1

yes

Implementing `true` and `false` demonstrated how to exit a Rust program with a specific status code, without returning a value from the `main()` function. A C program's `main()` function also takes an array of arguments (`argv`) and a count of arguments (`argc`) as parameters. With no arguments, the `yes` program continually prints lines with the "`y`" character until interrupted or killed. It also takes a single command line argument, providing an alternative string to print.

The `yes` utility was designed to (repeatedly) output a string to a program expecting input.[1] For example, to uninstall Rust, `rustup` asks for confirmation:

```
% rustup self uninstall

Thanks for hacking in Rust!

This will uninstall all Rust toolchains and data, and remove $HOME/.
cargo/bin from your PATH environment variable.

Continue? (y/N)
```

Answering the confirmation prompt automatically and keeping Rust installed by *piping* the output of `yes` with an "`N`" argument to `rustup` using the shell's pipe operator ("`|`"):

```
% yes N | rustup self uninstall
...
Continue? (y/N)
info: aborting uninstallation
```

1 Modern programs usually take a `-y` option to disable interactive confirmation prompts.

The `main()` function in `yes.c` implements `yes` with two branches, based on whether an argument is present, or not:[2]

usr.bin/yes/yes.c

```
int
main(int argc, char *argv[])
{
    if (argc > 1)
        for (;;)
            puts(argv[1]);
    else
        for (;;)
            puts("y");
}
```

The `libc` function `puts()` takes a string (`char *`) and prints it to `stdout`, followed by a newline character:

stdio.h

```
int puts(const char *str)
```

Calling the `println!` macro is the standard way to output a line of text in Rust.[3] Like `puts()`, `println!` adds a trailing newline.[4] Rust clearly identifies macros by ending their names with a *bang* (`!`).[5] Macro syntax is an advanced topic, and it's not necessary to write a macro to implement any of the programs in this book.

Calling `println!` in an infinite loop is the simplest way to implement `yes` in Rust. The idiomatic way of looping infinitely in Rust is with a `loop` expression, which repeatedly executes the code inside the block following the `loop` keyword. The block never returns, so the expression evaluates to the `never` type (`!`):[6]

src/bin/yes.rs

```
fn main() {
    loop {
       println!("y")
    }
}
```

2 The original version from 1979 by Ken Thompson is implemented in only six lines of code! https://github.com/dspinellis/unix-history-repo/blob/4c37048d6dd7b8f65481c8c86ef8cede2e782bb3/usr/src/cmd/yes.c
3 The macros defined in the `std` module are automatically available in all Rust programs without importing them with a `use` statement. See for details: https://stackoverflow.com/questions/57530128/are-all-macros-in-the-rust-std-library-included-in-the-prelude
4 The `print!` macro outputs a string without appending a newline.
5 The "`!`" is part of the string identifying the macro and does not represent the `never` type in this context.
6 Unless there is a `break` expression in the `loop`, which terminates the `loop` and returns a value or the unit type: `()`.

By default, `cargo build` compiles every binary target in a project. The `--bin` option to `cargo` takes an argument specifying a single *binary* target. Building just the `yes` program, running it, and exiting by typing `ctrl-c`:

```
% cargo build --bin yes
% target/debug/yes
y
y
y
...
y
^C
% echo $?
0
```

The `yes` program also prints a string argument instead of the default "y". Unix processes get their arguments from the `execve()` system call, which takes a path to the program, an array of arguments, and an array of environment variables, and starts a new process:

```
unistd.h
int
execve(const char *path, char *const argv[], char *const envp[]);
```

The `argv` parameter to a C program's `main()` function is a *vector* (array) of string arguments from `execve()`. The `argc` parameter contains the *count* of arguments. The first element of `argv` is the name of the running program, so `argc` is always at least `1`. Any command line arguments start at `argv[1]`.[7]

The `main()` function in Rust doesn't take any parameters: to access the arguments passed to `execve()`, the `std::env` module defines the `args()` *constructor* function, which returns an initialized `Args` struct containing the arguments:

```
std::env
fn args() -> Args
```

The convention in Rust is to use *lowercase* names for functions (`args()`) and *capitalized* names for structs (`Args`).[8] A struct is a *heterogeneous* data structure consisting of a group of named *fields*, where each field may be a different type. A struct's name is a new, custom, *type* representing the group of fields.

Everything in Rust is *private* by default: only visible within the module which defines it. The pub keyword (*public*) allows something to be visible outside the current module.[9] The fields of the `Args` struct are *private*, even though the struct itself is public:

```
std::env
pub struct Args { /* fields omitted */ }
```

Instead of accessing the arguments in the `Args` struct directly, a Rust program *iterates* through its arguments. An iterator in Rust represents a *sequence* of items. The `next()` *method* returns the next item in the sequence.

7 Arrays are zero-indexed in C (and Rust).
8 The naming conventions are documented in RFC #430: https://github.com/rust-lang/rfcs/pull/430
9 The functions and data structures used in this book are all declared as pub so the keyword is omitted from their definitions.

The `fn` keyword declares both functions and *methods*: a function associated with a data structure. The difference between a function and a method is that the first parameter to a method is the `self` keyword, representing the data structure that *receives* the method. A method's first parameter, `self`, is an exception to the requirement that a function's definition provides a type annotation for each of its parameters. A data structure *implements* methods so providing a type for `self` would be redundant.

