# Computer Science Foundations

## SEMESTER ONE

Seven modules. Fifty-six lessons.
Worked examples, 84 practice questions and answer guides.




Edition 01 / 18 November 2026

Original independent learning materials for first-year Computer Science. Module names checked against MUST Prospectus 2025/2026. This is not official university courseware or a guarantee of current examination coverage.

# Your study map

MS 8121 | Linear Algebra and Calculus 3

HS 8101 | Communication Skills 7

CS 8101 | Theory of Computation 12

CS 8140 | Operating Systems 17

CS 8138 | Programming Concepts in C 22

CS 8102 | Basics in Digital Circuit 27

CS 8103 | Electrical Fundamentals 32

Study plan and practical work 37

Resources and provenance 39

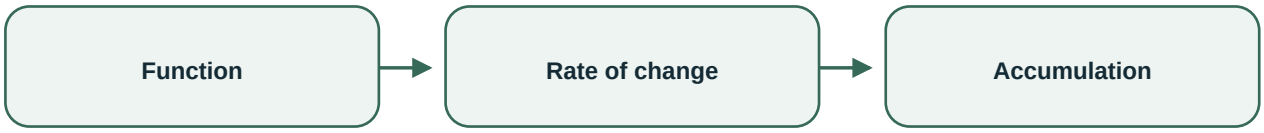
## How to use this book

Read a short lesson, close the page, explain the idea from memory, and solve an original example. Attempt each practice set before using its answer guide. Keep a record of the error, the repair and a fresh successful attempt.

The linked resources are for further study. All lesson text, examples and exercise sets here are original. Code files, a twelve-week study pathway, retrieval cards and a tracker are in the companion ZIP package.

Scope: detailed foundations for the seven current first-semester modules. Later-year modules are not presented as completed notes. Match the pack to your current lecturer outlines, assignments and laboratory rules.

# MS 8121 | Linear Algebra and Calculus




## Learning outcomes and route

By the end of this foundation module, you should be able to describe functions, solve small linear systems, interpret vectors and matrices, evaluate elementary limits, differentiate functions, and use integration to calculate accumulation. These are original foundation lessons based on the module title, not a reproduction of a lecturer's syllabus. Complete Lessons 1-8 in order; solve the practice questions before opening the answers.

Prerequisites: arithmetic with fractions, rearranging equations, powers, and basic coordinate geometry. If you cannot solve 3x + 4 = 19 or simplify x^2 / x for x != 0, revise algebra first.

## Lesson 1 | Functions are input-output rules

A function assigns exactly one output to each allowed input. The domain is the set of allowed inputs; the range is the set of outputs actually reached. Writing f(x) does not mean f multiplied by x. It names the output of f at x. A graph is a collection of input-output pairs, not simply a picture to memorize.

For a quotient, exclude values making the denominator zero. For a real square root, the quantity inside must be nonnegative. Composition means applying one rule after another: (f o g)(x) = f(g(x)). The order usually matters. An inverse reverses a one-to-one rule on a suitable domain; the reciprocal 1/f(x) generally does not.

Worked example: Let f(x) = sqrt(x - 2), g(x) = x + 5. The domain of f is x >= 2. The composition f(g(x)) = sqrt(x + 3) has domain x >= -3. Meanwhile g(f(x)) = sqrt(x - 2) + 5 still requires x >= 2. Checking the domain is part of the answer.

Try explaining to a friend why a function can have several inputs giving the same output, but cannot assign two different outputs to the same input.

## Lesson 2 | Vectors describe magnitude and direction

A vector can represent displacement, force or a list of features. In two dimensions, u = (u1, u2). Add component by component, and multiply each component by a scalar. The length is ||u|| = sqrt(u1^2 + u2^2). The dot product is u.v = u1v1 + u2v2. It also equals ||u|| ||v|| cos(theta), which connects arithmetic to angles. Two nonzero vectors are perpendicular when their dot product is zero.

Worked example: u = (3, 4), v = (4, -3). Their lengths are both 5. Their dot product is 12 - 12 = 0, so they are perpendicular. The unit vector in u's direction is (3/5, 4/5). Do not call u.v a vector: its result is a scalar.

In computing, vectors appear in graphics, optimization and data representations. These applications motivate the mathematics, but do not replace learning the operations by hand.

## Lesson 3 | Matrices and transformations

An m by n matrix has m rows and n columns. To add matrices, their dimensions must match. To multiply A of size m by n and B of size n by p, the inner dimensions n must match; AB then has size m by p. Each output entry is a row-column dot product. Matrix multiplication generally does not commute.

