Book of ec

Esthetic / Elegant C

From a first program to systems code

ec edition 1
Print edition · October 2026

Contents

The lessons follow the order of the online book. Chapter and section bookmarks provide direct navigation in this edition.

Learning ec . . . 3

1. A First Program . . . 6

2. Values and Expressions . . . 10

3. Making Decisions . . . 15

4. Repetition . . . 21

5. Functions and Recursion . . . 26

6. Arrays and Search . . . 31

7. Sorting and Matrices . . . 37

8. Records and States . . . 43

9. Text and Unicode . . . 50

10. Pointers and Views . . . 57

11. Errors and Cleanup . . . 63

12. Modules and Projects . . . 68

13. Storage and Allocators . . . 74

14. Collections and Callbacks . . . 79

15. Streams and Files . . . 84

16. Choosing Behavior with Function Values . . . 90

17. Reusing an Algorithm Across Types . . . 96

18. Deciding Work Before the Program Runs . . . 103

19. Sharing Work Without Losing Ownership . . . 110

20. Crossing the Boundary to C and Devices . . . 117

21. A Number Laboratory . . . 125

22. A Small Gradebook . . . 132

23. A Bounded Text Report . . . 139

24. Measuring and Testing . . . 147

Learning ec

A small program can be understood completely. You can follow each value from its first assignment to its last use, explain every branch, and say where every byte of storage comes from. As programs grow, keeping that clarity becomes harder. Learning to preserve it is the subject of this book.

ec means **Esthetic / Elegant C**. It combines direct control over values and memory with a source layout intended to make the shape of a program visible. Indentation marks its blocks. Types describe its data. Functions name its operations. Resources are acquired and released explicitly. There is room for compact code, but little room for an unexplained effect.

The *Book of ec* teaches ec edition 1 through problems: computing a bill, searching a sequence, sorting records, processing text, handling failures and measuring an algorithm. You will see a program take shape, examine why it works, and change it in exercises. The later chapters follow the same approach into generic algorithms, compile-time evaluation, concurrency and foreign interfaces.

The lessons build things. Begin with a greeting, turn it into a small bill calculator, and then learn to make decisions and repeat work. Later lessons construct searches, reports and components that manage their own resources. Each construction starts with a concrete goal, adds small pieces in order, and follows what those pieces do. When a listing replaces an earlier version, the text says so; two versions of one function are alternatives, not two definitions to paste into the same file.

You will also learn to write code for the person who has to understand it next. That person may be you a month from now. A name explains a role; columns show related facts; a blank line separates phases; an explicit allocation exposes a cost. We develop these habits while building the programs, explaining the choice at the point where it matters. ECS-1 gives the habits a consistent visual vocabulary. Their purpose is to make a program's structure, behavior and costs easier to see together.

Reading the Book

Begin with A First Program if you are new to ec. The foundations begin with output, then values, decisions, repetition and named calculations. The data chapters turn individual values into useful collections. The programs chapters explain how components, resources and external data fit together. Advanced chapters expose mechanisms that are easier to understand once you have something concrete to build with them.

The final part contains substantial worked projects. The number laboratory assembles arithmetic, iteration and argument validation. The gradebook develops a small record-processing application. The text report applies byte and Unicode reasoning. The measurement chapter shows how to test claims about both correctness and speed. Each project explains its limits as part of its design.

You do not have to master concurrency or foreign code before building an application. On a first reading, go straight through Foundations; come back to this table when you want to stop and build something.

After these lessons	Build this project	Ideas you will already have
All Foundations and Working with Data, then Errors and Cleanup and Modules and Projects	Number Laboratory	Small number functions, lists of values, reading numbers from text and reporting errors
The same route, especially Sorting and Matrices and Records and States	Gradebook	Grouping related values, putting them in order, calculating averages and displaying results
The same route, especially Text and Unicode and Pointers and Views	Text Report	Reading international text, counting its characters and preparing a report
Storage and Allocators, Collections and Callbacks, then Streams and Files	Extend one of those projects	Deciding who keeps and releases memory, passing functions to other functions and reading files
Collections and Callbacks, plus the earlier error lessons	Measuring and Testing	Checking expected answers, timing work and sorting a collection for the final assignment
The foundations, data and programs lessons	Advanced Techniques	Choosing functions, reusing algorithms for different types and keeping memory available while it is needed

Reading straight through is also useful; the projects gather ideas from earlier parts. The table simply gives you earlier places to stop and make something substantial. A pointer problem is easier to understand when you already have an array whose ownership you care about.

If you already know C, resist the temptation to read ec as C with different punctuation. Integer overflow wraps; expression evaluation has an explicit order; arrays have both value and decay contexts; `Text` is a view, not a zero-terminated character pointer. If you know Python, indentation will be familiar, but storage, integer widths and pointer iteration will require a different way of thinking.

You do not need prior knowledge of systems programming. You do need patience with examples. Before reading an explanation of a loop, try one iteration on paper. Before running a modified program, predict a result. An unexpected result becomes useful when it can be compared with a prediction.

Working with the Examples

A **complete program** includes its imports, helpers and application entry point. Save such a listing in its own module directory; unrelated complete examples must not be combined into a directory containing several `start` definitions. A **fragment** belongs inside the function or module described by its surrounding text. It is deliberately smaller than an application so that the idea under discussion remains visible.

Body additions retain the indentation at which they belong in the named function. A complete function begins at the left margin and belongs above the application entry point. A nested loop-body addition is indented more deeply; its introduction names the containing loop. Keep imports at the top of the file, once per imported name. Read a new listing as either a replacement for the stated piece or an additional piece with its own names; pasting every alternative example into one body can accidentally reassign an existing variable.

The first programs use the hosted language facilities shared by `arm64-macos` and `wasm32-browser`. File, process, thread and device examples state narrower requirements where needed. A browser instance receives output and arguments through its embedding; the language does not supply a universal browser console API.

A compiler translates ec source into a program for a selected target profile. A profile describes the environment where that program will run and which services it provides. Build and launch steps depend on the compiler and hosting environment; the source examples use the standard ec project layout.

Output shown after an example is the result expected from the stated input and the language rules, assuming the described external operations succeed. Exact error-message wording is not part of the lessons; the reason a program is rejected matters more than the sentence used to report it.

A Few Reading Habits

Read a program in three passes. First, identify its inputs and result. Next, follow the normal path. Finally, ask what happens at its boundaries: no input, one item, a full buffer, a failed operation. This is often faster than reading every line with equal attention from the beginning.

Keep a short trace when several values change together. A table with columns for the loop index, current element and accumulated result can reveal an error that remains invisible in a screenful of source. Later, the same habit helps distinguish a pointer from the storage it names and a resource descriptor from the resource it owns.

A short reading log can separate predictions from observations: note the example changed, the result expected, and what a paper trace or actual execution showed. If they disagree, find the first differing step before changing more code.

The exercises ask you to predict, modify and design. Their discussions are included so that you can check your reasoning without a separate answer key. Try the problem before reading the discussion. A correct answer matters, but an explanation of why it stays correct at the boundary matters more.

The worked example is the baseline for the next lesson. Save **On Your Own** changes in a separate example directory, or keep a separate copy of the worked file before experimenting. When a lesson says “continue,” it means continue that worked baseline, not an altered homework version.

Each lesson ends with an **On Your Own** assignment. It extends what you have just built and gives observable requirements against which to check your work. Make the change, predict the supplied cases, and explain one design choice before comparing behavior. The assignment has no complete solution immediately underneath it: this is your opportunity to assemble the ideas yourself.

The Other ec Books

The technical manual is organized for lookup. Use it when you know which feature you need and want its exact interface or restrictions. The reference documentation states the normative rules in prose. The formal definition gives their mathematical form and is authoritative when descriptions differ.

ECS-1 describes the coding style used here. Its central idea is that layout should reveal meaning. Related declarations form columns; independent steps do not acquire padding just because they are adjacent. A comment explains intent, while a section marker provides a landmark. The examples apply those distinctions rather than treating formatting as a decoration added afterward.

Optional further-reading notes appear at the ends of chapters. They name the relevant book and topic so that they remain useful in print. The lesson itself supplies the concepts and steps needed for its examples; you can follow the explanation without consulting the reference books.

A First Program

Our first program will greet a reader, then say what to do next. You will change its words, predict the order of its output, and finish with a greeting card of your own. The first useful result needs only an import and one function.

Write Something You Can Recognize

Create `hello.ec` in an otherwise empty example directory. Enter this complete starting program:

```
from IO import print


func start()
    print("Hello, ec\n")
```

The expected output is:

```
Hello, ec
```

Read the source from the bottom upward once. `print("Hello, ec\n")` asks for output. The quoted part is the text to write; `\n` ends the output line. `func start()` names the place where the host begins the program. The four spaces put the output statement inside that function. Finally, `from IO import print` makes the output operation available by its short name.

A **function** is a named piece of code. A **call**, such as `print(...)`, asks a function to do its work. The parentheses contain what we supply to that call. Empty parentheses in `start()` mean that this entry takes no inputs. Reaching the end of its body returns to the host.

For execution, use your ec implementation's documented build and launch procedure with a hosted target. The book specifies source and expected behavior; it does not supply a compiler command. You can also follow every step on paper: write the output of each call in order. A paper trace is a prediction, not evidence that a program has run.

Change only `Hello, ec` to `Hello, reader`. The next output should change accordingly. Keep the function name `start`, the import and the indentation unchanged. You have now changed the program's behavior by changing its input text.

Add a Second Step

Replace the whole `start` function with this version. Keep the import above it:

```
func start()
    print("Hello, reader\n")
    print("Let us build something.\n")
```

Before reading the next paragraph, write the two output lines in the order you expect. A program performs these statements from top to bottom:

```
Hello, reader
Let us build something.
```

Now remove only the first `\n`. The first call no longer finishes a line, so the second call continues on that same line:

```
Hello, readerLet us build something.
```

Restore the escape. This small mistake teaches a useful distinction: a source line ends a statement, while a newline inside the quoted text ends an output line. `print` does not add one for us. Keep the two calls on separate source lines so their order is visible.

If the source is rejected, compare it with the small listing before changing several things at once. `Print` and `print` are different names. A missing closing quote leaves the text unfinished. A call moved to the left margin is no longer in `start`'s body. Fix the differing spelling or indentation, then repeat the same prediction.

Make a Greeting Card

We can display several lines with one multiline literal. Replace both calls with this single call, keeping the four-space body indentation:

```
    print("""
        ──────────────────────────────────────────

         Hello, reader
         Let us build something.

        ──────────────────────────────────────────
        """)
```

Three quotes open the text and three close it. Everything between them is output text. The closing quotes sit eight spaces in; those eight source spaces are removed from each content line. The extra space before each greeting stays. This lets the source show the card's shape:

```
──────────────────────────────────────────

 Hello, reader
 Let us build something.

──────────────────────────────────────────
```

There is a newline after the bottom rule. Change one greeting line and predict the result again. If a line of ec code appears in the card, you placed it between the quotes: move it after the closing `""")` if you meant it to execute.

Keep the Finished File Readable

The card is our first finished result. Give its source a title and a visible entry-point landmark. These underscore-delimited regions are comments for the reader; they do not print anything. Replace `hello.ec` with this assembled version and check that the output stays the same:

```
________________________________________________________________________________

[ First Program ] Writes a greeting card to standard output.
________________________________________________________________________________

from IO import print

________________________________________________________________________________

[ Application Entrypoint ]
________________________________________________________________________________

func start()
    print("""
        ________________________________________

         Hello, reader
         Let us build something.
        ________________________________________
        """)
```

The title tells us what the file is for. The import tells us what it uses. The final landmark tells us where the application begins. Later helpers will go between the imports and that entry. This is the purpose of the ECS-1 style used throughout the book: help a reader find meaningful parts, without putting a large heading on every small operation.

Keep this file as the starting shape for later examples. Give each new application its own directory: selecting one source file also includes sibling ec files, so two unrelated `start` declarations must not share that directory. The first examples use either the native `arm64-macos` or hosted `wasm32-browser` profile. A profile names the program's execution environment; browser output is delivered through its host embedding. You do not need to implement that embedding to understand the source lesson.

These first output calls discard their success-or-failure result. Their displayed output assumes a successful write. In the text lesson we will check that result explicitly; Errors and Cleanup will teach how to carry a failure through a program.

Exercises

1. Exchange the two greeting lines. What changes in the output, and what stays the same in the program's structure?
2. Change the source comment title but leave the quoted text alone. Does the card change?
3. Add a second blank line inside the multiline literal. Where must it appear to leave a gap between the greeting lines?

Discussion

1. The lines exchange places because literal content is written in its given order. The import and entry point still do the same jobs.

2. Changing a comment title changes only what a source reader sees.

3. To separate the greetings, insert the blank content line between them, within the triple quotes; a blank source line outside the call would not produce output.

On Your Own

Make a visitor card with a name written using ASCII - the basic Latin letters, digits and punctuation - and one short message on separate lines. Keep the top and bottom rules and one space before each message. The Text and Unicode lesson will handle international text. Write the expected card before changing the file, then compare each displayed line with that prediction. Change the source title to describe your card. Explain which change affected output and which helped only the person reading the source.

Further Reading

ec Technical Manual: Program Execution; Literals.

Values and Expressions

Suppose a small workshop sells three parts at 125 cents each. We want the subtotal, then the amount left after paying with 500 cents. Using cents keeps this first calculation in integers and avoids rounding money through a binary real format.

Turn the Greeting into a Bill Calculator

Make a separate copy of `hello.ec` named `bill.ec` in its own directory. Keep the imports and `start`; change the file title and displayed greeting to identify a bill calculator. We will add the calculation after the banner-print call. The body listings keep their four-space indentation so they can be inserted directly into `start`.

A **variable** is named storage for a value that can change. In `quantity Int32 = 3`, `quantity` is the name, `Int32` says it holds a signed integer, and `3` is its initial value. First put the three facts we know into variables:

```
    quantity  Int32 = 3
    unitPrice Int32 = 125
    paid      Int32 = 500
```

`quantity` counts parts; `unitPrice` and `paid` count cents. These names state the roles we will use in the calculation. Writing `a`, `b` and `c` would force a reader to memorize a separate translation. We use camelCase for ordinary multiword names, so `unitPrice` reads as one name with two words.

The three declarations form a table of inputs. Their type and assignment columns line up because their jobs are parallel. Add the calculation immediately below them, separated by one blank line:

```
    subtotal = quantity * unitPrice
    change = paid - subtotal
```

Read the order aloud: find the price of all the parts, then subtract that price from the payment. `change` depends on the previous result. We leave these sequential steps unpadded rather than making them look like independent input rows. This is a useful habit: alignment promises a relationship, so use it only when that relationship is real.

Finally, add the two output calls:

```
    print("Subtotal: [ {subtotal} ] cents\n")
    print("Change:   [ {change} ] cents\n")
```

Inside a quoted output message, `{subtotal}` inserts the current value of `subtotal`; `{change}` does the same for `change`. This is **interpolation**. The surrounding square brackets are ordinary printed punctuation, and `\n` ends the line. The labels and brackets keep each amount next to its unit and make repeated output easy to scan. Before running or tracing the program, calculate the two answers on paper. The expected amounts are 375 and 125. A complete `start` checkpoint follows the next experiment.

Read `subtotal = quantity * unitPrice` as "calculate the right side and store its value under the name on the left." An **expression**, such as `quantity * unitPrice`, produces a value. A **statement**, such as the whole

assignment, performs a step. This distinction will matter when we put expressions in conditions and statements in loop bodies.

In this fragment, `subtotal` has no earlier binding, so the assignment introduces it. A later `subtotal = subtotal + 10` updates the existing variable. Reusing the name does not create a second subtotal. It also does not make a spreadsheet formula: if `quantity` changes afterward, the stored `subtotal` keeps its most recently assigned value: 375 before that update, or 385 after it. Changing an input does not recalculate the result.

The separate statement `quantity++` increases its stored value by one; `quantity--` decreases it by one. Each changes the variable when that statement runs. To see this, put `quantity++` immediately before the subtotal calculation: starting from 3 now charges for 4. Move it after both calculations and the printed amounts still describe 3. Restore the original statements before continuing.

Try changing only `quantity` to 4. The subtotal becomes 500 and the change zero. At 5 parts the result is -125: the arithmetic has represented a shortfall, but the program has not decided what to do about it. Keep that distinction in mind for the next lesson, where we add a decision to this bill.

Keep this checkpoint before trying the side experiments below. Replace your whole `start` with the following function if you want to compare the assembled version. Keep the file's import and source banners. There is only one `start`:

```
func start()

    print("""
        ____________________________________________

         Bill Calculator
        ____________________________________________
        """)

    quantity  Int32 = 3
    unitPrice Int32 = 125
    paid      Int32 = 500

    subtotal = quantity * unitPrice
    change = paid - subtotal

    print("Subtotal: [ {subtotal} ] cents\n")
    print("Change:   [ {change} ] cents\n")
```

A Type Makes a Promise

A **bit** has two possible states, conventionally written 0 and 1. Eight bits make a byte. `Int32` uses 32 bits, or four bytes, for a signed integer: it can represent negative numbers as well as zero and positive numbers. Its range is −2,147,483,648 through 2,147,483,647. An unsigned type, whose name starts with `Uint`, uses its range for nonnegative numbers instead. `Uint08` therefore represents 0 through 255.

A declared binding has one type throughout its life. Reassigning `quantity` changes its value, not the width or interpretation of its storage. Here **binding** means the association between a name and the variable or value it denotes; in our mutable local examples, you can read it as “named variable.”

When the right side already determines the type, `subtotal = quantity * unitPrice` can introduce a local without spelling it again. Both operands are `Int32`, so the result is `Int32`. A plain integer literal with no other context defaults to `Int32`; a real literal defaults to `Real32`.

An explicit type is useful when the width matters. `UintSize` is the type of lengths and indices; it follows the selected target's pointer width. `Uint08` holds a byte-sized unsigned integer. Fixed-width names make the intended representation visible, while size types describe quantities tied to addressable storage.

A declaration without an initializer, such as `total Int32`, does not supply zero. ec checks direct local reads for definite initialization, but you should begin with an initializer whenever the calculation naturally has one. The sum of no items is zero; that is a reason to write zero, not a property of uninitialized storage.

Arithmetic Has Boundaries

Integers are finite. For an N-bit integer, ec defines addition, subtraction and multiplication modulo 2^N: only the lowest N result bits are kept. A signed type interprets those bits in two's complement. For example, the following local calculation produces 4:

```
value Uint08 = 250
wrapped = value + 10
```

There are 256 possible byte values. After 255, adding one wraps to zero; adding four more reaches 4. Equivalently, 260 leaves remainder 4 after division by 256. For this eight-bit type, the modulus is $2^8 = 256$, not eight. Wrapping is predictable, but it is not necessarily the answer your application wanted. A quantity of 260 bytes cannot be represented by storing 4 in a `Uint08`. Choose a sufficient type and check a range when a value comes from outside your program.

Integer division discards the fractional part toward zero. Thus `17 / 5` is 3 and `17 % 5` is 2. The relationship `17 = 3 * 5 + 2` explains why quotient and remainder are useful together. With negative dividends, the remainder follows the dividend's sign: `-17 / 5` is -3 and `-17 % 5` is -2. Division by zero is a fatal error, so a divisor obtained from input needs a check.

Parentheses make grouping visible. Multiplication binds more tightly than addition: `2 + 3 * 4` is 14, whereas `(2 + 3) * 4` is 20. Operands evaluate left to right, but that does not change the expression's grouping. Precedence answers which operations are nested; evaluation order answers when their operands are evaluated.

Conversions Are Written Down

An integer can widen to a larger integer of the same signedness without a cast: passing an `Int32` to an `Int64` parameter preserves its value, and adding typed `Int32` and `Int64` values uses `Int64`. A narrowing or sign change still needs a conversion. ec also does not silently combine an `Int32` and a `Real64` into a common numeric type. Write a conversion when the calculation changes representation:

```
completed Int32 = 7
capacity  Int32 = 10

fraction = Real64(completed) / Real64(capacity)
```

This fragment assumes positive capacity and gives a binary approximation to 0.7. Converting after division would be too late: `Real64(completed / capacity)` first computes the integer quotient 0.

A conversion of a typed integer may wrap to the target width. A literal directly checked against that type must fit. Consequently, `Uint08(300)` is rejected, while converting an `Int32` variable holding 300 to `Uint08` produces 44. The two forms answer different questions: whether a literal fits its stated type, and how an existing value is represented in another type.

Binary real arithmetic follows IEEE 754. Many decimal fractions are approximate: just as one third has no finite decimal expansion, 0.1 has no finite binary expansion. `Real64` offers more precision than `Real32`, but neither stores every decimal fraction exactly. Real arithmetic can also produce positive or negative infinity; NaN means "not a number" and can arise from an invalid real operation such as zero divided by zero. This differs from integer division by zero, which is fatal. For now, use integers for counts and real types for measurements that require fractions. Formatting a number with fewer decimal places changes its display, not the stored value.

Constants and Bit Patterns

A local `const` binding cannot be reassigned after initialization:

```
    taxPercent const Int32 = 20
```

It is useful when a name expresses an invariant within a calculation. Constantness belongs to the binding; later we will distinguish a constant pointer from constant pointed-to storage.

Hexadecimal uses `#`, as in `#FF`. Binary uses a lowercase trailing `b`, as in `1111_0000b`. Separators can improve legibility inside a digit sequence. These spellings produce integers, not a separate kind of color or bit-string object.

Binary positions, from right to left, are worth 1, 2, 4, 8 and so on; `101b` is therefore 4 + 1, or decimal 5. Hexadecimal uses sixteen digits, `0` through `9` followed by `A` through `F` for ten through fifteen. `#2A` is 2 × 16 + 10, or decimal 42. Four binary digits fit one hexadecimal digit, which makes hex convenient when reading bytes: `#F0` and `1111_0000b` both mean 240.

Exercises

1. Compute the number of full hours and leftover minutes in 135 minutes.
2. Using the typed `Int32` variables `completed` and `capacity` above, explain why `Real64(completed / capacity)` does not produce 0.7.
3. A byte holds 255. What does incrementing it produce? Is that a sensible way to count arbitrary user records?

Discussion

1. One solution, placed inside `start` with `print` imported, is:

```
    minutes Int32 = 135

    hours     = minutes / 60
    remaining = minutes % 60

    print("Hours:   [ {hours} ]\n")
    print("Minutes: [ {remaining} ]\n")
```

The answer is two hours and fifteen minutes. Quotient and remainder describe the same division, so they form a natural pair of related declarations. Try 0, 59 and 60 as well: they distinguish an empty duration, an unfinished hour and an exact hour.

2. The division happened before conversion. Convert an operand to the desired real type before dividing, and ensure the other operand has that same type.

3. The byte wraps to zero. That behavior may be useful for a deliberately cyclic counter, but record counts require a suitable size type and a capacity policy. A well-defined arithmetic result is not automatically a valid application result.

A Closer Look: Literals and Their Surroundings

The earlier division used `completed` and `capacity`, two variables already declared as `Int32`. Dividing them computes an integer quotient. Once that division has discarded a fraction, converting the answer cannot recover it.

Written number literals can behave differently: they can take their type from the surrounding expression. In `Real64(7 / 10)`, the surrounding `Real64` supplies the type for the literal-only calculation, so the result is approximately 0.7. It does not perform an integer division and then recover its missing fraction.

Compare the operands when reading such an expression. The literals can receive a type here; the named variables already have one. For the calculator we built, converting the typed operands before dividing keeps that choice visible.

On Your Own

Extend `bill.ec` to sell two kinds of parts. Use separate quantities and unit prices, calculate each line total, then the combined subtotal and change. For two items at 150 cents and three at 80 cents, paid with 600 cents, expect subtotal 540 and change 60. Also check both quantities zero and an exact payment. Choose names that reveal which quantity belongs to which price, and use alignment to make the two input groups readable without disguising the dependent calculations.

Further Reading

ec Technical Manual: Types and Conversions; Expressions and Operators.

Making Decisions

A calculation follows one path. A useful program usually has several. A ticket price may depend on age, an input may be outside the accepted range, and a search may or may not find its target. A branch lets us express which path belongs to which condition.

Make the Bill Explain a Shortfall

Continue with the worked three-input `bill.ec` from Values and Expressions; keep the two-product homework in its separate directory. A negative change is meaningful to us, but a customer should receive an explanation. Keep the input and calculation statements, and replace the two output calls with these statements inside `start`:

```
    print("Subtotal: [ {subtotal} ] cents\n")

    if change < 0
        print("Still owed: [ {-change} ] cents\n")
    else
        print("Change: [ {change} ] cents\n")
```

The comparison asks whether `change` is negative. Exactly one body runs. With the original three parts and 500-cent payment, the second body prints change of 125 cents. With five parts, the first prints 125 cents still owed; `-change` displays the positive amount of the shortfall. With four parts, change is zero and the `else` body still runs. Those three cases exercise both sides and their boundary.

Each output action occupies its own line under the condition that chooses it. That layout makes the two possible effects visible before you read their text. Keep this completed bill calculator; the next construction will turn another pricing decision into a reusable helper.

Build a Ticket-Price Helper

Create `tickets.ec` in another example directory using the first program's file structure. Keep `print` and `start`, and change the displayed title to identify ticket prices. Begin with the simplest policy: every ticket costs 900 cents. Add this helper above the entry-point marker:

```
func ticketPrice(age Uint32) -> Uint32
    return 900
```

The parameter `age` receives the caller's age; this first version does not use it yet. The result annotation says the helper returns an unsigned 32-bit price. Put these two lines inside `start`, after its banner:

```
    price = ticketPrice(10)
    print("Price: [ {price} ] cents\n")
```

This first version reports 900. We now have a call whose result can reveal the next change. Add a child-price test before `return 900` in the helper:

```
    if age < 12
        return 500
```

Keep the indentation shown: the test belongs in the helper, and its return belongs under the test. Now age 10 returns 500 before reaching the ordinary price, while age 30 reaches 900. Add the senior-price test after the child test and before the final return. Here is the complete replacement helper; keep only this version in the file.

The final policy assigns a ticket price in cents. Children younger than twelve pay 500, adults pay 900, and people aged sixty-five or more pay 600. The function's input is a nonnegative age. This is a **module-level function fragment**: put it at the left margin, after the imports and before the application-entry banner and `start` in your file. It is a helper that `start` can call, not a declaration nested inside `start`.

```
func ticketPrice(age Uint32) -> Uint32

    if age < 12
        return 500

    if age >= 65
        return 600

    return 900
```

A function is a named calculation; its parameter is a local value supplied by the caller. The arrow states its result type. Read the body from top to bottom. Once a `return` runs, that call is finished. A ten-year-old never reaches the second test.

Keep the two caller lines already in `start`; they now use the completed policy:

```
    price = ticketPrice(10)
    print("Price: [ {price} ] cents\n")
```

The call supplies 10 to the parameter named `age`. The helper tests `10 < 12`, returns 500, and the call expression becomes that returned value. Only then is `price` initialized and the next statement printed. Defining the helper alone produces no ticket output. Keep two blank lines between neighboring top-level function declarations; a section banner already has its own spacing.

Indentation shows which statements a condition controls. `if` is four spaces inside the function, and its `return` is another four spaces in. The last `return 900` is back at the function-body level. It runs after both preceding conditions have failed, rather than belonging to the second condition.

These branches are expanded because each exits the function. ECS-1 does not hide a return at the end of a compact row. The result is still short, but the exits are easy to see.

Keep this checkpoint before trying the side experiments below. Replace your whole `start` with the following function if you want to compare the assembled version. Keep the file's import and source banners, and keep the helpers above the entry point. There is only one `start`:

```
func start()

    print("""
        ____________________________________________

         Ticket Prices
        ____________________________________________
        """)

    price = ticketPrice(10)
    print("Price: [ {price} ] cents\n")
```

Boundaries Tell the Story

Testing only an age of thirty would exercise the ordinary price and leave both boundaries unexplored. A more useful table is:

Age	Price in cents	Reason
0	500	Below twelve
11	500	Last child price
12	900	First ordinary price
64	900	Last ordinary price
65	600	First senior price

The choice of `< 12` rather than `<= 12` matters. Boundary examples expose that decision more clearly than a long verbal promise that the function handles ages correctly.

Conditions and Logic

Comparisons produce `Bool`, a type with exactly two values, `true` and `false`. Equality is `==`; assignment is `=`. The other comparisons are `!=` for not equal, `<` and `>` for strict ordering, and `<=` and `>=` to include equality. The logical operators combine answers: `&&` requires both true, `||` requires at least one true, and `^^` requires exactly one true. `!!` reverses true and false. A single `!` complements integer bits; it is not Boolean negation.

For a separate admission experiment, copy the checkpoint to another example directory. In that copy, replace the ticket calculation and output inside `start` with the following body fragment. Keep the banner and `print` import:

```
    age       Uint32 = 14
    hasTicket Bool   = true

    if (age >= 12) && hasTicket
        print("Admission allowed\n")
    else
        print("Admission refused\n")
```

The right side of `&&` is evaluated only when the left side is true. The right side of `||` is evaluated only when the left side is false. That is useful when the second question makes sense only after the first has been answered.

Here both questions answer true, so only the allowed line is printed. Change `hasTicket` to `false` to take the `else` path. Then try age 10 with a ticket: the first question fails, and the program refuses admission without needing the second question. An `else` pairs with the `if` at its indentation level; both outcomes continue after the branch unless they explicitly return or otherwise leave. Return to the original ticket checkpoint when you finish the experiment; its two caller lines are already intact.

When more than two outcomes should continue with later work, use `else if` to test another condition only after the earlier one fails. Our ticket helper instead uses early returns, so its two separate `if` statements already express the intended order. Changing independent `if` statements into an `else if` chain can change behavior when earlier branches do not return.

For example, `(divisor != 0) && (value % divisor == 0)` does not divide by zero: when the divisor is zero, the remainder expression is skipped. Reversing the operands would lose that protection. Short-circuiting is therefore part of program behavior, not merely a possible optimization.

Integers and pointers can also serve as conditions, with zero or null meaning false. Prefer a comparison when it explains the intended question. `if count > 0` says more about a count than relying on its numerical truth value. Text, slices and structs have no implicit truth value; ask for the property you need.

Choosing among Named Cases

A `case` is helpful when one selector is compared with several constant alternatives. Here is a module-level function that counts the days in a month while leaving February to an explicit flag:

```
func daysInMonth(month Uint32, leap Bool) -> Uint32

    case month
        4, 6, 9, 11
            return 30
        2
            return (leap ? 29 : 28)
        1, 3, 5, 7, 8, 10, 12
            return 31
        else
            return 0
```

Zero is a deliberate sentinel for a month outside 1 through 12. The example does not claim that zero is a day count for a valid month. The selector is evaluated once, and a chosen branch does not fall through to the next one.

A **sentinel** is a chosen value that stands for a special outcome. Here every real month has at least 28 days, so zero cannot be confused with a successful count. For `daysInMonth(2, true)`, month 2 selects the February branch, the true flag chooses 29, and the function returns 29. `daysInMonth(13, false)` selects `else` and returns the sentinel. The caller supplies the leap-year fact; this function does not calculate it from a year.

The grouped conditional `(leap ? 29 : 28)` chooses one value. It is useful for a small expression with two simple alternatives. A longer sequence deserves a branch. The parentheses are part of the admitted conditional form, not optional visual punctuation.

Keeping State Obvious

If both branches assign one local result, initialize it on every path that reaches its use. A local declared without an initializer is not automatically zero. A missing `else` can therefore be a missing value, not just a missing action.

Returning early often makes this easier to follow. In `ticketPrice`, each special case ends immediately and the ordinary result sits at the end. For other tasks, keeping one local and assigning it in an aligned branch table is clearer. Choose according to the shape of the problem, not a rule that every function must have one return.

A Compact Table When the Work Really Is a Table

Sometimes a branch only chooses a value and all paths continue with the same work. In the separate experiment copy, replace the calculation and output after the banner with this body fragment:

```
    age   Uint32 = 10
    price Uint32 = 0

    if      age < 12  : price = 500
    else if age >= 65 : price = 600
    else              : price = 900

    print("Selected price: [ {price} ] cents\n")
```

Each row asks one question and assigns one price; the last row covers the remaining ages. The aligned colons let you compare the policy as a table. This is useful compactness because the rows share a shape and no row exits the function. The helper's early-return version stays expanded: its exits are important actions in the reading path.

If a row grows to include validation, output or an exit, expand the whole group. A short line count is useful only while it keeps the decisions easy to see. Use the original checkpoint for the homework; the helper version is the reusable component we keep.

Exercises

1. Add a free ticket for age zero without changing the other prices.

2. What happens if `case month` has no `else` and none of its alternatives matches?

3. Write a condition meaning “the value is between 10 and 20 inclusive.” Why is `10 <= value <= 20` unsuitable?

Discussion

1. Insert this branch as the first phase of `ticketPrice`, before `if age < 12`:

```
    if age == 0
        return 0
```

The indentation shown puts the test inside the helper and the return inside the test. If this check came after the child-price branch, the earlier return would already have handled zero. Test ages 0, 1, 11 and 12 to check both the new special case and the boundary it must preserve.