Rust defines sets of one or more methods with the `trait` keyword.[10] Implementing the `Iterator` trait for a struct only requires defining an *associated type* `Item` and a single method, `next()`. The `next()` method takes a *mutable reference* (`&mut`) to the iterator (`self`) and returns an `Option` containing the next `Item` in the sequence:

std::env
```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

The `Option` returned by `next()` is one of Rust's fundamental data structures. An `Option` is an *enumeration* (`enum`) representing an *optional* value, which may or may not exist. An enum is composed of multiple *variants*, but each instance of an enum represents only one variant at a time.[11] The variants of a *public* enum are always visible. The names of enums and their variants are *capitalized*.

The `Option` enum has two variants, `Some` and `None`. The `None` variant represents a missing value. The `Some` variant represents some value `T`:

std::option::Option
```
enum Option<T> {
    None,
    Some(T),
}
```

Types enclosed in *angle brackets* "`<>`", are generic and represent any type.[12] The `Option` enum takes a *generic* parameter `<T>`, so the `Some` variant can store any type: in this case, the type of `Item` *yielded* by the `Iterator`.

Calling an iterator's `next()` method *yields* the next item in the sequence. Yielding an item from an iterator is the same as *returning* a value from `next()`. This book uses the term *yielding* to more clearly indicate that calling `next()` returns a different value each time.

The `std::env` module *implements* the `Iterator` trait `for` the `Args` struct with the `impl` keyword, yielding `String` items:[13]

10 Traits are called *interfaces* in other languages like Java.
11 Rust enums are stack allocated, tagged unions, that are the size of the largest variant (plus the tag).
12 The label "T" is arbitrary but the convention in Rust is to use "T" for generic types and "E" for generic errors. Type names follow the `UpperCamelCase` naming convention, with single letter types in uppercase.
13 The "`for`" keyword is also used in a different context to loop through an iterator.

```
                                                              std::env
impl Iterator for Args {
    type Item = String;
    fn next(&mut self) -> Option<String> {
        ...
    }
}
```

A `String` is a growable, heap allocated, UTF-8 string. Unlike strings in C (`char *`), Rust `Strings` are not null terminated. The first item *yielded* by the `Args` iterator is a `String` containing the program's name, like `argv[0]`. Subsequent calls to `next()` yield the command line arguments. The iterator yields a `None` value when there are no more arguments.

It's not possible to directly index into an iterator to access a specific item, because they may be generated *dynamically* and are evaluated *lazily*.[14] Instead of calling `next()` multiple times in a loop until reaching an index, the `Iterator` trait provides an `nth()` method which *consumes* the iterator up until the specified index, returning the item at that index.[15]

The `nth()` method takes two comma separated parameters: a *mutable reference* (`&mut`) to the iterator (`self`), and a `usize` index (`n`). The `usize` type is the platform's *unsigned* integer size, typically 32 or 64 bits. The `nth()` method returns an `Option` containing the item at index `n`, or `None`:[16]

```
                                                   std::iter::Iterator
fn nth(&mut self, n: usize) -> Option<Self::Item>
```

Calling the `nth()` method *consumes* arguments from the `Args` iterator. Assigning the output of `nth()` to a variable once is more efficient than creating a new `Args` struct with `args()` each time inside the `loop`.

A `let` *statement* binds a value to a named, lowercase identifier in the current block. Unlike expressions, statements in Rust don't evaluate to values and must be terminated with a semicolon (`;`).

When binding the `Args` struct returned from calling the imported `args()` function to the `argi` variable with a `let` statement, the compiler *infers* the type of the `argi` variable. Annotating a variable with the type of the value isn't required if the source code provides enough context for the compiler to *infer* the type itself.

14 The value of each item in an iterator may be computed, and evaluating each value is deferred until needed.

15 The `Iterator` trait only requires implementing the `next()` method, so the other trait methods like `nth()` are implemented just using `next()`.

16 The `n` parameter to `nth()` is zero indexed.

Importing the `Args` struct and calling its `nth()` method with the `argi` variable as the first parameter (`self`), assigning the output to the `y` variable, and printing it in a `loop` with `println!`:

```
src/bin/yes.rs
use std::env::args; // fn args()
use std::env::Args; // struct Args

fn main() {
    let argi = args();
    let y = Args::nth(argi, 1);

    loop {
        println!(y)
    }
}
```

Building the program returns an error from compiling the `println!` macro:

```
% cargo build --bin yes
error: format argument must be a string literal
 --> src/bin/yes.rs:9:18
  |
9 |         println!(y)
  |                  ^
  |
help: you might be missing a string literal to format with
  |
9 |         println!("{}", y)
  |                  ^^^^^
```

The `println!` macro *formats* the `String` yielded by `nth()` before printing it. The first parameter to `println!` is a *format string*, which must be a string literal. A *string literal* is a sequence of Unicode characters enclosed in double quotes (`""`). Importantly, since the compiler parses the format string at compile time, it can't be a variable. Rust's format strings are based on the Python language's syntax, using curly braces "`{}`" as *placeholders*.[17] The `println!` macro replaces the empty placeholder (`{}`) with the corresponding `y` parameter:

```
src/bin/yes.rs
use std::env::args; // fn args()
use std::env::Args; // struct Args

fn main() {
    let argi = args();
    let y = Args::nth(argi, 1);

    loop {
        println!("{}", y)
    }
}
```

17 The `libc` function `printf()` uses "`%`". To format a string, the equivalent of "%s" is "`{}`".

Adding a format string literal resolves the compiler error for `println!`, but the `self` parameter to the `nth()` method is the wrong type:

```
error[E0308]: mismatched types
 --> src/bin/yes.rs:6:23
  |
6 |     let y = Args::nth(argi, 1);
  |                       ^^^^
  |                       |
  |                       expected `&mut Args`, found struct `Args`
  |                       help: consider mutably borrowing here: `&mut
argi`
```

Calling `nth()` advances the iterator, modifying its internal state (consuming the first n items), so the `nth()` method *mutably borrows* its `self` parameter. The `mutable borrow` operator (`&mut`) creates an *exclusive* reference to the iterator so that nothing else in the program can access it, until `nth()` returns. This prevents undefined behavior caused by *data races*: where there are multiple, unsynchronized, references to a mutable value. Following the help message from the compiler and *mutably* borrowing the iterator with the `&mut` operator:

src/bin/yes.rs

```rust
use std::env::args; // fn args()
use std::env::Args; // struct Args

fn main() {
    let argi = args();
    let y = Args::nth(&mut argi, 1);
    //                ^^^^

    loop {
        println!("{}", y)
    }
}
```

Mutably borrowing the `Args` iterator resolves the type mismatch error.

However, it still doesn't compile because `println!` is unable to format the `Option` returned by `nth()`:

```
error[E0277]: `Option<String>` doesn't implement `std::fmt::Display`
 --> src/bin/yes.rs:9:24
  |
9 |         println!("{}", y)
  |                        ^ `Option<String>` cannot be formatted with
the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for
`Option<String>`
```

The `println!` macro expects a parameter implementing the `std::fmt::Display` trait, corresponding to the empty placeholder (`{}`) in the format string. Because the `Option` type is a generic container for other types, there's no default way to print one, so it doesn't implement `Display`.[18]

18 Since an `Option` is generic over any type, the Some variant may contain a type that isn't printable.

The use of `Options` is extremely common in Rust, and protects against bugs such as out of bounds array accesses. When writing a C program, it is the programmer's responsibility to check the length of the array (`argc`) first, ensuring that an element exists at that position in the array. This valid C program doesn't check the length of the `argv` array before printing the argument at index `1`:

tinyyes.c

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    for (;;)
        puts(argv[1]);
}
```

The code above compiles with no warnings, but the program accesses an invalid memory location and *segfaults* when run without an argument:

```
% gcc -Wall -o tinyyes tinyyes.c
% ./tinyyes y
y
y
y
...
y
^C
% ./tinyyes
[1]    16339 segmentation fault  ./tinyyes
```

Reading memory past the end of an array is a common mistake in C programs, which is prevented in Rust by using iterators. Calling `nth()` with an index parameter (`n`) greater than the number of arguments in the `Args` struct yields a `None` value. The `Option` returned from the `nth()` method is either `Some(String)`, or `None`, depending on whether an argument exists at the specified index, or not. The `{}` placeholder in the format string passed to `println!` expects a corresponding `String` parameter, not an `Option<String>`.[19]

The two basic strategies for extracting the value of an `Option` are *unwrapping*, or *destructuring*. The `unwrap()` method takes ownership of the `Option`, so it's *inaccessible* afterwards. It either returns the value of the `Some` variant, or *panics* and exits the program if the `Option` is `None`:

std::option::Option

```
fn unwrap(self) -> T
```

Idiomatic Rust programs don't directly call methods as functions with a `self` parameter. A *method call* is easier and more powerful. Method call expressions consist of a *receiver* expression followed by a dot ("`.`") and the method name, then the list of the method's parameters, not including the initial `self` parameter.[20] The compiler looks up the method's name in the list of methods implemented by the receiver, which doesn't require a `use` declaration.

The unidiomatic `Args::nth()` function call expression evaluates to an `Option<String>`, which then *receives* the `unwrap()` method call. The `unwrap()` method only takes a `self` parameter, so in

19 The `{}` placeholder can format any type implementing the `Display` trait.
20 The dot ("`.`") is sometimes called the *dot operator* but in this context it's technically part of a method call expression and isn't a true operator.

the context of a method call it takes an empty parameter list, and the compiler inserts the correct `self` parameter:

```
src/bin/yes.rs
use std::env::args; // fn args()
use std::env::Args; // struct Args

fn main() {
    let argi = args();
    let opt = Args::nth(&mut argi, 1);
    let y = opt.unwrap();

    loop {
        println!("{}", y)
    }
}
```

Now the `println!` macro is able to format the `String` returned by *unwrapping* the `Option`. But the `nth()` method can't *mutably* borrow an *immutable* variable:

```
error[E0596]: cannot borrow `argi` as mutable, as it is not declared as
mutable
 --> src/bin/yes.rs:6:23
  |
5 |     let argi = args();
  |         ---- help: consider changing this to be mutable: `mut argi`
6 |     let opt = Args::nth(&mut argi, 1);
  |                         ^^^^^^^^^ cannot borrow as mutable
```

Unless modified with the `mut` keyword, variables declared with a `let` statement are *immutable* by default. The `nth()` method consumes values from the iterator, which modifies it. To safely modify a mutable value, the `&mut` operator *exclusively* borrows it.

The main advantage of using a method call expression is that it automatically *borrows* the receiver as required. With the `self` parameter omitted from the method's parameter list, the compiler determines the correct ownership and calls the method.