Worked example: A = [[1, 2], [0, 1]], B = [[2, 0], [3, 1]]. AB = [[8, 2], [3, 1]], while BA = [[2, 4], [3, 7]]. Different results are not an arithmetic error: applying transformations in a different order changes the transformation.

The identity I leaves compatible vectors unchanged. An inverse A^-1, when it exists, undoes A. For a 2 by 2 matrix [[a,b],[c,d]], det(A) = ad - bc. If the determinant is nonzero, A^-1 = (1/det(A)) [[d,-b],[-c,a]]. A zero determinant signals loss of independent information; a unique inverse does not exist.

## Lesson 4 | Solving linear systems and understanding solutions

Write a system as an augmented matrix [A | b]. Elementary row operations swap rows, multiply a row by a nonzero number, or add a multiple of one row to another. They preserve the solution set. Eliminate variables, find pivots, then back-substitute. A row [0 0 | nonzero] is a contradiction. A free variable can give infinitely many solutions when the system is consistent.

Worked example: x + 2y = 7 and 3x + y = 11. Subtract three times the first equation from the second: -5y = -10, so y = 2 and x = 3. Substituting into both original equations checks the result: 3 + 4 = 7; 9 + 2 = 11.

Linear independence means that the only coefficients producing the zero vector are all zero. The vectors (1,2) and (2,4) are dependent because the second is twice the first. They do not provide two independent directions. Rank counts independent rows or columns. This helps explain why some systems do not identify a unique answer.

## Lesson 5 | Limits and continuity

A limit describes what outputs approach as inputs approach a value; it does not require the function to be defined at that value. Direct substitution works for many continuous expressions. A 0/0 result is an indeterminate form: simplify or use another valid method rather than declaring the limit zero.

Worked example: lim as x -> 2 of (x^2 - 4)/(x - 2). Factor the numerator: (x - 2)(x + 2). For x != 2 the quotient equals x + 2, so the limit is 4. The original expression remains undefined at 2. Defining its value there as 4 would remove the discontinuity.

A function is continuous at a when f(a) exists, the two-sided limit exists, and that limit equals f(a). Left and right limits must agree. For f(x) = |x|/x, the left limit at zero is -1 and the right limit is 1, so the two-sided limit does not exist.

## Lesson 6 | Derivatives measure local change

The derivative f'(a) is the limit of [f(a+h) - f(a)]/h as h -> 0, when the limit exists. It is the tangent slope and an instantaneous rate of change. The average slope over a finite interval is a different quantity.

Useful rules: d(x^n)/dx = n*x^(n-1); a constant differentiates to zero; the derivative of a sum is the sum of derivatives. The product rule is (uv)' = u'v + uv'. The quotient rule is (u/v)' = (u'v - uv')/v^2 where v != 0. The chain rule multiplies the outer derivative by the derivative of the inner expression.

Worked example: y = (3x + 1)^4. The outer derivative gives 4(3x + 1)^3, then the inner derivative contributes 3. Therefore y' = 12(3x + 1)^3. Missing the factor 3 is a common chain-rule error.

For s(t) = t^2 + 2t measured in metres, velocity is 2t + 2 metres per second and acceleration is 2 metres per second squared. Units help distinguish the quantities.

## Lesson 7 | Optimization is a reasoning process

To optimize on a closed interval, identify the quantity to maximize or minimize, express it using one variable, identify the allowed interval, find interior critical points, and compare their values with endpoint values. A derivative of zero is a candidate, not automatic proof of a maximum.

Worked example: A rectangle has perimeter 20 m. Let one side be x, so the other is 10 - x, with 0 <= x <= 10 for the closed comparison. Area A(x) = 10x - x^2. Its derivative is 10 - 2x, giving x = 5. A(0) = A(10) = 0, A(5) = 25. The maximum is a 5 m by 5 m square of area 25 m^2.

In an exam, state the domain and give units. Do not optimize a formula without explaining what its variables represent.

## Lesson 8 | Integration measures accumulation

An indefinite integral describes a family of antiderivatives, so include + C. For n != -1, integral x^n dx = x^(n+1)/(n+1) + C. The special case integral 1/x dx is ln|x| + C on intervals avoiding zero. Substitution reverses the chain rule.

A definite integral gives signed accumulation over an interval. If F' = f and the usual continuity conditions hold, integral from a to b of f(x) dx = F(b) - F(a). Areas below the horizontal axis contribute negatively; total geometric area may require splitting the interval and changing signs.

Worked example: A velocity v(t) = 2t + 1 m/s over 0 <= t <= 3 gives displacement [t^2 + t] from 0 to 3 = 12 m. Since v is positive throughout, distance is also 12 m. With a sign-changing velocity, displacement and distance can differ.