2. An unmatched case continues after the entire case. It does not invent a default result. Removing `else` from this particular `daysInMonth` without adding a later return leaves a path that reaches the function's end without its required `Uint32` result, so that altered function is rejected.

3. Write `(value >= 10) && (value <= 20)`. ec comparisons do not chain. The two comparisons make their two bounds explicit and join their Boolean results.

On Your Own

Add a group-ticket calculation to `tickets.ec`. Use the existing `ticketPrice` helper for one nonnegative age and multiply by a count of tickets. The count must be between 1 and 10; for a value outside that range, print a refusal instead of a price. Check age 11 with count 2 (1000 cents), age 12 with count 2 (1800), and counts 0 and 11 (refusal). Keep the refusal and successful output visibly separate. The accepted count bound keeps these prices within Uint32.

Further Reading

ec Technical Manual: Statements and Control Flow.

Repetition

A loop turns one useful step into a repeated calculation. Its hardest question is often not what to do in the body, but what must be true when the body begins and why repetition will end.

Add One Number, Then the Next

Create `sums.ec` in its own example directory using the first program's imports, banner and entry point. Change its titles to identify a sum calculator. Our goal is to add every integer from 1 through a chosen limit. Start inside `start`, after its banner, with this body fragment:

```
    total Uint64 = 0
    next  Uint64 = 1

    total += next
    next++

    print("Total: [ {total} ]; next: [ {next} ]\n")
```

The first two lines describe the current state: nothing added yet, and 1 waiting to be added. `+=` adds into the stored total. `++` advances the next value. The output is total 1, next 2. Repeat the two update statements once before the print, and the result becomes total 3, next 3. We have found the two-step action to repeat; copying it four or a thousand times would obscure that simple structure.

Put a Boundary Around the Repetition

Replace the calculation above, retaining its position after the banner, with:

```
    limit Uint32 = 4

    total Uint64 = 0
    next  Uint64 = 1

    loop next <= Uint64(limit)
        total += next
        next++

    print("Sum: [ {total} ]\n")
```

The condition is tested before every repetition. Both updates belong inside the loop; the print belongs outside, so it reports one final sum. Predict the result with limit 0 before reading on: the condition is false at once and no addition happens. The initialized zero is already the right answer.

Give the Calculation a Name

We want to ask for different sums without mixing output into the algorithm. Move the calculation into `sumTo`, a helper above `start`, and pass the limit as its parameter. To keep the arithmetic bounded, the helper accepts `Uint32` and accumulates in `Uint64`. This is its assembled definition:

```
func sumTo(limit Uint32) -> Uint64

    total Uint64 = 0
    next  Uint64 = 1

    loop next <= Uint64(limit)
        total += next
        next++

    return total
```

`loop` tests its condition before each body execution. A true answer enters the body; a false answer continues at the first statement after it, here `return total`. The body uses `total += next`, shorthand for adding `next` to the current total and storing the result back. Then `next++` advances the counter by one. Neither variable is recreated on each iteration: their declarations precede the loop.

In `start`, replace the temporary variables, loop and final print with these two lines after its banner:

```
    answer = sumTo(4)
    print("Sum: [ {answer} ]\n")
```

The file now contains the import, `sumTo`, and the entry point that displays its answer. The algorithm can be reused without printing. We extracted a named calculation with a clear input and result, rather than hiding each individual addition in a separate function.

For a limit of four, the trace is small enough to inspect completely:

Before the body: next	Before the body: total	After adding next
1	0	1
2	1	3
3	3	6
4	6	10

The fifth condition is false, so the result is 10. For a zero limit, the first condition is already false and the result is zero. A loop is allowed to execute no body iterations at all.

Keep this checkpoint before trying the side experiments below. Replace your whole `start` with the following function if you want to compare the assembled version. Keep the file's import and source banners, and keep the helpers above the entry point. There is only one `start`:

```
func start()

    print("""
        ──────────────────────────────────────────

         Sum Calculator
        ──────────────────────────────────────────
        """)

    answer = sumTo(4)
    print("Sum: [ {answer} ]\n")
```

A Useful Invariant

Before each iteration, `total` is the sum of all integers from 1 up to but not including `next`. That sentence is the loop invariant. It is true initially because the sum before 1 is empty and `total` is zero. Adding `next` and incrementing it restores the same relationship for the next iteration.

When the condition becomes false, `next` has passed the limit. The invariant then tells us that `total` is the complete requested sum. This gives a reason for correctness, rather than relying on four favorable sample values.

The widths matter. The maximum sum for a `Uint32` limit fits `Uint64`, and `next` can represent one more than the largest `Uint32`. Using a `Uint32` loop variable would wrap when incrementing the maximum limit and could keep the condition true. A loop's progress argument depends on its arithmetic, not just its written `++`.

Leaving a Loop

A bare `loop` has no condition and repeats until something leaves it. `break` leaves the nearest enclosing loop. `continue` begins its next iteration. Neither jumps across a function boundary. In this function-body fragment, with `print` imported, the counter reaches three and then execution continues after the loop:

```
    count Uint32 = 0

    loop
        if count == 3
            break
        count++

    print("Count: [ {count} ]\n")
```

`break` ends only the loop; the print still runs. A `return` at the same point would end the function, skipping that later statement. If the increment were missing, the count would stay zero and this loop would never reach its break condition.

A **factor** divides an integer with remainder zero. For example, 3 is a factor of 12 because `12 % 3` is zero. A prime number is an integer greater than one whose only positive factors are one and itself; a composite number has a factor in between. The following helper searches for such a factor. It returns zero when the supplied number is below two or has no divisor between two and its square root:

```
func smallFactor(number Uint32) -> Uint32

    if number < 2
        return 0

    divisor Uint32 = 2

    loop divisor <= number / divisor
        if number % divisor == 0
            return divisor
        divisor++

    return 0
```

The quotient test avoids multiplying `divisor * divisor`, which could overflow a fixed-width integer. The divisor starts at two, so neither division uses zero. A number such as 49 returns 7; a prime such as 47 returns zero.

Follow 15 through the search. With divisor 2, the loop test is `2 <= 15 / 2`, or `2 <= 7`; the remainder is 1, so the body advances the divisor. With divisor 3, the test is `3 <= 5` and the remainder is zero, so the function returns 3 immediately. With input 2, the first test is `2 <= 1`, false, and the function returns zero without entering the loop. This is the correct no-factor outcome for the smallest prime.

Zero here means "no factor found in this search." It does not distinguish a prime from the intentionally excluded inputs 0 and 1. The caller must retain that distinction if it wants to classify numbers. We will do so in the Number Laboratory.

Choosing a Loop Shape

The sum began with `loop` so that we could build its condition and updates one step at a time. Once those parts are understood, a counting `for` can collect them into one header. This is an ordinary way to write a counted calculation, including a single loop with a short body:

```
    total Uint64 = 0

    for next Uint64 = 1; next <= Uint64(limit); next++: total += next
```

Read the clauses as initialize once, test before each iteration, then advance after the body. This fragment computes the same sum as the earlier calculation. Here `next` belongs to the loop and cannot be used after it; the earlier declaration kept it in the surrounding scope. The compact body is readable because it does one simple addition. A longer body belongs on indented lines.

Keep `loop condition` when repetition is best described by changing state, such as searching until a node is found or retrying while a device is busy. Once an array or slice is available, `for item in items` expresses visiting its elements. That form supplies element pointers and knows the length of the sequence; the data chapters explain how to read through those pointers. Choose the form that exposes the task rather than mechanically converting every loop to one spelling.

A counting `for` also keeps its step on a `continue` path. Here is a function that sums only odd integers below a small bound:

```
func sumOddBelow(limit Uint32) -> Uint64

    total Uint64 = 0

    for value Uint64 = 0; value < Uint64(limit); value++
        if value % 2 == 0
            continue
        total += value

    return total
```

Translating this mechanically into a `loop` with `value++` at the bottom would skip the increment whenever `continue` runs. Changing the shape of a loop requires checking its control flow as well as its appearance.

Read the three clauses in order: initialize `value` once; test before each iteration; execute the increment after the body, including after `continue`. For limit 5, values 0, 2 and 4 take the skip, while 1 and 3 are added. The result is 4. Once value reaches 5, the test fails. A `break` would leave without executing another increment.

Exercises

1. Trace `sumTo(3)` without looking at the earlier table. State the invariant before every body execution.

2. Why does `smallFactor` need no trial divisor larger than the square root?

3. Modify the odd-sum function to sum values divisible by three. Predict the result for a limit of ten.

Discussion

1. The totals before the three bodies are 0, 1 and 3; the returned total is 6. In each row, the total contains precisely the integers below the next value.

2. If a composite number is a product of two factors and both factors exceeded its square root, their product would exceed the number. At least one factor is therefore within the searched range.

3. Skip a value when `value % 3 != 0`. The included values below ten are 0, 3, 6 and 9, giving 18. Including zero changes nothing, but noticing it prevents a silent mismatch between the condition and the explanation.

On Your Own

Add a `sumSquaresTo(limit Uint32) -> Uint64` helper to `sums.ec`. Sum the squares from 1 through the supplied limit; the helper's accepted range is 0 through 10,000. In `start`, reject a larger limit before calling it. Leave `sumTo` intact so the caller can report both answers for one accepted input. For limits 0, 1, 3 and 4, the new helper must return 0, 1, 14 and 30. Use Uint64 for the value being squared and the accumulator. State what the accumulator contains before each iteration, and explain why the accepted input bound matters even with a wide type.

Further Reading

ec Technical Manual: Statements and Control Flow.

Functions and Recursion

A function gives a calculation a name, a set of inputs and a result. Good function boundaries make a program easier to explain: “find the greatest common divisor” is a meaningful step; “perform lines seven through twelve” is not.

We have already called helpers from `start`. Now we will follow what happens *inside* a call, then use the same model when a function calls itself. All named function definitions in this lesson belong at module level. The short call and assignment fragments belong inside a caller such as `start`.

Start with One Reduction

Create `divisors.ec` in its own directory with the greeting program's import and entry-point structure. Rename the banners for a divisor calculator. We will build a calculation that can simplify a ratio: for 48 and 18, the common divisor 6 reduces the pair to 8 and 3.

Two integers have a greatest common divisor: the largest positive integer dividing both, except that we must choose a convention for the pair zero and zero. Euclid's algorithm replaces a pair `(a, b)` with `(b, a % b)` until the second value is zero.

Try just one reduction inside `start`, after its banner:

```
    a Uint32 = 48
    b Uint32 = 18

    remainder = a % b
    a = b
    b = remainder

    print("Reduced pair: [ {a} ], [ {b} ]\n")
```

The output is 18 and 12. The remainder must be saved before assigning `a = b`, because that assignment replaces the old 48. These three statements are a sequence of changes, so we leave them as plain steps. Lining up their assignment signs would add the appearance of a table without explaining their dependency.

Repeat the Reduction in a Helper

Now use the previous lesson's loop idea: repeat while `b` is nonzero, then return `a`. Add this definition above the entry-point marker. It replaces the one-step experiment with a reusable calculation:

```
func greatestCommonDivisor(a Uint32, b Uint32) -> Uint32

    loop b != 0
        remainder = a % b
        a = b
        b = remainder

    return a
```

This function defines the result for `(0, 0)` as zero. When only one argument is zero, it returns the other. Every remainder operation has a nonzero divisor because of the loop condition.

For `(48, 18)`, the pairs are `(48, 18)`, `(18, 12)`, `(12, 6)` and `(6, 0)`. The final answer is 6. The important fact is that replacing a pair with the second value and the remainder preserves its common divisors.

Why does it preserve them? Integer division writes `a = q * b + remainder`. Any integer dividing both `a` and `b` must also divide the remainder. Conversely, an integer dividing both `b` and the remainder divides `a` too. The pair changes, but its set of common divisors does not. This is the mathematical reason the repeated reduction works.

Parameters Are Local Values

The assignments to `a` and `b` change this call's parameters. They do not change the variables that supplied the arguments. Ordinary parameters are passed by value. Later, a pointer parameter will give a function a way to modify a caller's storage explicitly.

Keep the banner call in `start`, but replace the one-step variables, updates and print with this caller. Keep `print` imported:

```
    first  Uint32 = 48
    second Uint32 = 18

    divisor = greatestCommonDivisor(first, second)
    print("GCD: [ {divisor} ]; inputs: [ {first} ], [ {second} ]\n")
```

The output reports a GCD of 6 and inputs still equal to 48 and 18. The names `first` and `second` belong to the caller; `a` and `b` belong to the called function. An **argument** is a value supplied at the call; a **parameter** is the name receiving it inside the function. A local such as `remainder` is visible only in the block where it is declared and its nested blocks. The caller cannot reach it by writing the same name.

The temporary `remainder` matters because assignments occur in order. Writing `a = b` and then `b = a % b` would use the new `a` and produce zero immediately. One alternative is tuple destructuring:

```
        a, b = (b, a % b)
```

A **tuple** groups a fixed number of values in order. In `(b, a % b)`, the first component is the old `b`, and the second is the remainder calculated from the old pair. Both are evaluated before either target is assigned. The comma-separated targets receive the components in the same order; this is called **destructuring**. The fragment can replace the three body statements above, but the longer version is helpful while learning the algorithm.

The file now calculates a divisor without making its caller's inputs disappear. To reduce the ratio, add `print("Reduced ratio: [ {first / divisor} ] : [ {second / divisor} ]\n")` after the GCD print for these nonzero inputs. It reports 8 : 3. The general caller must check for a zero divisor before using that division, because our helper deliberately returns zero for `(0, 0)`.

The iterative implementation and the recursive version below are alternatives worth comparing. Keep the iterative helper and add the recursive helper under its distinct name; do not move either definition inside `start`.

Keep this checkpoint before trying the side experiments below. Replace your whole `start` with the following function if you want to compare the assembled version. Keep the file's import and source banners, and keep the helpers above the entry point. There is only one `start`:

```
func start()

    print("""
        ____________________________________________

         Divisor Calculator
        ____________________________________________
        """)

    first  Uint32 = 48
    second Uint32 = 18

    divisor = greatestCommonDivisor(first, second)
    print("GCD: [ {divisor} ]; inputs: [ {first} ], [ {second} ]\n")
    print("Reduced ratio: [ {first / divisor} ] : [ {second / divisor} ]\n")
```

A Function Can Call Itself

The same mathematical reduction gives a recursive function:

```
func greatestCommonDivisorRecursive(a Uint32, b Uint32) -> Uint32

    if b == 0
        return a

    return greatestCommonDivisorRecursive(b, a % b)
```

The first branch is the base case. Without it, the next remainder could divide by zero. The recursive case makes progress because the nonnegative remainder is smaller than the previous nonzero second argument.

Each recursive call has its own `a` and `b`. Follow `(48, 18)` inward and then back out:

Call receives	What it waits for	What it returns
48, 18	The call with 18, 12	6
18, 12	The call with 12, 6	6
12, 6	The call with 6, 0	6
6, 0	Nothing: the base case returns immediately	6

The innermost call supplies the answer first. Its caller then returns that answer to its own caller, until the original call completes. A **call stack** is the record of these active calls and where each should return. This example does no extra arithmetic on the way out, but the return journey still matters.

Recursion does not imply a special recursive object or captured environment. It is another function call. Its parameters and return path have to exist while the call is active. Do not assume that a compiler eliminates the call merely because it occurs in a return expression. A stack limit remains part of the execution environment.

For Euclid's algorithm, the iterative version is straightforward and makes storage use obvious. Recursion becomes more attractive when the data itself has a recursive shape, such as a tree, or when a divide-and-conquer algorithm naturally describes smaller subproblems. Choosing recursion is a decision about clarity and resources, not a badge of sophistication.

Interfaces That Explain Their Use

A result annotation such as `-> Uint32` describes the returned value. ec can infer some results, but a named algorithm often benefits from an explicit interface. Parameters always need types in an ordinary named declaration. Callers do not infer a function's parameter types from whichever arguments happen to be used first.

Defaults can make a common argument convenient. Named arguments can make a less common argument clear. This function repeats an ordinary integer step:

```
func advanced(value Int32, step Int32 = 1) -> Int32
    return value + step
```

A call `advanced(10)` returns 11; `advanced(10, step = 5)` returns 15. The default belongs to the declaration, not to a function-pointer value. A parameter with a default follows required parameters so that the interface has a clear fixed part.

Do not hide a substantial operation behind a misleadingly harmless name. The function above only computes an integer. A function that allocates, writes a file or changes shared state needs an interface and explanation that make those effects apparent.

Decide What Deserves a Function

`greatestCommonDivisor` earns its name because it answers one useful question, can be called from several calculations, and can be tested independently. Its short reduction steps belong together: a reader needs to see the saved remainder and both assignments at once. Extracting each assignment into a one-use helper would send the reader away without adding an idea.

For the same reason, a larger function can keep several related phases when those phases form one task. Separate the phases with a blank line; add a local heading when a phase needs a name for navigation. Extract when the boundary names a useful concept, enables reuse or independent testing, or separates infrequent work from a frequently executed loop. Readability and performance begin with knowing what work happens where; neither follows automatically from making functions shorter.

Testing a Small Algorithm

A test case should make a claim. For greatest common divisors, useful claims include symmetry, handling zero, exact divisibility and reduction of a nontrivial pair. Check `(48,18)`, `(18,48)`, `(0,9)`, `(9,0)`, `(9,9)` and `(0,0)`. Do not claim symmetry proves the algorithm correct: a function returning zero for every input is symmetric too.

The strongest explanation combines examples with the argument that the remainder step preserves common divisors and terminates. Tests catch mistakes in the written program; the argument explains the algorithm you intended to write.

Exercises

1. Trace the recursive calls for `(18, 48)`.
2. Write a function returning both quotient and remainder for a nonzero `Uint32` divisor.
3. The Fibonacci sequence begins 0, 1; each next term is the sum of the previous two. A naive recursive function computes `F(n - 1) + F(n - 2)` until it reaches `F(0) = 0` or `F(1) = 1`. Expand `F(5)` by two levels. Which calculation repeats, and why is this a poor default for a large input?

Discussion

1. The first reduction gives `(48,18)`, after which the earlier trace applies. A smaller first argument is accepted without a special case.

2. This module-level helper requires `divisor != 0`:

```
func quotientAndRemainder(value Uint32, divisor Uint32) -> (Uint32, Uint32)
    return (value / divisor, value % divisor)
```

In `start`, `quotient, remainder = quotientAndRemainder(17, 5)` introduces two locals holding 3 and 2. The return type lists both component types in order; the returned expression lists their values. The caller must establish the nonzero divisor before making the call. The later error-handling chapter shows how a function can report unsuitable input as a recoverable failure instead.

3. `F(5)` calls `F(4)` and `F(3)`. Expanding `F(4)` calls `F(3)` again, as well as `F(2)`. The duplicated `F(3)` branches each repeat their own smaller calls. An iterative version instead retains the previous two values and performs one step per new term. It still needs an explicit input bound because a fixed-width integer cannot contain arbitrarily large Fibonacci numbers.

On Your Own

Extend `divisors.ec` with a helper that simplifies a nonnegative numerator and a positive denominator, returning a pair of Uint32 values. Reuse `greatestCommonDivisor`; do not copy its loop. Check 48/18 -> 8/3, 7/7 -> 1/1 and 0/9 -> 0/1. In the caller, detect denominator zero before calling the new helper and print a refusal. Give the new function a name that describes its mathematical job, and keep the validity check visible at the call site.

Further Reading

ec Technical Manual: Functions.

Arrays and Search

An instrument has recorded five readings. We want a program that answers two questions: where does a requested reading first occur, and how can sorted order make that search faster? We will build `readings.ec`, first keeping the measurements, then adding a search helper, and finally adding a search for ordered input.

Start a new application directory with a copy of the first chapter's greeting program named `readings.ec`. Keep `from IO import print`, the source banners and `start`; change the banner titles to Readings. Every body addition below goes inside `start`, after its runtime banner. The helpers go at the left margin before the application-entry banner. The assembled file appears after the construction steps.

Keep the Readings Together

An array stores a fixed number of elements of one type in order. Add these two lines to `start`, at the indentation shown:

```
    readings [5]Int32 = [18, 21, 19, 21, 17]
    readings[2] = 20
```

The positions are numbered from zero. Position 2 is the third reading, so the resulting sequence is `18, 21, 20, 21, 17`. The count is `readings.len`, a `UintSize` value equal to 5. The last valid position is 4. There is no element at position 5 and no negative-index shortcut for the last element. The array's type fixes its count; an assignment to an element does not change that type.

An initialized declaration can infer the count: `readings []Int32 = [18, 21, 19, 21, 17]`. The fully inferred `readings = [18, 21, 19, 21, 17]` also creates an array, using the ordinary integer literal type. Write the type when it communicates a requirement; write the count when that count is part of the requirement.

Searching One Position at a Time

Our first question is simple: where is the first reading equal to a requested value? A linear search examines positions in order. It can stop as soon as it finds an answer.

We can now name the search operation. Insert the following helper above the application-entry banner, leaving the input in `start`. Its result will be a position, so we use the same `UintSize` type as the array length. Its parameter `[:]Int32` is a *slice*: the location of the first element together with a count of elements. The location is called a pointer; for this chapter, think of the slice as a window onto existing array storage. At a call, `readings[:]` supplies a view of the whole array. This does not copy the readings or allocate another array. For now, the caller keeps the array alive until the call returns; Pointers and Views will examine that promise closely.

```
func findFirst(values [:]Int32, wanted Int32) -> UintSize

    index UintSize = 0
```

Listing continued

```
    loop index < values.len
        if values[index] == wanted
            return index
        index++

    return values.len
```

Why return `values.len` on failure? Every successful position is smaller than the length, so the length is a distinct *sentinel*: a reserved answer meaning "not found." It works even for an empty slice: the answer is zero, and zero is also its length. No special negative value needs to fit an unsigned index.

Return to `start`. Immediately after the two reading lines, add this call and its output:

```
    position = findFirst(readings[:], 21)
    found = position < readings.len

    print("First match: [ {position} ]; found: [ {found} ]\n")
```

For target 21 the new line reports position 1 and `true`. Change only the call's target to 99: the line should report position 5 and `false`. Restore 21 before continuing. These two trials distinguish a successful position from the end sentinel.

The helper tests the bounds before indexing. Its loop condition therefore carries an essential part of the argument that the read is valid. Do not rely on a runtime bounds check to repair a mistaken condition: out-of-range array or slice access is invalid, and ordinary indexing does not promise an inserted check.

Trace the earlier readings while searching for 21:

Position tested	Value	Decision
0	18	Advance
1	21	Return 1

If a search for 99 appears to keep going, trace the index column first: every unsuccessful body must advance it once. If a search reports the second 21 instead of the first, inspect where the successful return occurs. These are different failures; changing the sentinel cannot repair either loop mistake.

The later 21 is irrelevant because the function promises the *first* occurrence. Searching for 99 examines all five elements and returns 5. The useful invariant is: before testing position `index`, every earlier element has already been shown unequal to `wanted`. Advancing preserves that statement. Reaching the end proves absence.

For `n` elements, the search makes at most `n` comparisons and uses a fixed amount of local storage. That is a count of algorithmic work, not a measured execution time. If the data is small or changes often, this straightforward method can be all the program needs.

Add a Search That Uses Sorted Order

A sorted sequence gives us another piece of information. If its middle value is too small, every value before that middle is too small too. Binary search repeatedly excludes a region that cannot contain the answer.

Keep `findFirst`; it remains useful for unordered readings. Insert the next helper after it, with two blank lines between the functions. We will give this new helper its own ordered input in `start`.

It is easiest to make the interval half-open: `low` is included, `high` is excluded. The function below finds the first position whose value is at least `wanted`, often called a *lower bound*. Its precondition - a promise the caller must satisfy - is that the slice is sorted in nondecreasing order: each value is at least the previous one, so equal values are allowed. The elements must remain unchanged throughout the call. For example, `[0, 5)` means positions 0 through 4; the closing parenthesis says position 5 is excluded.

```
func lowerBound(values [:]Int32, wanted Int32) -> UintSize

    low  UintSize = 0
    high UintSize = values.len

    loop low < high
        middle = low + (high - low) / 2

        if values[middle] < wanted
            low = middle + 1
        else
            high = middle

    return low
```

There are two reasons to move a boundary. If the middle value is too small, it cannot be the answer, so `low` advances past it. Otherwise the middle might be the first qualifying position, so `high` moves to it without skipping it. The invariant is that every position below `low` is too small, while every position at or beyond `high` qualifies if it exists. When the boundaries meet, that position is the first possible answer.

These are ordinary integer operations and structured branches. `high - low` cannot underflow while the interval invariant holds. Computing the midpoint as `low + (high - low) / 2` also avoids adding two large endpoints first. Because `low < high`, the midpoint is strictly below `high`, so its array position exists.

With sorted values `17, 18, 20, 21, 21` and target 21, the intervals proceed as follows:

Interval	Middle value	Next interval
`[0, 5)`	20 at position 2	`[3, 5)`
`[3, 5)`	21 at position 4	`[3, 4)`
`[3, 4)`	21 at position 3	`[3, 3)`

The answer is 3. Equality moves the right boundary inward instead of returning immediately, which is why duplicate values lead to their first position. Each iteration reduces the nonempty interval; roughly doubling the element count adds only one more comparison to the worst case. This is what logarithmic growth means here.

A lower bound does not itself promise a match. Add the following group after the first-match output in `start`. It supplies sorted values explicitly; the next chapter will calculate sorted order for us. Check both the returned position and its value:

```
    sortedReadings [5]Int32 = [17, 18, 20, 21, 21]
    wanted         Int32    = 21

    position = lowerBound(sortedReadings[:], wanted)
    found = (position < sortedReadings.len) && (sortedReadings[position] == wanted)

    print("Sorted match: [ {position} ]; found: [ {found} ]\n")
```

Here `position` and `found` already exist, so these assignments replace their earlier values. The new line should report 3 and `true`; the earlier output stays unchanged. `&&` evaluates its second operand only when its first is true, so a position at the end causes no element read. Reversing those operands would lose that protection.

Copies, Views and Assumptions

Before copying this program's data for another experiment, decide whether you need independent elements or another way to reach the same elements. Both `backup = readings` and `backup [5]Int32 = readings` make an independent array copy. `view = readings[:]` makes a slice, while `first = &readings[0]` makes an element pointer. The latter two share the original elements. An array does not implicitly become a slice at the search call; keep the visible `[:]`.

Binary search on unsorted data still performs its written operations, but its conclusion is unreliable because the sorted-order argument no longer holds. That is a broken algorithmic precondition, separate from an out-of-range memory access. The next chapter gives us a way to establish sorted order ourselves.

Put the Program Together

Your file now has two helpers and one entry point. The helpers describe the searches; `start` keeps the inputs and visible results together. The expected answer lines are `First match: [ 1 ]; found: [ true ]` and `Sorted match: [ 3 ]; found: [ true ]`. As in the first program, these introductory output calls leave output failures unhandled; the text chapter develops the checked form.

```
_____________________________________________________________________________

[ Readings ] Finds the first matching measurement.
_____________________________________________________________________________

from IO import print


func findFirst(values [:]Int32, wanted Int32) -> UintSize

    index UintSize = 0

    loop index < values.len
        if values[index] == wanted
            return index
        index++

    return values.len


func lowerBound(values [:]Int32, wanted Int32) -> UintSize
```

Listing continued

```
    low  UintSize = 0
    high UintSize = values.len

    loop low < high
        middle = low + (high - low) / 2

        if values[middle] < wanted
            low = middle + 1
        else
            high = middle

    return low


_____________________________________________________________________________

[ Application Entrypoint ]
_____________________________________________________________________________

func start()

    print("""
        ___________________________________________

         Readings
        ___________________________________________
        """)

    readings [5]Int32 = [18, 21, 19, 21, 17]
    readings[2] = 20

    position = findFirst(readings[:], 21)
    found = position < readings.len
    print("First match: [ {position} ]; found: [ {found} ]\n")

    sortedReadings [5]Int32 = [17, 18, 20, 21, 21]
    wanted         Int32    = 21

    position = lowerBound(sortedReadings[:], wanted)
    found = (position < sortedReadings.len) && (sortedReadings[position] == wanted)
    print("Sorted match: [ {position} ]; found: [ {found} ]\n")
```

Exercises and Discussed Answers

1. What does `findFirst` return for an empty slice, a one-element match, and a one-element miss? Explain why the caller must compare the answer with the length.

2. Change the question to “how many readings equal the target?” What information replaces the early return?

3. Trace `lowerBound` on `2, 2, 2, 2` for targets 1, 2 and 3. Which result is not a valid element position?

Discussion

1. The answers are 0, 0 and 1. The same integer can mean success or absence for different lengths; the sentinel belongs to the particular input. Never index with it before checking.

2. Start a `UintSize` counter at zero, examine every element, increment only on equality, and return the counter after the loop. The maximum count is the slice length. Keeping the early return would count only the first match. Here is a module-level solution using the same slice contract:

```
func countMatches(values [:]Int32, wanted Int32) -> UintSize

    count UintSize = 0
    index UintSize = 0

    loop index < values.len
        if values[index] == wanted
            count++
        index++

    return count
```

For the original readings and target 21, it returns 2; for an empty slice it returns 0.

3. Targets 1 and 2 both produce position 0; checking equality distinguishes absence from a match. Target 3 produces 4, the insertion position after the last element. Empty intervals let the algorithm express that answer without reading outside the array.

Continue with Sorting and Matrices.

On Your Own

Extend Readings with `findLast`, returning the last matching position or the input length when no match exists. Keep `findFirst` so the program can print both answers for the same target. For the original readings after the correction, target 21 should give first position 1 and last position 3. Target 99 should give the length for both. Passing `readings[:0]` should give zero without reading any element.

Keep each helper's search logic out of `start`; let the entry choose input and print the comparison. Before writing code, decide what must be remembered when a match is found if the search is going to continue.

Further Reading

ec Technical Manual: Arrays and Slices.

Sorting and Matrices

The previous program searched a sequence that we had already written in sorted order. This time the program must do the arranging. We will start with `5, 2, 4, 2`, sort a copy, and print the result. Then we will extend the same program to total the columns of a small sensor table.

Create `sorting.ec` in a separate application directory from the first chapter's greeting file. Keep its import and banners, and use Sorting as the banner title. Add the following input group after the runtime banner in `start`:

```
    original [4]Int32 = [5, 2, 4, 2]
    ordered  [4]Int32 = original
```

The explicit array type on `ordered` asks for an independent copy. We can change its elements while keeping `original` for comparison. The shorter declaration `ordered = original` would also copy the array; `view = original[:]` would share its elements instead. Making the intended copy visible is part of making this experiment understandable.

A Sorted Prefix

Before writing nested loops, do the first insertion by hand. The first value, 5, is already a sorted one-element prefix. The next value is 2. If we write 5 over its slot immediately, where will the 2 go? Save it as the **key** first:

Step	Array contents	Saved key
Save position 1	`5, 2, 4, 2`	2
Move the larger 5 right	`5, 5, 4, 2`	2
Fill the opened position	`2, 5, 4, 2`	no longer needed

The middle row is allowed: the missing 2 is safe in a local variable. We now have two sorted values. For the next insertion, save 4 and ask which of those two predecessors must move. Only 5 moves; 2 stays. This produces `2, 4, 5, 2`.

These repeated steps are insertion sort: keep the values already considered in order, save the next key, move larger values right, then place the key. The outer loop chooses the next key; the inner loop makes room for it. Add this helper above the application-entry banner. It sorts the supplied elements in place, using the copy that `start` already owns. The `key` local saves the value that shifting would otherwise overwrite.

```
func insertionSort(values [:]Int32)

    index UintSize = 1

    loop index < values.len
        key = values[index]
        position = index
```

Listing continued

```
        loop (position > 0) && (values[position - 1] > key)
            values[position] = values[position - 1]
            position--

        values[position] = key
        index++
```

The outer loop begins at one because a prefix containing one value is already sorted. A *prefix* is the part beginning at position zero. Before each iteration, positions below `index` are in nondecreasing order: each value is at least the previous one. `key` saves the next value before any element is overwritten. The inner loop shifts larger predecessors one place to the right, opening a position for the saved value.

The guard has two jobs. `position > 0` prevents unsigned subtraction from wrapping when calculating the predecessor. Only after that succeeds does `&&` permit the comparison with `values[position - 1]`. The destination `values[position]` also exists: position begins at the current valid `index` and only decreases.

Now add the call and output after the input group in `start`:

```
    insertionSort(ordered[:])
    print("Sorted: [ {ordered[0]}, {ordered[1]}, {ordered[2]}, {ordered[3]} ]\n")
```

Follow the whole outer iterations:

Saved key	Shifts	Sequence after insertion
2, from position 1	Move 5 to position 1	`2, 5, 4, 2`
4, from position 2	Move 5 to position 2	`2, 4, 5, 2`
2, from position 3	Move 5, then 4 right	`2, 2, 4, 5`

During shifting, the array can temporarily contain duplicate copies of a moved value. That is harmless because the displaced `key` still exists in a local variable. At the end of the iteration, storing it restores exactly the original values, including the original number of occurrences of each value. The proof needs both parts: the prefix is ordered, and no input value has been lost.