Using a method call, the *receiver* of the `nth()` method is the `Args` struct returned by the `args()` function. The `nth()` method *mutably borrows* its `self` parameter. First, the compiler tries passing an `Args` struct as the `self` parameter, which fails, then `&Args`, which also fails, then finally `&mut Args`, which succeeds:

src/bin/yes.rs

```
use std::env::args; // fn args()
use std::env::Args; // struct Args

fn main() {
    let mut argi = args();
    let opt = argi.nth(1);
    let y = opt.unwrap();

    loop {
        println!("{}", y)
    }
}
```

The method call expression resolves the `nth()` method without needing to import the `Args` struct:[21]

```
warning: unused import: `std::env::Args`
 --> src/bin/yes.rs:2:5
  |
2 | use std::env::Args; // struct Args
  |     ^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default
```

Chaining multiple method calls together, so the receiver of each method is the value returned from the previous method, is more concise than multiple `let` statements. Chaining the methods so that the `Args` struct returned by the `args()` function is the receiver of the `nth()` method, and the `Option` returned by `nth()` is the receiver of the `unwrap()` method:

src/bin/yes.rs

```
use std::env::args;

fn main() {
    let y = args().nth(1).unwrap();

    loop {
        println!("{}", y);
    }
}
```

Chaining multiple methods also eliminates the need to define the `y` variable as *mutable* (since the value never changes).

21 The `nth()` method is defined in the `Iterator` trait which is included in the prelude.

This program compiles and runs, but *panics* if there isn't an argument present:

```
% target/debug/yes no
no
no
no
...
no
^C
% target/debug/yes
thread 'main' panicked at 'called `Option::unwrap()` on a `None`
value', src/bin/yes.rs:5:13
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
% echo $?
101
```

Panicking terminates the program *safely*, before it accesses invalid memory. A more robust way of handling an `Option` than unwrapping it, possibly causing a panic, is with *pattern matching*.

A `match` expression takes a *target* value and a block containing a comma separated list of *arms*. Each `match` *arm* binds a candidate pattern to an expression with the *fat arrow* (`=>`) operator. The `match` expression evaluates to the value returned from the expression bound to the *first* pattern in the list that matches the target value.

The `std::prelude` module includes a set of frequently used data structures, traits, and functions from the Rust standard library. The `prelude` includes the `Option` enum and both variants (`Some` and `None`). Every Rust program automatically imports *everything* in the `prelude` into its namespace, so `Some` and `None` may be used without a `use` declaration.

Along with unwrapping, *destructuring* is the other common way of working with an `Option`. Destructuring uses a *pattern* to extract a matching value, and bind it to a local variable. A pattern in a `match` arm *destructures* the target, and binds it to a variable with the scope of the arm's expression.

Matching on the `Option` returned by `nth()`, and *destructuring* the value contained in the `Some` variant to a local variable `y`:[22]

src/bin/yes.rs

```rust
use std::env::args;

fn main() {
    match args().nth(1) {
        Some(y) => loop {
            println!("{}", y)
        },
    }
}
```

22 The `rustfmt` formatter adds a trailing comma after `match` arms that aren't blocks.

But, the patterns in a `match` statement must always be *exhaustive*, covering all possible values of the target, or the program won't compile:

```
error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/bin/yes.rs:4:11
    |
4   |     match args().nth(1) {
    |           ^^^^^^^^^^^^^ pattern `None` not covered
    |
    = help: ensure that all possible cases are being handled, possibly
by adding wildcards or more match arms
    = note: the matched value is of type `Option<String>`
```

An `Option` can only be either `Some`, or `None`. Adding a second arm to handle the `None` case which prints "y" if there is no argument:

```rust
src/bin/yes.rs
use std::env::args;

fn main() {
   match args().nth(1) {
       Some(y) => loop {
           println!("{}", y)
       },
       None => loop {
           println!("y")
       },
   }
}
```

Now, this `yes` program continuously prints either the argument or "y", until quit with `ctrl-c`. Typically, the shell *pipes* the output of `yes` to another command reading from `stdin`. Piping `yes` to the `head` utility prints the first 10 lines, and reveals a problem:

```
% target/debug/yes | head
y
y
y
y
y
y
y
y
y
y
thread 'main' panicked at 'failed printing to stdout: Broken pipe (os
error 32)', library/std/src/io/stdio.rs:935:9
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
% echo $?
0
```

After reading the first 10 lines, the `head` process exits, closing its `stdin` filehandle, and "*breaking*" the pipe. The operating system sends a `SIGPIPE` signal to a process when it writes to a broken pipe. The default action when receiving a `SIGPIPE` signal is to silently terminate the process. However, if the process configures itself to *ignore* `SIGPIPE` signals, then the `write()` system call fails, returning an `EPIPE` error:[23]

```
sys/sys/errno.h
#define EPIPE        32       /* Broken pipe */
```

Signals aren't thread safe: in a multithreaded program, a *random* thread receives the `SIGPIPE` signal, not necessarily the thread that wrote to the broken pipe. Rust assumes that all programs are multithreaded, so to make broken pipe errors thread safe, the Rust compiler inserts code to *ignore* `SIGPIPE` signals into the startup code of all Rust programs.[24] The next time `yes` prints a line after `head` exits, the underlying `write()` system call returns an `EPIPE` error, causing the `println!` macro to panic the program, printing an error message.[25] Rust programs should handle write errors, instead of panicking when writing to a broken pipe.[26]

The shell variable `$?` contains the exit status of the *previous* process, which in this example is `head`. The `head` program exits with a `0` status after successfully printing 10 lines of input. To find the status of *all* the processes in a pipeline in `zsh`, the `pipestatus` array contains the exit status of each process. The `${}` syntax selects the entire array:

```
% echo ${pipestatus}
```

In `bash`, the `PIPESTATUS` array contains each exit status, and the `[@]` syntax selects all the elements in an array:

```
% echo ${PIPESTATUS[@]}
```

Rust programs always exit with a status of `101` after panicking:[27]

```
% target/debug/yes | head
y
...
thread 'main' panicked at 'failed printing to stdout: Broken pipe (os
error 32)', library/std/src/io/stdio.rs:935:9
note: Run with `RUST_BACKTRACE=1` environment variable to display a
backtrace.
% echo ${pipestatus}
101 0
```