## Practice set | Attempt independently

1. Find the real domain of h(x) = sqrt(5-x)/(x-2).

2. For u=(2,-1), v=(1,3), find u+v, 2u and u.v.

3. Find the length of (6,8) and a unit vector in its direction.

4. Multiply [[1,3],[2,0]] by the column vector (2,4).

5. Find the determinant and inverse of [[2,1],[1,1]].

6. Solve 2x+y=8 and x-y=1. Show an elimination step and check.

7. Explain the solution set of x+y=2 and 2x+2y=5.

8. Evaluate lim as x -> 3 of (x^2-9)/(x-3).

9. Differentiate x^3 - 4x + 7 and (x^2+1)^3.
10. For position s(t)=3t^2-t, find velocity at t=2 and acceleration.
11. Find the minimum of f(x)=x^2-6x+10 on [0,4].
12. Evaluate integral from 0 to 2 of (3x^2+2) dx and explain its units if x is seconds and the integrand is m/s.

## Answers and reasoning

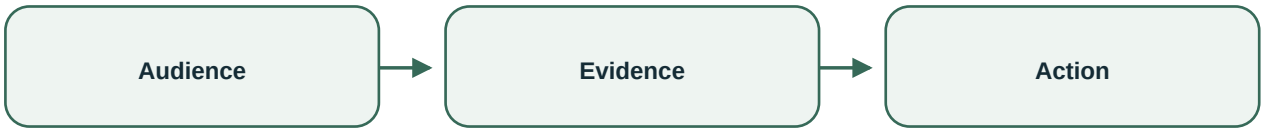
1. The square root needs x <= 5; the denominator excludes x=2. Domain: (-infinity,2) union (2,5].
2. u+v=(3,2); 2u=(4,-2); u.v=2-3=-1.
3. Length 10. Divide each component by 10: (0.6,0.8).
4. First entry 1*2+3*4=14; second 2*2+0*4=4. Result (14,4).
5. Determinant 2*1-1*1=1. Inverse [[1,-1],[-1,2]]. Multiplying by the original gives I.
6. Add the equations: 3x=9, so x=3. Then y=2. Check 6+2=8 and 3-2=1.
7. Doubling the first equation gives 2x+2y=4, contradicting 5. No solution.
8. Cancel x-3 for x != 3 to obtain x+3. The limit is 6.
9. 3x^2-4; and, by the chain rule, 6x(x^2+1)^2.
10. v(t)=6t-1, so v(2)=11; acceleration a(t)=6. Use m/s and m/s^2 if s is metres and t is seconds.
11. f'(x)=2x-6 gives x=3. Endpoint values are f(0)=10 and f(4)=2; f(3)=1 is the minimum.
12. Antiderivative x^3+2x gives 8+4=12. The result is metres, representing displacement; the integrand is positive, so distance agrees.

## Check your understanding

Explain matrix multiplication without saying "multiply corresponding entries." Explain why a limit can exist where the function is undefined. Solve a new derivative and a new integral without looking at the rules. If you can only repeat these examples, choose different coefficients and try again.

Further study: MIT 18.06 Linear Algebra, https://ocw.mit.edu/courses/18-06-linear-algebra-spring-2010/ ; OpenStax Calculus Volume 1, https://openstax.org/books/calculus-volume-1/pages/preface . These are independently published resources; the examples and exercises above were created for this pack.

# HS 8101 | Communication Skills




## Learning outcomes and route

Communication in computing means helping a specific audience understand something accurately and act on it. This module develops purposeful reading, clear paragraphs, technical explanations, reports, source use, email, presentations and collaborative feedback. Work through eight lessons and produce the written evidence requested. This is an original foundation course, not an official assessment guide.

Prerequisites: willingness to revise and to explain an idea in your own words. You can plan an explanation in English or Kiswahili, but follow the language and format required by your lecturer for submission.

## Lesson 1 | Purpose, audience and channel

Before writing, answer three questions: Who will read this? What do they already know? What should they understand or do afterwards? A lecturer assessing a lab needs a reproducible method and evidence. A classmate trying to install a compiler needs ordered instructions and a way to verify success. A manager approving a change needs the problem, options, consequences and recommendation.

A channel should fit the purpose. A short email works for a clear request; a diagram can explain relationships; a report records evidence; a live conversation helps resolve ambiguity. Do not assume more technical vocabulary makes writing more professional.

Worked example: "The application has issues" gives no decision-useful information. A clearer defect report is: "On the registration page, submitting an empty email field creates an account instead of showing a validation message. Expected: reject the form and explain the missing field. Reproduced in three attempts." It separates observation from expectation without inventing a cause.

Task: Explain a C loop to two audiences: a new student and an experienced programmer. Change the explanation, not the underlying truth.