Notice the strict > comparison. An element equal to the key stays before it. If these values were keys attached to records, this choice would preserve the original order of equal keys: insertion sort would be *stable*. Replacing > with >= would still sort numbers, but would reverse that preservation rule.

Boundaries and Cost

For an empty or one-element slice, `index < values.len` is false immediately. Already sorted input never enters the inner body, so it needs only the outer comparisons. Reverse order makes the first inserted element move across one predecessor, the next across two, and so on. The total is `n × (n - 1) / 2` shifts in that case. This explains quadratic worst-case work without making a claim about a particular machine's speed.

The algorithm needs only the key and a few indices, so the extra storage it uses does not grow with the input length. It can be a useful choice for small or nearly sorted sequences. Large arbitrary collections motivate other algorithms and measured comparisons; knowing insertion sort's limits is part of understanding it.

Our opening copy makes the effect easy to inspect. After sorting, `ordered` contains `2, 2, 4, 5`, while `original` still contains `5, 2, 4, 2`. Temporarily change the call to `insertionSort(original[:])` and predict which array changes; then restore it. The slice chooses the storage the helper will modify.

Rows Are Arrays Too

Let us extend the file with another question: what total did each sensor record over two days? Below the sorted-output line in `start`, add this table. Its rows are days and its columns are sensors:

```
    samples [2][3]Int32 = [[4, 7, 2],
                           [6, 1, 8]]
```

Read the type from the outside inward: two elements, each containing three `Int32` values. `samples[1][2]` is 8. `samples.len` is 2; `samples[0].len` is 3. Each row has the same shape because that shape is part of the type. The nested literal must supply exactly those counts.

The new helper needs to walk a row and then walk the columns in that row. Insert `columnSums` after `insertionSort`, before the entry banner. A parameter of `[:][3]Int32` accepts any number of rows, each with three readings:

```
func columnSums(matrix [:][3]Int32) -> [3]Int64

    totals [3]Int64 = [0, 0, 0]

    for row UintSize = 0; row < matrix.len; row++
        for column UintSize = 0; column < totals.len; column++
            totals[column] += Int64(matrix[row][column])

    return totals
```

The two counting headers have matching roles: one selects a row, the other a column. Each keeps its initialization, condition and increment together. The loop bodies remain indented so that the nesting stays visible. The explicit conversion to `Int64` emphasizes the wider accumulator; same-sign widening would also accept the `Int32` reading directly.

For each fixed `row`, the inner loop starts `column` at zero again. It visits all three columns before the outer loop advances to the next row. The running totals therefore develop as follows:

Row just processed	Added readings	Totals
None	None	`0, 0, 0`
0	`4, 7, 2`	`4, 7, 2`
1	`6, 1, 8`	`10, 8, 10`

Finish the extension by adding these lines after `samples` in `start`:

```
    totals = columnSums(samples[:])
    print("Column totals: [ {totals[0]}, {totals[1]}, {totals[2]} ]\n")
```

The output should contain `10, 8, 10`. The call returns an array value, and the inferred declaration preserves that array type, just as it would when copying a named array. Each loop has one visible job, and the matching row and column headers make their relationship easy to inspect.

Wider accumulation does not prove that every possible input total fits. For this function, require every mathematical column sum to fit `Int64` if mathematical totals are the intended result. Otherwise integer addition wraps according to the language rule. For the two-row example, even extreme `Int32` readings fit easily.

A zero-row slice returns three zeros. There is no `matrix[0]` query, so an empty input never requires a first row. That small design choice makes the boundary case natural.

Choosing What the Loops Mean

Rows are contiguous arrays. A row-first scan follows the declared structure, completing one row before the next. A column-first scan can compute the same totals, but visits storage in a different pattern. That distinction may matter in a later performance investigation; it is not a reason to promise a speedup without measurements.

It is also easy to exchange the *meaning* of the loops accidentally. Adding every reading to `totals[row]` would calculate row totals only while the destination happened to be large enough. With more rows, it could become an invalid access. Naming indices by their role makes such errors easier to see than using two unrelated letters.

Put the Program Together

This is the assembled file. The input copies, sorting call, table and column-total call remain visible as separate phases. No allocation is hidden in either helper: the arrays are supplied or returned as fixed-size values.

```
________________________________________________________________________________

[ Sorting ] Orders readings and totals sensor columns.
________________________________________________________________________________

from IO import print


func insertionSort(values [:]Int32)

    index UintSize = 1

    loop index < values.len
        key = values[index]
        position = index

        loop (position > 0) && (values[position - 1] > key)
            values[position] = values[position - 1]
            position--

        values[position] = key
        index++


func columnSums(matrix [:][3]Int32) -> [3]Int64

    totals [3]Int64 = [0, 0, 0]
```

Listing continued

```
    for row UintSize = 0; row < matrix.len; row++
        for column UintSize = 0; column < totals.len; column++
            totals[column] += Int64(matrix[row][column])

    return totals


________________________________________________________________________________

[ Application Entrypoint ]
________________________________________________________________________________

func start()

    print("""
        ________________________________________

         Sorting
        ________________________________________
        """)

    original [4]Int32 = [5, 2, 4, 2]
    ordered  [4]Int32 = original

    insertionSort(ordered[:])
    print("Sorted: [ {ordered[0]}, {ordered[1]}, {ordered[2]}, {ordered[3]} ]\n")

    samples [2][3]Int32 = [[4, 7, 2],
                           [6, 1, 8]]

    totals = columnSums(samples[:])
    print("Column totals: [ {totals[0]}, {totals[1]}, {totals[2]} ]\n")
```

Exercises and Discussed Answers

1. Trace insertion sort on `3, 1, 2`. During the second outer iteration, where is the saved value while 3 moves?

2. What changes if the inner guard uses >=? Give the two equal values labels so their order can be observed.

3. Compute row sums for the sample matrix. How would the destination's shape differ from `columnSums`?

Discussion

1. Inserting 1 produces `1, 3, 2`. Next, `key` holds 2 while 3 is copied to the final position. The array temporarily reads `1, 3, 3`; storing the saved key gives `1, 2, 3`. The temporary duplicate is evidence that the local variable is necessary.

2. Label equal keys `2a` and `2b`, with `2a` earlier. Using `>=` moves `2a` right when inserting `2b`, leaving `2b` first. Numeric order is correct, stability is lost.

3. The row sums are 13 and 15. For this fixed example, the following body fragment uses the earlier `samples` array. The crucial change is the destination index: a row sum accumulates into `rowTotals[row]` while columns vary.

```
    rowTotals [2]Int64 = [0, 0]

    for row UintSize = 0; row < samples.len; row++
        for column UintSize = 0; column < samples[row].len; column++
            rowTotals[row] += Int64(samples[row][column])
```

A function accepting arbitrary row count would need a caller-supplied destination slice or another explicit storage policy; it cannot return a runtime-sized fixed array.

Next, Records and States attaches meaning to related values instead of identifying everything by a numeric position.

On Your Own

Add a helper that transposes the fixed two-by-three sample into a three-by-two array: the original columns become its rows. For `[[4, 7, 2], [6, 1, 8]]`, the result must be `[[4, 6], [7, 1], [2, 8]]`, and the original table must remain unchanged. Print each result row on its own output line.

Choose names for the two indices that make their roles clear. The hint is a relationship, not a complete loop: the value at original row `r`, column `c` belongs at result row `c`, column `r`. Check an all-zero table as well as one with six distinct values.

Further Reading

ec Technical Manual: Arrays and Slices; Statements and Control Flow.

Records and States

The sensor examples identified data by position. A delivery task needs names for different properties: its route, the number of completed parcels and its current state. We will build `tasks.ec` to create a task, advance its state and report the change. Then we will inspect what copying and shared representation mean for those records.

Create the file in its own application directory using the greeting program as a shell. Keep `from IO import print` and both source banners; change the titles to Tasks. We will keep the task types beside the helper that advances them, place the separate representation experiment after that group, and keep `start` last. The order follows the work rather than putting every type in one distant collection.

Give the Data a Shape

Insert these declarations after the import and before the application-entry banner:

```
type TaskState Uint08
    Ready
    Running
    Finished

struct Task
    name      Text
    completed Uint32    = 0
    state     TaskState = TaskState.Ready
```

`TaskState` is an enumeration: a distinct type with named values. Its underlying storage is `Uint08`; without explicit values, these members receive 0, 1 and 2. Write `TaskState.Ready` to name a member of that type, and `first.state` to select a field of one record. `Task` is a struct with three named fields. The defaults belong to the field declarations, so every initializer that omits `completed` or `state` obtains those values. There is no default for `name`.

After the runtime banner inside `start`, add these records. Each named initializer says which property receives its value:

```
    first  = Task(name = "north route")
    second = Task(name = "south route", completed = 3, state = TaskState.Running)
```

For an initial check, add `print("Initial state: [ {first.state} ]\n")` after the records. It should report `Ready`. The named entries let the construction describe itself. `first.completed` is zero and `first.state` is `Ready`. A local declaration `task Task` would instead leave its storage uninitialized. Field defaults are applied by construction; they are not a blanket promise that every declaration of storage is zeroed or completed for us.

State Changes as Functions

An enumeration names states, but it does not enforce a transition policy. We must decide what “advance” means. For this small example, advancing a ready task starts it, advancing a running task finishes it, and advancing a finished task leaves it finished.

Add `advance` after the two type declarations and before the entry banner. We first copy the input to `changed`, then replace only the state selected by the case:

```
func advance(task Task) -> Task

    changed = task

    case task.state
        TaskState.Ready
            changed.state = TaskState.Running
        TaskState.Running
            changed.state = TaskState.Finished
        TaskState.Finished
            ;
        else
            ;

    return changed
```

The parameter arrives by value. `changed = task` copies the struct's field values; assigning its state does not modify the caller's record. We can now use the helper. Replace the initial-state print in `start` with these lines:

```
    started = advance(first)
    print("Copy: [ {started.state} ]; original: [ {first.state} ]\n")

    first = advance(first)
    print("Updated original: [ {first.state} ]\n")
```

The first output shows `Running` and `Ready`. The assignment after that print explicitly replaces `first`, so the final output shows `Running`. This makes the state change visible at the call site. Later, pointers will let a function modify existing caller storage directly, but value passing is enough for this example.

The empty `;` is a deliberate empty body. It is legal ec syntax, not missing work. Why also include `else`? An enum has every value of its underlying integer type, not only its named members. An explicit conversion such as `TaskState(Uint08(99))` can produce an unnamed state. Our policy preserves that value unchanged; a stricter application could return a checked failure instead. The type alone does not validate a protocol received from somewhere else.

Follow one task through three calls:

Before	**After** `advance`	**Completed count**
Ready	Running	Unchanged
Running	Finished	Unchanged
Finished	Finished	Unchanged

Separating the state from the parcel count matters. This function does not claim that finishing a task proves any particular count. If that is a business rule, the transition must check it explicitly.

Records Inside Arrays

Next we want to keep several tasks together. Add this group below the state outputs in `start`. It uses the array idea from the previous chapters:

```
    tasks []Task = [Task(name = "north route"),
                    Task(name = "south route")]

    tasks[0] = advance(tasks[0])
    print("Task zero: [ {tasks[0].state} ]; task one: [ {tasks[1].state} ]\n")
```

Indexing first selects a task; `.state` or `.completed` then selects its property. A whole-record copy copies the inline numbers and the `Text` descriptor - the small value recording the location and byte length of the name. It does not copy the text bytes. Here the names are literals with static backing, so both records can safely refer to their names for the program's lifetime. A record containing a view of temporary storage would have a different lifetime obligation.

A partially constructed struct can be useful while filling fields, but reading or passing the whole stored record requires all its value fields initialized. Do not construct `Task()` and immediately pass it to `advance`: `name` has no default. Giving fields meaningful defaults reduces repetition; it does not excuse missing information.

At this point the delivery-task program is complete: it creates records, advances them and demonstrates independent copies. The homework below extends those operations. Save this version before studying representation; unions and packed layout are a separate extension, not prerequisites for using a struct.

Extension: When Two Interpretations Share Bytes

Our task record now works. Let us add a small representation experiment to the same file so we can distinguish a struct from a union. A union overlays members at the same address: they are different ways to interpret one shared region of storage, rather than separate fields stored one after another. After `advance`, add a two-line `____` / `____ [ Representation Experiments ]` section header, then declare `RealBits` below it. It gives us two views of the four-byte binary32 representation of a `Real32`:

```
union RealBits
    real Real32
    bits Uint32
```

Then append the experiment after the task-array output inside `start`:

```
    value = RealBits(real = 1.0)
    encoding = value.bits
    print("Representation bits: [ {encoding} ]\n")
```

The specified `encoding` is `#3F80_0000`, printed as decimal 1065353216 by the ordinary integer placeholder. There is no numerical conversion from 1.0 to an integer here: the bytes written for the real are read as a `Uint32`. Both types occupy four bytes, and every initialized bit pattern is a valid representation for each type.

A union has no hidden tag recording which member was last written. That makes it unsuitable as a self-validating replacement for `TaskState`. If an application uses a union to store different payload forms, it must maintain its own state field and follow it before interpreting the payload.

Nor does a small write initialize a larger member. In a union containing `Uint08` and `Uint32`, writing the byte leaves the other three bytes without newly established values. Reading the integer would require them initialized too. Restricted representations add another concern: arbitrary integer bits do not necessarily form a valid `Bool` or Unicode `Char`. Copying the union transports its representation and initialization state; it does not repair either problem.

Extension: Layout Is a Separate Question

Ordinary structs retain field declaration order and add alignment padding where needed. *Alignment* means that a field starts at an address divisible by the boundary its type requires. *Padding* is the gap inserted to reach such a boundary. The fields' values are the record's meaning; padding is not another field. Consequently, dumping a struct's raw bytes is not a portable file format. A file format needs explicit decisions about widths, ordering and encoding.

A boundary format may also need an exact layout. Add this last declaration immediately after `RealBits`, in the representation-experiment section. It does not change our task processing; it gives us a small record whose byte positions we can predict:

```
struct Header packed
    kind   Uint08
    count  Uint16
    offset Uint32 align 4
```

The layout is relative to the start of the record:

Byte offsets	Contents
0	`kind`
1-2	`count`
3	Padding before the explicitly aligned field
4-7	`offset`

`packed` removes ordinary field-alignment padding, while the explicit `align 4` still places `offset` at a multiple of four. `count` begins at byte 1, and `offset` begins at byte 4. Direct access to `count` remains meaningful, but its under-aligned position cannot produce an ordinary `*Uint16` pointer. Packing is not a general instruction to make every record smaller: it changes which addressing operations are valid.

Bitfields describe still smaller representation units. Their grouping, reserved bits and read-modify-write behavior deserve a specific boundary contract. Ordinary application state usually needs only named fields and an enum.

Put the Program Together

The assembled program below includes both the task lesson and its representation extension. To keep only the task checkpoint, omit the Representation Experiments section and the final three `value`/`encoding`/print lines in `start`. The task output is unaffected.

The assembled program keeps `TaskState`, `Task` and `advance` together, then groups the representation experiments separately before the entry point. That locality lets a reader find the operation next to the data it

understands. Check the outputs in order: `Running/Ready`, then `Running`, then `Running/Ready`, then the integer representation 1065353216. The unused `Header` declaration records our layout experiment; it allocates no task object.

```
_____________________________________________________________________________

[ Tasks ] Builds records and makes their state changes explicit.
_____________________________________________________________________________

from IO import print

type TaskState Uint08
    Ready
    Running
    Finished

struct Task
    name      Text
    completed Uint32    = 0
    state     TaskState = TaskState.Ready


func advance(task Task) -> Task

    changed = task

    case task.state
        TaskState.Ready
            changed.state = TaskState.Running
        TaskState.Running
            changed.state = TaskState.Finished
        TaskState.Finished
            ;
        else
            ;

    return changed


____
____ [ Representation Experiments ]

union RealBits
    real Real32
    bits Uint32

struct Header packed
    kind   Uint08
    count  Uint16
    offset Uint32 align 4


_____________________________________________________________________________

[ Application Entrypoint ]
_____________________________________________________________________________

func start()

    print("""
        ________________________________________
```

Listing continued

```
         Tasks
        ____________________________________
        """)

    first  = Task(name = "north route")
    second = Task(name = "south route", completed = 3, state = TaskState.Running)

    started = advance(first)
    print("Copy: [ {started.state} ]; original: [ {first.state} ]\n")

    first = advance(first)
    print("Updated original: [ {first.state} ]\n")

    tasks []Task = [Task(name = "north route"),
                    Task(name = "south route")]

    tasks[0] = advance(tasks[0])
    print("Task zero: [ {tasks[0].state} ]; task one: [ {tasks[1].state} ]\n")

    value = RealBits(real = 1.0)
    encoding = value.bits
    print("Representation bits: [ {encoding} ]\n")
```

Exercises and Discussed Answers

1. Add a `Paused` state between running and finished. State your transition policy before changing `advance`.

2. Does copying a `Task` duplicate its name's UTF-8 bytes? What happens if both tasks refer to the same literal?

3. Why is `RealBits(real = 1.0).bits` conceptually different from converting a typed real value to `Uint32`?

Discussion

1. One possible policy is `Ready -> Running -> Paused -> Finished`, with finished and unnamed states unchanged. Add `Paused` to the enumeration, change the `Running` branch to assign `TaskState.Paused`, and add a `Paused` branch assigning `TaskState.Finished`. Leave the `Ready`, `Finished` and `else` branches as written. Another sensible policy might resume a paused task instead; the code cannot decide that requirement for you. Trace every state, including the final one.

2. The descriptor is copied, so both names refer to the same static bytes. The records remain independent in their inline fields. Static literal storage makes this sharing valid without any allocation or release.

3. The union yields representation bits, `#3F80_0000`, whereas numerical conversion of a typed `Real32` value holding 1.0 yields integer 1. Choosing between those operations begins with the question you intend to answer.

Continue with Text and Unicode, where the distinction between a value and its borrowed bytes becomes visible even in a short word.

On Your Own

Add `completeParcel(task Task) -> Task`. It returns a copy with the completed count increased by one only when the state is Running. Ready, Finished and unnamed states remain unchanged. At the maximum Uint32 count, leave the count unchanged rather than wrapping it to zero. The state itself never changes.

Use a running task with count 3 to demonstrate that the returned record has count 4 while the original still has 3. Repeat with a ready task and with the maximum count. Keep the state transition helper separate: advancing a workflow and recording a parcel are different operations.

Further Reading

ec Technical Manual: Aggregate Types.

Text and Unicode

Our next program will inspect a name and print a label for it. We want to know how many bytes the name occupies, decode one scalar, and produce `Name [ ... ]` in storage of a known size. The same name will pass through every stage, so we can see when we have bytes, decoded values or rendered text.

Create `names.ec` in its own application directory from the greeting program. Keep its banners and `from IO import print`, add `import Strings` above that import, and change the titles to Names. We will also start handling output failure. Change the entry signature to `func start() -> Int32`: it will return 0 when its work succeeds and 1 when an operation fails. The assembled file is provided after the steps.

First Check One Ordinary Write

Before adding Unicode work, make the new return convention do one visible job. Keep the file banners and imports; replace `start` with this complete checkpoint:

```
func start() -> Int32

    output = print("""
        ──────────────────────────────────────

         Names
        ──────────────────────────────────────
        """)

    if !!output.ok
        return 1

    output = print("Name [ Ada ]\n")

    if !!output.ok
        return 1

    return 0
```

A successful write returns a status whose `.ok` is true. `!!output.ok` asks whether that success is false. On that path, `return 1` ends the entry with a failure code. Otherwise execution reaches the next statement. A successful pair of writes displays the banner and `Name [ Ada ]`, then returns zero. Unlike our first greeting, this program will not report successful termination after a failed write.

Keep this checkpoint. To begin the inspection below, remove only the Ada write and its following failure check. Leave the checked banner and final `return 0`; the following additions go between them. We will use the same check each time, so the new text operations do not also require inventing a new error policy.

Inspect the Input Before Rendering It

A Unicode scalar is a numeric text value; ec stores one in a `Char`. The notation U+0041 identifies a scalar by its hexadecimal number; that one represents `A`. UTF-8 encodes a scalar as one to four bytes. Decoding means reading those bytes and recovering the scalar. A displayed symbol can combine several scalars, so we will keep these terms separate.

Single quotes write a character literal: `letter Char = 'A'` stores one scalar. Double quotes write text: `name Text = "A"` refers to its UTF-8 bytes. The two values describe the same visible letter in different forms. A character literal can also use an escape, such as `'\n'` for line feed or `'\u{E9}'` for U+00E9.

Add these declarations after the runtime banner in `start`:

```
    name Text = "A\u{E9}\u{1F642}"
    byteCount = name.len
```

The source file remains ASCII. Its escapes produce U+00E9 and U+1F642 in the value. There are seven UTF-8 bytes: one for `A`, two for U+00E9 and four for U+1F642. The text contains three Unicode scalars. `name.len` is 7 because lengths and ranges count bytes.

Scalar	Starting byte offset	Bytes used
U+0041	0	1
U+00E9	1	2
U+1F642	3	4
End	7	0

To inspect the length, add the following after those declarations. Keep the existing `return 0` as the last statement of `start` as we extend it; the listing includes it to show the insertion boundary:

```
    output = print("Bytes: [ {byteCount} ]\n")

    if !!output.ok
        return 1

    return 0
```

`print` returns a status value. Its `.ok` property is true on success; `!!` reverses that answer. We return 1 on failure before continuing. The checkpoint already checks the banner. The assembled listing calls that first result `banner`; either name is fine because each check reads the status of the write immediately before it.

`name[1]` reads a `Uint08` byte, not the whole second scalar. It cannot be assigned or addressed. A `Text` is also not directly iterable with `for`; decoding is a separate operation.

Decoding and a Small Introduction to Failure Values

We have already imported `Strings`. Its `Strings.scalarCount(name)` returns 3 for this input. To walk scalars while retaining byte positions, use `Strings.nextScalar(text, offset)`. Its successful value is a tuple containing a `Char` and the next byte offset.

Decoding may also report that an offset is past the end or starts inside a scalar. The function therefore returns `Result[(Char, UintSize)]`. A `Result[T]` holds either a successful `T` or an error. Read `.ok` first; `.value` is valid only on success, and `.error` only on failure. This is an ordinary returned value, not an exception. The errors chapter will develop the full pattern.

Now insert the next group before the final `return 0` in `start`. We handle failure first, just as for output, and only then read the successful tuple:

```
    step = Strings.nextScalar(name, 1)

    if !!step.ok
        return 1

    character = step.value.0
    nextOffset = step.value.1
    output = print("Scalar: [ {character} ]; next byte: [ {nextOffset} ]\n")

    if !!output.ok
        return 1
```

The `.0` and `.1` projections select the first and second components of the successful tuple; they are not text indices. The local `character` is U+00E9 and `nextOffset` is 3. Starting a walk at zero would return offsets 1, then 3, then 7; at 7 the loop stops without requesting another scalar. Starting at offset 2 instead returns `INVALID_ENCODING`, because that byte continues the previous scalar. Starting at 7 returns `END_OF_INPUT`; starting beyond 7 returns `OUT_OF_RANGE`. To follow the failure path, change only the call offset from 1 to 2 in a copy of the checkpoint: the decode fails, `!!step.ok` becomes true, and `return 1` runs before either tuple projection or the scalar-output call. Restore offset 1 to continue building the label.

A walking loop normally tests `offset < text.len` and advances to the returned next offset. Advancing by one would walk bytes rather than scalars.

A scalar is still not necessarily one displayed character. The text `"e\u{301}"` has two scalars: a letter and a combining accent. `"\u{E9}"` has one. They can look alike, but ec text comparison uses their bytes without Unicode normalization or locale collation. `scalarCount` is not a grapheme counter.

Cutting Only at Boundaries

Let us make one controlled comparison before building the label. A text range also returns a `Result[Text]`. Add the following two assignments before the final return; reuse the existing `name`, rather than declaring it again:

```
    wholeScalar = name[1:3]
    splitScalar = name[1:2]

    output = print("Whole cut: [ {wholeScalar.ok} ]; split cut: [ {splitScalar.ok} ]\n")

    if !!output.ok
        return 1
```

This check should print `true` and `false`. `wholeScalar` succeeds and borrows the U+00E9 bytes. `splitScalar` reports `INVALID_ENCODING`. Bad numeric bounds report `OUT_OF_RANGE`. Even an empty range must have endpoints on scalar boundaries: `name[2:2]` is not a valid text cut. Required constant bounds on constant text are checked during translation instead.

The successful range does not allocate a new string. Its lifetime follows the original bytes. Text literal storage lasts throughout the program and is read-only. Other text may refer to a buffer that will change or to local storage that will end. A successful UTF-8 check establishes an encoding fact at that moment; it does not extend storage lifetime.

Formatting Is a Request to Render

The expression `"Name [ {name} ]"` can contain a runtime placeholder. In a consuming call, that is a `Format` recipe: evaluate `name` now, then render its value. It is not automatically a heap-allocated string. A recipe cannot be stored in an ordinary local variable or returned as a retained string.

When we need the rendered bytes to remain available, supply storage. `Strings.fixed` turns a writable byte slice into a fixed `TextBuffer`. It initially has length zero and does not read or clear the supplied bytes. `buffer.format` writes a new rendering and returns a view of it, or reports an error such as `NO_SPACE`.

We have enough pieces to make the label. Insert `showLabel` above the application-entry banner; the imports are already at the top of the file, so do not repeat them. It owns its temporary rendering storage and reports success or failure to its caller:

```
func showLabel(name Text) -> Int32

    storage [64]Uint08
    buffer = Strings.fixed(storage[:])
    formatted = buffer.format("Name [ {name} ]")

    if !!formatted.ok
        return 1

    output = print("{formatted.value}\n")

    if !!output.ok
        return 1

    return 0
```

The `!!` operator means logical negation. The first branch handles a formatting failure before reading its value; the second handles output failure. `print` adds no newline, so the recipe supplies `\n`. The name's borrowed bytes and the local buffer both remain alive through rendering and output.

Replace the final `return 0` of `start` with `return showLabel(name)`. That connects the last stage and forwards its success or failure as the application result. For a simple trial, temporarily change `name` to `"Ada"`; calling `showLabel("Ada")` prints `Name [ Ada ]` followed by a newline and returns 0 when output succeeds. Both cuts are valid for Ada because each byte boundary is also a scalar boundary. Restore the escaped sample after the trial. The buffer has room for 64 bytes, including the fixed label text. The prefix `Name [ ` uses seven bytes and the suffix `]` uses two, leaving 55 bytes for the name. A 56-byte ASCII name cannot fit. The newline is added by the later print call, so it consumes no buffer capacity. It does not promise room for 64 Unicode scalars. No terminator is counted or required by this rendering operation. If formatting fails, its contents, length, capacity and storage remain unchanged. This transaction rule is especially useful when replacing a previously valid display value.

Appending differs from formatting: `buffer.append(text)` adds to existing contents, while `buffer.format(text)` replaces them. Both return checked results. Neither permits an input view overlapping any of the destination's capacity range. Thus appending `buffer.view()` to the same buffer is rejected even if spare capacity appears sufficient. The source and destination relationship is part of the operation's contract, not a guess about which copying direction might work.

Holding a View Safely

A `buffer.view()` is borrowed text. An append that fits in place preserves views of the earlier prefix. Replacing contents, resetting the buffer or growing its storage removes the relevant preservation guarantee. Obtain a fresh view after replacing contents, and keep the buffer and its backing alive while consuming it.

A fixed buffer borrows its array, so there is no heap allocation to release in `showLabel`. Returning its text view would be wrong: the local array would end before the caller used the bytes. The next chapter makes this lifetime reasoning explicit for all pointers and slices.

Put the Program Together

The assembled file follows the data through inspection, decoding, boundary checks and rendering. The sample produces byte count 7, scalar U+00E9 with next offset 3, then `true`/`false` for the cuts, followed by the label. An expected failed cut is printed as a fact; an unexpected decode, rendering or output failure returns 1. Each check sits directly after the operation whose result it protects.

```
____________________________________________________________________________

[ Names ] Decodes text and renders a bounded label.
____________________________________________________________________________

import Strings

from IO import print


func showLabel(name Text) -> Int32

    storage [64]Uint08
    buffer = Strings.fixed(storage[:])
    formatted = buffer.format("Name [ {name} ]")

    if !!formatted.ok
        return 1

    output = print("{formatted.value}\n")

    if !!output.ok
        return 1

    return 0


____________________________________________________________________________

[ Application Entrypoint ]
____________________________________________________________________________

func start() -> Int32

    banner = print("""
        ____________________________________

         Names
        ____________________________________
        """)
```

Listing continued

```
    if !!banner.ok
        return 1

    name Text = "A\u{E9}\u{1F642}"
    byteCount = name.len
    output = print("Bytes: [ {byteCount} ]\n")

    if !!output.ok
        return 1

    step = Strings.nextScalar(name, 1)

    if !!step.ok
        return 1

    character = step.value.0
    nextOffset = step.value.1
    output = print("Scalar: [ {character} ]; next byte: [ {nextOffset} ]\n")

    if !!output.ok
        return 1

    wholeScalar = name[1:3]
    splitScalar = name[1:2]
    output = print("Whole cut: [ {wholeScalar.ok} ]; split cut: [ {splitScalar.ok} ]\n")

    if !!output.ok
        return 1

    return showLabel(name)
```

Exercises and Discussed Answers

1. For `"x\u{20AC}y"`, give the byte length, scalar count and byte range selecting U+20AC.

2. Why can an empty text range fail, while an empty in-range array slice needs no scalar-boundary check?

3. Suppose a fixed buffer currently contains `"ok"`. What must remain true after an oversized `format` attempt fails? May the program still print the old view?

Discussion

1. The first answer is five bytes, three scalars, and range `[1:4]`. The following `y` starts at byte 4. This can be worked out from UTF-8 widths without inspecting a terminal's character cells.

2. A text boundary represents a position between whole scalars. Choosing the same continuation-byte offset twice does not turn it into such a position. An ordinary array has elements whose boundaries are already fixed by their type.

3. Failure leaves the buffer and its storage unchanged. With no other mutation or lifetime change, the earlier `"ok"` view remains usable. A successful replacement would require different reasoning, even if it reused the same address.

Continue with Pointers and Views.

On Your Own

Turn Names into a label inspector that accepts any valid name length, including empty text, up to the label buffer's capacity. Replace the fixed-offset decoding and cutting experiment in `start` with a byte-count and scalar-count report using `Strings.scalarCount`; retain the checked `showLabel` call.

For `"Ada"`, both counts should be 3. For `"A\u{E9}\u{1F642}"`, they should be 7 and 3. Empty text should report two zeros and print `Name [ ]`. An ASCII name of 55 bytes should fit; one of 56 bytes should make label rendering return failure. Check every output result. The counts describe the input, so do not include the label prefix or suffix in them.

Further Reading

ec Technical Manual: Text and Formatting; Strings.

Pointers and Views

The sorting helper changed an array supplied by its caller; the task helper returned a changed copy. We will now build one program that makes this difference visible. It will update a score through an address, double readings through a slice, and change the first half through a returned view.

Create `views.ec` in its own application directory using the checked entry pattern from Names. Keep `from IO import print`, the source and runtime banners, and `func start() -> Int32`; change the titles to Views. Remove the `Strings` import and name-processing helpers. In `start`, keep the checked banner write, then replace all following name-processing statements with `return 0`. Every body addition below goes between them; the final listing shows the whole file.

Change One Existing Integer

First add these lines to `start`:

```
    score Int32 = 10
    address = &score
    address.* = 14
    observed = score
```

`&score` forms the address of the existing variable. Its type is `*Int32`. The postfix `.*` designates the storage at that address; on the left of assignment it is written, and in a value expression it is read. `observed` is therefore 14. Forming the address itself reads no integer value and makes no copy of the storage.

It helps to distinguish the variable holding an address from the object at that address:

Expression	What it denotes after the fragment
`score`	The integer value 14
`address`	The address of `score`
`address.*`	The integer storage also named by `score`
`observed`	An independent integer value copied from `score`

Changing `score` afterward changes what `address.*` reads, but does not change `observed`. Reading through a pointer is called *dereferencing* it.

Do not confuse `*Int32` in a type with multiplication in an expression. Nor does a pointer allocate an integer: it holds an address, and the integer in this example already exists as `score`.

We can give this update a name. Add `addTo` above the application-entry banner. The pointer must lead to an `Int32` that still exists, may be written and already holds a value. Its address must also meet the alignment required by `Int32`; taking `&score` gives us that correctly aligned address:

```
func addTo(value *Int32, amount Int32)
    value.* += amount
```

After the four score lines in `start`, add `addTo(&score, 3)`. Then add the following checked output before the final return:

```
    output = print("Score: [ {score} ]; earlier copy: [ {observed} ]\n")

    if !!output.ok
        return 1
```

The output should be 17 and 14. The pointer parameter is itself copied by value, but both copies contain the address of the same integer. The function changes that integer to 17. The compound assignment reads the old value before writing, which is why initialized storage is part of its precondition. An ordinary store need not read the previous contents.