23 A process ignores a signal by calling one of the `libc` functions `signal()` or `sigaction()` and setting the signal handler to `SIG_IGN`.
24 On Unix platforms. This was committed in: https://github.com/rust-lang/rust/pull/13158.
25 This behavior is documented in https://github.com/rust-lang/rust/issues/24821.
26 Not handling `SIGPIPE` gracefully is a known issue in Rust: https://github.com/rust-lang/rust/issues/46016 and https://github.com/rust-lang/rust/issues/62569.
27 See the discussion here: https://www.reddit.com/r/rust/comments/3qdvez/guarantees_about_exit_codes/

The BSD version exits with a status of `141`:

```
% /usr/bin/yes | head
y
...
% echo ${pipestatus}
141 0
```

The original code in `yes.c` doesn't handle errors returned from writing to `stdout` so the process dies when it receives the `SIGPIPE` signal. When a program dies due to an *unhandled* signal, the `zsh` and `bash` shells set the exit status to `128` plus the signal number, in this case `13`:[28]

```
sys/sys/signal.h
#define SIGPIPE 13 /* write on a pipe with no one to read it */
```

The `writeln!` macro returns errors instead of panicking, allowing the program to handle a broken pipe, without changing how it handles the signal.[29] The `writeln!` macro takes a destination *writer* and a format string, or just a writer (to print a newline).[30] The writer is something *implementing* the `std::io::Write` trait.[31]

Several structs in the `std::io` module implement the `Write` trait, such as `Files`, `TcpStreams`, and `Stdout`.[32] The `stdout()` function in the `std::io` module returns a `Stdout` struct representing a handle to the current process's `stdout`:

```
std::io
fn stdout() -> Stdout
```

Binding the `Stdout` struct returned by the `stdout()` function to the `out` variable with a `let` statement, and passing it as the first parameter to the `writeln!` macro:

```
src/bin/yes.rs
use std::env::args;
use std::io::stdout;

fn main() {
    let out = stdout();

    match args().nth(1) {
        Some(y) => loop {
            writeln!(out, "{}", y)
        },
        None => loop {
            writeln!(out, "y")
        },
    }
}
```

28 This behavior isn't standard - the `ksh` (Korn) shell sets it to the signal number (`13`).
29 The Rust standard library doesn't include a module for handling signals to change this behavior.
30 The difference between `writeln!` and `println!` is similar to the difference between `fprintf()` and `printf()` in `libc`.
31 Actually the macro doesn't check if the parameter implements the `Write` trait, it just directly calls the trait's `write_fmt()` method.
32 The `Implementors` section of the documentation for each trait lists all of the types that implement it.

But, the compiler isn't able to find the `write_fmt()` method:

```
error[E0599]: no method named `write_fmt` found for struct `Stdout` in
the current scope
    --> src/bin/yes.rs:9:13
     |
9    |             writeln!(out, "{}", y)
     |             ^^^^^^^^^^^^^^^^^^^^^^ method not found in `Stdout`
     |
     = help: items from traits can only be used if the trait is in
scope
     = note: this error originates in the macro `writeln` (in Nightly
builds, run with -Z macro-backtrace for more info)
help: the following trait is implemented but not in scope; perhaps add
a `use` for it:
     |
1    | use std::io::Write;
     |
```

A macro is a code *generator*: code which writes other code. The compiler evaluates the macro's definition and includes the code it generates in the final program. The code generated from the `writeln!` macro calls the `Write` trait's `write_fmt()` method, which the compiler is unable to find:

```
                                                          std::fmt::Write
fn write_fmt(
    &mut self,
    args: Arguments,
) -> Result<(), std::io::Error>
```

It's possible for multiple traits to define a method with the same name (and even with the same parameter and return types). To prevent ambiguity about which method the progam will, each trait must be brought into scope with a `use` declaration to access its methods.[33]

Accepting the suggestion from the compiler's error message and importing the `io::Write` trait with a `use` declaration:

```
                                                          src/bin/yes.rs
use std::env::args;
use std::io::stdout;
use std::io::Write;
//           ^^^^^
fn main() {
    ...
```

33 Calling the `nth()` method on the `Args` struct didn't require importing the `Iterator` trait because it's included in the prelude.

Importing the `Write` trait enables the compiler to find the correct `write_fmt()` method.

But, the code generated by the `writeln!` macro causes errors in each arm of the `match` (only the first error is shown here):

```
error[E0308]: mismatched types
  --> src/bin/yes.rs:10:13
   |
8  | /     match args().nth(1) {
9  | |         Some(y) => loop {
10 | |             writeln!(out, "{}", y)
   | |             ^^^^^^^^^^^^^^^^^^^^^^ expected `()`, found enum
`Result`
11 | |         },
...  |
14 | |         },
15 | |     }
   | |     -- help: consider using a semicolon here
   | |_____|
   |       expected this to be `()`
   |
   = note: expected unit type `()`
                found enum `Result<(), std::io::Error>`
   = note: this error originates in the macro `writeln` (in Nightly
builds, run with -Z macro-backtrace for more info)
```

A `loop` expression executes its body an infinite number of times, so each iteration doesn't return a new value, and must evaluate to `()`. The `println!` macro returns `()`, but the code generated from the `writeln!` macro returns a `Result` enum.