## Lesson 2 | Active reading and useful notes

Read with a question. First preview headings, definitions and examples. Then read a short section closely. Close it and reconstruct the main idea from memory. Finally, compare your reconstruction with the source and correct gaps. Copying paragraphs is a record of reading, not evidence of understanding.

Use a note structure with four fields: concept, own explanation, original example, question remaining. Record the source immediately so you can return to it. Mark uncertainties explicitly; a question mark is better than a confident error that becomes part of your revision.

Worked example: After reading about processes, write: "A process is an executing program with its own execution context and resources. Two launches of the same executable can be two processes. Question: what is shared between threads in one process?" This is more useful than copying a definition without an example.

For revision, turn a heading into a retrieval question. Replace "Ohm's law" with "When is V=IR a suitable model, and how do I check the units?"

## Lesson 3 | Sentences and paragraphs

A useful sentence makes its actor, action and object clear. Prefer a concrete verb: "The program validates the input" is clearer than "Input validation is carried out by the program." Passive voice is still useful when the action or result matters more than the actor, as in a method description.

A paragraph develops one main point. It often begins with a topic sentence, follows with explanation or evidence, and ends by connecting the point to its purpose. Every sentence should contribute to that point. Split paragraphs when the subject changes.

Worked example: "We tested it and it was okay" becomes: "The program correctly classified 0, 50 and 100 as valid marks. It rejected -1 and 101. These boundary tests support the input-range check; they do not establish that every possible input is handled correctly." Notice the limited, precise claim.

Editing pass: remove repeated ideas, vague "this" references, unnecessary introductions and unexplained abbreviations. Keep qualifications that affect meaning. Shorter is useful only if the result remains accurate.

## Lesson 4 | Explaining a technical process

Explain the purpose before the mechanism. Introduce required terms, show the steps, then demonstrate with a small example. Distinguish what is always true from what holds only under stated assumptions. Use units, input ranges and expected outputs where relevant.

Worked example: To explain an average, first say it distributes the total equally across the count. Then show 10+20+30=60; divide by 3 to get 20. In C, explain that integer division discards the fractional part of a nonnegative result, so a real-valued average requires a floating-point operation. The concept and the language behavior are related but different ideas.

For instructions, start each step with an action and include a check: "Compile the file with the command shown. If compilation succeeds, run the executable. Confirm that the output is 20.00." Avoid "click there" without naming the control. A screenshot can support an instruction but should not be the only source of information.

## Lesson 5 | Reports and evidence

Follow the structure required by the assignment. A typical lab report includes title, objective, relevant background, apparatus or software, method, results, discussion, conclusion and references. Results state what was observed or calculated. Discussion interprets the result, considers errors and limitations, and compares it with the intended model. The conclusion answers the objective using the evidence actually collected.

Worked example: A resistor experiment might report nominal resistance 1.0 kOhm and measured resistance 0.98 kOhm. The absolute difference is 0.02 kOhm; relative to the nominal value it is 2%. This is

not automatically a malfunction. The discussion needs resistor tolerance, instrument resolution and test conditions. Do not invent measurements to make a report look complete.

A reproducible method names enough detail for another person to repeat it. For a program, record source-file version, compiler flags, test inputs and outputs. For a circuit calculation, show the diagram, assumptions, equations, substitutions and units.

## Lesson 6 | Summarizing, paraphrasing and citing

A summary selects the main points and is shorter than its source. A paraphrase restates a specific idea using your own sentence structure and wording. Neither removes the obligation to credit the source when the idea came from someone else. A quotation preserves the exact words and must be marked and cited; use quotations sparingly.

A practical workflow is read, close, write, compare, cite. Closing the source makes it harder to merely exchange a few words. Compare for accuracy, then add the source using the required referencing style. Keep the author or organization, title, year where available, URL or publication information, and any page or section needed to locate the claim.

Worked example: If an operating-systems textbook explains virtual memory, a report can cite that textbook for the concept and separately label your own diagram and example. Do not claim that a generated explanation is a verified source. Check factual claims against reliable material and follow the institution's rules on assistance and academic integrity.

## Lesson 7 | Professional email and collaborative communication

Use a specific subject, a courteous greeting, a concise context, a clear request and a close. Supply the details the recipient needs, but avoid unnecessary personal information. Send a follow-up after a reasonable interval appropriate to the deadline; do not interpret silence as approval.

