the `unwrap` call after the extract and then asserting that the minimum value we got is really the minimum value that we put in the heap. Additionally, you can add multiple source code blocks and they will each be extracted out as separate test cases.

Like all previous tests, you can run the documentation tests using the same `cargo test` command. When they run, a separate section shows up in the test output that looks like the following:

```
    Doc-tests binheap

running 3 tests
test MinBinaryHeap<T>::insert_0 ... ok
test MinBinaryHeap<T>::insert_1 ... ok
test MinBinaryHeap<T>::extract_0 ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

In the output, you can see that three documentation tests were extracted from the source code. In this case, the source code had two source code examples associated with the `insert` method and a single example associated with the `extract` method.

BENCHMARKING

The final type of testing we are going to discuss is slightly different than other types of testing since there is no way to explicitly check for success and failure. Benchmarking your crate is a good way to ensure that the code you wrote is performant. In the case of our example crate, we want to make sure that using our minimum binary heap to find the minimal value is faster than sorting a vector of numbers to find the minimal value. Let us take a look at some example benchmarking code:

```
// benches/bench.rs
#![feature(test)]
extern crate test;
extern crate binheap;
extern crate rand;

#[cfg(test)]
mod tests {
    use rand::{thread_rng, sample};
    use binheap::*;
    use test::Bencher;

    fn gen_sample() -> Vec<usize> {
        let mut rng = thread_rng();
        sample(&mut rng, 1..1000, 100)
    }

    #[bench]
    fn bench_extract_min(b: &mut Bencher) {
        let data = gen_sample();

        b.iter(|| {
            let mut heap = MinBinaryHeap::new();
            for x in &data {
                heap.insert(x);
            }
            heap.root();
        });
    }
}
```

The first thing to notice in the benchmarking code is that it lives in the `benches/` directory and requires the `test` feature gate to be enabled. The test feature gate allows us to access the built-in test module which is used

to import the benchmarking data types and functions; unfortunately, we cannot enable the test feature gate using the stable compiler since the test module API has not been completely stabilized by the compiler team – that is, the Rust developers are still trying to ensure that they have gotten the API correct before they agree to fully support it. So, to run benchmarking tests, we will have to run the benchmarks with the nightly compiler. While there is a risk that the benchmark API found in the `test` module of the nightly compiler might change in the future, the `Bencher::iter` function has been fairly stable in the past and being able to run benchmarking tests is too useful to pass up. We can easily run the benchmarking tests with the nightly compiler using `rustup`:

```
$ rustup run nightly cargo bench
running 2 tests
test tests::bench_extract_min ... bench:         725 ns/iter (+/- 549)
test tests::bench_min_iter    ... bench:       1,402 ns/iter (+/- 383)

test result: ok. 0 passed; 0 failed; 0 ignored; 2 measured
```

Running `cargo` using the `rustup` command in this way will allow us to only use the nightly compiler for running benchmarks while using the stable compiler for all other builds.

Getting back to our source example, we can see that benchmark tests are marked with `#[bench]` instead of `#[test]`. Additionally, the benchmark tests all have the following signature: `fn(&mut Bencher)`. The benchmark runner passes in the `Bencher` type to the test. The `Bencher` type exposes a method called `iter` that calls the passed-in closure many times — as many iterations as it takes for the timing per iteration to statistically stabilize. Since it is up to our code to call the `iter` method, we can run as much setup code per test that we need to without the setup code being benchmarked as well — only the code that is run in the `iter` closure is benchmarked.

KEY TAKEAWAYS

- Testing is an important part of the crate-creation process. Rust makes it easy to integrate unit, integration, and documentation tests into your build process.
- Using the external `quickcheck` crate, you can augment your unit testing with property-based testing, making it easier to generate test cases for verifying the invariants that your code guarantees.
- Rust has unstable support for running benchmark performance tests. It makes it easy to write performance tests, although you will have to run a nightly compiler to use the built-in functionality.