In Rust, terminating an expression with a semicolon (`;`) *suppresses* its value and evaluates to the unit type `()`. The compiler's help message suggests terminating the `match` expression with a semicolon, but that doesn't address the two `loop` expressions. Terminating both lines invoking `writeln!` with semicolons suppresses each returned `Result`:

```
src/bin/yes.rs
use std::env::args; // fn args()
use std::io::stdout;
use std::io::Write;

fn main() {
    let out = stdout();

    match args().nth(1) {
        Some(y) => loop {
            writeln!(out, "{}", y);
            //                    ^
        },
        None => loop {
            writeln!(out, "y");
            //                ^
        },
    }
}
```

Terminating the lines invoking `writeln!` with semicolons resolves the errors with the `loop` expressions in the `match` arms, but the borrow checker generates an error because the `out` variable containing a `Stdout` struct is not *mutable*. The error indicates that the `writeln!` macro is trying to *mutably borrow* the `out` variable:

```
error[E0596]: cannot borrow `out` as mutable, as it is not declared as
mutable
  --> src/bin/yes.rs:10:22
   |
6  |     let out = stdout();
   |         --- help: consider changing this to be mutable: `mut out`
...
10 |             writeln!(out, "{}", y);
   |                      ^^^ cannot borrow as mutable
```

Variables in Rust are *immutable* by default: their values can't be changed once they've been bound to a name in a `let` statement. Writing to the `Stdout` struct *updates* its internal buffer.[34] Adding the `mut` keyword as suggested by the error message modifies the `let` statement to create a *mutable* binding:

```
src/bin/yes.rs
fn main() {
    let mut out = stdout();
    //  ^^^

    match args().nth(1) {
        ...
```

34 The `Stdout` struct is line buffered, storing input in an internal buffer until it receives a newline character then *flushing* the buffer to `stdout`.

This change resolves the ownership error, enabling the program to compile, but with a warning for each call to `writeln!`:

```
warning: unused `Result` that must be used
  --> src/bin/yes.rs:10:13
   |
10 |             writeln!(out, "{}", y);
   |             ^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be
handled

warning: unused `std::result::Result` that must be used
 --> src/bin/yes.rs:13:13
   |
13 |             writeln!(out, "y");
   |             ^^^^^^^^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be
handled
```

When `head` closes `stdin` and exits, this version of `yes` keeps executing the `loop`, ignoring each `Result` returned from `writeln!`. After suspending execution by typing `ctrl-z`, the shell shows that `head` is "done", and piping the output of the `ps` command to `grep` confirms that no process with that `id` is running:

```
% target/debug/yes | head -1
y
^Z
[1]  + 23631 suspended  target/debug/yes |
       23632 done       head -1
% ps | grep 23632
%
```

At this point the structure of the Rust program is the same as the original `yes.c`: a loop in each arm of a conditional based on whether there's an argument string or not. Conditionally setting a variable containing the string to be printed allows all of the error handling to be condensed into a single `loop`. The `unwrap_or()` method returns either the `Some` value of an `Option`, or the value of the `default` parameter:

```
                                                   std::option::Option
fn unwrap_or(self, default: T) -> T
```

Initializing a variable to "y" and reassigning it if there's a command line argument would require a *mutable* variable. However, after the program checks for an argument when starting, it never modifies the variable again. Assigning the value returned from `unwrap_or()` to a variable once doesn't require it to be mutable:

```
src/bin/yes.rs
fn main() {
    let mut out = stdout();
    let y = args().nth(1).unwrap_or("y");

    loop {
        writeln!(out, "{}", y);
    }
}
```

But `nth()` returns an `Option<String>`, and the string literal "y" is a `&str`:

```
error[E0308]: mismatched types
 --> src/bin/yes.rs:9:37
  |
9 |     let y = args().nth(1).unwrap_or("y");
  |                                     ^^^
  |                                     |
  |                                     expected struct `String`, found
`&str`
  |                                     help: try using a conversion
method: `"y".to_string()`
```

Rust has two fundamental string types: `String` and `&str`. A `String` *owns* its heap-allocated contents, which are mutable and resizeable. A *string slice* (`&str`) is an immutable reference to a string allocated elsewhere in memory: as part of the program's executable file, on the stack, or as a `String` on the heap.

The `unwrap_or()` method takes a parameter `default` of type `T`, the generic type of the `Option`. In this case the `Args` struct yields an `Option<String>`. The string literal "y" is a reference (`&str`) to a string which is *statically* compiled into the program. To make the types match, the compiler suggests calling `"y".to_string()`:[35]

```
std::string::ToString
fn to_string(&self) -> String
```

Making a copy of the static string "y" on the heap as a `String` with `to_string()`:

```
src/bin/yes.rs
fn main() {
    let mut out = stdout();
    let y = args().nth(1).unwrap_or("y".to_string());

    loop {
        writeln!(out, "{}", y);
    }
}
```

35 Structs such as `String` are named using `UpperCamelCase`, but methods like `to_string()` are in `snake_case`. The convention is to convert the name to lowercase, conforming to `snake_case`, when using `UpperCamelCase` types in method names.

compiles (with a warning about the unused `Result` from `writeln!`), but running `clippy` returns a warning:

```
warning: use of `unwrap_or` followed by a function call
 --> src/bin/yes.rs:8:27
  |
8 |     let y = args().nth(1).unwrap_or("y".to_string());
  |                           ^^^^^^^^^^^^^^^^^^^^^^^^^^ help: try this:
`unwrap_or_else(|| "y".to_string())`
  |
  = note: `#[warn(clippy::or_fun_call)]` on by default
  = help: for further information visit https://rust-lang.github.io/
rust-clippy/master/index.html#or_fun_call
```

The `unwrap_or()` method *eagerly* evaluates its `default` parameter, allocating a `String` for "y" even if the `Option` has `Some` value. The suggested `unwrap_or_else()` method *lazily* computes a default value only if the `Option` is `None`.