Worked example:

Subject: CS 8138: clarification on file-input format

Dear Lecturer, I am working on the file-processing exercise. The brief names a marks file but does not state whether each line contains one mark or a student name followed by a mark. Could you confirm the required format? I have attached the relevant part of the brief for context. Thank you, Privatus Cosmas

For group work, record decisions, owners and due dates. "Someone will finish the report" is not an assignment. "Asha will draft the method by Friday; Juma will check the results table on Saturday" makes responsibility visible. Use actual agreed names and dates, not assumptions.

## Lesson 8 | Presentations, listening and feedback

A presentation needs a clear question, a small number of supporting points and a conclusion the audience can remember. Begin with the problem, show one concrete example, then explain the result. Slides should support speech rather than reproduce an essay. Label diagrams and keep enough contrast and text size for the room.

Practise to time. A three-minute explanation might allocate 30 seconds to context, 90 seconds to the example, 40 seconds to interpretation and 20 seconds to the takeaway. These are planning proportions,

not compulsory rules. Rehearse with a timer and adjust to your actual pace.

Worked example: Present the integer-division bug by showing the expected average, the original expression, its actual output and the corrected expression. Ask the audience to predict the result before revealing it. During questions, restate the question to check understanding. Say what you know and what needs checking; guessing is not professionalism.

Useful feedback describes the artifact and its effect: "The graph lacks units, so I cannot interpret the slope." It is more actionable than "This is bad." Listen without interrupting, clarify the requested change and decide how to revise.

## Practice set | Produce evidence

1. State an audience and intended action for a compiler-installation guide.
2. Rewrite: "The system is very good and works perfectly."
3. Write a four-sentence paragraph explaining why boundary tests matter.
4. Turn "memory management" into three retrieval questions.
5. Separate observation and inference: "The app crashed because the database is broken."
6. Design a results table for five C test cases.
7. Explain the difference between results and discussion in a lab report.
8. Outline a citation record for an online textbook chapter.
9. Write a professional email asking a lecturer to clarify an ambiguous submission format.
10. Create a three-minute presentation outline on a topic from another module.
11. Give constructive feedback on a diagram with no labels or units.
12. Write a 150-word explanation of one concept, then shorten it by about one third without losing essential meaning.

## Answer guide and assessment rubric

1. Example: first-year students who need to compile and run one C file. Intended action: run a supplied command and verify the expected output. Specify the operating system.
2. Replace praise with evidence: "The program passed the five listed tests. Performance and invalid-file handling have not yet been evaluated." The exact claim depends on actual tests.
3. Include a topic sentence, a boundary example such as 0 and 100, an explanation of what these cases can expose, and a limit on the conclusion. Do not claim that five tests prove every case.
4. Examples: What does virtual memory give a process? How does an address translate? Why can a page fault be expensive? Questions should require an explanation, not only a term.
5. Observation: the app crashed. Inference: a database problem caused it. State the steps, error message and evidence needed before assigning cause.

6. Columns: case ID, input, expected result, actual result, pass/fail and explanation. A useful test table makes disagreement visible.
7. Results document findings. Discussion explains their meaning, limitations and relation to the objective. A screenshot alone rarely supplies an adequate discussion.
8. Record organization or author, title, date if present, chapter or section, URL and access date when required by the selected style. Use the lecturer's specified citation format.
9. A strong answer includes a specific subject, relevant context, one answerable question, courtesy and identification. Do not invent an emergency or blame the lecturer.
10. A strong outline has one central idea, one worked example, clearly timed sections and a final takeaway. Demonstrate the explanation aloud.
11. Example: "Please label both axes and add voltage/current units. Without them I cannot tell whether the plotted slope represents resistance or conductance."
12. Assess accuracy, audience fit, structure, evidence and concision, each from 0 to 2. A score is a self-assessment aid, not an institutional grade. Compare drafts and identify one concrete improvement.

## Further study and weekly artifact

Read Google Technical Writing One, https://developers.google.com/tech-writing/one , and Purdue OWL's paragraph guidance, https://owl.purdue.edu/owl/general_writing/academic_writing/paragraphs_and_paragraphing/index.html . Use the sources as further reading; all examples in this pack are original.

Keep one revised artifact each week: email, explanation, test report, diagram caption or presentation outline. Preserve the first draft so improvement is visible.

# CS 8101 | Theory of Computation




Even number of 1s: start in E, accept E.

## Learning outcomes and route

This module asks what computational models can recognize, what problems are solvable in principle, and how we reason about those claims. Learn to specify a language, build and trace finite automata, translate simple regular expressions, read grammars and distinguish computability from efficiency. The eight lessons form a foundation route; the lecturer's outline determines assessment depth.

Prerequisites: sets, functions, logical statements and simple proofs. A machine diagram without a precise language description is not a complete specification.