The arithmetic still wraps at `Int32` width. Passing an address does not turn it into checked arithmetic, an atomic update or an ownership transfer.

Null Is a Value, Not Storage

Before moving to arrays, add a small optional-update experiment after the score output and its check. A typed pointer may hold `null`:

```
    selected *Int32 = null

    if selected != null
        selected.* = 7
```

This fragment performs no write because the condition is false. Null can be copied, compared and truth-tested. Reading or writing an ordinary object at address zero is invalid. Integer zero is not implicitly a pointer; use the actual `null` literal in pointer context.

A null check is useful but limited. A nonnull address can still be misaligned, refer outside accessible storage, or reach bytes with no initialized value. “Nonnull” answers one question, not all of them. For every dereference, ask what storage is currently there and what operation that storage permits.

The Length a Pointer Does Not Carry

Next add an array to `start`, after the null experiment. We will name both the address of its first element and the address of the whole array:

```
    values [4]Int32 = [10, 20, 30, 40]
    elementPointer = &values[0]
    arrayPointer = &values
```

`elementPointer` has type `*Int32`. `arrayPointer` has type `*[4]Int32`, a pointer to the entire array. The element pointer carries no bound. Adding one steps by one `Int32`; adding one to the whole-array pointer steps by the size of the complete array. The type determines the scale, not a hidden remembered array length.

Both `inferred = values` and `copy [4]Int32 = values` copy the four elements into independent storage. If `pointer` points to the whole array, `copy = pointer.*` does the same. To share elements, write `first = &values[0]` or `window = values[:]`. An explicitly written `[N]T` function parameter adjusts to `*T`; the count is not passed or checked. Use `[:]T` when the function needs a count, or `*[N]T` when the entire array type matters.

To update all readings, insert `doubleValues` after `addTo`, with two blank lines between the helpers:

```
func doubleValues(values [:]Int32)
    for item in values
        item.* *= 2
```

Each `item` is an element pointer, not a copied element. `item.*` makes the update visible in the caller's array. Add `doubleValues(values[:])` after the array/address group in `start`. It changes the four-element example to `20, 40, 60, 80`: the first iteration points at position zero, updates that element, then iteration advances to the next element.

When the loop starts, the whole range claimed by a nonempty slice must fit in one block of memory that is still available, with the alignment its elements need. A loop that touches only a few elements does not make an oversized slice valid. Our doubling loop also needs every element it reads and updates to hold an initialized, writable `Int32`.

A View Borrows Its Backing

Let us now take a window onto those doubled values. A slice contains a pointer and a length. Copying it copies those two fields. Add the following lines after the doubling call; do not redeclare the array:

```
    left = values[:2]
    alias = left
    alias[1] = 99
```

Both views now observe 99 at their second position, and `values[1]` is 99. The array owns the inline elements; neither slice owns a separate copy.

We can also calculate the window in a helper. Add `firstHalf` after `doubleValues`. It returns the first half of a supplied slice, rounding the count down:

```
func firstHalf(values [:]Int32) -> [:]Int32
    return values[:values.len / 2]
```

Replace `left = values[:2]` in `start` with `left = firstHalf(values[:])`; keep `alias = left` and the assignment through it. Finish the experiment with this output and check:

```
    output = print("Readings: [ {values[0]}, {values[1]}, {values[2]}, {values[3]} ]\n")

    if !!output.ok
        return 1
```

It should report `20, 99, 60, 80`. The first helper doubled every element; the view update then replaced only position one. The caller must keep the original backing alive while using the answer. An empty input produces an empty view, and a one-element input also produces an empty view. There is no allocation or element copy.

Now contrast this with a local array created *inside* a function. Returning its slice would copy the descriptor, but function exit would end the local array's storage. A later access through the returned view would have no right to that ended storage. A return value cannot carry the local storage's lifetime out with it. The same reasoning explains why returning a fixed text buffer's local backing was unsafe in the previous chapter.

Plain Addresses and Real Obligations

ec pointers are plain addresses without hidden provenance or generation tags: a pointer carries no secret record of which allocation first produced its numeric address. Arithmetic can form an address outside an object, and address arithmetic wraps at the pointer width. Formation alone does not access memory. A later access is judged by its final address and the live storage there, not by the route used to calculate it.

That rule should make the model clearer, not encourage guessing addresses. An access still needs a suitable live extent, correct alignment, the right initialized representation for reads, writable storage for writes, and compatible const/device/atomic rules. A cast changes none of those facts. Storage reused at the same numeric address is judged by its current conditions. Separately, allocator release operations require the allocator's current ownership descriptor; a reused address does not authorize a second release through an obsolete one.

Constantness also follows the storage being protected. A constant array seals its inline elements. A constant slice binding prevents replacing its pointer and length, but does not make independently writable backing constant. Conversely, a mutable pointer cannot lawfully write to a constant object just because the pointer itself is mutable.

The language supplies mandatory checks for certain statically known mistakes, but does not promise a runtime check for every pointer or slice operation. It also does not let an optimizer assume away your later explicit checks merely because an earlier access would have needed them.

Put the Program Together

The complete file keeps the storage declarations in `start` and the operations above it. Names such as `observed`, `address`, `left` and `alias` distinguish a copied value from an address or view. The expected lines report score 17 with earlier copy 14, then readings `20, 99, 60, 80`. All output writes are checked.

```
________________________________________________________________________________

[ Views ] Updates existing storage through pointers and slices.
________________________________________________________________________________

from IO import print


func addTo(value *Int32, amount Int32)
    value.* += amount


func doubleValues(values [:]Int32)
    for item in values
        item.* *= 2


func firstHalf(values [:]Int32) -> [:]Int32
    return values[:values.len / 2]

________________________________________________________________________________

[ Application Entrypoint ]
________________________________________________________________________________

func start() -> Int32
```

Listing continued

```
    banner = print("""
        ──────────────────────────────────────────

         Views
        ──────────────────────────────────────────
        """)

    if !!banner.ok
        return 1

    score Int32 = 10
    address = &score
    address.* = 14
    observed = score
    addTo(&score, 3)
    output = print("Score: [ {score} ]; earlier copy: [ {observed} ]\n")

    if !!output.ok
        return 1

    selected *Int32 = null

    if selected != null
        selected.* = 7

    values [4]Int32 = [10, 20, 30, 40]
    elementPointer = &values[0]
    arrayPointer = &values
    doubleValues(values[:])

    left = firstHalf(values[:])
    alias = left
    alias[1] = 99
    output = print("Readings: [ {values[0]}, {values[1]}, {values[2]}, {values[3]} ]\n")

    if !!output.ok
        return 1

    return 0
```

Exercises and Discussed Answers

1. After `other = address`, does assigning `other = null` change `address`? Does assigning `other.* = 8` before that change `score`?
2. What does `firstHalf` return for lengths zero, one, two and five? Who owns the elements?
3. A function returns a slice into its caller's array. Is that always safe for the caller to store forever?

Discussion

1. Copying the pointer creates another address value. Replacing one pointer leaves the other unchanged. Dereferencing either before replacement reaches the shared integer, so that store changes `score`. Dereferencing after setting `other` to null would be invalid.

2. The returned lengths are zero, zero, one and two. For input `[10, 20, 30, 40, 50]`, the last case returns a view of `[10, 20]`; assigning 99 to its first element changes the original array's first element too. Every returned element still belongs to the original backing. The function owns no surviving element storage.

3. The view remains usable only while its backing satisfies the access contract. The caller's array can itself be local to a shorter-lived scope. Returning a view is a borrowing operation, not a lifetime extension.

The next part begins with Errors and Cleanup, then makes allocation and release explicit. You now have the storage vocabulary needed to understand why a cleanup call belongs to a particular resource and scope.

On Your Own

Add `clearValues(values [:]Int32)` and use it on the first half of a new five-element array `[1, 2, 3, 4, 5]`. The final array should be `[0, 0, 3, 4, 5]`. Passing an empty slice should change nothing; passing the whole slice should clear all five values.

The helper must change caller storage, not an independent array copy. Explain why its ordinary zero store does not need to read an element's previous value, while `doubleValues` does. Keep the array declaration and the call using its view close enough that the backing storage is easy to identify.

Further Reading

ec Technical Manual: Pointers and Memory.

Errors and Cleanup

We are going to build a quantity checker. Give it `"12"`, `"7"` and `"3"`, and it reports a total of 22. Give it `"seven"`, and it stops with an error instead of inventing a number. Before it returns, it must also signal that its calculation has finished. That last requirement will let us see cleanup happen on both successful and unsuccessful paths.

Start a new application directory for this exercise. We will use the file banner and entry-point arrangement from the first program, but change `start` to return `Core.Status`. A status represents success or failure without a successful payload; returning `Core.ok()` means success. The complete source appears after we have constructed its parts. The earlier blocks are additions to those named functions, not separate files.

Create `quantities.ec` with the usual file banner. Its inventory needs `import Core` for results and errors, `import Strings` for parsing, and `from IO import print` for the final output. Put those imports above the functions, with the `from` import in its own group. We will construct the calculation below the inventory first.

First decide what the calculation can promise

A plain integer cannot distinguish “the correct total is zero” from “there is no total because parsing failed.” Our helper will return `Core.Result[Uint64]`: either a Uint64 value or an error. The brackets select the successful value's type. The result's `ok` property tells us which alternative exists; read `value` only on success and `error` only on failure.

We will accept at most 1,000 quantities, each from 0 through 100. Those are application rules. Put the count guard first in the body of `totalQuantities(items [:]Text) -> Core.Result[Uint64]`:

```
    if items.len > 1000
        return Core.OUT_OF_RANGE

    total Uint64 = 0
```

The accumulator line belongs at the same indentation as the `if`; the assembled source below shows the exact layout. Returning an error from this Result function constructs failure. Returning an integer constructs success. These conveniences belong to `return`, not to every assignment involving Result.

The guard leaves before any parsing or addition, so an oversized list cannot produce a partial total.

Add one successful iteration before considering the whole list

Inside `for item in items`, the loop variable is a pointer to a Text element. The text to parse is therefore `item.*`. `Strings.parseInt[Uint32]` asks for a Uint32 result, not an integer of whatever type the parser happens to choose. Add this sequence to the loop:

```
        parsed = Strings.parseInt[Uint32](item.*)

        if !!parsed.ok
            return parsed.error

        quantity = parsed.value
```

Predict the path for `"7"`: parsing succeeds, the failure branch is skipped, and `quantity` becomes 7. For `"seven"`, the branch returns the parser's error and the value projection is never read. `!!` is logical negation, so the branch reads “if parsing did not succeed.”

Now add the application check and the addition immediately after `quantity`:

```
        if quantity > 100
            return Core.OUT_OF_RANGE

        total += Uint64(quantity)
```

The parsed value is Uint32, while our accumulator is Uint64. The explicit conversion makes that choice visible; same-sign widening would also accept it directly. Even the largest accepted list totals only 100,000, so this accumulator cannot overflow. Finish the function with `return total` after the loop, at the loop's indentation level.

Trace the prefixes of `12, 7, 3`: the totals are 12, 19 and 22. For `12, seven, 3`, only the first addition occurs. Nothing returns the partial total, and the third item is not visited. Empty input visits nothing and succeeds with zero.

Parsing and application validation are separate decisions. `"101"` parses successfully and fails our limit; leading spaces or an empty string fail parsing. A decimal value outside Uint32 produces the parser's `OVERFLOW`. A `Core.Error` carries a domain identifying an error family and a code identifying its condition; it is a returned value, not a printed message.

Replace repetition with propagation

The parse/check/project sequence does exactly one thing on failure: return the error unchanged. Replace that entire sequence, from `parsed =` through `quantity =`, with:

```
        quantity = Strings.parseInt[Uint32](item.*)?
```

Keep the quantity limit and addition after it. The postfix `?` yields the successful value or leaves the containing error-bearing function with the failure. It neither retries parsing nor undoes earlier actions. The complete function below uses this shorter version. Use an explicit check when you need recovery or a different policy; use `?` when forwarding the error is the complete decision.

Make a scope-exit action observable

A resource is something acquired and later released, such as storage or a file. Before working with either, we can learn the scheduling rule using a Boolean flag. Write `markFinished(finished *Bool)` as a one-statement helper that assigns true through its pointer. Then write `totalAndMark(items [:]Text, finished *Bool) -> Core.Result[Uint64]` to wrap the calculation.

Its first phase is:

```
    finished.* = false
    defer markFinished(finished)
```

Its last statement is `return totalQuantities(items)`. The deferred call runs when this wrapper's scope ends, after its return value has been determined. The pointer is copied when `defer` is registered; the Boolean at that address is changed when the call runs. That is why the caller must keep the Boolean alive through the call.

Try to predict the flag for both input lists. It becomes true after either return. The flag is a demonstration of a scope-exit action, not a memory resource itself. Later, the same placement will put a release beside the successful acquisition it protects. This locality is an ECS-1 choice with a practical purpose: a reader can see the obligation and its cleanup together.

Assemble the application

Here is the completed `quantities.ec`. It includes the shorter parse step, both cleanup helpers and an error-aware entry point. Helpers precede their consumers, and `start` is last under its own part marker. The output banner uses the same visual language as the source.

```
________________________________________________________________________________

[ Quantity Checker ] Parses a bounded list and observes scope cleanup.
________________________________________________________________________________

import Core
import Strings

from IO import print


func totalQuantities(items [:]Text) -> Core.Result[Uint64]

    if items.len > 1000
        return Core.OUT_OF_RANGE

    total Uint64 = 0

    for item in items
        quantity = Strings.parseInt[Uint32](item.*)?

        if quantity > 100
            return Core.OUT_OF_RANGE

        total += Uint64(quantity)

    return total


func markFinished(finished *Bool)
    finished.* = true


func totalAndMark(items [:]Text, finished *Bool) -> Core.Result[Uint64]

    finished.* = false
    defer markFinished(finished)

    return totalQuantities(items)

________________________________________________________________________________
```

Listing continued

```
[ Application Entrypoint ]
________________________________________________________________________________

func start() -> Core.Status

    print("""
        ________________________________________

         Quantity Checker
        ________________________________________
        """)?

    items []Text = ["12", "7", "3"]

    finished Bool = false
    result = totalAndMark(items[:], &finished)

    Core.assert(finished)

    if !!result.ok
        return result.error

    print("Total: [ {result.value} ]\n")?

    return Core.ok()
```

The assert checks our invariant: the wrapper must have marked completion before returning. A false assertion causes a fatal assertion failure. The parse error remains recoverable: the explicit result check returns it from `start`. Hosted startup then reports its domain and code on standard error and terminates unsuccessfully. With valid input, the program prints the banner and total 22.

Change only `"7"` to `"seven"`. The wrapper still marks completion, the assertion still holds, and no total line is printed. Change it to `"101"`: the same path carries the application error instead. Output operations also return Status, so their `?` failures can leave the entry point.

Use cleanup without giving it extra powers

Deferred calls execute in reverse registration order when ordinary control leaves their scopes. Register cleanup after successful acquisition; a registration never reached cannot run. No destructor releases resources automatically. Fatal termination and immediate process exit do not promise ordinary scope cleanup.

Arguments are saved at registration. Deferring release with a slice saves that slice now; assigning a resized allocation to the variable later does not update the saved value. A pointer argument saves an address and can observe later changes in the live object there. Neither form extends the lifetime of borrowed data.

A deferred call's result is discarded, and `defer close()?` is invalid. If a close failure must change the answer, perform that close explicitly before choosing the answer. Ignoring a result is legal, but not equivalent to success. ECS-1's `_ =` marks a failure deliberately accepted at the risky call; we will use it for best-effort cleanup in the file-copy chapter.

One unusual result type needs an explicit choice. In `Core.Result[Core.Error]`, an Error may itself be the successful payload. A bare Error at `return` would not say which role you intend, so write `Core.success[Core.Error](error)` for that payload or `Core.failure[Core.Error](error)` for failure. The latter requires a failure error, not `Core.NO_ERROR`. Our integer result has no such ambiguity.

Exercises: Try Changes, Then Explain Them

1. Predict the total or error for an empty list, `["0", "100"]`, `[" 7"]` and `["101"]`.
2. Require every quantity to be positive. Change one condition without obscuring the parser's own errors.
3. Suppose cleanup calls are registered as `first`, then `second`. Which runs first, and can the later one change a return value already chosen?

Discussion

1. Empty input succeeds with zero; the second list totals 100; the leading space fails parsing; 101 fails the application limit.

2. For positivity, replace the limit condition with `quantity == 0 || quantity > 100`, retaining `Core.OUT_OF_RANGE`. Parsed `-0` is zero too, so this rejects it. Parsing still runs first, preserving its errors.

3. Deferred calls run `second` then `first`, after the return value is fixed. They can change live storage reached by saved pointers, but cannot replace that chosen return value.

On Your Own

Extend the quantity checker with a caller-supplied `limit Uint64` on the total. Keep the existing item-count and per-item limits. Return `Core.OUT_OF_RANGE` as soon as adding the next valid quantity would exceed the total limit, and keep the completion flag true after every ordinary return. With `12, 7, 3`, limit 22 must succeed with 22 and limit 21 must fail. Empty input with limit zero must succeed with zero. With `7, seven` and limit 5, the first quantity must cause the limit failure before the malformed second item is visited. With a sufficient limit, malformed text must preserve the parser error. Decide where the new guard belongs so that the successful total is never replaced by an out-of-limit intermediate total.

Further Reading

ec Technical Manual, Errors and Cleanup.

Modules and Projects

Our next application reports three facts about a batch of quantities: how many values it contains, their total and their maximum. We will build it in three files. One describes the answer, one calculates it, and one presents it. By the end, replacing the output format will not require changing the calculation.

Use a new directory named `QuantityReport`, separate from the quantity checker. The inputs in this chapter are already Uint32 values; parsing has been done by the caller. We will accept between one and 1,000 values. Empty input is a failure because this interface promises a maximum and we have not defined one for an empty batch.

Draw the boundary before writing the functions

Create this structure:

```
QuantityReport/
    src/
        main.ec
        Analysis/
            summary.ec
            calculate.ec
```

A module is a directory. The two files inside `Analysis` contribute to one namespace: the set of declarations that can use one another's names without importing another module. The source-root directory is a separate module. `Analysis` can compute without knowing anything about the root module's output.

There is no significance to the alphabetical order of these files. Declarations within a module are order-independent. We choose their arrangement for readers: the public answer and the work producing it stay together, while presentation stays outside.

Give the result a name

What must survive the function call? We need the number of values, their sum and their maximum. Name that answer `Summary`, and put this complete source in `Analysis/summary.ec`:

```
________________________________________________________________________________

[ Quantity Summary ] The answer produced by a batch analysis.
________________________________________________________________________________

struct Summary
    count   UintSize
    total   Uint64
    maximum Uint32
```

The field-type column makes three parts of one answer visible as a group. `UintSize` matches slice lengths. `Uint64` holds the sum of up to 1,000 Uint32 values, including their largest possible values. The maximum has the same type as one input.

This declaration is public because it stands outside a `namespace` block, in the module's principal namespace. Other modules can name it after importing `Analysis`. Its fields are public wherever the type is accessible. We have chosen a simple visible record, not an opaque object whose representation is protected.

Build the calculation in three phases

The function will be named `summarize`, with parameter `values [:]Uint32` and result `Core.Result[Summary]`. Add `import Core` to `Analysis/calculate.ec`, then put this function below the inventory. First reject sizes the interface cannot accept:

```
    if values.len == 0 || values.len > 1000
        return Core.INVALID_ARGUMENT
```

This must precede any read of `values[0]`. Now start the two accumulators:

```
    total   Uint64 = 0
    maximum Uint32 = values[0]
```

The maximum begins with a real input value, while the total begins with no values added. These different starting points prevent a common mistake: initializing the maximum to zero would be unsuitable for a later version accepting only negative readings.

What should one iteration do? Add the current value to the total. Replace the maximum only if the new value is greater. Add the following loop after the accumulators, and finish the function afterward with a named-field `Summary` construction:

```
    for value in values
        total += Uint64(value.*)

        if value.* > maximum
            maximum = value.*

    return Summary(count = values.len, total = total, maximum = maximum)
```

Predict the state after each element of `[4, 9, 2]`. It is `(total 4, maximum 4)`, then `(13, 9)`, then `(15, 9)`. Reading the first value once for initialization and again during traversal does not add it twice: initialization changes only the maximum. Each addition occurs in the loop.

Here is the assembled `Analysis/calculate.ec`:

```
_____________________________________________________________________________

[ Batch Analysis ] Computes a bounded integer summary.
_____________________________________________________________________________

import Core


func summarize(values [:]Uint32) -> Core.Result[Summary]

    if values.len == 0 || values.len > 1000
        return Core.INVALID_ARGUMENT
```

Listing continued

```
    total   Uint64 = 0
    maximum Uint32 = values[0]

    for value in values
        total += Uint64(value.*)

        if value.* > maximum
            maximum = value.*

    return Summary(count = values.len, total = total, maximum = maximum)
```

There is no import of `summary.ec`: sibling files already share the module's declarations. The explicit numeric conversion widens each Uint32 before addition. The loop borrows its input; it does not retain a pointer or change a value. The caller can reuse its storage after the call.

Write the consumer without moving the algorithm

In `main.ec`, we need the module name `Analysis`, the result type from `Core`, and `print` from `IO`. The calculation is called as `Analysis.summarize(values)`, not through a source filename.

Construct `report(values [:]Uint32) -> Core.Status` in two phases. First obtain the successful record:

```
    summary = Analysis.summarize(values)?
```

Then print its three fields and return success:

```
    print("Count:   [ {summary.count} ]\n")?
    print("Total:   [ {summary.total} ]\n")?
    print("Maximum: [ {summary.maximum} ]\n")?

    return Core.ok()
```

The messages share a shape, so their important values line up. The calls remain separate: each may fail, and each `?` must remain visible. If the second output fails, the first line stays written. Printing is not a transaction that can erase earlier output.

Use this completed `src/main.ec`:

```
_____________________________________________________________________________

[ Quantity Report ] Presents a summary from the Analysis module.
_____________________________________________________________________________

import Core

import Analysis

from IO import print


func report(values [:]Uint32) -> Core.Status

    summary = Analysis.summarize(values)?
```

Listing continued

```
    print("Count:   [ {summary.count} ]\n")?
    print("Total:   [ {summary.total} ]\n")?
    print("Maximum: [ {summary.maximum} ]\n")?

    return Core.ok()


──────────────────────────────────────────────────────────────────────────────

[ Application Entrypoint ]
──────────────────────────────────────────────────────────────────────────────

func start() -> Core.Status

    print("""
        ──────────────────────────────────────────

         Quantity Report
        ──────────────────────────────────────────
        """)?

    values []Uint32 = [4, 9, 2]

    return report(values[:])
```

The three source files now form one usable application. `start` delegates the final status to `report`; adding another `Core.ok()` after that return would serve no purpose. With successful output, the report contains count 3, total 15 and maximum 9.

Try replacing the final argument with `values[:0]`. The empty view reaches `summarize`, which returns before accessing its first element. `report` propagates the failure before printing any summary line. The banner has already been printed by `start`. Following this chain is more useful than saying vaguely that “the program handles errors.”

Make the dependency direction deliberate

An import belongs to the file that contains it. A second reporting file needs its own `import Analysis`, even though it shares the root module's declarations with `main.ec`. Conversely, both files within `Analysis` see `Summary` automatically. File-local imported names and shared module declarations are different mechanisms.

A selective import such as `from Analysis import summarize` gives this file the short name `summarize`; it does not re-export that function to consumers of the root module. Importing a name and publishing a declaration are separate actions.

If `Analysis` imported the root module to print messages, the dependency cycle would be rejected. Returning data and errors keeps the direction simple: presentation depends on analysis. A test can call the same calculation without capturing output.

A helper that should be internal belongs in a namespace block, such as `Internal`. Within the module it can be named `Internal.checkLimit`; other modules cannot access it. We did not extract our single size condition into such a helper because doing so would scatter a simple decision. ECS-1 keeps helpers near consumers and extracts a function when it names useful work, not just to shorten a body.

Do not move `Summary` into an internal namespace while keeping this public function signature. Public declarations cannot expose internal types. The boundary includes the answer's representation as well as the function's name.

Record a project choice only when needed

The conventional structure works without a manifest. A manifest is project data, separate from executable source. To record the native target explicitly, add this complete `QuantityReport/project.ecp`:

```
____
____ [ Project ] - quantity report

edition = 1
kind    = application
target  = "arm64-macos"
```

Every field and the manifest itself is optional. Without a target here, the translation request must select one; if both places name it, they must agree. The title uses the two-line header form, and the related settings share an assignment column. No executable calls belong in this data file.

Local source libraries may live under `lib/src`; compiled libraries have a declared export interface and compatibility requirements. An import does not download a package. Keep products out of authored source directories so the tree continues to describe what the program is made from.

Exercises: Try Changes, Then Explain Them

1. Move `Summary` to another file in `Analysis`. Which imports change?

2. Add a second root file that calls the calculation. What belongs in that file's inventory?

3. Replace the three inputs with `[0, 0, 0]`. Then remove them all. Why are those cases different?

Discussion

1. Moving the record within the same module changes no imports.

2. The second consumer needs `import Analysis`, not an import of either filename.

3. Three zeros have a defined maximum of zero and produce a successful record with count 3 and total zero. An empty sequence has no maximum under our chosen interface, so it fails. Removing the guard would not make that case valid; it would permit an invalid read of `values[0]`.

On Your Own

Extend the report with a minimum reading. Update the `Summary` record, the calculation and the consumer in their existing files; keep one direction of dependency. For `[4, 9, 2]`, the report must add minimum 2. A singleton `[7]` must report minimum and maximum 7, while `[0, 0, 0]` must report both as zero. Empty input must retain its error before any first-element access. Keep the related record fields and accumulator declarations visually aligned, and verify that every return constructing a Summary initializes the new field. Do not make the analysis module print its own result.

Further Reading

ec Technical Manual, Modules and Imports; Projects and Configuration; Library Interfaces.

Storage and Allocators

Build a reading filter that preserves values at or above a threshold. For `[3, 8, 2, 8]` and threshold 5, its answer must be `[8, 8]`, in that order. The new difficulty is not the comparison. It is deciding where the answer lives after the filtering function returns.

Create a new application file, `readings.ec`. We will construct a producer named `selectReadings`, then a caller named `runFilter` that owns the storage, and finally connect the caller to `start`. The inventory needs `import Core`, `import Memory` and, in its own group, `from IO import print`. The completed source appears after those steps.

Decide who owns the answer

Returning a view into a producer's local array would leave the caller with an address whose storage has ended. Instead, the caller will supply an allocator and receive the obligation to release the resulting allocation.

An allocator reserves suitably sized and aligned storage and accepts it back when released. `Memory.Allocator` is a small interface describing how to reach that service. We will use a fixed arena: an allocator that carves blocks from a byte buffer supplied by us. That buffer is its backing storage; there is no hidden unlimited source of bytes.

The producer's result is `Core.Result[[:]Int32]`. Read that from the inside out: a slice of Int32 values, wrapped as either success or error. The type describes the answer's shape.

The caller must keep the input available, initialized and unchanged while the function runs. It must not share any bytes with the backing memory that the allocator may write, including memory used to keep track of allocations. The caller must later give the returned memory back through the same allocator.

Count before allocating

We could grow an answer one item at a time, but that would introduce capacity decisions. For this first allocation example, traverse twice. The first pass counts exactly how many elements we will need.

Inside `selectReadings`, whose parameters are `values [:]Int32`, `threshold Int32` and `allocator Memory.Allocator`, begin with:

```
    count UintSize = 0

    for value in values
        if value.* >= threshold
            count++
```

Predict `count` after each value in our sample: 0, 1, 1, 2. It can never exceed `values.len`, so UintSize is sufficient. The loop visits element pointers; reading `value.*` gets the integer.

Now append one allocation request after the loop:

```
    selected = Memory.allocateArray[Int32](allocator, count)?
    index UintSize = 0
```

The type argument asks for Int32 elements, while `count` is the number of elements, not bytes. The library checks the required extent. The returned storage is uninitialized. It is not yet an array of usable readings merely because an address exists.

The request may fail. The visible `?` returns that error before the fill loop can use `selected`; reserving storage must succeed before we write its elements.

Fill exactly the slots counted

Add the second traversal after `index`:

```
    for value in values
        if value.* >= threshold
            selected[index] = value.*
            index++

    return selected
```

Because the input stays unchanged, the second pass accepts exactly the same values as the first. Each accepted value writes one slot and advances the index. The returned slice therefore contains only initialized elements. No operation after allocation can report a recoverable error in this algorithm.

Why did we not defer release inside the producer? A successful result intentionally transfers the release obligation to the caller. Releasing it before that caller receives the slice would defeat the design. Cleanup belongs beside the acquisition in the scope that actually owns the result's use.

Build that owning scope

Define `runFilter() -> Core.Status` after the producer. In its body, declare the sample readings and backing buffer first:

```
    readings []Int32 = [3, 8, 2, 8]
    backing  [256]Uint08
```

These arrays are local to `runFilter`. Their type columns align because they describe the two pieces of storage in this example. One holds input values; the other will be used only through the arena. We do not read uninitialized backing bytes as integers.

Next construct the arena and immediately pair it with destruction:

```
    arena = Memory.fixedArena(backing[:])
    defer Memory.destroyArena(&arena)
```

The arena must remain at this address while its interfaces and allocations use it. Once established there, obtain an interface and call the producer:

```
    selected = selectReadings(readings[:], 5, Memory.allocator(&arena))?
    defer Memory.release(&arena, selected)
```

Notice the order. If filtering fails, its release registration is never reached, but arena destruction was already registered. On success, release runs first at scope exit, then arena destruction. The allocator interface can be copied; its context still points to the same stationary arena.

Finally, read the successful result and print it. This is the last phase of `runFilter`, followed by `return Core.ok()`. Output failures also leave through the registered cleanup calls. The storage lifetime does not depend on whether reporting succeeds.

Assemble and trace the program

The complete `readings.ec` is below. The producer signature fits on one line, keeping its input, limit and allocator visible together. Its small loops stay close to their shared count and output state. The final return is a separate phase.

```
________________________________________________________________________________

[ Reading Filter ] Selects readings using caller-owned arena storage.
________________________________________________________________________________

import Core
import Memory

from IO import print


func selectReadings(values [:]Int32, threshold Int32, allocator Memory.Allocator) -> Core.Result[[:]Int32]

    count UintSize = 0

    for value in values
        if value.* >= threshold
            count++

    selected = Memory.allocateArray[Int32](allocator, count)?
    index UintSize = 0

    for value in values
        if value.* >= threshold
            selected[index] = value.*
            index++

    return selected


func runFilter() -> Core.Status

    readings []Int32 = [3, 8, 2, 8]
    backing  [256]Uint08

    arena = Memory.fixedArena(backing[:])
    defer Memory.destroyArena(&arena)

    selected = selectReadings(readings[:], 5, Memory.allocator(&arena))?
    defer Memory.release(&arena, selected)

    print("Selected count: [ {selected.len} ]\n")?

    for index, value in selected
        print("Reading [ {index} ]: [ {value.*} ]\n")?

    return Core.ok()
```

Listing continued

```
_____________________________________________________________________________

[ Application Entrypoint ]
_____________________________________________________________________________

func start() -> Core.Status

    print("""
        ____________________________________________

         Reading Filter
        ____________________________________________
        """)?

    return runFilter()
```

The result count is 2, with 8 at indices 0 and 1. Change the threshold argument from 5 to 9. The producer counts zero, returns a successful empty slice without an allocated block, and the output loop does nothing. Releasing that empty slice is valid. Reading its first element would not be valid, which is why reporting follows the length through iteration.

Next restore threshold 5 and shrink backing to one byte. The two Int32 results cannot fit: filtering returns `NO_SPACE`, no selected-release call is registered, and arena destruction still runs. This is a concrete failure test using a defined error, not an invalid pointer.

Preserve the information needed to release

For a fixed arena, releasing a child allocation ends that child's lifetime. Its backing bytes stay reserved until rewind, reset or destruction. Destruction returns the backing to its original owner; it does not free an automatic array to the heap.

Keep the exact allocation slice. A shorter view such as `selected[:1]` does not become an independently releasable block. Copying the descriptor creates another view of the same allocation, not another owner entitled to release it.

Resize deserves special care. Requesting the current count returns the same slice without calling the allocator. For the Int32 storage used here, a successful change from one positive length to another ends the old allocation, even when the numerical address is reused. Resizing to zero releases; resizing an empty slice to a positive count allocates. Failure preserves the previous allocation. Newly added elements require initialization.

Consequently, a deferred release that saved `items` before a size-changing resize would still hold the old descriptor. A resizing design must arrange cleanup around its current owner, perhaps through a stable owner record passed by pointer. Our count-then-allocate design avoids that complication. Its simplicity is an engineering reason to choose it, not merely a formatting preference.