The definition of the `unwrap_or_else()` method includes a `where` clause. A `where` clause specifies *bounds* on a generic type in the parameter list. The bounds for the type `F` in `unwrap_or_else()` specify that it's a function (`FnOnce`) taking an empty parameter list `()` and returning a value of type `T` (the same type as the `Option<T>`):[36]

```
std::option::Option
fn unwrap_or_else<F>(self, f: F) -> T
where
    F: FnOnce() -> T,
```

Instead of defining short functions that are called in only one place in the program, idiomatic Rust programs define anonymous functions called *closures*.[37] A *closure expression* creates a closure from a list of comma separated parameters enclosed between pipes (`||`), followed by an expression body.

A closure expression taking an empty parameter list (`||`) and returning the `String` created by calling `to_string()` on the static string "y" satisfies the bounds specified in the `where` clause for `unwrap_or_else()`. Passing the closure as the parameter to `unwrap_or_else()` resolves the warning from `clippy`, since `unwrap_or_else()` only executes the closure if the `Option` is `None`.

```
src/bin/yes.rs
fn main() {
    let mut out = stdout();
    let y = args().nth(1).unwrap_or_else(|| "y".to_string());

    loop {
        writeln!(out, "{}", y);
    }
}
```

However the compiler still warns that the program doesn't use the `Result` returned by `writeln!`.

36 The `FnOnce` type is a function that can only be called once. The differences between different function types are outside the scope of this book.

37 Closures are called lambdas or lambda functions in some programming languages.

The `Option` and `Result` enums are two of the fundamental data types in Rust. An `Option` represents a value which may or may not be present. A `Result` represents the output of a *fallible* operation: one that can fail. In idiomatic Rust, functions which can either succeed or fail with an error return `Result`s.

The `Result` enum is composed of two *variants*: the successful `Ok`, wrapping a generic value `T`, or an `Err` containing a generic error value `E`.[38] The Rust standard library annotates the enum's definition with the `must_use` *attribute*, which causes the compiler to issue a warning when the program doesn't handle the value of the `Result`:

```
std::result
#[must_use = "this `Result` may be an `Err` variant, which should be handled"]
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Like `Option`s, `Result`s also provide an `unwrap()` method which returns their `Ok` value or panics if they're an `Err`:

```
std::result::Result
fn unwrap(self) -> T
```

Unwrapping the `Result`s returned by `writeln!` is the simplest way to use their values:

```
src/bin/yes.rs
fn main() {
    let mut out = stdout();
    let y = args().nth(1).unwrap_or_else(|| "y".to_string());

    loop {
        writeln!(out, "{}", y).unwrap();
    }
}
```

Unwrapping an error (such as a broken pipe) safely handles it by panicking the program:

```
% target/debug/yes | head -1
y
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value:
Os { code: 32, kind: BrokenPipe, message: "Broken pipe" }', src/bin/
yes.rs:10:32
% echo ${pipestatus}
101 0
```

The error message provides more details about the specific *kind* of error that caused the panic: a `BrokenPipe`.[39] The names of structs, enums, or variants use `UpperCamelCase`, where each capitalized word is joined to the next *without* an underscore ("_"): `BrokenPipe`, `ErrorKind`, `NotFound`.

38 The `Result` enum and both variants are included in the Prelude.
39 Code 32 is the error number for `EPIPE`.

Several variants, representing different categories of errors, compose the `std::io::ErrorKind` enum:

```
                                                    std::io::ErrorKind
enum ErrorKind {
    NotFound,
    PermissionDenied,
    ...
    BrokenPipe,
    AlreadyExists,
    ...
}
```

Instead of unwrapping every `Result` returned by `writeln!` and panicking, the program should exit silently with a status of `141` when encountering a `BrokenPipe` error, emulating the effect of being killed with a signal.[40]

In order to examine each `Result` to determine if it's a `BrokenPipe`, in each iteration of the `loop` a `let` statement assigns the `Result` returned by `writeln!` to the `res` variable. The `match` expression uses the `res` variable as the target value. The pattern in the first `match` arm binds an `Ok` value to the variable `t` and returns it unmodified. The pattern in the second `match` arm binds an `Err` value to the local variable `e` and unwraps the `Result`, causing a panic:

```
                                                    src/bin/yes.rs
fn main() {
    let mut out = stdout();
    let y = args().nth(1).unwrap_or_else(|| "y".to_string());

    loop {
        let res = writeln!(out, "{}", y);

        match res {
            Ok(t) => t,
            Err(e) => res.unwrap(),
        }
    }
}
```

This code doesn't compile, first raising a *warning* because the `match` arm for the `Err` case doesn't use the local variable `e`:

```
warning: unused variable: `e`
  --> src/bin/yes.rs:16:17
   |
16 |             Err(e) => res.unwrap(),
   |                 ^ help: if this is intentional, prefix it with an
underscore: `_e`
```

Unused variables are a common source of bugs and generate warnings from the Rust compiler. The `match` arm isn't using the specific value of the `Err` (yet). The underscore symbol "_" is the *wildcard* pattern and matches any value, without binding it to a name. The compiler *ignores* variable names

40 This emulation isn't perfect, see for details: https://github.com/rust-lang/rust/issues/62569#issuecomment-774546256

prefixed with "_", but renaming the variable to _e doesn't resolve the error from the borrow checker:

```
error[E0382]: use of partially moved value: `res`
  --> src/bin/yes.rs:16:24
   |
16 |             Err(_e) => res.unwrap(),
   |                 --     ^^^ value used here after partial move
   |                 |
   |                 value partially moved here
   |
   = note: partial move occurs because value has type `std::io::Error`,
which does not implement the `Copy` trait
help: borrow this field in the pattern to avoid moving `res.0`
   |
16 |             Err(ref _e) => res.unwrap(),
   |                 ^^^
```

The pattern in the `match` arm handling the `Err` case *partially* moves ownership of `res` into the `_e` variable. The move is *partial* because the pattern only moves the `Err` variant of the `Result`.

Rust's ownership system manages the lifetime of a value by ensuring that each value only has one owner at a time. This memory management doesn't incur a performance penalty at runtime because the borrow checker statically analyzes the program at compile time and inserts calls to *drop* (free) values when their owner goes out of scope.

The `let` statement creates a new binding of the `res` variable on each iteration of the `loop`. The compiler *drops* the `Result` at the end of each `loop` when its owner, the `res` variable, goes out of scope. However, invoking the `writeln!` macro doesn't move ownership of the argument string `y`, which would cause the compiler to drop it at the end of the first iteration of the `loop`.

Macros may *generate* code which calls functions which borrow their parameters. So that printing a variable doesn't take ownership of it, making it inaccessible afterwards, the `writeln!` and `println`! macros implicitly *borrow* their parameters.[41]

Trying to unwrap `res` after the pattern in the `match` arm partially moves it to the `_e` variable generates the error from the borrow checker. However, Rust's ownership rules do allow the creation of multiple shared references to an immutable value. The borrow checker guarantees that the owner of the value outlives all of the references to it, preventing *use after free* bugs: where the program

41 The `print!`, `eprint!`, `eprintln!`, `write!`, and `format!` macros also all implicitly borrow their parameters.

frees (drops) a value but there are still existing references to it. Instead of moving the value, the `ref` keyword binds a name to a *reference* in a pattern:

src/bin/yes.rs

```
fn main() {
   ...
   loop {
       let res = writeln!(out, "{}", y);

       match res {
           Ok(t) => t,
           Err(ref _e) => res.unwrap(),
           //  ^^^
       }
   }
}
```

This compiles and produces a program which panics by unwrapping the `Err`. To query the type of an error, the `io::Err` variant of the `Result` returned by `writeln!` implements a `kind()` method which returns an `ErrorKind` enum:

std::io::Error

```
fn kind(&self) -> ErrorKind
```

Importing the `BrokenPipe` variant into the program's namespace with a `use` declaration and matching the `ErrorKind` returned from calling `kind()` on the `Err` value (after removing the leading "_" from the e variable):

src/bin/yes.rs

```
use std::env::args; // fn args()
use std::io::stdout;
use std::io::ErrorKind::BrokenPipe;
use std::io::Write;
use std::process::exit;

fn main() {
    let mut out = stdout();
    let y = args().nth(1).unwrap_or_else(|| "y".to_string());

    loop {
        let res = writeln!(out, "{}", y);

        match res {
            Ok(t) => t,
            Err(ref e) => match e.kind() {
                BrokenPipe => exit(141),
            },
        }
    }
}
```

But the `match` expression must handle all `ErrorKind` variants:

```
error[E0004]: non-exhaustive patterns: `_` not covered
  --> src/bin/yes.rs:16:33
   |
16 |             Err(ref e) => match e.kind() {
   |                                 ^^^^^^^^ pattern `_` not covered
   |
   = help: ensure that all possible cases are being handled, possibly
by adding wildcards or more match arms
   = note: the matched value is of type `ErrorKind`, which is marked as
non-exhaustive
```

The `match` expression should unwrap any `Result` that is an `Err` with a *kind* other than `BrokenPipe`, causing a panic.[42] As the error message shows, the underscore ("_") is a *wildcard* pattern that matches any value, without binding it to a variable. Adding a wildcard arm which *unwraps* the `Result`:

```
src/bin/yes.rs
fn main() {
    ...
    loop {
        let res = writeln!(out, "{}", y);

        match res {
            Ok(t) => t,
            Err(ref e) => match e.kind() {
                BrokenPipe => exit(141),
                _ => res.unwrap(),
            },
        }
    }
}
```

The arms of a `match` expression are evaluated in order, so first the target value is compared to a `BrokenPipe`, and then any other value matches the underscore wildcard. With this change, the Rust implementation of `yes` successfully emulates the behavior of the original when piping its output to `head`:

```
% target/debug/yes n | head -1
n
% echo ${pipestatus}
141 0
```

42 The original `yes.c` ignores errors and just keeps looping.

Any other errors cause the program to panic, for example when trying to write to a full device. The `/dev/full` device file on Linux simulates a full disk and always returns write errors. The shell's redirection operator ">" *redirects* the output of `yes` to the `/dev/full` file:

```
% target/debug/yes > /dev/full
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value:
Os { code: 28, kind: Other, message: "No space left on device" }', src/
bin/yes.rs:24:14
% echo ${pipestatus}
101
```

Error code 28 is ENOSPC:

```
sys/sys/errno.h
#define ENOSPC    28    /* No space left on device */
```

The `yes` program reads a single *argument* from the command line. The next chapter implements the `head` utility which takes a command line *option* specifying the number of lines to read from each input before exiting.

```
todo!()
```

The `map_err()` method of a `Result` applies a closure to an `Err` value, passing through any `Ok` value. Replacing the `match` on the `Result` with a call to `map_err()` reduces the number of lines of code to 10, the same as the original `yes.c`.