## Lesson 1 | Sets, alphabets, strings and languages

An alphabet Sigma is a finite set of symbols, such as {0,1}. A string is a finite sequence of symbols from that alphabet. Its length is written |w|. The empty string, epsilon, has length zero. Sigma* is the set of all finite strings, including epsilon. A language over Sigma is any subset of Sigma*; it need not contain all strings or contain epsilon.

Worked example: Over Sigma={0,1}, let L be the strings that end in 01. Then 01, 101 and 1101 are in L, while epsilon, 0, 10 and 011 are not. State whether the empty string is accepted whenever its treatment affects your construction.

Concatenation joins strings in order. For u=10 and v=01, uv=1001, but vu=0110. Language union combines members; intersection keeps shared members. The empty language has no strings. It is different from {epsilon}, which has one member.

## Lesson 2 | Deterministic finite automata

A DFA has a finite set of states Q, an alphabet Sigma, a transition function delta: Q x Sigma -> Q, a start state q0 and a set F of accepting states. Each state-symbol pair has exactly one next state. A string is accepted if, after consuming all its symbols, the machine is in an accepting state.

Worked example: Recognize binary strings with an even number of 1s. Use E for even and O for odd. Start in E, and make E accepting. Reading 0 leaves the parity unchanged; reading 1 toggles E and O.

| State | Input 0 | Input 1 | Accepting? |
|---|---|---|---|
| E (start) | E | O | Yes |
| O | O | E | No |

Trace 1011: E -> O -> O -> E -> O, so reject. Trace 1010: E -> O -> O -> E -> E, so accept. Epsilon is accepted because the initial number of 1s is zero, which is even.

## Lesson 3 | State meaning is more useful than guessing circles

When designing a DFA, state what information must be remembered from the prefix already read. For "ends in 01," remember whether the current suffix is empty or irrelevant, is 0, or is 01. These are meanings of states, not just labels.

Worked example: q0 means no useful suffix; q1 means the prefix ends in 0; q2 means it ends in 01. Start in q0 and accept only q2.

| State | Input 0 | Input 1 |
|---|---|---|
| q0 | q1 | q0 |
| q1 | q1 | q2 |
| q2 | q1 | q0 |

Trace 1001: q0 -> q0 -> q1 -> q1 -> q2, so accept. Trace 010: q0 -> q1 -> q2 -> q1, so reject. Repeated zeroes matter: from q1 on 0, stay in q1 because the most recent symbol still gives a useful 0 suffix.

To justify the construction, explain how each transition preserves the stated meaning. Then test short accepted and rejected strings, including epsilon. Tests help find mistakes; an invariant argument explains all input lengths.

## Lesson 4 | Nondeterminism and subset construction

An NFA may allow several next states for a symbol, or none. Some definitions also permit epsilon transitions that consume no input. A string is accepted when at least one complete path consumes the string and ends in an accepting state. It is not necessary for every path to accept.

Worked example: To recognize strings containing 01 somewhere, an NFA can remain in a scanning state while reading any symbol, branch to a second state after seeing a 0, then enter an accepting state when the next symbol is 1. Once the substring has been found, it may consume remaining symbols in an accepting loop.

Subset construction converts an NFA to a DFA. A DFA state represents a set of possible NFA states. Start with the epsilon closure of the NFA start state; for each symbol, collect reachable states and then take epsilon closure again. A subset is accepting if it contains at least one NFA accepting state. Include the empty subset when it is reachable; it acts as a rejecting sink.

Nondeterminism changes the description, but DFAs and NFAs recognize the same class of languages: the regular languages. This does not mean that every language is regular.

## Lesson 5 | Regular expressions and closure

In formal-language notation, union is often written with |, concatenation by adjacency, and repetition by *. The star permits zero repetitions. These are mathematical regular expressions; features such as backreferences in some programming-language regex engines go beyond this model.

Worked example: (0|1)*01 describes all binary strings ending in 01. The starred prefix can be any binary string, including epsilon. By contrast, (01)* describes epsilon, 01, 0101 and so on. They are different languages.

Regular languages are closed under union, intersection, concatenation, star and complement relative to the chosen Sigma*. For complementing a DFA, first ensure that its transition function is total, adding a sink if necessary, then exchange accepting and nonaccepting states. Forgetting the alphabet or missing transitions can give a wrong complement.

For intersection, a product construction tracks both DFA states at once. Accept when both components accept. A state pair means "where both machines would be after reading the same prefix."

## Lesson 6 | Why some languages need more memory

A finite automaton has a fixed amount of state memory, regardless of input length. Some tasks require keeping track of unbounded information. The language {0^n 1^n : n >= 0} requires matching the number of zeroes and ones, and is not regular.