The two-pass filter performs O(n) comparisons and O(k) copies, where n is the input length and k is the accepted count, with one typed allocation request. This notation describes how work grows: O(n) comparisons means their count is bounded by a constant multiple of the input length as the input grows; O(k) copies depends on the accepted length in the same way. These are growth bounds, not durations in seconds. A single-pass growable collection trades that second input traversal for capacity management and possible copying.

Choose another storage policy when the task changes

Our filter needs one exact-size allocation. A task with many short-lived intermediate results may fit an arena with checkpoints. `Memory.mark` records the current allocation cursor. A later `Memory.rewind` ends allocations made after that mark and recovers their space together. Save the mark before the temporary work, and stop using every affected pointer before rewinding. The mark must come from this arena since its last reset, with no intervening rewind to an earlier position; it is not an arbitrary byte count or a transferable ticket. Check the returned status. Reset ends all child allocations.

A growable arena takes backing blocks from a parent allocator when needed. Constructing it does not allocate a block immediately, but later allocations may fail with the parent's error. The parent must outlive the arena. Reset keeps the first backing block for reuse and returns later blocks; destruction returns them all. This changes where the storage comes from, not the obligation to end every use before its allocation ends.

A pool addresses a different pattern: individual fixed-size objects that are acquired and returned independently. `Memory.pool[T]` uses part of the supplied storage for one occupancy bit per usable slot, so its capacity can be smaller than the input slice length. Read the resulting capacity instead of assuming every input element becomes an object slot. A pool allocation creates one object; it is not a general replacement for our multi-element result array. Like an arena, the pool object must stay at its established address while its allocations or allocator interfaces are in use.

Exercises: Try Changes, Then Explain Them

1. Set the threshold to 2. Predict the returned sequence and output indices.

2. Could `runFilter` return its nonempty `selected` slice instead of printing it?

3. Why not allocate `values.len` elements and release only the accepted prefix?

Discussion

1. Threshold 2 keeps all four readings in their original order, indexed 0 through 3.

2. Returning this local arena's slice would leave the caller without live backing after `runFilter` ends. A lasting result needs an allocator and owner whose lifetime extends through its use.

3. We could allocate room for the whole input instead. Then we must keep the full slice returned by the allocator, separately from the shorter slice of accepted readings. Release the full slice through the same allocator; releasing only the shorter piece is an error.

On Your Own

Extend `selectReadings` to accept inclusive lower and upper bounds. Return `Core.INVALID_ARGUMENT` when the lower bound exceeds the upper bound, before allocating. Preserve order and duplicates. With `[3, 8, 2, 8]`, bounds 3 and 8 should produce `[3, 8, 8]`; bounds 9 and 10 should produce an empty result. Keep both passes governed by the same condition and keep the caller's release paired with the successful result. Test the reversed bounds with the small backing buffer too: the argument error should win before storage exhaustion is considered.

Further Reading

ec Technical Manual, Memory; Pointers, Lifetime and Memory.

Collections and Callbacks

Build a program that counts distinct readings without changing the caller's input. The batch `[8, 3, 8, 2, 3]` contains three distinct values. We will copy it into a fixed collection, sort the copy, then count the beginnings of equal-value groups.

Use a new file named `distinctReadings.ec`. Its inventory will import `Collections` and `Core`, followed by `from IO import print`. We will construct two helpers, `compareReading` and `countDistinct`, before assembling the application. There is room for at most 64 readings; exceeding that bound will be a normal `NO_SPACE` failure.

Make an empty collection over visible storage

An array of storage and a collection's current sequence are different things. `Collections.Array[T]` has capacity for elements of type T and a current length. A fixed collection borrows storage already supplied by its caller and never grows by allocating elsewhere.

Start the body of `countDistinct(values [:]Int32) -> Core.Result[UintSize]` with:

```
    storage [64]Int32
    buffer = Collections.arrayFixed(storage[:])
    buffer.append(values)?
```

After construction, capacity is 64 and length is zero. Construction does not read uninitialized `storage`. Append then copies the supplied values into the sequence. The library infers Int32 from the storage slice; no explicit `[Int32]` is necessary in this call.

The receiver call `buffer.append(values)` means `Collections.append(&buffer, values)`: pass the existing descriptor's address first. The dot does not introduce an object system or a hidden owner. Similarly, `buffer.view()` will mean `Collections.view(&buffer)`.

Predict the state after appending five readings: length 5, capacity 64, original order retained. Try 65: the operation reports `NO_SPACE` and preserves the empty collection. Our `?` returns before sorting. Keep those construction and append steps on separate lines, because one establishes storage and the other is a fallible mutation; padding their assignment signs would not reveal a common table.

Describe an order the sort can call

The sort knows how to rearrange elements, but it needs our definition of which element comes first. It calls a supplied function with two element pointers. The function returns a negative value for less, zero for equivalent and a positive value for greater.

Before writing the function, fill in three predictions: comparing 3 with 8 should be negative; comparing 8 with 3 positive; comparing 8 with 8 zero. This helper implements those choices:

```
func compareReading(_ *Void, left *Int32, right *Int32) -> Int32

    if left.* < right.*
        return -1
```

Listing continued

```
    if left.* > right.*
        return 1

    return 0
```

Put it before `countDistinct`, after the imports. The first argument is a context pointer required by the sort's interface. `*Void` carries an address without naming a readable element type. Our function needs no extra state, so _ marks that argument unused. The sort's default context is null.

Do not implement comparison as `left.* - right.*`: signed subtraction can wrap at the limits and give the wrong sign. Direct comparisons work across the full Int32 range. Each successful comparison ends the callback immediately; zero is reached only when neither value is smaller.

`&compareReading` is the helper's function address. Passing it gives the sort a callback: a function it can call during its work. No thread or background activity is implied. Ordinary function values do not capture the caller's runtime locals. A callback needing state receives it explicitly, with a lifetime covering its calls.

Sort the copy, then find each group's first element

Return to `countDistinct`. Immediately after append, add:

```
    ordered = buffer.view()
    Collections.sort(ordered, &compareReading)
```

The view points into the local collection's initialized sequence. It does not copy those values again. Sorting that view rearranges the copy in `storage`, leaving the caller's input untouched.

For our five readings, write the sorted row `[2, 3, 3, 8, 8]`. Mark every element whose value differs from the element immediately before it. Index zero needs a mark too, because it starts the first group. Those marks are at indices 0, 1 and 3.

Add this final phase after the sort:

```
    count UintSize = 0

    for index, value in ordered
        if index == 0 || value.* != ordered[index - 1]
            count++

    return count
```

Short-circuit evaluation is part of the reasoning. When index is zero, the first condition is true, so the predecessor expression is not evaluated. Later indices can safely subtract one. Empty input performs no iteration and returns zero; a singleton starts one group and returns one.

The sort compares values according to one stable ordering relation. A comparator that changes its answer on alternate calls breaks that requirement. A callback is part of the algorithm's contract, not just a function with convenient argument types.

Assemble the program

Here is the complete source. The imports appear once at the top, then the comparison helper, then its consumer, then the application entry point. That order follows how the parts work together.

```
________________________________________________________________________________

[ Distinct Readings ] Counts value groups in a sorted local copy.
________________________________________________________________________________

import Collections
import Core

from IO import print


func compareReading(_ *Void, left *Int32, right *Int32) -> Int32

    if left.* < right.*
        return -1

    if left.* > right.*
        return 1

    return 0


func countDistinct(values [:]Int32) -> Core.Result[UintSize]

    storage [64]Int32
    buffer = Collections.arrayFixed(storage[:])
    buffer.append(values)?

    ordered = buffer.view()
    Collections.sort(ordered, &compareReading)

    count UintSize = 0

    for index, value in ordered
        if index == 0 || value.* != ordered[index - 1]
            count++

    return count


________________________________________________________________________________

[ Application Entrypoint ]
________________________________________________________________________________

func start() -> Core.Status

    print("""
        ________________________________________

         Distinct Readings
        ________________________________________
        """)?
```

Listing continued

```
    readings []Int32 = [8, 3, 8, 2, 3]
    count = countDistinct(readings[:])?

    print("Distinct values: [ {count} ]\n")?

    return Core.ok()
```

With successful output, the count is 3. Change the call to `countDistinct(readings[:0])`: it becomes zero. Keep the array but pass `readings[:1]`: it becomes one. In each case, the local collection owns no separately allocated block, and its Int32 elements own no resources. Its automatic backing remains alive throughout the call and ends afterward.

A direct check of input preservation is also useful. After the helper returns, `readings[0]` remains 8; sorting the caller's array would have made it 2. Keeping the copy visible in the source makes that behavioral promise easy to find.

Further Study: When the Fixed Batch Stops Fitting

The distinct-count program is complete. You can do its homework now with the fixed collection just constructed. The rest of this section compares changes that a different workload might require; none needs to be added to the working batch counter.

A growable collection retains an allocator and expands capacity when needed. Its failures preserve the existing collection, but successful growth can end earlier view guarantees. Obtain a new view after growth and do not keep element pointers across mutations without checking the particular operation's promise.

A copied collection descriptor is not a second owner. Clearing or releasing it also does not follow pointers inside elements and release their referents. If elements own resources, the program must arrange their cleanup before discarding their last owner.

Nonempty slice append rejects overlap with the destination array's backing capacity. Appending a view of the same collection is therefore different from appending a scalar value, which is captured before growth. Our independent local copy avoids that overlap issue.

The sort performs O(n log n) comparisons in the worst case and uses O(log n) stack space without heap allocation. Here n is the number of elements, and log n grows like the number of times n can be halved before reaching one, as in binary search. The product n log n grows faster than n but slower than the n squared worst-case work of insertion sort. These bounds describe growth, not the exact number of instructions or a measured speed. Copying and counting are linear. Equal elements may change relative order: the sort is not stable. That does not affect an integer group count, but it would matter if equal-key records carried an arrival order you needed to preserve.

A queue would serve an arrival-order task; a hash map would serve repeated key lookups with explicit hash and equality callbacks. For a map, equal keys must produce the same hash. Both callbacks must give stable answers for unchanged keys and context while those keys reside in the map, and stored keys must not change. A text key that borrows mutable bytes therefore needs an ownership decision before insertion: changing those bytes would change the key the map relies on. Our bounded batch needs neither a queue nor a map. Choosing a collection starts with the operation the program needs, not with the most elaborate available structure.

Exercises: Try Changes, Then Explain Them

1. Predict the count for `[5, 5, 5]` and `[-2, 0, -2, 4]`.

2. Remove the copy and sort the input directly. Which lines change, and what promise changes?

3. Why does the predecessor expression not read outside the slice at index zero?

Discussion

1. The first input forms one group; the second sorts to `[-2, -2, 0, 4]` and forms three.

2. For the direct-sort version, remove `storage`, `buffer` and append; replace the view assignment with `ordered = values`. Keep the sort and counting phase. The input must now be writable, and the caller must accept rearrangement; the fixed capacity limit also disappears.

3. The index-zero case is safe because `||` skips its right operand when the left operand is true, not because indexing clips an invalid subscript.

On Your Own

Extend the answer to a record containing the number of distinct values and the number of distinct values occurring at least twice. For `[8, 3, 8, 2, 3]`, those fields should be 3 and 2. For `[5, 5, 5]`, they should be 1 and 1, not 1 and 2. Empty input should produce two zeros. Keep the original-input preservation and 64-element failure policy. Decide what state must survive from one sorted element to the next, and check both the first group and the final group. Keep the related counters visually grouped, while leaving fallible append and sort as separate phases.

Further Reading

ec Technical Manual, Collections; Functions and Calls.

Streams and Files

We will build a file copier that creates a new destination and refuses to overwrite an existing one. Before involving files, we will make the transfer fail in a controlled memory example. That small experiment tells us what the copier must preserve when it reports an error.

Create a new file, `copyFile.ec`. Its final inventory will import `Core`, `IO` and `System`, plus `print` from IO. The complete application needs the `files` and `standardStreams` capabilities, available on the native `arm64-macos` profile. A capability is a service made available by the selected profile. The memory experiment alone needs no filesystem service.

Start with bytes that we can follow

A stream is a source or destination transferring a sequence of bytes. A file can provide one, but so can an array. Reader and Writer adapters expose the same operations for either case; each carries a context pointer and a callback. They borrow the context rather than keeping it alive automatically.

Define `testShortCopy()` with no parameters or result. We will make it use five source bytes, three output slots and two scratch slots. Begin that function's body with:

```
    sourceBytes []Uint08 = [65, 66, 67, 68, 69]
    outputBytes [3]Uint08
    scratch     [2]Uint08
```

The source bytes are ASCII A through E. Scratch is a temporary chunk buffer, reused during the copy; its length limits a chunk, not the entire source. The aligned type column reveals these three storage roles. None overlaps another.

Next add the two memory stream records:

```
    source      = IO.MemoryReader(bytes = sourceBytes[:], position = 0)
    destination = IO.MemoryWriter(bytes = outputBytes[:], position = 0)
```

Their positions count bytes from the beginning. Both records must stay at these addresses while the adapters borrow them. Constructing these records transfers no bytes. The next call performs the transfer and advances their positions.

Predict a partial transfer

Now add the copy call:

```
    progress = IO.copy(IO.memoryReader(&source), IO.memoryWriter(&destination), scratch[:])
```

`IO.copy` reads a chunk and writes all of that chunk before reading another. Its answer is `IO.Progress`, a record with `count`, `error` and `end`. Unlike Result, all three fields are meaningful together; there is no payload to unwrap with `?`.

Before writing assertions, trace the operation:

Chunk	Read from source	Written to destination	Positions afterward
First	A, B	A, B	source 2, destination 2
Second	C, D	C, then no room	source 4, destination 3

The copy reports three bytes written, `NO_SPACE`, and `end = false`. D has already been consumed but not written; E has not been read. Add assertions for those facts. They will appear in the completed program below.

This is why an error cannot be reduced to “nothing happened.” It is also why restarting the same operation without a position policy is not recovery. The stream state has changed.

Give the same algorithm file owners

We can now reuse `IO.copy` with file adapters. It must be the File records, not their copied adapters, that own the close obligation. The new helper will be `copyNewFile(sourceText Text, destinationText Text) -> Core.Result[UintSize]`.

Start with path validation:

```
    sourcePath      = System.path(sourceText)?
    destinationPath = System.path(destinationText)?
```

These calls reject a zero byte in a path. They do not open files or inspect their contents. Next open the source for reading and pair it with cleanup:

```
    source = System.openFile(sourcePath, System.FILE_READ)?
    defer closeIfOpen(&source)
```

We need to define that cleanup helper before its consumer:

```
func closeIfOpen(file *System.File)
    if file.open
        _ = System.closeFile(file)
```

The helper checks the original descriptor and deliberately accepts a closing failure. `_ =` is on the risky call itself. It is not placed on the defer: the defer's job is to schedule cleanup, while this helper states its error policy.

Now append destination acquisition to `copyNewFile`:

```
    flags = System.FILE_WRITE | System.FILE_CREATE | System.FILE_EXCLUSIVE
    destination = System.openFile(destinationPath, flags)?
    defer closeIfOpen(&destination)
```

Bitwise OR combines three independent options: writing, creation and a requirement that the file be newly created. An existing destination yields `ALREADY_EXISTS`, rather than an overwrite. If this open fails, the source cleanup was already registered. If it succeeds, cleanup order is destination then source.

Decide which errors the caller should receive

The transfer phase uses fresh scratch storage and adapters pointing to our stationary File records:

```
    scratch [4096]Uint08
    progress = IO.copy(System.fileReader(&source), System.fileWriter(&destination), scratch[:])

    if progress.error != Core.NO_ERROR
        return progress.error
```

The visible error check is necessary because Progress is not Status. The count includes bytes written even if an error accompanies them. Our function chooses to return only a successful byte count; an application that needs partial counts on failure would choose a richer result record.

After successful transfer, append these final steps:

```
    System.closeFile(&destination)?
    System.closeFile(&source)?

    return progress.count
```

Why close explicitly when cleanup is registered? Here success means transfer and closing both succeeded. `closeFile` consumes its owner and marks it closed even when it returns an error. The deferred helper therefore sees that owner closed and avoids closing it again. On earlier failure paths, cleanup failures are accepted so they do not replace the selected primary error.

This policy does not erase external effects. A newly created destination may remain with a prefix if transfer or closing fails. Exclusive creation is not an atomic replacement operation.

Assemble the copier

Use the completed source below. Before running the native application, replace the two path literals in `start` with your chosen existing source and absent destination paths. Using full paths avoids assuming a particular working directory. The source may contain arbitrary bytes; no UTF-8 decoding is involved.

```
________________________________________________________________________________

[ File Copier ] Copies into a new file and checks partial-transfer behavior.
________________________________________________________________________________

import Core
import IO
import System

from IO import print


func testShortCopy()

    sourceBytes []Uint08 = [65, 66, 67, 68, 69]
    outputBytes [3]Uint08
    scratch     [2]Uint08

    source      = IO.MemoryReader(bytes = sourceBytes[:], position = 0)
    destination = IO.MemoryWriter(bytes = outputBytes[:], position = 0)

    progress = IO.copy(IO.memoryReader(&source), IO.memoryWriter(&destination), scratch[:])
```

Listing continued

```
    Core.assert(progress.count == 3)
    Core.assert(progress.error == Core.NO_SPACE)
    Core.assert(!!progress.end)
    Core.assert(source.position == 4)
    Core.assert(destination.position == 3)


func closeIfOpen(file *System.File)
    if file.open
        _ = System.closeFile(file)


func copyNewFile(sourceText Text, destinationText Text) -> Core.Result[UintSize]

    sourcePath      = System.path(sourceText)?
    destinationPath = System.path(destinationText)?

    source = System.openFile(sourcePath, System.FILE_READ)?
    defer closeIfOpen(&source)

    flags = System.FILE_WRITE | System.FILE_CREATE | System.FILE_EXCLUSIVE
    destination = System.openFile(destinationPath, flags)?
    defer closeIfOpen(&destination)

    scratch [4096]Uint08
    progress = IO.copy(System.fileReader(&source), System.fileWriter(&destination), scratch[:])

    if progress.error != Core.NO_ERROR
        return progress.error

    System.closeFile(&destination)?
    System.closeFile(&source)?

    return progress.count

_______________________________________________________________________________

[ Application Entrypoint ]
_______________________________________________________________________________

func start() -> Core.Status

    print("""
        ________________________________________

         File Copier
        ________________________________________
        """)?

    testShortCopy()
    written = copyNewFile("input.bin", "output.bin")?

    print("Copied bytes: [ {written} ]\n")?

    return Core.ok()
```

The memory test runs before opening either path. A false assertion is a broken program invariant and causes fatal assertion failure; a normal file error remains a returned failure. After a successful copy, a second attempt with the same destination should fail because the destination now exists.

The copying helper does not retry an interrupted transfer. A reader returning zero bytes without error or end causes `WOULD_BLOCK`, so the helper does not spin. A short transfer is otherwise valid. Writer error takes precedence over the reader status for the chunk being written; if writing succeeds, a reader error or end stops the copy.

Empty source data is a normal success when the reader reports end. Empty scratch storage gives `INVALID_ARGUMENT`. Cumulative count overflow is checked before another byte beyond the representable total is consumed. A zero-length read or write skips its callback, so it cannot probe whether an underlying file is open.

If you later add a buffered writer, remember that acceptance into its queue is different from delivery to the underlying file. Flush and check that queue explicitly. Closing the file does not flush an unrelated IO buffer, just as returning an error does not undo a previously written prefix.

Standard output already uses a small shared buffer, so repeated `print` calls do not each need a trip to the operating system. A successful print has consumed our source bytes, but some output may still be waiting in that buffer. Use `IO.flush()?` when the next action depends on delivery, or when we need to handle its error here. On a terminal, a newline flushes the line; reading stdin also flushes the text asking for input, even without a newline. Normal program completion flushes the remaining output and reports a failing exit status if that final delivery fails. Our caller-owned file buffer is separate: `IO.flush(&buffer)` empties that buffer, while `IO.flush()` empties standard output.

Exercises: Try Changes, Then Explain Them

1. Increase the memory destination to five bytes. Predict count, error and end.

2. Restore three destination bytes but increase scratch to five. How far does the source advance?

3. Why must cleanup inspect the original File rather than a copy made before explicit close?

Discussion

1. Five destination bytes allow the whole source to be written: count 5, `NO_ERROR`, end true.

2. Five scratch bytes can consume the whole source before only three fit in the output, so source position becomes 5 while the copy still reports three written bytes.

3. A copied File does not create a second resource owner; after explicit close, its old handle bits would be stale. The deferred pointer reaches the original descriptor whose closed state was updated.

On Your Own

Add `testCopyBoundaries` and call it after `testShortCopy`, before any file is opened. Cover an empty source with nonempty scratch, a nonempty source with empty scratch, and a zero-capacity memory destination. Check both progress and positions. The empty source should copy zero successfully with end true. Empty scratch should fail with `Core.INVALID_ARGUMENT` before either position changes. For the zero-capacity destination, choose two-byte scratch and verify source position 2, destination position zero, count zero and `Core.NO_SPACE`. Keep each case's storage and stream records together, and do not reuse a stream position accidentally between cases.

Further Reading

ec Technical Manual, IO; System.

Choosing Behavior with Function Values

A command interpreter begins innocently. One command adds a value, another replaces it, and an `if` chain chooses between them. As the command set grows, the same decision appears in several places: which name is recognized, which operation runs, perhaps which description is printed. A table can keep those facts together, provided a row can hold behavior as well as data.

You have already used callbacks through collection and stream interfaces. Here we build a small callback interface ourselves. The important design question is what a callback needs to know. A function address identifies code. A separate context identifies the particular state on which that code should act. Keeping those two roles visible makes dispatch tables useful without introducing hidden captures or hidden ownership.

Start with Two Ordinary Operations

We will build a command runner whose input is a name and an integer. Its result will be an updated counter, together with a count of recognized changes. Work in one module with no imports; the declarations below are a reusable component, not an application entry point. Start with just the state:

```
struct Counter
    value   Int32
    changes UintSize
```

The two aligned fields describe different parts of the same state. Keeping them together lets us pass one address whenever an operation must update both. Add these functions directly below the type, leaving two blank lines between functions:

```
func addValue(counter *Counter, amount Int32)
    counter.value += amount
    counter.changes++


func setValue(counter *Counter, value Int32)
    counter.value = value
    counter.changes++
```

Before introducing a callback, make the operations work by calling them directly. Add this small consumer after them:

```
func runCommands() -> Counter

    counter = Counter(value = 10, changes = 0)

    addValue(&counter, 5)
    setValue(&counter, 3)

    return counter
```

Follow the state: the first call produces 15 and one change; the second produces 3 and two changes. The pointer makes each mutation visible at the call site. We have a useful starting version. It knows which functions to call, however, while our command input contains names such as `"add"`.

Give a Table Row Some Behavior

A growing chain of name tests would repeat the relationship between a name and its operation. We will put that relationship into rows. Insert the following declarations after `Counter`, before the operation functions:

```
type Operation func(*Counter, Int32)

struct Command
    name  Text
    apply Operation
```

`Operation` names a function-pointer type. Read `func(*Counter, Int32)` as "a function taking a counter address and an integer, returning no value." Omitting a result here means `Void`. This description is the function's **signature**: the parameter and result types that a call must match. Both callbacks satisfy it. The alias earns its place because it represents a concept used by the command record and by any future command implementation.

There is a useful distinction between choosing a function and calling it. `&addValue` chooses the code and produces a value that can be stored in a row. `addValue(&counter, 5)` calls that code immediately. The dispatcher does the two jobs at different times: table construction stores an address; a later name match calls the stored address with the counter and argument for this request.

Now add the dispatcher after `setValue` and before `runCommands`. Its job is to find one row and call it; it knows nothing about addition or replacement.

```
func dispatch(commands [:]Command, counter *Counter, name Text, argument Int32) -> Bool

    for command in commands
        if command.name == name
            command.apply(counter, argument)
            return true

    return false
```

The loop binds `command` as a pointer to an element, as in earlier array lessons. Field access reaches its name and function pointer. Calling `command.apply(counter, argument)` calls that pointer with exactly the two written arguments. It does not secretly pass `command` as another argument: a function-pointer field is different from an associated function selected through receiver notation.

The table has a small but useful contract. Its entries contain nonnull, correctly typed callback addresses, and every callback accepts the supplied live `Counter`. Names are compared as text values. The first matching row runs once; no match changes nothing and returns `false`. Duplicate names are therefore a policy choice for the table's author, not an ambiguous language operation.

Follow Three Commands

Replace the whole earlier `runCommands` declaration with this version. Keep the types, two callbacks, and dispatcher. This replacement puts the table next to its use and keeps all counter storage local:

```
func runCommands() -> Counter

    commands = [Command(name = "add", apply = &addValue),
                Command(name = "set", apply = &setValue)]

    counter = Counter(value = 10, changes = 0)

    added = dispatch(commands[:], &counter, "add", 5)
    replaced = dispatch(commands[:], &counter, "set", 3)
    recognized = dispatch(commands[:], &counter, "multiply", 2)

    return counter
```

The three Boolean bindings let us follow recognition separately from the returned counter. They are not three configuration rows: the calls are successive actions whose order changes the state, so we do not align them into an artificial table.

The explicit `&` makes the function addresses visible. `commands[:]` supplies a slice of the table; passing an array does not silently create a slice. `&counter` supplies the changing state. Each dispatch returns before the next begins.

Following each call gives these expected state changes:

Request	Matching row	Counter value afterward	Changes	Returned value
add 5	`addValue`	15	1	true
set 3	`setValue`	3	2	true
multiply 2	none	3	2	false

The call sites do not need to know each callback's body. They do need the shared contract. For example, replacing `addValue` with a callback that stores `counter` for later use would introduce a new lifetime obligation that this synchronous dispatch design does not supply. An interface should say whether it merely calls now or retains a callback/context pair for later.

The integer addition uses ordinary wrapping `Int32` arithmetic. The example does not promise overflow detection. The `changes` counter also wraps at its width. If either limit matters to an application, the command contract must define and implement the desired check.

State Is an Argument

Suppose there are two independent counters. Both can use the same `commands` array. As an experiment, insert these statements in `runCommands` immediately before its final return. They use its existing table and do not replace `counter`:

```
    first  = Counter(value = 10, changes = 0)
    second = Counter(value = 100, changes = 0)

    dispatch(commands[:], &first, "add", 2)
    dispatch(commands[:], &second, "add", 7)
```

Both calls find the same table row and call the same `addValue` function. The first receives `&first`, so it leaves that counter at 12. The second receives `&second`, so it leaves that counter at 107. Each counter records one change. Neither call needs to remember what the other did. The address selecting the state is what we mean by **context** here: one behavior can serve many objects.

Some library interfaces use `*Void` so they can accept many unrelated context types. At that boundary, convert explicitly to the interface pointer type and recover a typed pointer through a named alias in the adapter. Such a conversion preserves an address; it does not check that a live object of the intended shape is there. Our own interface needs only counters, so keeping `*Counter` is simpler and gives callers a stronger type check.

Neither a function address nor a context pointer owns what it refers to. Our local table and local counter remain alive for all three synchronous calls. A stored callback might instead outlive its context. A function defined in a dynamically loaded library also needs that library to remain loaded. Copying the pointer solves neither problem.

Anonymous Does Not Mean Capturing

An anonymous function is useful when a small operation belongs at one call site. It is still a function pointer. Continue the same experiment by inserting this declaration after both experimental dispatch calls and before the final return. It creates an operation that replaces a counter value; it can be used wherever an `Operation` is expected.

```
    replace Operation = func(counter *Counter, value Int32)
        counter.value = value
        counter.changes++
```

This declaration creates a function value; it does not yet change a counter. Add `replace(&first, 20)` immediately after the declaration to invoke it. The first counter becomes 20 and records its second change; the second counter stays 107. The original `counter`, which the function returns, stays 3 with two changes. The function uses its own parameters. It cannot read an ordinary local variable from the surrounding function. Writing a callback inside a scope does not package that scope's runtime state. If a callback needs a limit, a prefix or an accumulator, give it an explicit parameter or put that information in the context object.

Forming this anonymous function value allocates nothing, and repeated evaluations of the same anonymous function have the same code address. It can refer to permitted module declarations, enclosing type parameters and constants, but not capture surrounding runtime locals. A named function remains preferable when the operation deserves a reusable name or a substantial explanation.

What We Have Built

The assembled component consists of `Counter`, `Operation`, `Command`, `addValue`, `setValue`, `dispatch`, and the replacement `runCommands`. Calling `runCommands()` returns a counter with value 3 and two changes. The two-counter and anonymous-function experiments can be removed together without changing that result. A caller with its own live counter and table can use `dispatch` directly; no global registry or allocator is required.

The abstraction paid for itself when names became data. With only the two fixed calls in the first version, it would have added a table and search for no benefit. Now adding a command changes a callback and one row, while recognition and invocation remain in one place.

See the Component Produce a Result

The component has no output of its own. To observe it as an application, keep its declarations in a separate example directory and give the file a source banner titled Command Runner. Add `import Core` and `from IO import print` to the inventory at the top of the component file.

```
import Core

from IO import print
```

Append this entry-point section after the component's final function. It calls the lesson consumer, prints its result, and checks output failure. Keep only the final versions of declarations that the lesson replaced:

```
________________________________________________________________________________

[ Application Entrypoint ]
________________________________________________________________________________

func start() -> Core.Status

    print("""
        ________________________________________

         Command Runner
        ________________________________________
        """)?

    result = runCommands()
    print("Value: [ {result.value} ]; changes: [ {result.changes} ]\n")?

    return Core.ok()
```

After the banner, successful execution is expected to report:

```
Value: [ 3 ]; changes: [ 2 ]
```

Exercises and Discussion

1. Add a command that subtracts. Write another `Operation` callback and add a `"subtract"` row. What must change in `dispatch`?

The callback can perform `counter.value -= amount` followed by `counter.changes++`. Its signature matches `Operation`, so only the table gains a row; the search algorithm stays unchanged. Keep the initializer columns aligned with the neighboring rows. This is a useful extension because the existing abstraction corresponds to a real shared behavior.

2. Give two rows the name `"add"`. Which one executes? Does the second run afterward?

The first matching row executes, and `return true` leaves the dispatcher immediately. The second does not run. A table builder could reject duplicates, but that would be another explicit validation step. Do not assume the language supplies one.

3. A callback must multiply by a limit chosen at runtime. Why can it not simply mention an enclosing local named `limit`?

Anonymous functions do not capture that local. One solution is a context struct containing the counter pointer and multiplier, with a callback type accepting that struct's pointer. The owner must keep both the context and the counter alive until the last call. Changing the callback signature is an honest change to what the interface needs; hiding a dangling address in global state is not a substitute.

In the next chapter, we retain the idea of passing behavior explicitly while removing duplication caused only by element types. Runtime selection and type specialization solve different problems, and a small search algorithm will let us see both at work.

On Your Own

Extend the command runner to process a batch. Define a request record containing a command name and an `Int32` argument, then write `dispatchBatch` taking the command table, a counter pointer, and a slice of requests. It must process requests in order and return the number recognized. An unknown request must leave the counter unchanged and must not stop later requests.

Use the original add/set table. Starting at 10, the requests add 5, unknown 2, set 3, and add 4 must finish at 7 with three changes and a recognized count of three. An empty batch must return zero and leave both counter fields unchanged. Borrow the table and requests only during the call; do not allocate or retain their pointers.

Hint: reuse `dispatch` so the name-matching policy has one implementation. Keep each request initializer on one aligned row so its name and argument can be checked together.

Further Reading

ec Technical Manual: Functions.

Reusing an Algorithm Across Types

Imagine that a report first searches a list of integer identifiers, then a list of records. Both searches walk from left to right, stop at the first match and report whether anything was found. Copying the loop creates two places to repair an off-by-one error. Forcing both lists into one representation creates conversions and loses useful type information.

A generic function lets the loop retain its structure while its element type changes. It does not decide what matching means. We will pass that policy as a function value, building on the previous chapter. This separates two choices: the concrete type is selected during translation, while the callback can be selected at runtime.

Start with an Integer Search

A search can fail normally. Returning zero alone is ambiguous because element zero may be the answer. Returning a pointer introduces a borrowing interface when the caller may only need a position. We will use a small record with a Boolean and an index. When `found` is false, the index is zero by our chosen convention and must not be used as a successful result.

Build a reusable search component in one module with no imports. Start with the concrete integer version below. We already know how to compare these values; there is no reason to introduce a type parameter yet.

```
struct SearchResult
    found Bool
    index UintSize


func findFirstInteger(values [:]Int32, wanted Int32) -> SearchResult

    for index, item in values
        if item.* == wanted
            return SearchResult(found = true, index = index)

    return SearchResult(found = false, index = 0)
```

The loop visits elements from left to right. `item` is an element pointer, so `item.*` reads the value. Finding a match returns immediately. The last line gives both empty input and unsuccessful searches the same deliberate result.

Add a consumer after the search function:

```
func integerSearch() -> SearchResult

    values = [12, 7, 12, 9]

    return findFirstInteger(values[:], 12)
```

Following this call gives `found = true, index = 0`. The second 12 never needs inspection. Change the requested value to 9 and the index becomes three; change it to 8 and `found` becomes false. Restore 12 before continuing.