The pumping lemma gives a necessary property of regular languages. If L is regular, there is a pumping length p such that every sufficiently long string s in L can be split s=xyz with |xy| <= p, |y| > 0 and xy^i z in L for all i >= 0. To prove nonregularity using it, assume regularity, choose a suitable string, and handle every permitted decomposition.

Worked example: Choose s=0^p1^p. Every valid y lies among the first p zeroes and contains at least one zero. Pumping down with i=0 removes zeroes but no ones, so the result is not in L. This contradicts the required property.

Do not use the lemma backwards: finding one pumpable string does not establish regularity. A regular-language construction, such as a DFA, is a more direct positive proof.

## Lesson 7 | Grammars and pushdown intuition

A grammar generates strings using production rules. A context-free grammar has a single nonterminal on the left of each rule. Derivation repeatedly replaces a nonterminal using an available production. A parse tree records the structure of the derivation.

Worked example: S -> 0S1 | epsilon generates exactly 0^n1^n for n >= 0. To derive 0011: S -> 0S1 -> 00S11 -> 0011. Each use of 0S1 adds one zero at the front and one one at the back. The epsilon rule stops the expansion.

A pushdown automaton adds a stack, which provides a last-in, first-out form of unbounded memory. For 0^n1^n, push a marker for each 0 and pop one for each 1, enforcing the correct phase order. This explains why the language can be context-free even though it is not regular.

Ambiguity means that some string has more than one parse tree or appropriate derivation structure. A grammar for arithmetic may need precedence and associativity rules; recognizing valid tokens is not enough to determine the intended expression structure.

## Lesson 8 | Computability and complexity are different questions

A Turing-machine model adds an unbounded tape and a head that can read, write and move. It is a mathematical model, not a claim that real computers have infinite physical memory. A decider halts on every input and answers yes or no. A recognizer must accept members, but may run forever on some nonmembers.

The halting problem asks whether an arbitrary program halts on an arbitrary input. There is no algorithm that correctly decides all such program-input pairs. This does not mean that we can never prove a particular program halts; simple bounded loops are often easy to analyze.

Complexity asks about resource use, such as time or memory, for a solvable task as input size grows. A linear scan has a different growth pattern from checking all subsets. Being computable does not automatically mean being practical. Big-O gives an asymptotic upper bound, not the exact running time on a particular laptop.

Worked comparison: Determining whether a finite binary string ends in 01 is decidable and can be handled by a DFA. Proving that a general-purpose halting checker cannot exist concerns impossibility across all programs. These are fundamentally different kinds of claims.

## Practice set

1. Distinguish epsilon, the empty language, and {epsilon}.
2. List all binary strings of length exactly 2.
3. Trace 11010 through the even-ones DFA. Accept or reject?
4. Trace 00101 through the ends-in-01 DFA.
5. Design a DFA for strings with an odd number of 0s. State whether epsilon is accepted.
6. Give a formal regular expression for binary strings starting with 1 and ending with 0.
7. Is epsilon in (01)*? Is 010 in it? Explain.
8. How do you identify accepting states during subset construction?
9. What must you fix before complementing an incomplete DFA?
10. Give a derivation of 000111 using S -> 0S1 | epsilon.
11. Explain why the pumping argument for 0^n1^n must consider every allowed split.
12. Distinguish a decider, a recognizer, and an efficient algorithm.

## Answers and reasoning

1. Epsilon is a string of length zero. The empty language has no members. {epsilon} has one member, the empty string.
2. 00, 01, 10 and 11. There are 2^2=4 possibilities.
3. E -> O -> E -> E -> O -> O. Reject: three 1s were read.
4. q0 -> q1 -> q1 -> q2 -> q1 -> q2. Accept: the final suffix is 01.

5. Use Even0 and Odd0. Start in Even0; accept Odd0. On 0 toggle; on 1 stay. Reject epsilon because zero zeroes is even.

6. 1(0|1)*0. The shortest accepted string is 10. A one-symbol string cannot satisfy both required endpoints here.

7. Epsilon is included by zero repetitions. 010 is not a whole number of repetitions of 01.

8. A subset state accepts if it contains any accepting state of the NFA. Do not require every state in the subset to accept.

9. Make transitions total over the declared alphabet, using a rejecting sink for missing transitions, then swap final/nonfinal states.

10. S -> 0S1 -> 00S11 -> 000S111 -> 000111.

11. The lemma allows a decomposition to exist if the language is regular. To contradict it, the chosen string must fail for all valid decompositions; disproving one arbitrary split is insufficient.