Make the Second Use Earn the Generic

Now the report also needs to search records identified by a code. Copying the loop would preserve its structure while changing the element type and the comparison. Those are exactly the two choices we should expose. Keep `SearchResult`; replace `findFirstInteger` with the following function:

```
func findFirst[T](values [:]T, wanted T, equal func(T, T) -> Bool) -> SearchResult

    for index, item in values
        if equal(item.*, wanted)
            return SearchResult(found = true, index = index)

    return SearchResult(found = false, index = 0)
```

Only three parts have changed: `[T]` introduces the element type, the parameter types use it, and the loop asks `equal` instead of assuming an operator. We keep the search and its early exit together rather than extracting a helper for each line.

Our callback must accept two ordinary copyable values of `T`, return whether they match, and leave the searched sequence usable and unmodified throughout the call. This is our interface's contract; a function-pointer type alone cannot enforce that policy.

Read `[T]` as a type parameter, not a runtime argument. After a caller chooses `Int32`, the parameter `values` is a slice of `Int32`, `wanted` is an `Int32`, and `equal` accepts two `Int32` values. The body follows exactly the same control flow as a concrete integer search. The element pointer `item` is dereferenced because the callback takes values.

To read a specialization, replace every occurrence of `T` consistently. For `findFirst[Int32]`, the resulting parameter types are:

Parameter	Written in the generic	After choosing `Int32`
`values`	`[:]T`	`[:]Int32`
`wanted`	`T`	`Int32`
`equal`	`func(T, T) -> Bool`	`func(Int32, Int32) -> Bool`

The result stays `SearchResult` because its definition does not mention `T`. This is why the same result-handling code can serve integer and record searches even though their callbacks have different concrete types.

The early return is visible on its own line. The final return handles both an empty sequence and a sequence with no match. We never initialize an index and read it after an unsuccessful loop, so the empty case requires no special repair.

This function's no-mutation callback requirement is our interface contract, not a restriction automatically imposed on every function pointer.

Specialize It for Integers

Add this module-level helper after the search declaration, separated from the preceding function by two blank lines.

```
func sameInteger(left Int32, right Int32) -> Bool
    return left == right
```

Replace the entire `integerSearch` declaration with this consumer of the generic version. Keep `sameInteger` before it, so the comparison policy is visible before the call that selects it:

```
func integerSearch() -> SearchResult

    values = [12, 7, 12, 9]

    return findFirst[Int32](values[:], 12, &sameInteger)
```

The request still means “find the first 12.” The explicit type argument chooses the integer version of the algorithm; `&sameInteger` supplies its matching policy. We changed how the operation is assembled without changing its answer.

At index zero, the callback receives 12 and 12 and returns true. The search returns `found = true, index = 0` immediately; it does not inspect the later duplicate. Searching for 9 visits indices zero through three. Searching for 8 visits all four and returns `found = false, index = 0`. Searching an empty slice calls the callback zero times.

With a match at index `k`, the algorithm makes `k + 1` callback calls. With no match it makes `values.len` calls. The search itself allocates no storage, but its total time depends on the callback's work. Calling an expensive comparison “constant time” merely because it is one line would hide the real cost.

For these arguments, `findFirst(values[:], 12, &sameInteger)` can omit `[Int32]`. The slice and function pointer provide consistent type evidence; the literal is checked against the resulting parameter type. Starting with the explicit spelling is useful while learning because it shows what is specialized.

In ordinary code, prefer the shorter call once that relationship is clear. Deduction also looks inside a pointer to a generic record: an argument of type `*Tree[Int32, Uint64]` determines both `K` and `V` for a parameter `*Tree[K, V]`. There is no need to repeat those two types at each insertion. An explicit list still matters when a type appears only in the result, because the expected result does not supply deduction evidence.

Reuse the Walk, Change the Meaning

A record has more than one possible notion of a match. A report may identify a reading by its code while deliberately ignoring its current score. Add `Reading` and `sameCode` after `integerSearch`, as a new record-search region. They supply the second matching policy without asking the language to compare whole records.

```
struct Reading
    code  Uint32
    score Int32


func sameCode(left Reading, right Reading) -> Bool
    return left.code == right.code
```

Add this consumer immediately after `sameCode`. It is an addition, not a replacement for the integer example:

```
func readingSearch() -> SearchResult

    readings = [Reading(code = 40, score = 8),
                Reading(code = 70, score = 3),
                Reading(code = 90, score = 6)]

    wanted = Reading(code = 70, score = 0)

    return findFirst[Reading](readings[:], wanted, &sameCode)
```

The aligned rows let us compare codes and scores as a small table. The wanted record is a separate step because it is the query, not another stored reading.

The comparison at index zero is false; at index one it is true. The wanted score of zero is irrelevant to this callback. The returned index is one, and the original score remains three. This is a policy decision stated in code, not an accidental conversion between record types.

There is no built-in general record equality that the generic function can assume. If we had written `item.* == wanted` directly, this particular struct use would require an appropriate defined operation. Passing a callback makes the required operation explicit and allows several matching policies for the same type. Ordinary name lookup inside a generic still belongs to the declaration's context; callers cannot silently replace an internal helper by defining another helper with the same name.

What Specialization Does and Does Not Buy

`findFirst[Int32]` and `findFirst[Reading]` are different concrete function specializations. Repeating the same resolved type arguments refers to the same specialization. Integers stay ordinary integers and readings stay ordinary records; they are not put into a shared wrapper representation just to enter the loop. A generic body still has to obey its concrete types: noncopyable objects cannot be copied merely because their name was replaced by `T`.

Our callback takes both values by value. That is convenient for small integers and our small record. For large records, it copies values for each comparison. A separate design could accept pointers in its callback, but it would need a clear read-only-by-contract policy and careful treatment of the wanted value's lifetime. Choosing that interface is an engineering decision, not something specialization guesses for us.

You can apply the same idea to data definitions. The following optional declaration is independent of our search component; do not replace `SearchResult` with it. It describes a pair whose two fields always have the same chosen type:

```
struct Pair[T]
    first  T
    second T
```

`Pair[Int32]` is a complete concrete type; `Pair[Reading]` is another. Type arguments are types, never runtime counts. Generic functions and structs are supported; generic aliases and unions are outside this edition. When a bare parameter `T` is replaced by an array type, its value semantics preserve the array rather than silently decaying it.

Check the Assembled Search Component

The shared region now contains `SearchResult` and `findFirst`; the integer region contains `sameInteger` and `integerSearch`; the record region contains `Reading`, `sameCode`, and `readingSearch`. Remove the old `findFirstInteger` when making the replacement. Calling `integerSearch()` returns index zero; calling `readingSearch()` returns index one. Each result is an index, so neither function returns a view into its local array.

For an empty-input check, change only the call in `integerSearch` to pass `values[:0]`. The result has `found = false`; do not use its index to claim that element zero matched. Restoring `values[:]` restores the original search. We can now vary type, comparison, and searched range independently while keeping one loop to review.

See the Component Produce a Result

The component has no output of its own. To observe it as an application, keep its declarations in a separate example directory and give the file a source banner titled Generic Search. Add `import Core` and `from IO import print` to the inventory at the top of the component file.

```
import Core

from IO import print
```

Append this entry-point section after the component's final function. It calls the lesson consumer, prints its result, and checks output failure. Keep only the final versions of declarations that the lesson replaced:

```
# ____________________________________________________________________________

# [ Application Entrypoint ]
# ____________________________________________________________________________

func start() -> Core.Status

    print("""
        ________________________________________

         Generic Search
        ________________________________________
        """)?

    integer = integerSearch()
    reading = readingSearch()

    print("Integer: found [ {integer.found} ]; index [ {integer.index} ]\n")?
    print("Reading: found [ {reading.found} ]; index [ {reading.index} ]\n")?

    return Core.ok()
```

After the banner, successful execution is expected to report:

```
Integer: found [ true ]; index [ 0 ]
Reading: found [ true ]; index [ 1 ]
```

An unsuccessful search still needs its `found` check before indexing. This caller prints the result fields without using an unsuccessful index as an element position.

Exercises and Discussion

1. Change the algorithm to find the last match. What extra state is necessary?

Initialize a result before the loop, replace it at each match, and return it after the loop. Here is the resulting module-level function, using the existing `SearchResult` declaration and callback contract:

```
func findLast[T](values [:]T, wanted T, equal func(T, T) -> Bool) -> SearchResult

    result = SearchResult(found = false, index = 0)

    for index, item in values
        if equal(item.*, wanted)
            result = SearchResult(found = true, index = index)

    return result
```

For `[12, 7, 12, 9]` searching for 12, the saved index first becomes zero, then two. No return inside the loop stops the second update. Empty input leaves the initialized result untouched. This change trades early exit for the information needed to choose the last match.

2. Call the integer specialization with `wanted` stored in an `Int64` variable. Will the expected result type fix the disagreement?

No. A supplied `[Int32]` requires an `Int32` argument under the ordinary argument rules; deduction without explicit arguments cannot make incompatible occurrences of `T` agree. Convert deliberately if the narrowing/wrapping semantics fit the application. The expected `SearchResult` does not supply type-deduction evidence.

3. Take the search function's address. Is `&findFirst` enough when the destination has a function-pointer type?

Name the full specialization, such as `&findFirst[Int32]`. Expected function-pointer context does not infer omitted generic arguments at a function-address expression. This differs from inference at an ordinary generic call.

4. Why should `sameCode` avoid changing the `readings` array indirectly?

The loop's simple first-match meaning assumes the sequence remains usable and its comparison policy stable during the call. A callback could reach mutable state through other addresses, so its type alone does not enforce this. Our stated callback contract rules that behavior out. Write down such obligations when you design reusable algorithms; generic syntax does not replace an interface contract.

The next lesson moves a different boundary: deciding when computation runs and when source structure is selected. A generic changes types. Required evaluation computes values, and a macro substitutes syntax. Keeping those jobs separate makes all three easier to reason about.

On Your Own

Add `countMatches[T]` beside `findFirst`, using the same view, wanted-value, and comparator contract. It must return a `UintSize` count and visit the entire slice. Keep the search functions available; this operation answers a different question.

For `[12, 7, 12, 9]`, matching 12 must return two, matching 8 must return zero, and an empty slice must return zero without calling the comparator. For the lesson's readings, keep `sameCode` and add a second policy requiring both code and score to match. With wanted code 70 and score zero, the first policy must count one and the second zero. With wanted code 70 and score three, both must count one.

Do not change stored readings or introduce record-wide equality. Use the two policies to demonstrate that the reusable walk and the meaning of a match remain separate decisions.

Further Reading

ec Technical Manual: Generics.

Deciding Work Before the Program Runs

A report format needs storage for ten entries. The number is derived from a small rule: rows one through four contribute one, two, three and four entries. We could write `10` and explain it in a comment, but then a changed format might leave the count and explanation disagreeing. We could calculate the count every time the report runs, but an array type needs its count before execution.

ec can evaluate ordinary functions during translation when a context requires a value then. This is different from hoping an optimizer removes a calculation. It is a language obligation: produce a suitable value during translation or reject the program. Later in this lesson we will select target-dependent source and examine macros. All three happen around translation, but they solve different problems.

Compute the Size Where It Is Defined

Our finished component will build ten report entries and return their total. It will also give us a place to try an optional host operation and compare a macro with an ordinary function. Work in one module. Start with this ordinary helper, using only the loop and accumulator techniques from earlier lessons:

```
func triangular(count UintSize) -> UintSize

    total UintSize = 0
    remaining = count

    loop remaining > 0
        total += remaining
        remaining--

    return total
```

We could call this helper with a runtime argument. For the report's fixed shape, instead insert this typed module constant above the helper, in the file's inventory. Its small Report Size heading identifies the group of settings that determine the report's shape:

```
____
____ [ Report Size ]

ENTRY_COUNT const UintSize = triangular(4)
```

Module declarations are order-independent, so this placement is legal. Leave two blank lines between the constant and the following helper. The inventory shows the owned size immediately; the calculation stays in a named helper that we can follow separately. When we add imports, this heading will follow the last import directly, keeping the inventory compact.

For the argument four, the loop adds four, three, two and one. The total progresses through 4, 7, 9 and 10; the next test sees zero remaining and returns ten. The module constant's initializer is a required-evaluation context, so this work happens during translation. There is no special annotation on `triangular`: it is an ordinary function that could also be called with a runtime argument.

Now add a complete consumer after `triangular`. Its first loop initializes the entries with 1 through 10; its second loop totals them:

```
func reportTotal() -> Int32

    entries [ ENTRY_COUNT ]Int32

    next Int32 = 1

    for entry in entries
        entry.* = next
        next++

    total Int32 = 0

    for entry in entries
        total += entry.*

    return total
```

The array count is ten before the function runs. Its local integers are not initialized merely by reserving the array, so the writing loop comes before the reading loop. After the first loop, the entries are 1, 2, 3, through 10. The totals after the first three reads are 1, 3, and 6; after the last read the function returns 55.

Keep these two loops together. They are two visible phases of one small report operation, not two reusable concepts needing separate helpers. The blank lines separate preparing storage, filling it, and consuming it without adding a heading to every few statements.

The count is small enough for the arithmetic and evaluation budget. For arbitrary large arguments, `UintSize` arithmetic wraps and evaluation takes one iteration per unit; the function is not an unbounded mathematical integer formula. A production format calculator should state and check its admitted range. The example's input is exactly four.

Constant Binding and Required Computation

`const` attached to a binding says it cannot be reassigned. A local constant may receive a value computed at runtime. Conversely, `const(expression)` can force a computation during translation without making the receiving variable immutable. To try that distinction in our component, temporarily insert the next two statements at the start of `reportTotal`. They do not replace `ENTRY_COUNT` or resize `entries`:

```
    count = const(triangular(3))
    count += 1
```

The local `count` becomes seven; the report still contains ten entries and returns 55. Remove these two experimental statements before continuing. The word appears in two different grammatical positions, and each position matters. If `triangular` instead received a runtime parameter inside `const(...)`, required evaluation could not reach that runtime value. It would reject the request rather than postpone it.

Keep two questions separate: **when must the initial value be computed**, and **may this binding change afterward**? Assume `requested` is a function parameter supplied at runtime:

Declaration	Initial value	May the binding be reassigned?
module-level `ENTRY_COUNT const UintSize = triangular(4)`	required during translation	no
local `count = const(triangular(3))`	required during translation	yes
local `limit const UintSize = requested`	supplied by the runtime call	no
local `count = triangular(requested)`	computed from the runtime argument	yes

The local `limit` row is the important contrast. Its value cannot change after initialization, yet the caller may choose a different value on each call. Immutability alone does not make an unknown runtime input available during translation.

Required evaluation follows the selected target's integer widths, layouts and arithmetic. It does not quietly use the translating machine's pointer width. It can manipulate its own locals and call ordinary ec functions, but it cannot print, read a clock, run foreign code, access a device, or perform thread and atomic operations. Declared input files have their own explicit mechanism. An accidental printing call inside an otherwise suitable helper makes that evaluation fail.

The evaluator also has bounded steps, call depth and live storage. A nonterminating loop reaches an evaluation limit; it does not become a runtime loop. Returning an address into a dead evaluator local is not made valid by doing the work early. **Materialization** is the step that turns an evaluation's result into data the program can keep. For `ENTRY_COUNT`, it keeps the integer ten; the helper's temporary loop variables are not part of that result. Pointers and views need additional rules because keeping an address alone cannot keep dead storage alive.

Select Source for the Target

Suppose filling the report is one phase in a larger operation and we want to offer the host a chance to run another thread before totaling it. This is called yielding: the host's scheduler decides which thread runs next. `System.yield` requires the `blockingWait` capability, a named feature that the selected profile either supplies or lacks. We want the same helper name to work on profiles with and without that feature. A runtime condition cannot make an unsupported operation legal: both runtime branches are part of the selected program.

Add the two imports above `ENTRY_COUNT` and add `yieldIfAvailable` between `triangular` and `reportTotal`. This translation-time choice keeps the component usable with any current profile by excluding the unsupported call where the capability is absent:

```
import Core
import System


func yieldIfAvailable()
    const if Core.hasCapability("blockingWait")
        System.yield()
    else
        ;
```

In `reportTotal`, insert `yieldIfAvailable()` after the filling loop and before `total Int32 = 0`, with a blank line on each side. The answer stays 55. On `arm64-macos`, the selected body calls `System.yield`. On the browser and

current freestanding profiles, it has an empty body at that point. The function does not promise a delay, fair scheduling or a browser yielding interface. Its fallback is deliberately no operation.

The selected branch's items enter the surrounding scope; `const if` does not add a hidden lexical scope. Unselected branches still need valid tokens, balanced delimiters and correct indentation, but their names and operations are not resolved or translated. This is why target-specific source can be present without demanding that every target support it. Conditions depending on generic type parameters are selected separately for each specialization.

A Macro Receives Syntax

An ordinary function call evaluates its arguments once and passes values. A macro receives source components and substitutes them into a template. The difference is easy to miss when both are written with parentheses.

The report now needs twice each generated entry: 2, 4, through 20. Before choosing an implementation, compare two ways to express doubling. Add the following macro and helpers before `reportTotal`; they use no additional imports. The macro is deliberately small enough to inspect every use of its argument.

```
macro twice(value expr) -> expr
    value + value


func nextValue(counter *Int32) -> Int32

    counter.*++

    return counter.*


func doubled(value Int32) -> Int32
    return value + value
```

Temporarily insert these statements at the beginning of `reportTotal`, before `entries`. They are an experiment; the existing report loops do not use these bindings:

```
    counter Int32 = 0
    expanded = twice(nextValue(&counter))

    counter = 0
    called = doubled(nextValue(&counter))
```

In the declaration, `value expr` means that the argument supplies expression syntax; `-> expr` means the replacement is also an expression. These are syntax categories, not value types such as `Int32`. The expanded form of the first assignment is:

```
    expanded = nextValue(&counter) + nextValue(&counter)
```

Expansion writes that structure during translation. The two counter updates occur later, when the containing function executes. A macro is therefore not automatically a request to compute its argument during translation.

Left-to-right evaluation calls `nextValue` once to obtain one, then again to obtain two. `expanded` is three, and the counter reaches two. After the reset, the ordinary function call evaluates its argument once, obtaining one. `doubled` adds that value to itself, so `called` is two and the counter is one.

Structured substitution preserves expression grouping: `twice(a + b)` means two copies of the whole addition. It does not preserve single evaluation. **Name hygiene** solves another problem: a temporary named by a structured template does not accidentally replace or refer to a caller's local with the same spelling. That protects names, not evaluation count. Our macro can be hygienic and still call `nextValue` twice.

Remove those four experimental statements after following the trace. For the report, use the function. A macro is justified when an interface genuinely needs a source category such as a declaration, a type expression or a supplied statement body. A block macro's `return`, propagation, `break`, `continue` and `defer` participate in the caller's control flow and scopes. It does not automatically create a function-call boundary. Structured template-local names are fresh; argument names retain the caller's meaning. Text macros have different binding and precedence rules and deserve a separate explicit reason to use them.

Finish the Report with One Evaluation per Entry

Replace the entire `reportTotal` declaration with this final version. Keep `ENTRY_COUNT`, `triangular`, `yieldIfAvailable`, `nextValue`, and `doubled`; the experimental `twice` macro is no longer needed by the component.

```
func reportTotal() -> Int32

    entries [ ENTRY_COUNT ]Int32

    counter Int32 = 0

    for entry in entries
        entry.* = doubled(nextValue(&counter))

    yieldIfAvailable()

    total Int32 = 0

    for entry in entries
        total += entry.*

    return total
```

At the first iteration, `nextValue` changes the counter from zero to one and `doubled` produces two. The second produces four. At the tenth, the counter becomes ten and the entry becomes twenty. The final loop returns 110. The array's shape was chosen during translation; its entries and total are produced when `reportTotal()` is called.

This is the assembled outcome. Imports come first, then the count constant, then the helpers before their consumer. There is one `reportTotal`, using the last replacement. All storage is local, no pointer escapes, and the optional yield neither changes the entries nor supplies their initialization. Keeping the generated value as one function argument makes its single evaluation visible without creating a temporary helper solely for layout.

If we had left `twice(nextValue(&counter))` in the filling loop, the entries would instead begin 3, 7, and 11: each iteration would advance the counter twice. A syntactic shortcut would have changed the report. The small experiment established the design choice before we incorporated it into the finished operation.

See the Component Produce a Result

The component has no output of its own. To observe it as an application, keep its declarations in a separate example directory and give the file a source banner titled Report Entries. Keep the existing Core and System imports and add `from IO import print` after them. The finished inventory is:

```
import Core
import System

from IO import print
```

Append this entry-point section after the component's final function. It calls the lesson consumer, prints its result, and checks output failure. Keep only the final versions of declarations that the lesson replaced:

```
________________________________________________________________________________

[ Application Entrypoint ]
________________________________________________________________________________

func start() -> Core.Status

    print("""
        ________________________________________

         Report Entries
        ________________________________________
        """)?

    total = reportTotal()
    print("Total: [ {total} ]\n")?

    return Core.ok()
```

After the banner, successful execution is expected to report:

```
Total: [ 110 ]
```

Exercises and Discussion

1. What happens if `ENTRY_COUNT` becomes `triangular(0)`? The loop executes zero times, returns zero and forms a zero-length array. There is no element zero to initialize or read. The loop's initial `total` makes the empty case intentional.

2. Would an ordinary `if Core.hasCapability("blockingWait")` be an adequate substitute in `yieldIfAvailable`? No. It tests a value at runtime while leaving both branches subject to ordinary checking. Use `const if` to exclude the unsupported operation from the selected program.

3. Make the macro call evaluate `nextValue` once without changing the macro. First assign `value = nextValue(&counter)`, then call `twice(value)`. This duplicates a harmless read of the saved integer. It also makes the evaluation decision obvious to the reader. For this particular operation, the ordinary function is still the clearer interface.

4. Can a required evaluation call `nextValue`? Yes, if its pointer addresses a counter created within that evaluation. Mutating evaluator-local storage is permitted. Passing a pointer to an ordinary runtime local is not. The operation's effect and the storage it reaches matter more than the function's name.

The next chapter returns to runtime work that genuinely overlaps. Its central question is again about boundaries: when may one thread observe another thread's writes, and who keeps the shared storage alive?

On Your Own

Make the report's row count a typed module constant named `ROW_COUNT`, and derive `ENTRY_COUNT` from it. Keep the final single-evaluation filling loop and optional yield. For this exercise the admitted row count is zero through 16; no runtime input or allocation is needed.

Follow the complete report for three configurations: four rows must still produce ten entries and return 110; six rows must produce 21 entries and return 462; zero rows must return zero without calling `nextValue`. Explain which work changes during translation and which runs on each call. Keep the related count constants together in the inventory so changing the format does not require searching through the implementation.

Hint: a zero-length array makes both collection loops empty. Check the initialized accumulator before adding a special branch.

Further Reading

ec Technical Manual: Compile-Time Evaluation; Macros.

Sharing Work Without Losing Ownership

A long list can be summed in two halves. One thread handles the first half while the original thread handles the second. The arithmetic is familiar; the difficult questions are about time. When is the worker's answer visible? Can the original function return while the worker still uses a local object? What happens if waiting for the worker is interrupted?

A **thread** is a separate sequence of function calls and statements within the same program. Each has its own current call stack, but both can reach the same memory through pointers. Their steps may alternate or run on different processor cores. Reading the worker's next source line tells us what it does next, not when it happens relative to the caller. We need explicit ordering where one thread depends on the other's work.

This chapter uses a small parallel sum to answer those questions. We build a reusable component for `arm64-macos`, the current profile with the `threads` capability. It is not browser code, and it does not promise that two threads are faster than one. Thread creation has a cost; a useful speed comparison would require a real workload and measurement. Here the goal is a correct protocol whose storage and synchronization can be followed on paper.

First Make the Arithmetic Work

We will build `sumConcurrently`, a reusable operation returning either an `Int64` total or a thread-creation error. Begin in one module with the sequential helper below. It needs no imports yet:

```
func sumValues(values [:]Int32) -> Int64

    total Int64 = 0

    for value in values
        total += Int64(value.*)

    return total
```

For `[2, 4, 6, 8]`, the accumulator visits 2, 6, 12, and 20. Empty input returns zero because `total` is initialized before the loop. Each element is widened before addition. The total still uses wrapping `Int64` arithmetic; we have not promised an unbounded mathematical sum.

This helper earns its name because both halves will use exactly the same arithmetic. We will keep splitting, starting, and finishing the thread together in a separate consumer so its lifetime can be read from top to bottom.

Give the Worker a Borrowed Job

Add the following imports at the top of the file. Insert the type declarations before `sumValues`:

```
import Core
import System

struct SumJob
```

Listing continued

```
    values [:]Int32
    total  Int64

type SumJobPointer *SumJob
```

`values` describes the input; `total` is the answer slot. The job does not own the slice's backing array. Our caller must keep that array alive and unmodified until the operation returns. Other readers are harmless; unsynchronized writers would break the contract.

The thread API accepts a callback with a `*Void` context. Add this adapter after `sumValues`:

```
func sumWorker(context *Void)
    job = SumJobPointer(context)
    job.total = sumValues(job.values)
```

The alias makes the conversion back to the agreed job type explicit. It changes the pointer type, not the object's lifetime, alignment, or initialization. We will pass the address of a live `SumJob`, and the worker will write its answer once. For now, the caller will wait for the worker to finish before reading that answer.

Establish the Return Boundary Before Starting

Starting a thread gives the caller a **handle**, a record identifying the host thread and carrying the obligation to join or detach it. This is separate from the `SumJob` containing our input and answer. A successful join supplies a completion boundary. It waits for completion, makes the worker's effects happen-before the return from joining, and consumes the owning thread descriptor: the descriptor becomes closed and cannot be used to join again.

Joining can return `INTERRUPTED` while retaining the handle. A local job cannot be abandoned on that path. Add the following helper after `sumWorker`. It retries interruption and treats any other error as a broken invariant.

Pass the thread handle this caller owns while it is still open, for a thread other than the one making this call. Keep that handle under this caller's sole control throughout the wait: no other code may join, detach or otherwise use it at the same time. With those conditions met, an interrupted wait is the recoverable error to retry.

```
func joinWorker(thread *System.Thread)
    loop
        joined = System.joinThread(thread)
        if joined.ok
            return

        Core.assert(joined.error == Core.INTERRUPTED, "unexpected join failure")
```

This helper does not return while the thread is live. Repeated interruptions can keep it waiting; the language does not promise fair scheduling or bounded completion. A failed assertion ends the program rather than returning past a borrowed live job. We avoid `defer System.joinThread(...)` here because discarding an interrupted join at function exit would not establish the required lifetime boundary.

The assertion uses the defined `Core.Error` comparison by domain and code, so `Core.INTERRUPTED` is compared as an error value rather than by an invented host error number.

Assemble the First Threaded Version

Add the following consumer after `joinWorker`. This version uses completion as its only synchronization boundary:

```
func sumConcurrently(values [:]Int32) -> Core.Result[Int64]

    middle = values.len / 2
    job = SumJob(values = values[:middle], total = 0)

    thread = System.spawnThread(&sumWorker, Core.VoidPointer(&job))?
    localTotal = sumValues(values[middle:])
    joinWorker(&thread)

    return Core.success(localTotal + job.total)
```

Follow `[2, 4, 6, 8]`. The split index is two, so the job borrows `[2, 4]` and the caller sums `[6, 8]`. A successful spawn orders the initialized job before the worker starts. The worker computes six; the caller computes fourteen. Their relative speed does not matter: `joinWorker` completes before the caller reads `job.total`, and success contains twenty.

The `?` is safe at spawn: failure creates no worker that could retain the local address. After a successful spawn, every path goes through the join helper before returning. Keeping those steps adjacent makes the storage proof visible. Replacing join with an unchecked deferred call would hide the interrupted-wait case at precisely the point where returning becomes dangerous.

Add this small consumer after `sumConcurrently` if you want a fixed input to follow through every revision:

```
func sumExample() -> Core.Result[Int64]

    values = [2, 4, 6, 8]

    return sumConcurrently(values[:])
```

The array remains alive throughout the call, including the worker's join. The result returns a number or error, never a view into that array. At this stage the component is already useful. If all you need is the final sum, keep this version.

Extend the Job to Publish Before Completion

Now add a different requirement: the caller should be able to ask whether the answer is available before joining. A worker can have written its answer while still finishing its return. We need to distinguish “may read the answer” from “may release the worker's storage.”

An **atomic** load or store accesses its value indivisibly: another thread cannot observe half of that operation. Atomicity alone is not enough to order an ordinary answer field. We will use a flag with release/acquire ordering to establish that relationship.

Make the next job, worker, and consumer replacements as one revision; call the example again after all three are in place. First replace the earlier `SumJob` declaration with this one; keep the pointer alias:

```
struct SumJob
    values [:]Int32
    total  Int64
    ready  Core.Atomic[Bool]
```

Replace `sumWorker`, then insert `readPublished` immediately after it:

```
func sumWorker(context *Void)
    job = SumJobPointer(context)
    job.total = sumValues(job.values)
    Core.atomicStore(&job.ready, true, Core.MemoryOrder.Release)


func readPublished(job *SumJob, total *Int64) -> Bool

    if !!Core.atomicLoad(&job.ready, Core.MemoryOrder.Acquire)
        return false

    total.* = job.total

    return true
```

The release store follows the ordinary total write. An acquire load that reads that published `true` synchronizes with the release, so its subsequent ordinary read can safely use the answer. A false load returns without touching the output. The worker never changes the total after publication.

Happens-before names the ordering that gives us permission to depend on an earlier write, including across threads. Merely waiting some time establishes no such relationship. A **data race** is a conflicting access, including a write, without the ordering or atomic-access conditions required by the memory model. Seeing a plausible number does not repair one.

Changing both orders to `Relaxed` would keep indivisible flag access but remove the ordering needed by the ordinary total read. Moving the store before the total assignment would put the write on the wrong side of the release. The short callback stays expanded so those effects can be followed in their required order.

Replace the Consumer and Follow Both Schedules

Finally replace the whole earlier `sumConcurrently` declaration with the following version. Retain `sumValues`, the replacement job and worker, `readPublished`, `joinWorker`, and the unchanged `sumExample`:

```
func sumConcurrently(values [:]Int32) -> Core.Result[Int64]

    middle = values.len / 2
    job = SumJob(values = values[:middle], total = 0, ready = Core.Atomic[Bool](false))

    thread = System.spawnThread(&sumWorker, Core.VoidPointer(&job))?
    localTotal = sumValues(values[middle:])

    workerTotal Int64 = 0
    published = readPublished(&job, &workerTotal)
    joinWorker(&thread)

    if !!published
        workerTotal = job.total

    return Core.success(localTotal + workerTotal)
```

The `SumJob` is initialized directly in its local storage. The explicit `Core.Atomic[Bool](false)` initializes the flag in that same storage; it does not copy an atomic value. The job contains an atomic, so it is not copied into the new thread or returned by value. The spawn receives only its address, explicitly converted to `Core.VoidPointer`. Successful thread creation orders the already completed initialization before the worker begins; the worker therefore sees its slice, zero total and false flag.

The original thread sums the second half while the worker sums the first. Afterward it makes one readiness probe. If the probe succeeds, acquire ordering permits reading the answer even before the worker has fully returned. If it fails, the caller still joins and then reads the answer through join's completion guarantee. Both paths retain `job` and the caller's backing array through the join.

For `[2, 4, 6, 8]`, `middle` is two. The worker computes six and the original thread computes fourteen, so success contains twenty. Scheduling changes whether the readiness probe succeeds, but not the successful sum. An empty input gives two empty slices and two zero totals. The example uses wrapping `Int64` addition; it does not turn an arbitrarily large sum into checked mathematical arithmetic.

Follow the two possible probe results separately:

Point in the caller	Worker has published before the probe	Probe reads false
`readPublished` returns	copies six into `workerTotal`; returns true	leaves `workerTotal` at zero; returns false
`joinWorker` returns	worker has ended, even if it was paused after publishing	worker has completed the sum, published and ended
`if !!published`	skips the body because six was already read safely	reads six from `job.total`, now safe after join
final addition	fourteen plus six	fourteen plus six

There are two different uses of “ready” in everyday speech. The answer can be ready to read while the worker still needs to finish returning. Our atomic flag answers the first question; joining answers the second. The job's local storage survives both questions because `sumConcurrently` does not return between them.

The probe is included to expose the distinction between publication and completion. A production routine that needs only the final sum can omit the flag and probe, join once with the same interruption handling, and then read `job.total`. The simpler protocol uses join for both visibility and lifetime. A routine that displays intermediate progress could have a reason to retain publication separately.

There is no implied thread pool, copied slice backing or automatic cleanup on return.

The final assembly contains one declaration of each helper and only the replacement `SumJob`, `sumWorker`, and `sumConcurrently`. The first version and this extension are alternatives, not two overload sets to paste together. Calling the unchanged `sumExample()` exercises the completed protocol and returns success containing twenty when spawning succeeds.

See the Component Produce a Result

The component has no output of its own. To observe it as an application, keep its declarations in a separate example directory and give the file a source banner titled Concurrent Sum. Keep the existing Core and System imports and add `from IO import print` after them. The finished inventory is:

```
import Core
import System

from IO import print
```

Append this entry-point section after the component's final function. It calls the lesson consumer, prints its result, and checks output failure. Keep only the final versions of declarations that the lesson replaced:

```
// ------------------------------------------------------------------------

// [ Application Entrypoint ]
// ------------------------------------------------------------------------

func start() -> Core.Status

    print("""
        ----------------------------------------

         Concurrent Sum
        ----------------------------------------
        """)?

    total = sumExample()?
    print("Total: [ {total} ]\n")?

    return Core.ok()
```

After the banner, successful execution is expected to report:

```
Total: [ 20 ]
```

Use `arm64-macos` for this caller. A spawn failure returns before the total line; a successful spawn reaches that line only after joining. The value alone does not tell you which readiness-probe path occurred.

Exercises and Discussion

1. Replace `joinWorker` with `System.joinThread(&thread)?`. Why is the seemingly ordinary error path unsafe here?

If the join is interrupted, propagation returns from `sumConcurrently` while the worker may still use `job`, whose local storage then ends. The caller also retains an unconsumed thread handle. Error propagation is correct only when the resources and borrowed storage on that path remain valid; it is not an automatic lifetime solution.

2. Can detaching the thread repair that error path?

No. Detaching consumes the handle and arranges release of the thread's host resources when it ends, but does not wait and does not own or copy `job`. A detached design needs independently managed context and input storage, with an explicit agreement about who releases them after the worker's last use.

3. What if the worker updates `total` repeatedly after setting `ready`?

Our one-publication proof no longer applies. The reader could race with a later ordinary write. A stream of updates needs a new protocol, such as a mutex protecting the whole shared state, or an appropriate atomic representation and update scheme. Reusing the same flag does not automatically acknowledge consumption.

4. Should the browser version silently run this thread code on the page?

No. The browser profile has no threads or atomics. A project can select a separate sequential implementation with `const if`, as taught in the previous chapter. Merely importing a module is different from actively using an unavailable facility, but an active threaded implementation cannot acquire browser services that the profile does not define.

Sharing work safely meant making a small boundary precise: the pointer passed to another execution, the synchronization that permits a read, and the completion that permits release. The next chapter applies the same discipline to a different boundary: code and devices outside ordinary ec functions.

On Your Own

Give `sumConcurrently` a sequential path for inputs shorter than eight elements. On that path it must return the result of `sumValues` as a success without creating a thread. For eight or more elements, retain the complete publication and join protocol from the final version. Keep the lesson's `arm64-macos` profile and input-lifetime contract.

An empty input must return zero. The integers one through seven must return 28 through the sequential path; one through eight must return 36 through the threaded path when spawning succeeds. A spawn failure on the threaded path must propagate, and a successful spawn must still be joined before any return. Mark the split between the two paths clearly enough that these obligations can be checked by reading the function.

Eight is this exercise's chosen boundary, not a measured performance threshold. Explain what measurements would be needed before choosing a threshold for a real workload.

Further Reading

ec Technical Manual: Concurrency; System.

Crossing the Boundary to C and Devices

A useful program rarely owns every component it calls. A numerical routine may already exist in C; a kernel may need to write a device register; a small machine operation may have no ordinary source spelling. At each boundary, the missing information becomes our responsibility. A declaration tells ec what an external object looks like, but cannot prove that the external object actually follows that declaration.

Our goal is a small sample-reporting component: ask an existing C routine for the largest sample, then offer that value to a device with a bounded number of status reads. As with the worker's context in the previous chapter, the values, lifetimes, and effects must be explicit. We will build the C wrapper first, then the device writer, then join them. Assembly and variadic calls will be optional investigations after that working design, not required pieces to paste into it.

This lesson is a contract-and-trace exercise unless your environment supplies the stated C library and mapped device registers. Do not run the device code with guessed addresses. Unlike the preceding lessons, its expected effects come from explicit external assumptions. You can complete the exercise on paper by recording the foreign-call count, status reads and data writes; the trace below supplies all the inputs needed for that work.

Start at the Supplied C Boundary

Assume an application supplies a C library built for the selected `arm64-macos` profile. Its header, after including the C headers that define `int32_t`, `uint32_t` and `size_t`, contains these declarations:

```
int32_t sample_max(const int32_t *values, size_t count);
extern uint32_t sample_calls;
```

You do not need to know C to follow this interface. `int32_t` and `uint32_t` name signed and unsigned 32-bit integers. `size_t` is the C implementation's size type. The `*` marks a pointer, and the `const` before its element type says this function does not modify those elements through that parameter. The semicolons end C declarations. The second line declares a shared variable rather than a function.

For this example, the supplied library's contract is precise: `sample_max` requires at least one readable element, reads exactly within that array, returns its largest integer, retains no pointer, and does not modify the elements. It increments `sample_calls`. The application permits only one call or counter access at a time; the counter is not atomic. These are application-specific symbols, not standard ec APIs or services supplied by a profile.

Start an ec component in one module with these external declarations. They need no imports. The selected C implementation uses the stated fixed-width integer mappings and address-sized `size_t`. You are describing an already supplied library, not defining its implementation:

```
extern sampleCalls Uint32 symbol "sample_calls"


extern func sampleMaximum(values *Int32, count UintSize) -> Int32 symbol "sample_max"
```

The external declaration has no body and states its result explicitly. `extern` selects the C calling convention here. `symbol` names the C-level entity while allowing a normal camelCase ec name. The variable declaration

creates no second counter: it describes storage owned by the foreign definition, and therefore has no initializer in ec.

A **calling convention** is the machine-level agreement about how a call passes arguments and returns its answer. Together with data layouts and symbol naming, it forms the **application binary interface**, or ABI. Both sides must agree because the other side receives the machine representation, not the explanatory type names in this chapter. `symbol "sample_max"` identifies which external function to call; it does not by itself establish that agreement.

Put the Empty Case on Our Side

Our ec caller already works with slices, which can be empty. The C routine forbids an empty input. Give that case a local meaning before exposing the wrapper to callers. Add this function after the external declaration:

```
func maximumOr(values [:]Int32, fallback Int32) -> Int32

    if values.len == 0
        return fallback

    return sampleMaximum(values.ptr, values.len)
```

The early return is the important boundary: it prevents an invalid foreign call rather than hoping C checks the count. We keep it on its own line so the no-call path is easy to see.

The wrapper accepts an ordinary slice. For a nonempty slice, it passes pointer and count separately because a slice descriptor is not a C argument by value. For an empty slice, it returns the fallback without making the foreign call. Given `[4, 9, 2]` and fallback zero, the result is nine under the supplied C contract. Given an empty slice, the result is zero and `sample_calls` does not change.

The C `const` qualifier is part of the foreign promise that the data is not modified; ec does not invent a new pointer qualification conversion for it. An ec declaration must agree with actual widths, signedness, layout, calling convention and symbol identity. A wrong declaration is an invalid call; ec does not automatically translate an incorrectly described memory representation into the right one.

Add a fixed consumer below `maximumOr` to follow the first assembled version:

```
func maximumExample() -> Int32

    values = [4, 9, 2]

    return maximumOr(values[:], 0)
```

Under the supplied contract, `maximumExample()` returns nine and increments the foreign counter once. Temporarily change the call to `maximumOr(values[:0], 0)`: the result becomes zero and the foreign counter is not touched. Restore the full slice before continuing. The local array survives the synchronous call, and the C routine's no-retention promise means there is no later use to support.

Add One Device Write

A **device register** is a location through which the program observes or commands a device. It is not ordinary storage for a local variable. Reading it may ask hardware for status; writing it may start an action. The platform must first make its addresses accessible, which is what mapped means here.

For the second half of the same component, suppose the host has already mapped two aligned 32-bit device registers into this native process and the caller supplies their addresses. The device contract permits repeated status reads; bit zero is one while busy. A zero bit permits one whole data-register write, and this particular interface requires no additional barrier between that status read and write. The caller excludes competing writers. This is an explicit hypothetical device contract, not a claim about a named machine.

Start with one observation. Add this first version of `tryWrite` after `maximumExample`:

```
func tryWrite(status *device Uint32, data *device Uint32, value Uint32) -> Bool

    if (status.* & 1) != 0
        return false

    data.* = value

    return true
```

A busy status returns false without writing; a clear busy bit permits exactly one write. The early exit keeps the successful path at the outer indentation. Each source access matters here: we must not read the data register just to prepare a whole-value write.

Give the Writer a Transaction Budget

One observation may be too soon. We want a small number of retries, but an endless wait would give the caller no way to regain control. Replace the entire first `tryWrite` declaration with this version, which adds an explicit attempt budget:

```
func tryWrite(status *device Uint32, data *device Uint32, value Uint32, attempts UintSize) -> Bool

    remaining = attempts

    loop remaining > 0
        if (status.* & 1) == 0
            data.* = value
            return true

        remaining--

    return false
```

Each test performs one 32-bit device read. The mask `& 1` keeps bit zero and discards the other status bits. A status word can therefore be nonzero while the busy bit is clear. With three allowed attempts, consider these successive readings:

Attempts remaining before the read	**Status word**	`status & 1`	**Action**
3	5 (binary 101)	1	busy; reduce remaining to two
2	3 (binary 011)	1	busy; reduce remaining to one
1	2 (binary 010)	0	write `value` once and return true

The last test succeeds even though the status word is two rather than zero. With three busy responses the function instead returns false without a data write. Zero attempts performs no transaction.

The attempt count bounds transactions, not wall-clock time. Real device contracts determine access latency and any required ordering. Writing a whole `Uint32` admits every register bit pattern and avoids the read that a compound update or bitfield assignment would introduce. A write-only register cannot safely be updated by reading its old bits first.

The pointers do not map memory. `*device` distinguishes device transactions from ordinary memory access, and atomic fences do not substitute for a required device fence.

Connect the Two Halves

Add this consumer after the replacement `tryWrite`. Keep `maximumOr` and its external declarations. The zero-attempt exit comes before the foreign call as well as before any device access:

```
func sendMaximum(values   [:]Int32,
                 fallback Int32,
                 status   *device Uint32,
                 data     *device Uint32,
                 attempts UintSize) -> Bool

    if attempts == 0
        return false

    maximum = maximumOr(values, fallback)

    return tryWrite(status, data, Uint32(maximum), attempts)
```

The result says whether the device write happened. It does not say whether the C routine ran: for nonempty input and a positive budget, C runs before polling, even if every poll reports busy. Empty input uses the fallback without C. The explicit integer conversion encodes the signed maximum into 32 bits; negative values convert modulo 2^{32}. Our hypothetical data register accepts all bit patterns. A real protocol with a narrower value range would need a check before this conversion.

Finish with a consumer whose device addresses are parameters. No invented mapping service is required:

```
func sendExample(status *device Uint32, data *device Uint32) -> Bool

    values = [4, 9, 2]

    return sendMaximum(values[:], 0, status, data, 3)
```

Follow the whole call using the status sequence 5, 3, 2 from the table. C reads the three samples, returns nine, and increments its counter once. The writer observes two busy responses, then a clear busy bit, writes nine once, and returns true. Three busy responses instead return false after the same single C call and no data write. Passing an attempt budget of zero to `sendMaximum` performs neither foreign nor device work.

The assembled component has the external variable and function, `maximumOr`, the replacement `tryWrite`, and `sendMaximum`; `maximumExample` and `sendExample` are its two small lesson consumers. The caller supplies the library, mapped addresses, live sample storage, and exclusive access promised earlier. All activity completes within the calls. This is a usable component under those stated inputs, not a freestanding device driver or a claim that ordinary processes already possess these addresses.

We keep calculation and device polling as two helpers because they have different external contracts and can be used independently. We keep their order visible in `sendMaximum` because an error or retry policy depends on which effects have already happened.

Investigate Other Foreign Interfaces

The main component is finished. The following questions help you adapt it when a real library supplies a different header. They require no changes to the code above.

When the Header Uses Other Types

Do not replace a C type with a familiar ec type by eye. C `long` is 64 bits on the current native 64-bit profiles but 32 bits on `wasm32-browser`. An `extern type CLong scalar "long"` declaration asks for the profile's actual mapping. Other foreign scalars remain distinct types even when their storage width resembles an ec integer.

A struct crossing by value needs the C layout specified by `callconv C` or an appropriate binding description. An ordinary ec struct with similar field names is not automatically interchangeable. For an opaque C object, a pointer can be sufficient; learning its fields is unnecessary unless the interface requires their layout. A pointer still needs its separate valid-storage contract.

Callbacks cross in the opposite direction. An ec function supplied to C must have the C calling convention, with the exact expected parameter and result types. An ordinary ec-convention function pointer does not become compatible merely because its parameters look the same. Callback context and lifetime remain explicit; C retaining a pointer to a local after the local ends is still a lifetime error. Foreign allocation also keeps its own release obligation: use the foreign release operation that belongs to it, not an unrelated ec allocator.

On the browser profile, C interoperability covers code linked into the same WebAssembly module. It does not supply native dynamic loading, arbitrary host symbols or a JavaScript callback API. Selecting a profile is not equivalent to obtaining every library used by examples elsewhere.

When a Library Requests Variadic Arguments

A C declaration ending in `...` accepts additional arguments without a fixed type at each position. That flexibility moves type knowledge into the function's separate contract. A format string or count might describe the arguments, but the ec call cannot infer arbitrary foreign conventions from those values.

Small integer types and `Bool` are **promoted** to C `int` under the defined profile mappings: their values are passed in that larger integer type. `Real32` is promoted to `Real64`. For example, an extra argument holding `Int16(-3)` arrives as `Int32(-3)`, preserving the value but changing the argument type. The receiving ec C-variadic function must ask for `Core.vaArg[Int32]`, not `Core.vaArg[Int16]`. `Text`, slices and ordinary ec function pointers are not accepted as raw variadic values. A null pointer needs an explicit pointer type, and an array cannot be spread into a C variadic call.

When implementing a C variadic function in ec, a `CVaList` acts as a cursor through the extra arguments. `Core.vaStart` starts that cursor, `vaArg` reads and advances it, `vaCopy` makes an independent cursor at the current position, and `vaEnd` finishes one. Each started or independently copied list must be ended once. A list borrowed through a C `va_list` parameter is not an ordinary copy: the callee cannot retain or end that borrowed list, and after the call the caller's list permits only its required end operation. This lifecycle is a reason to prefer a typed slice interface when designing a new ec API.

An Optional Assembly Experiment

Keep this experiment separate from `sendMaximum`: copying a register would not improve that component. Processor registers are the small storage locations named by machine instructions. Inline assembly lets a function supply such instructions while describing how their values connect to ec expressions. For a native Arm64 profile whose documented assembler accepts the shown `mov`, this self-contained function fragment moves an integer through an output register. It illustrates operand declarations, not a performance improvement over `return value`.

```
func registerCopy(value Uint64) -> Uint64

    copied Uint64
    asm r"mov %{destination}, %{source}" (
        in  source      = value  register "gpr";
        out destination = copied register "gpr"
    )

    return copied
```

The `r` before the quote makes the assembly text a raw string: backslashes are kept as written instead of introducing text escapes. `%{source}` and `%{destination}` refer to the operand names below it; `"gpr"` asks for a general-purpose processor register rather than naming one particular register. `out` locates `copied` without reading an old value, so leaving it uninitialized before the assembly is appropriate. The input and output bindings expose the register effects. There is no memory item because the payload reads and writes no memory. If the result is unused, nothing here requires retaining the copy.

A call `registerCopy(9)` returns nine under the stated assembler contract, just as an ordinary return would. Inspecting the operands is the point of this experiment; do not insert this copy into the device path as an imagined optimization.

An inline assembly statement must declare every relevant register, flag and memory effect. `observable` is required for certain system operations and preserves the stated observable ordering; it is not a license for undeclared access. A memory clause is not a synchronization barrier. The payload cannot change the stack pointer or escape into another ec statement. Browser code cannot use inline assembly.

Owning the Whole Function

Now consider a different job: writing the entry and return sequence yourself. An inline statement sits inside a compiler-managed function, so it cannot take over that frame. Instead, replace the entire body with `asm` after the signature:

```
func nativeAdd(left Int64, right Int64) -> Int64 callconv C asm
    r"""
    add x0, x0, x1
    ret
    """
```

For this arm64 example, the ABI puts the arguments in x0 and x1. The first instruction adds them into x0; `ret` returns that register as the answer. An ordinary ec caller can write `nativeAdd(20, 22)` and receive 42. The parameter names describe the interface; the raw instructions use ABI registers directly. `callconv C` selects that machine agreement without introducing any C source.

There is an important exchange of responsibility. The compiler creates the function symbol but inserts no frame setup, register saves, stack checks or cleanup into its body. If you need a frame or call another function, you must preserve the return address and required registers yourself and restore the stack correctly before returning. The return type must be explicit, and this form is for named nongeneric functions with C-compatible signatures. It is not available in the browser or during compile-time evaluation.

Use the ordinary ec body for ordinary logic and inline assembly for a small machine operation within it. Use a complete assembly body when controlling the entry, stack and exit is the actual task. A wrong return sequence is a bug in your function even if the instructions assemble successfully.

Exercises and Discussion

1. Why does `maximumOr` check emptiness before calling C? The C routine requires at least one element. The wrapper gives empty input a defined local meaning and never asks the foreign side to accept a contract it did not promise.

2. Can two threads read and increment `sample_calls` as an ordinary variable? Not without the required synchronization. `extern` identifies storage; it does not make accesses atomic. Use the library's documented synchronization interface or a correctly synchronized calling policy.

3. Change `data.* = value` to `data.* |= value`. What changes? The update reads the device register and then writes it. It is not an atomic transaction pair and violates our intended whole-write interface if reads are forbidden. A compact expression can have very different external effects.

4. Does returning false after three busy reads prove the device is broken? No. It proves only that the permitted observations did not allow a write within this attempt budget. The caller decides whether to retry, report a timeout under a separate time policy, or choose another action.

5. Suppose the C library changes its count parameter from `size_t` to `uint32_t`, but the ec declaration keeps `UintSize` on `arm64-macos`. Is passing a small count enough to make the declaration correct? No. A count of three fits both types, but their declared widths differ on this profile. The ABI agreement concerns the parameter type, not only today's argument value. Update the ec declaration to match the supplied C interface, then handle any size conversion deliberately in the wrapper.

The advanced mechanisms now have distinct jobs: function values choose behavior, generics reuse type-dependent structure, required evaluation computes early, synchronization orders shared work, and foreign declarations describe external contracts. In the book's projects, use each only when the problem creates a real need for it.

On Your Own

Replace the Boolean result of the bounded `tryWrite` with a `WriteReport` containing `written Bool` and `reads UintSize`. Return the exact number of status transactions performed. Adapt `sendMaximum` and `sendExample` to return that record, preserving the early zero-budget exit before C is called.

A zero budget must report false and zero reads. A first status of two must report true and one read. Statuses 5, 3, 2 with budget three must report true and three reads; three busy responses must report false and three reads. Every successful report must correspond to exactly one whole data-register write, and every unsuccessful report to none. Keep the original foreign-call policy: positive-budget, nonempty input calls C once even when polling fails.

Do not add a status read merely to construct the report. Count the transactions already required by the algorithm; a diagnostic should not change the device interaction it describes.

Further Reading

ec Technical Manual: C Interoperability; Assembly and Devices.

A Number Laboratory

The earlier arithmetic chapters developed algorithms one at a time. We can now put several into an application with one coherent input policy. Our number laboratory accepts a decimal integer from zero through one million, classifies it as prime or composite when that distinction applies, computes its greatest common divisor with 360, and sums the positive integers up to it.

The program is deliberately bounded. One million is not a language limit; it is our workload limit. It keeps the repeated sum practical to inspect and prevents a user from accidentally requesting billions of iterations. Parsing, application validation and computation remain separate steps.

Read Errors and Cleanup and Modules and Projects before assembling the application. The source below is a complete program for a hosted profile with standard streams and process arguments. Save it as `numberLab.ec` in its own module directory.

Start with One Accepted Number

Create `numberLab.ec` in its own module directory. Begin with a file banner titled Number Laboratory, then `import Core`, `import Strings` and `from IO import print`. Keep the application-entry banner and put `start` last. The final listing below supplies this outer layout in full.

Before writing any calculation, settle the boundary. Put `MAXIMUM const Uint32 = 1_000_000` under an Input Bound inventory header after the imports. The entry will check the argument count before indexing, parse the text, then apply this application limit. Add these lines inside the checked entry, after its runtime banner:

```
    if args.len != 1
        return Core.INVALID_ARGUMENT

    number = Strings.parseInt[Uint32](args[0])?

    if number > MAXIMUM
        return Core.OUT_OF_RANGE
```

Use the signature `func start(args [:]Text) -> Core.Status`. During this first stage, finish the body with `print("Accepted: [ {number} ]\n")?` and `return Core.ok()`. A host-supplied argument `"49"` should be accepted; `"1000001"` should fail the limit check; two arguments should fail before either is parsed. This is a small useful program before any number theory is added.

Give Each Calculation Its Own Helper

Place the helpers before the entry banner in an Integer Algorithms section. First copy the greatest-common-divisor helper from the functions lesson:

```
func greatestCommonDivisor(a Uint32, b Uint32) -> Uint32

    loop b != 0
        remainder = a % b
        a = b
```

Listing continued

```
        b = remainder

    return a
```

For 49 and 360 it will return one. We keep that question separate from classifying 49: common divisibility and primality are different results, even when they use related arithmetic.

Next add the factor search from Repetition, with two blank lines after the first helper:

```
func smallFactor(number Uint32) -> Uint32

    if number < 2
        return 0

    divisor Uint32 = 2

    loop divisor <= number / divisor
        if number % divisor == 0
            return divisor
        divisor++

    return 0
```

The quotient in the loop condition avoids a possibly overflowing square. A nonzero return identifies a factor; zero still needs interpretation by the caller because it also covers inputs zero and one. Finally add the bounded summation:

```
func sumTo(limit Uint32) -> Uint64

    total Uint64 = 0
    next  Uint64 = 1

    loop next <= Uint64(limit)
        total += next
        next++

    return total
```

The accumulator and advancing value are visibly wider than the input. The loop's work is also visible: one addition for every integer through the limit. We can later replace this helper without changing how arguments are accepted or how the report is presented.

Turn the Results into a Report

Add `reportNumber` after the helpers and before the entry banner. Start by collecting a factor, a category, the common divisor and the sum. The category begins as prime and is overridden for the two special cases:

```
    factor = smallFactor(number)
    category Text = "prime"

    if      number < 2  : category = "neither prime nor composite"
    else if factor != 0 : category = "composite"
```

Listing continued

```
    common = greatestCommonDivisor(number, 360)
    total = sumTo(number)
```

Those rows are a useful compact table because both branches merely assign the same property. The function then prints one multiline report and returns success. Here is the whole helper, including its signature and checked write:

```
func reportNumber(number Uint32) -> Core.Status

    factor = smallFactor(number)
    category Text = "prime"

    if      number < 2  : category = "neither prime nor composite"
    else if factor != 0 : category = "composite"

    common = greatestCommonDivisor(number, 360)
    total = sumTo(number)

    print("""
        Number:   [ {number} ]
        Class:    [ {category} ]
        Factor:   [ {factor} ]
        GCD 360:  [ {common} ]
        Sum:      [ {total} ]
        """)?

    return Core.ok()
```

Now replace the temporary Accepted print and `return Core.ok()` at the end of `start` with `return reportNumber(number)`. The entry has become an input boundary followed by one clearly named operation. It forwards that operation's status to the host.

Read the Boundary First

`start` is the assembly point: it accepts the input, asks the parser to turn text into a number, applies our bound, then delegates to the report. The helpers appear before it so each operation is defined before the application combines them. `[Uint32]` in `parseInt[Uint32]` selects the integer type to produce; `?` unwraps success or returns the error from `start`.

The argument slice excludes the executable name. The program therefore expects a slice containing just one text, such as `"49"`. Think of the host passing a one-element slice whose element is `"49"`; `args[0]` is then that text. A no-argument invocation supplies a zero-length slice. An empty slice and a two-element slice both fail before indexing. An argument containing spaces is not silently trimmed; the parser requires the entire text to be a valid number.

There are two different numerical checks. `parseInt[Uint32]` rejects malformed or unrepresentable input. The application then rejects an otherwise valid value above `MAXIMUM`. Keeping the checks separate preserves useful errors and makes the application's chosen limit visible in its inventory.

The banner is printed before validation. A failure can therefore leave that banner as observable output. Nothing promises to roll the output back. If you wanted absolutely no output on invalid input, move the reporting phase after the input checks; that would be a deliberate behavior change.

Every output call uses `?`. A failed write leaves the current error-bearing function and reaches the host's normal failed-entry handling. The program owns no file or heap allocation to release. No cleanup ceremony is needed where no resource was acquired.

Follow One Input

For input `49`, the report after the banner is:

```
Number:   [ 49 ]
Class:    [ composite ]
Factor:   [ 7 ]
GCD 360:  [ 1 ]
Sum:      [ 1225 ]
```

`smallFactor` tries integers only while the divisor is at most the quotient. The trial results show why seven is the first answer:

Divisor	`49 / divisor`	`49 % divisor`	Decision
2	24	1	Advance
3	16	1	Advance
4	12	1	Advance
5	9	4	Advance
6	8	1	Advance
7	7	0	Return 7

Seven divides 49, so the function returns it. Euclid's algorithm reduces `(49, 360)` to a greatest common divisor of one. The sum loop adds 1 through 49. Its result can also be checked independently as `49 * 50 / 2`, performed in a sufficiently wide type.

For a prime such as 47, the factor field is zero. That is the helper's sentinel, not a claim that zero is a divisor. For zero and one, the explicit category check overrides the initial `"prime"` value. This is why a helper's return value must be interpreted together with its preconditions and the caller's question.

Notice that choosing a category does not end the report: both GCD and sum are calculated for every accepted number, including zero and one.

Check More Than One Happy Path

A useful test set includes `0`, `1`, `2`, `49`, `360` and `1000000`, plus absent input, repeated arguments, `"hello"`, `"-1"` and `"1000001"`. These cases distinguish mathematical boundaries, parser failures and the application limit.

For 360, the greatest common divisor with 360 is 360. For zero, it is also 360 under the helper's convention; the sum is zero and the category is neither prime nor composite. The maximum accepted input has sum 500,000,500,000, which fits `Uint64` but not `Uint32`.

Do not infer a performance result from a correct numerical answer. The sum loop performs one iteration per number. A formula can reduce that work, but it introduces an intermediate multiplication that must still fit. The

measurement chapter explains how to compare implementations without confusing an argument about complexity with a measured duration.

Put the Program Together

The construction produces this complete file. Compare its input boundary and helper order with your assembled version; the temporary Accepted output has been replaced by the final report.

```
_____________________________________________________________________________

[ Number Laboratory ] Examines one bounded nonnegative decimal integer.
_____________________________________________________________________________

import Core
import Strings

from IO import print
____
____ [ Input Bound ]

MAXIMUM const Uint32 = 1_000_000


____
____ [ Integer Algorithms ]

func greatestCommonDivisor(a Uint32, b Uint32) -> Uint32

    loop b != 0
        remainder = a % b
        a = b
        b = remainder

    return a


func smallFactor(number Uint32) -> Uint32

    if number < 2
        return 0

    divisor Uint32 = 2

    loop divisor <= number / divisor
        if number % divisor == 0
            return divisor
        divisor++

    return 0


func sumTo(limit Uint32) -> Uint64

    total Uint64 = 0
    next  Uint64 = 1

    loop next <= Uint64(limit)
        total += next
        next++
```

Listing continued

```
    return total


____
____ [ Number Report ]

func reportNumber(number Uint32) -> Core.Status

    factor = smallFactor(number)
    category Text = "prime"

    if      number < 2  : category = "neither prime nor composite"
    else if factor != 0 : category = "composite"

    common = greatestCommonDivisor(number, 360)
    total = sumTo(number)

    print("""
        Number:   [ {number} ]
        Class:    [ {category} ]
        Factor:   [ {factor} ]
        GCD 360:  [ {common} ]
        Sum:      [ {total} ]
        """)?

    return Core.ok()

_____________________________________________________________________________

[ Application Entrypoint ]
_____________________________________________________________________________

func start(args [:]Text) -> Core.Status

    print("""
        ________________________________________

         Number Laboratory - ec
        ________________________________________
        """)?

    if args.len != 1
        return Core.INVALID_ARGUMENT

    number = Strings.parseInt[Uint32](args[0])?

    if number > MAXIMUM
        return Core.OUT_OF_RANGE

    return reportNumber(number)
```

Exercises and Discussion

1. Replace the sum loop with a constant-work formula while preserving the whole accepted input range.

2. Add the number of decimal digits to the report, treating zero as one digit.

3. Decide whether a leading plus sign should be accepted, and locate the rule that controls the current answer.

Discussion

1. Convert the input to `Uint64` before computing `n * (n + 1) / 2`. Under the one-million application bound, the product fits easily. Converting after a narrow multiplication would not repair an already wrapped result.

2. For the digit count, this module-level helper can go before `reportNumber`. The parameter itself is a local copy, so dividing it cannot change the report's original number.

```
func decimalDigits(value Uint32) -> Uint32

    count Uint32 = 1

    loop value >= 10
        value /= 10
        count++

    return count
```

The initial one handles every one-digit value, including zero. Each division removes one decimal digit: 1,000 becomes 100, then 10, then 1, yielding four digits. In `reportNumber`, compute `digits = decimalDigits(number)` before printing and add a `Digits: [ {digits} ]` line to the report. The accepted upper bound, 1,000,000, has seven digits.

3. The standard parser accepts an optional ASCII `+` or `-`, followed by digits. A nonzero negative value cannot fit `Uint32`; `-0` can. If the application's text policy should be stricter than that, add an explicit spelling check rather than claiming the parser has a different contract.

On Your Own

Add an integer-only perfect-square test and a Boolean Square field to the report. Preserve the existing input policy and one-million workload bound. The field should be true for 0, 1, 49 and 1,000,000, and false for 2 and 50. Malformed input and values above the bound must continue to fail before calculations begin.

A perfect square has the form `root * root`. Before choosing a loop condition, decide how to prevent division by zero and how large a product can become under the application's bound. Keep the test in its own helper so its cases can be examined independently of text parsing and output.

Further Reading

ec Technical Manual: Strings; Errors and Cleanup.

A Small Gradebook

A number laboratory asks several questions about one value. A gradebook asks related questions about several records: which student has each score, what is the class average, and what order makes the report useful?

Our application uses a fixed set of student records. It validates scores, computes a summary and prints a stable descending ranking. There is no hidden database, input parser or persistence layer. Keeping the input in the source lets us concentrate on the relationship between records, array storage and algorithms. A later variation can supply the same records from another source without changing the calculations.

Read the chapters on Records and States, Sorting and Matrices and Errors and Cleanup first. Save this complete program as `gradebook.ec` in its own module directory. It uses a hosted profile with floating point enabled.

Begin with the Records and the Question

Create `gradebook.ec` in a new application directory with a Gradebook source banner, `import Core` and `from IO import print`. Use the checked runtime banner and a `func start() -> Core.Status` entry beneath its own entry banner. We will fill in that entry after its data operations are ready; the assembled file follows the lesson.

Put these types after the imports. `Student` is one input record. `Summary` is the answer to four class-wide questions:

```
struct Student
    name  Text
    score Uint32

struct Summary
    count   UintSize
    total   Uint64
    lowest  Uint32
    highest Uint32
```

In `start`, below the checked runtime banner, add the supplied records:

```
    students []Student = [Student(name = "Ada", score = 86),
                          Student(name = "Ben", score = 72),
                          Student(name = "Cy",  score = 86),
                          Student(name = "Dee", score = 95)]
```

The repeated initializer columns let us check each name against its score. The literal names need no allocation or cleanup. For the first stage, print one known field with `print("First score: [ {students[0].score} ]\n")?` and end with `return Core.ok()`. This should report 86. Remove this temporary output when connecting the final report.

Validate and Accumulate Before Sorting

Place `summarize` after the types. We need a nonempty class to define an average and extrema, and we limit the class size to 1,000. After those checks, a total starts at zero, a minimum starts at 100 and a maximum at zero. Each accepted score updates all three. The operator `><` chooses the smaller of two values, and `<>` chooses the larger. Thus the two assignments below retain the lowest and highest scores seen so far; they do not swap values.

```
func summarize(students [:]Student) -> Core.Result[Summary]

    if (students.len == 0) || (students.len > 1000)
        return Core.OUT_OF_RANGE

    total   Uint64 = 0
    lowest  Uint32 = 100
    highest Uint32 = 0

    for student in students
        if student.score > 100
            return Core.OUT_OF_RANGE

        total += Uint64(student.score)
        lowest = lowest >< student.score
        highest = highest <> student.score

    return Summary(count = students.len, total = total,
                   lowest = lowest, highest = highest)
```

To connect this stage, add `summary = summarize(students[:])?` after the records in `start`. Temporarily print `summary.total`; the four scores sum to 339. Change Ben's score to 101 and the helper should return an error before sorting exists. Restore 72. Testing validation now keeps a later rearrangement from hiding where bad input was first accepted.

Add Stable Ranking

Insert `rankStudents` after the summary helper. It is insertion sort from the data chapter with two changes: the saved value is a whole Student, and comparison uses its score in descending order.

```
func rankStudents(students [:]Student)

    position UintSize = 1

    loop position < students.len
        held = students[position]
        insertion = position

        loop insertion > 0
            if students[insertion - 1].score >= held.score
                break
            students[insertion] = students[insertion - 1]
            insertion--

        students[insertion] = held
        position++
```

The saved `held` record keeps its name and score together while predecessors move. The inner test stops at an equal score, so tied students keep their earlier order. Add `rankStudents(students[:])` immediately after successful summarization in `start`. That call changes the existing array; the already computed summary remains valid because sorting does not change the scores themselves.

Report the Completed Work

The last helper consumes the ordered records and the summary. Add it after `rankStudents`:

```
func reportClass(students [:]Student, summary Summary) -> Core.Status

    average = Real64(summary.total) / Real64(summary.count)

    print("""
        Students: [ {summary.count} ]
        Average:  [ {average:.2f} ]
        Lowest:   [ {summary.lowest} ]
        Highest:  [ {summary.highest} ]
        """)?

    for index, student in students
        rank = index + 1
        print("Rank [ {rank} ]  {student.name}: [ {student.score} ]\n")?

    return Core.ok()
```

Its average converts the total and count before division. The loop prints a one-based rank while keeping the array's zero-based index local. Remove the temporary score/total prints in `start` and finish it with `return reportClass(students[:], summary)`. The application now has one clear path: supply records, validate and summarize, rank, report.

Following the Application

Begin at `start`: the array is the input, `summarize` reads and validates it, `rankStudents` changes its order, and `reportClass` reads the changed order. The summary is computed only once. Reordering cannot change its count, total, minimum or maximum, so it is still the correct summary afterward.

In `for student in students`, `student` is a pointer to one record. The field expression `student.score` selects the pointed-to record's field; it is the convenient member-access form of `student.*.score`. The reporting loop also asks for an index, which begins at zero. Adding one gives the displayed ranks beginning at one.

One Record, Two Kinds of Storage

A `Student` stores its score inline and a `Text` descriptor for its name. Copying the record copies both values. It does not create another copy of the name's bytes. The names in this program are literals with static storage, so they remain valid while records move during sorting.

That property would need attention with dynamically constructed names. Moving a record is harmless only if its borrowed name remains usable. A sorting algorithm cannot fix the lifetime of the data supplied to it. The explicit resource chapters explain how to arrange storage when records must outlive an input buffer.

The array itself is local to `start`. Every `students[:]` creates a slice describing those same elements. The ranking function therefore changes the array that the report function subsequently reads. No second array is allocated. If

preserving original order were a requirement, we would first request an explicit array copy or supply separate storage.

Validate before Rearranging

`summarize` rejects an empty class and classes larger than 1,000. It also rejects every score above 100. Those choices make the result meaningful and keep the maximum total at 100,000. The `Uint64` accumulator has ample room; the proof comes from the accepted input bounds rather than hope about typical class sizes.

Validation and summary happen before ranking. A bad score therefore returns a failure before any records are rearranged. The local running total may have changed, but it is not exposed as a successful result. This is a useful transaction boundary: validate the data while reading, then perform the requested mutation.

`><` selects the minimum and `<>` the maximum. Their appearance may be unfamiliar, so read the two assignments aloud: the new lowest is the smaller of the previous lowest and this score; the new highest is the larger. The initial values are justified by the accepted score range.

Why the Ranking Is Stable

Insertion sort keeps an already sorted prefix. The held record moves left only past records with a strictly smaller score. The test `>= held.score` stops at an equal score, preserving the original order of equal-score students. That is the meaning of a stable sort.

Follow the records after each insertion; the scores travel with their names:

Held record	Records shifted right	Order after insertion
Ben, 72	None: Ada already ranks above Ben	Ada, Ben, Cy, Dee
Cy, 86	Ben; stop at equal-scoring Ada	Ada, Cy, Ben, Dee
Dee, 95	Ben, Cy, Ada	Dee, Ada, Cy, Ben

For our input, the ranking is Dee, Ada, Cy, Ben. Ada stays before Cy because both scored 86 and Ada appeared first. A condition that shifted equal scores too would still produce descending numerical order, but it would lose this promised tie behavior.

The report after the banner is expected to be:

```
Students: [ 4 ]
Average:  [ 84.75 ]
Lowest:   [ 72 ]
Highest:  [ 95 ]
Rank [ 1 ]  Dee: [ 95 ]
Rank [ 2 ]  Ada: [ 86 ]
Rank [ 3 ]  Cy: [ 86 ]
Rank [ 4 ]  Ben: [ 72 ]
```

The average divides real conversions of the integer total and count. It does not first truncate an integer division. `.2f` asks for two fractional decimal digits in the output; it does not change the stored `Real64` value.

Put the Program Together

Here is the complete gradebook after replacing the temporary outputs. Its source order follows the construction: data shapes, summary, ordering, report, then the entry that connects them.

```
____________________________________________________________________________

[ Gradebook ] Validates scores and prints a stable descending ranking.
____________________________________________________________________________

import Core

from IO import print


____
____ [ Class Records ]

struct Student
    name  Text
    score Uint32

struct Summary
    count   UintSize
    total   Uint64
    lowest  Uint32
    highest Uint32


func summarize(students [:]Student) -> Core.Result[Summary]

    if (students.len == 0) || (students.len > 1000)
        return Core.OUT_OF_RANGE

    total   Uint64 = 0
    lowest  Uint32 = 100
    highest Uint32 = 0

    for student in students
        if student.score > 100
            return Core.OUT_OF_RANGE

        total += Uint64(student.score)
        lowest = lowest >< student.score
        highest = highest <> student.score

    return Summary(count = students.len, total = total,
                   lowest = lowest, highest = highest)


func rankStudents(students [:]Student)

    position UintSize = 1

    loop position < students.len
        held = students[position]
        insertion = position
```

Listing continued

```
        loop insertion > 0
            if students[insertion - 1].score >= held.score
                break
            students[insertion] = students[insertion - 1]
            insertion--

        students[insertion] = held
        position++


func reportClass(students [:]Student, summary Summary) -> Core.Status

    average = Real64(summary.total) / Real64(summary.count)

    print("""
        Students: [ {summary.count} ]
        Average:  [ {average:.2f} ]
        Lowest:   [ {summary.lowest} ]
        Highest:  [ {summary.highest} ]
        """)?

    for index, student in students
        rank = index + 1
        print("Rank [ {rank} ]  {student.name}: [ {student.score} ]\n")?

    return Core.ok()

_____________________________________________________________________________

[ Application Entrypoint ]
_____________________________________________________________________________

func start() -> Core.Status

    print("""
        ___________________________________________

         Gradebook - ec
        ___________________________________________
        """)?

    students []Student = [Student(name = "Ada", score = 86),
                          Student(name = "Ben", score = 72),
                          Student(name = "Cy",  score = 86),
                          Student(name = "Dee", score = 95)]

    summary = summarize(students[:])?
    rankStudents(students[:])

    return reportClass(students[:], summary)
```

Exercises and Discussion

1. Give Ben a score of 101. Which operations have already happened when the program returns failure?

2. Change the ranking to ascending order while keeping it stable.

3. Add a count of passing scores, with a pass defined as at least 60. Decide whether that belongs in the summary or the reporting function.

Discussion

1. The banner has been printed. Ada's valid score has made the local total 86 and both extrema 86. The next iteration reads Ben's 101 and returns before adding it. These local values are discarded as the function returns failure. The ranking has not begun, and no summary is returned successfully. This distinction is why the order of validation and mutation matters.

2. For an ascending stable sort, stop shifting when the preceding score is `<= held.score`. Equal scores again stop the movement. With the original input, the ascending result should be Ben, Ada, Cy, Dee: Ada still precedes Cy. That pair checks stability as well as numeric order.

3. A passing count is a property of the class data, so the summary is a natural home. Add `passing UintSize` to `Summary`, initialize `passing UintSize = 0` alongside the other summary accumulators, and after each score passes validation increment it under `if student.score >= 60`. Include `passing = passing` in the `Summary(...)` return and render `{summary.passing}` in the report. All four supplied scores pass; changing Ben's score to 59 should make the count three while leaving the class valid. The report should display that computed fact rather than silently repeat the algorithm. This keeps the question and its answer in one place.

On Your Own

Change the printed ranks so equal scores share a rank, with a gap afterward. The supplied descending order should report Dee at 1, Ada and Cy at 2, and Ben at 4. Preserve the stable order of tied students and the existing summary. With every score equal, every displayed rank should be 1.

You need the position in the ordered array and some information about the previous record; decide when a new score starts a new rank. Retain the nonempty-class validation and test one student, two tied students and four distinct scores. Do not change the sort merely to change how ranks are displayed.

Further Reading

ec Technical Manual: Aggregate Types; Arrays and Slices.

A Bounded Text Report

A useful text tool begins by defining what it counts. “Characters” might mean bytes, Unicode scalars or displayed grapheme clusters. “Words” might require a language-specific tokenizer. We will build a smaller, precise tool: it counts UTF-8 bytes, Unicode scalars, ASCII-separated words, LF line breaks and the longest word measured in scalars. It also reports whether the input is entirely ASCII.

The input is zero or one text argument. With none, the program uses a supplied sample. One empty argument means empty text. More arguments are rejected instead of being silently joined. Input is limited to 4096 bytes, so the program has a clear boundary and every counter has a simple upper bound.

Our word separators are exactly space, tab, CR and LF. CR is carriage return (`\r`); LF is line feed (`\n`), the line break this report counts. A run of all other scalars is one word. Punctuation belongs to a word, and a non-ASCII space belongs to a word too. This is an intentional counting policy, not a claim to implement Unicode word segmentation.

Set Up a Small Text Tool

Create `textReport.ec` in its own module directory. Use a Text Report source banner, `import Core`, `import Strings`, `from Core import Result, Status` and `from IO import print`. The entry will be `func start(args [:]Text) -> Status`, last beneath the application-entry banner, with a checked runtime banner. The final listing shows the entire file.

First place these constants under an Input Defaults inventory header after the imports:

```
MAX_BYTES    const UintSize = 4096
DEFAULT_TEXT const Text     = "A caf\u{E9}\nB"
```

In `start`, after the banner, accept zero or one argument. No argument selects the supplied sample; one argument selects exactly that text, including an empty one:

```
    if args.len > 1
        print("usage: supply zero or one text argument.\n")?
        return Core.INVALID_ARGUMENT

    text Text = DEFAULT_TEXT

    if args.len == 1
        text = args[0]
```

For this first stage, add `print("Input bytes: [ {text.len} ]\n")?` and `return Core.ok()`. With no arguments, expect 9. This verifies input selection before we write a scanner. Remove that temporary print when we connect the report.

Build the State the Scan Needs

Below the inventory, add this type. Zero is the natural starting count; ASCII-only starts true and becomes false only when contrary evidence appears:

```
struct TextStats
    bytes       UintSize = 0
    scalars     UintSize = 0
    words       UintSize = 0
    lineBreaks  UintSize = 0
    longestWord UintSize = 0
    ascii       Bool     = true
```

Next put the separator decision in a small helper. This is the program's word policy in one place:

```
func isSeparator(value Char) -> Bool
    return (value == ' ') || (value == '\t') || (value == '\r') || (value == '\n')
```

The report record contains final totals. Two local variables describe progress: `offset` is the next UTF-8 byte position, and `wordLength` is the number of scalars in the current word. Before each iteration, all bytes before `offset` have been decoded and counted.

`Strings.nextScalar` returns both the scalar and its next byte offset. Advancing to that returned offset preserves the boundary invariant. A separator resets `wordLength` to zero. A nonseparator begins a new word exactly when `wordLength` was zero, then extends it by one scalar. Updating the maximum after each extension avoids a special end-of-input correction.

The helper returns `Result[TextStats]`, and `?` propagates a failed decode. Although valid `Text` and a maintained boundary make those failures absent under this helper's contract, the code still respects the library's returned-result interface. The entry returns `Status`, so formatting and output failures propagate in the same way. Review Errors and Cleanup if this control flow is unfamiliar.

Turn One Decoded Scalar into Updated Counts

Before writing the full loop, follow one iteration. This is the body we will place inside `analyze`'s `loop offset < value.len`; `value` is its Text parameter and `stats`, `offset` and `wordLength` are its initialized locals:

```
        decoded = Strings.nextScalar(value, offset)?
        character = decoded.0
        offset = decoded.1

        stats.scalars++

        if Uint32(character) > #7F
            stats.ascii = false

        if character == '\n'
            stats.lineBreaks++
```

Decoding supplies the next byte boundary; counting increases by one scalar, not by the number of bytes consumed. The ASCII and LF questions are independent, so they are separate branches. A scalar can affect more than one report field.

After those lines, the word update asks whether the scalar separates words or extends one:

```
        if isSeparator(character)
            wordLength = 0
        else
            if wordLength == 0
```

Listing continued

```
                stats.words++
            wordLength++
            stats.longestWord = stats.longestWord <> wordLength
```

The zero-length test counts a word when it begins. A second separator does not invent an empty word. Updating the maximum after every extension also handles a final word with no trailing newline.

Now enclose those operations in the complete helper below and insert it after `isSeparator`. This is the only definition of `analyze` to add; the two earlier groups showed the pieces of its body.

```
func analyze(value Text) -> Result[TextStats]

    if value.len > MAX_BYTES
        return Core.OUT_OF_RANGE

    stats = TextStats(bytes = value.len)
    offset     UintSize = 0
    wordLength UintSize = 0

    loop offset < value.len
        decoded = Strings.nextScalar(value, offset)?
        character = decoded.0
        offset = decoded.1

        stats.scalars++

        if Uint32(character) > #7F
            stats.ascii = false

        if character == '\n'
            stats.lineBreaks++

        if isSeparator(character)
            wordLength = 0
        else
            if wordLength == 0
                stats.words++
            wordLength++
            stats.longestWord = stats.longestWord <> wordLength

    return stats
```

In `start`, add `stats = analyze(text)?` after choosing the input. For a checkpoint, temporarily print `stats.words`; the default sample should give three. The byte limit is enforced by the helper before its loop, so a larger input never begins analysis.

Supply Storage for the Report

The facts are computed; now give their text rendering somewhere to live. Put `renderReport` after `analyze`:

```
func renderReport(stats TextStats, buffer *Strings.TextBuffer) -> Result[Text]
    return buffer.format("""
        bytes        [ {stats.bytes} ]
        scalars      [ {stats.scalars} ]
        words        [ {stats.words} ]
        LF breaks    [ {stats.lineBreaks} ]
```

Listing continued

```
        longest word [ {stats.longestWord} ]
        ASCII only   [ {stats.ascii} ]
        """)
```

Its caller supplies a buffer rather than receiving a view into a dead local array. In `start`, replace the checkpoint prints with these lines after the analysis call:

```
    reportStorage [256]Uint08
    reportBuffer = Strings.fixed(reportStorage[:])
    report = renderReport(stats, &reportBuffer)?
    print(report)?

    return Core.ok()
```

Keep the storage and its descriptor next to each other: that layout shows which bytes back the returned report. Printing finishes before `start` returns and ends that local storage. Every operation with a failure result is checked or propagated.

Read the helpers as three jobs: `isSeparator` defines the word rule, `analyze` computes facts, and `renderReport` turns those facts into text. `start` joins them. In the scanner, `decoded.0` is the recovered scalar and `decoded.1` is the next byte offset. `#7F` is 127, the highest ASCII value. The `<>` operator selects the larger of its two operands, so the longest-word field never decreases.

There is no heap allocation or owned resource requiring a deferred release. The buffer and returned report view stay alive until `print` has consumed them.

`renderReport` returns a `Text` view into storage supplied by its caller. It does not return a view into its own local array. This is the same safe borrowing pattern developed in Pointers and Views.

Reading the Expected Result

With no arguments, the selected text is `A`, a space, `café`, LF and `B`. The source spelling uses an escape for the non-ASCII scalar. After the banner, the report's specified values are:

```
bytes        [ 9 ]
scalars      [ 8 ]
words        [ 3 ]
LF breaks    [ 1 ]
longest word [ 4 ]
ASCII only   [ false ]
```

The byte count exceeds the scalar count by one because U+00E9 takes two UTF-8 bytes. The words are `A`, `café` and `B`. The longest contains four scalars. The LF is counted as a line break and as a separator, but never as a word scalar.

Trace the changes around the middle word. After `A`, `wordLength` is one. The space resets it to zero. Reading `c` increments the word count and begins a new length of one. The next three scalars extend that length to four, updating the maximum. LF resets it again. The final `B` begins the third word. No special “finish last word” operation is necessary because the word was counted when it began. Here is the same scan with the byte position made visible; each row shows state after consuming that scalar:

Consumed scalar	Next byte offset	Words	Current word length	Longest
`A`	1	1	1	1
Space	2	1	0	1
`c`	3	2	1	1
`a`	4	2	2	2
`f`	5	2	3	3
U+00E9	7	2	4	4
LF	8	2	0	4
`B`	9	3	1	4

Failure and Boundary Behavior

An empty argument produces zero for every count and `true` for ASCII-only: there is no non-ASCII scalar in empty input. Repeated separators produce no empty words. A final word needs no trailing separator. CRLF counts as one LF break, while CR alone counts as a separator but contributes no LF break. The label says exactly what the program counts.

The byte limit applies before decoding. At most 4096 scalars and 4096 words can occur within 4096 bytes, so all counters fit `UintSize` on either hosted profile. No counter relies on wrapping. A larger input returns `OUT_OF_RANGE` before its analysis loop runs. Startup validates text arguments before calling `start`; invalid UTF-8 arguments follow that startup failure contract rather than entering this scan.

Formatting is bounded by the fixed 256-byte array. The six short labels, Boolean and at-most-four-digit counts fit that capacity, but the call is still checked. If a future report grows beyond it, formatting returns a failure without changing the buffer. Output is different: a failed print may already have written a prefix. The program propagates the first output failure; it does not claim to roll back the banner or earlier bytes.

Put the Program Together

The final file replaces all checkpoint prints with the report. Use it to compare the assembled stages: input selection, bounded analysis, caller-owned rendering storage and checked output.

```
_____________________________________________________________________________

[ Text Report ] Counts a bounded UTF-8 input using explicit word rules.
_____________________________________________________________________________

import Core
import Strings

from Core import Result, Status
from IO   import print
____
____ [ Input Defaults ]
```

Listing continued

```
MAX_BYTES    const UintSize = 4096
DEFAULT_TEXT const Text     = "A caf\u{E9}\nB"


____
____ [ Text Analysis ]

struct TextStats
    bytes       UintSize = 0
    scalars     UintSize = 0
    words       UintSize = 0
    lineBreaks  UintSize = 0
    longestWord UintSize = 0
    ascii       Bool     = true


func isSeparator(value Char) -> Bool
    return (value == ' ') || (value == '\t') || (value == '\r') || (value == '\n')


func analyze(value Text) -> Result[TextStats]

    if value.len > MAX_BYTES
        return Core.OUT_OF_RANGE

    stats = TextStats(bytes = value.len)
    offset     UintSize = 0
    wordLength UintSize = 0

    loop offset < value.len
        decoded = Strings.nextScalar(value, offset)?
        character = decoded.0
        offset = decoded.1

        stats.scalars++

        if Uint32(character) > #7F
            stats.ascii = false

        if character == '\n'
            stats.lineBreaks++

        if isSeparator(character)
            wordLength = 0
        else
            if wordLength == 0
                stats.words++
            wordLength++
            stats.longestWord = stats.longestWord <> wordLength

    return stats
```

Listing continued

```
func renderReport(stats TextStats, buffer *Strings.TextBuffer) -> Result[Text]
    return buffer.format("""
        bytes        [ {stats.bytes} ]
        scalars      [ {stats.scalars} ]
```

Listing continued

```
        words        [ {stats.words} ]
        LF breaks    [ {stats.lineBreaks} ]
        longest word [ {stats.longestWord} ]
        ASCII only   [ {stats.ascii} ]
        """)

____________________________________________________________________________

[ Application Entrypoint ]
____________________________________________________________________________

func start(args [:]Text) -> Status

    print("""
        ________________________________________

         Text Report - v. 1.0
        ________________________________________
        """)?

    if args.len > 1
        print("usage: supply zero or one text argument.\n")?
        return Core.INVALID_ARGUMENT

    text Text = DEFAULT_TEXT

    if args.len == 1
        text = args[0]

    stats = analyze(text)?

    reportStorage [256]Uint08
    reportBuffer = Strings.fixed(reportStorage[:])
    report = renderReport(stats, &reportBuffer)?
    print(report)?

    return Core.ok()
```

Exercises and Discussed Answers

1. Predict the report for the text represented by `"one\t\ttwo\n"` before tracing the loop.

2. Make punctuation a separator too. What must be decided before editing `isSeparator`?

3. Add a count of non-ASCII scalars without scanning the text twice.

4. Why should the program not return `report` from `start` for another component to consume later?

Discussion

1. The answer is 9 bytes, 9 scalars, two words, one LF break, longest word three and ASCII-only true. The second tab sees a zero current-word length and keeps it zero; no word is created.

2. First list the exact punctuation scalars to include. A condition for `'.'`, `','` and `'!'` is a different policy from all Unicode punctuation. State the chosen policy in the input contract and test leading, repeated and trailing separators. Do not silently describe a small ASCII list as universal tokenization.

3. Add `nonAscii UintSize = 0` to `TextStats` and place `stats.nonAscii++` in the existing `Uint32(character) > #7F` branch. Add a report line containing `{stats.nonAscii}`. The supplied sample should report one; the all-ASCII exercise input should report zero. Keep the Boolean assignment there as well, or derive ASCII-only from a zero count when rendering. The invariant still accounts for every decoded scalar exactly once.

4. For the fourth, `start` has a specified hosted result type, and the report borrows an automatic array whose lifetime ends on return. A longer-lived consumer would need longer-lived caller-owned storage or an explicit owned allocation. Copying the descriptor alone solves neither problem.

Continue with Measuring and Testing, using the empty, boundary and separator cases here as a starting test set.

On Your Own

Extend the report with an ASCII-digit count without adding another scan. Count only scalars from `'0'` through `'9'`. The text `"A1 20\n"` should produce a digit count of 3; empty input and `"abc"` should produce zero. U+00B2, written `"\u{B2}"` in source, should not count even though it is displayed as a superscript digit.

Add the field to the record, update it during decoding and render it with the other totals. Keep the existing byte limit and verify that the enlarged report still fits its fixed buffer. Explain where the new conditional belongs so that digits also retain their existing contribution to word and scalar counts.

Further Reading

ec Technical Manual: Text and Formatting; Strings.

Measuring and Testing

Build a small measurement application whose result you can explain. It will count occurrences of an integer key, check that counting is correct, collect eight elapsed durations and report their summary. The point is to connect a known calculation to a stated workload, not to produce an impressive-looking number.

Create `measureReadings.ec` in its own application directory. The final program uses `Core`, `IO` and `System`, and imports `print` from IO. Select a hosted profile with `clocks` and `standardStreams`. We will construct the calculation, its tests, the measurement loop and the caller in that order, then assemble the source.

Start with a result that can be checked by hand

Our calculation accepts `values [:]Int32` and a key, returning a UintSize count. Before writing the loop, predict the answer for `[4, -1, 4, 0]` with keys 4, -1 and 9: respectively 2, 1 and 0.

The accumulator starts at zero. Each matching element adds one, so it can never exceed the slice length. This gives the complete first helper:

```
func countMatches(values [:]Int32, key Int32) -> UintSize

    count UintSize = 0

    for value in values
        if value.* == key
            count++

    return count
```

Its invariant says that `count` equals the number of matching values in the visited prefix. That gives an O(n) work bound and O(1) additional storage: the number of visits grows with the input length n, while the counter and loop state need a fixed amount of space. The input is borrowed, initialized and unchanged.

Turn predictions into tests

Create `testCountMatches()` with no parameters or result after the calculation. It will own the four-element fixture above. Begin its body with that array and these assertions:

```
    values []Int32 = [4, -1, 4, 0]

    Core.assert(countMatches(values[:0], 4) == 0)
    Core.assert(countMatches(values[:], 4) == 2)
    Core.assert(countMatches(values[:], -1) == 1)
    Core.assert(countMatches(values[:], 9) == 0)
```

An assertion records a condition the program must satisfy; false causes a fatal assertion failure. This is suitable for our own test expectations. Unsuitable user input normally belongs in a recoverable Result, rather than being declared impossible by an assertion.

Add three boundary checks after these: `values[1:3]` contains one 4, `values[2:3]` contains one 4, and `values[1:2]` contains none. The singleton cases are especially good at exposing a loop that misses its final element.

Next test a relationship, not just another answer. Splitting the input into adjacent parts must preserve the sum of match counts. Append this phase to the same test function:

```
    split UintSize = 2

    whole = countMatches(values[:], 4)
    parts = countMatches(values[:split], 4) + countMatches(values[split:], 4)

    Core.assert(whole == parts)
```

Both counts are 2. Try split zero and split `values.len`: one part then becomes empty. The sum cannot overflow because the two parts together contain exactly the original number of elements. These paired calculations align because they are the two sides of one property. They are not unrelated steps decorated with arbitrary spacing.

Such properties help compare a simple implementation with a candidate optimization. Keep the explanation of the simple version's correctness; two implementations sharing one boundary mistake are poor evidence for each other.

Define one measurement interval

Wall-clock time answers calendar questions and may move backward after adjustment. A monotonic clock answers elapsed-time questions within one process or browser instance. An Instant marks a point; a Duration is an interval represented in signed nanoseconds. Ordered readings never decrease, but equal readings are allowed. Clock resolution is the granularity limiting how much a very short interval can tell us.

Name the helper `measureCounts`. It takes `values [:]Int32`, `key Int32` and writable `samples [:]System.Duration`, and returns `Core.Result[System.TimingStats]`: either the collected summary or a failure. Input and sample storage must be disjoint and live throughout. Reject an empty sample slice at the start of the function:

```
    if samples.len == 0
        return Core.INVALID_ARGUMENT
```

Then iterate `for sample in samples`. Construct its body in three phases. First make the inputs opaque to translation:

```
        input     = Core.blackBox(values)
        searchKey = Core.blackBox(key)
```

`blackBox` returns the same initialized value, while preventing translation from treating its result as known from its argument. Its observable event prints nothing. It is not a clock, memory fence or instruction to disable all optimizations.

Next start the interval, perform the calculation and make the result observable through the same facility:

```
        started = System.monotonic()

        matches = countMatches(input, searchKey)
        Core.blackBox(matches)
```

Finally append `sample.* = System.elapsed(started)?`. The closing clock read occurs inside `elapsed`. This sample includes the counting work and result-observation event; input preparation occurred before the first reading. There is no output, allocation or file operation in the interval.

The elapsed difference can report `OVERFLOW`; a clock failure itself is fatal under its contract. Propagation leaves already written samples intact but does not initialize later slots. After the loop, return `System.summarize(samples)`. It calculates count, minimum, maximum, total and integer mean, rejecting empty/negative samples and total overflow.

Give the loop a workload and an owner

The caller will allocate a 4,096-element input and eight sample slots as local arrays. Fill each input element with its index modulo 17. Then key 4 occurs 241 times: 240 complete groups of 17 account for 4,080 elements, and the remaining 16 include one more 4.

Check that count before measuring. That is an untimed verification call on the same workload, so it also acts as warm-up. Record that fact when comparing another implementation; equal conditions matter more than pretending preparation has no effect.

After successful collection, call `System.reportTiming`. It returns IO.Progress, not Status: inspect its error field. A failed report may already have written a prefix. Reporting belongs outside every timed interval, otherwise we would be comparing output activity as part of the calculation.

Assemble the application

The completed source contains every construction step. The inventory precedes helpers, each helper precedes its consumer, and `start` remains the last function. Storage for the workload and samples is visible in `runMeasurements` rather than concealed inside a utility.

```
________________________________________________________________________________

[ Reading Measurements ] Checks and measures a linear match counter.
________________________________________________________________________________

import Core
import IO
import System

from IO import print


func countMatches(values [:]Int32, key Int32) -> UintSize

    count UintSize = 0

    for value in values
        if value.* == key
            count++

    return count


func testCountMatches()
```

Listing continued

```
    values []Int32 = [4, -1, 4, 0]

    Core.assert(countMatches(values[:0], 4) == 0)
    Core.assert(countMatches(values[:], 4) == 2)
    Core.assert(countMatches(values[:], -1) == 1)
    Core.assert(countMatches(values[:], 9) == 0)
    Core.assert(countMatches(values[1:3], 4) == 1)
    Core.assert(countMatches(values[2:3], 4) == 1)
    Core.assert(countMatches(values[1:2], 4) == 0)

    split UintSize = 2

    whole = countMatches(values[:], 4)
    parts = countMatches(values[:split], 4) + countMatches(values[split:], 4)

    Core.assert(whole == parts)


func measureCounts(values [:]Int32, key Int32, samples [:]System.Duration) -> Core.Result[System.TimingStats]

    if samples.len == 0
        return Core.INVALID_ARGUMENT

    for sample in samples
        input     = Core.blackBox(values)
        searchKey = Core.blackBox(key)

        started = System.monotonic()

        matches = countMatches(input, searchKey)
        Core.blackBox(matches)

        sample.* = System.elapsed(started)?

    return System.summarize(samples)


func runMeasurements() -> Core.Status

    testCountMatches()

    values  [4096]Int32
    samples [8]System.Duration

    for index, value in values
        value.* = Int32(index % 17)

    Core.assert(countMatches(values[:], 4) == 241)

    stats = measureCounts(values[:], 4, samples[:])?

    progress = System.reportTiming(IO.stdout(), "countMatches", stats)

    if progress.error != Core.NO_ERROR
        return progress.error

    return Core.ok()
```

Listing continued

```
_____________________________________________________________________________

[ Application Entrypoint ]
_____________________________________________________________________________

func start() -> Core.Status

    print("""
        ___________________________________________

         Reading Measurements
        ___________________________________________
        """)?

    return runMeasurements()
```

The sample count should be 8, and each calculation should find 241 matches. No particular duration follows from those facts. If collection fails, reporting is skipped. If tests fail, measurement is never reached. The 4,096 input elements are initialized by the fill loop before either verification or measurement reads them; the conversion is safe because each remainder lies from 0 through 16.

Read a report without overstating it

The summary describes the supplied observations. Integer mean truncates toward zero. The fields do not establish how much future variation to expect or whether a small difference between versions is repeatable.

For a purely arithmetic fixture of durations 2, 4 and 9 nanoseconds, the total is 15 and mean 5. These are test inputs, not timings of our program. On that sorted fixture, `System.percentile` with argument 500 selects the middle sample, 4. Its argument uses thousandths: 500 means halfway, 1000 means the maximum. Nearest rank selects an actual sample rather than interpolating. Input must be sorted, nonnegative and nonempty; the function checks those conditions.

A zero duration can mean two readings occurred within one clock tick. Make the workload larger or measure a stated batch; do not conclude that the algorithm takes no time. Record the target, input size and distribution, sample count, relevant build configuration, and preparation or warm-up policy. Use the same choices for both implementations.

Process/thread CPU time and counters answer different questions and have their own availability and interpretation rules. Elapsed time is not automatically CPU time, allocation count or generated code size. If a proposed optimization sorts once to make repeated searches cheaper, include sorting whenever the actual workload pays that setup cost.

Exercises: Try Changes, Then Explain Them

1. Which existing test exposes a counter that skips its last element?
2. Why is replacing `monotonic` with `utcNow` not a harmless substitution?
3. Compare arithmetic sample sets `[4, 4, 4]` and `[3, 100, 100]`. Does the smaller minimum identify the preferable result?

Discussion

1. The singleton matching slice must count one, so skipping its final element produces zero immediately.

2. Wall-clock adjustments can move calendar time backward, whereas the interval code needs the process monotonic domain.

3. The second sample set has the smaller minimum but a much larger total. Compare the full observations under stated conditions, not only the most flattering field. Testing fixes the required answer; measurement helps evaluate the cost of obtaining that answer.

On Your Own

Add median reporting without destroying the original chronological samples. Copy the eight durations into separate fixed storage, sort the copy using a callback that compares `nanoseconds` directly, and obtain the nearest-rank median with `System.percentile`. Keep sorting and reporting outside the measured intervals. First check the new path with the arithmetic fixture 2, 4 and 9, whose median must be 4. For eight samples, nearest rank selects the fourth sorted element; do not replace it with an average of two elements. Also check a singleton fixture and an empty fixture: the former selects its only value, while the latter must report `Core.INVALID_ARGUMENT`. Explain the extra storage and sort cost beside your results, and propagate errors from the percentile or output operations.

Further Reading

ec Technical Manual, Core; System; IO.