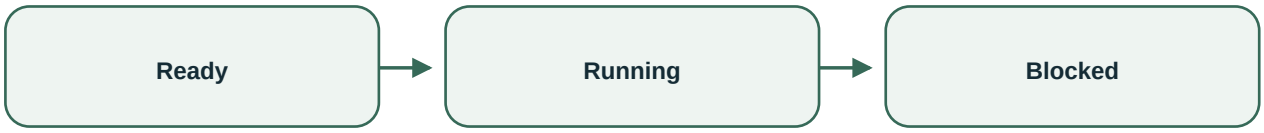
12. A decider always halts with a decision. A recognizer may fail to halt on nonmembers. Efficiency concerns the resources an algorithm uses; termination alone does not establish a useful resource bound.

## Mastery checkpoint and further study

Build a new DFA from a precise specification, state the meaning of every state and trace five strings. Explain why your construction handles all prefixes. Then contrast finite-state memory with stack memory using your own example.

Further study: MIT Automata, Computability, and Complexity, https://ocw.mit.edu/courses/6-045j-automata-computability-and-complexity-spring-2011/ . The original lessons here are a foundation pathway; consult the course notes and your lecturer for formal proof depth.

# CS 8140 | Operating Systems




Selected transitions only: dispatch, wait; completion returns blocked work to ready.

## Learning outcomes and route

An operating system coordinates access to hardware and provides abstractions that make programs usable. This module develops a working understanding of processes, scheduling, memory, concurrency, files and protection. Use the eight lessons, paper calculations and a safe shell sandbox. These foundation topics are inferred from the module title, not an official weekly outline.

Prerequisites: the difference between source code and an executable, binary units, and basic C variables and functions. A terminal is an interface to a shell; it is not itself the operating system.

## Lesson 1 | Abstractions and the kernel

The kernel is the privileged core that manages resources and enforces protection. User applications normally request services through system calls rather than directly controlling devices. Examples include creating a process, reading a file or requesting memory mappings. A library function may wrap a system call, perform work entirely in user space, or do both.

Useful abstractions include a process for executing work, virtual memory for a process's address space, a file for named persistent data and a thread for a stream of execution. Abstractions simplify programming, but their implementation still has costs and limits.

Worked example: A program writes a line to a file. At the application level, this looks like a call to a standard I/O function. Beneath that interface, buffering, kernel calls, filesystem metadata and storage operations may be involved. Returning from a buffered write is not necessarily proof that data has reached durable storage. Distinguish the interface promise from assumptions about its implementation.

## Lesson 2 | Programs, processes and threads

A program is stored instructions and data. A process is an executing instance with execution state and resources. The same executable can run in several processes. Typical process states include running, ready and blocked; specific operating systems use more detailed state models.

A running process may become ready when preempted, blocked when waiting for I/O, and ready again when the event completes. A context switch saves one execution context and restores another. This is overhead needed to share execution resources, not application work itself.

Threads within a process share an address space and many resources, while each thread has its own execution context and stack. Sharing makes communication convenient, but also creates coordination problems. Processes normally have stronger address-space separation, although they can communicate

through controlled mechanisms.

Worked example: An editor can have one thread for the interface and another for background indexing. A blocked indexing operation need not stop all useful interface work if the design and runtime allow progress. This does not mean threads automatically make every program faster.

## Lesson 3 | Scheduling and performance measures

A scheduler chooses among ready work. First-come, first-served uses arrival order. Shortest-job-first favors the shortest known burst in its nonpreemptive form. Round robin gives each ready process a quantum and rotates unfinished work. Real schedulers must balance responsiveness, throughput, fairness and overhead; no single policy is ideal for every workload.

Turnaround time = completion time - arrival time. Response time = first execution time - arrival time. For simple single-burst exercises with no I/O, waiting time = turnaround time - CPU burst. State these assumptions before using that shortcut.

Worked example: All jobs arrive at time 0. A needs 5 units, B needs 3, C needs 1. FCFS in order A,B,C completes them at 5,8,9. Turnaround times are 5,8,9 and waiting times 0,5,8; average waiting is 13/3. SJF runs C,B,A, completing C at 1, B at 4 and A at 9. Waiting times for C,B,A are 0,1,4, averaging 5/3. These numbers depend on the stated arrival pattern and no switching cost.

## Lesson 4 | Virtual memory and address translation

Virtual addresses belong to a process's address space. Hardware and the operating system translate them to physical memory using structures such as page tables. Paging divides the address space into fixed-size pages and physical memory into corresponding frames. The offset identifies a position within a page; the page number identifies which translation to use.

Worked example: With 4096-byte pages, virtual address 10000 has page number floor(10000/4096)=2 and offset 1808. If virtual page 2 maps to physical frame 7, the physical address is 7*4096+1808=30480.

A translation lookaside buffer caches address translations. A page fault means the current access requires operating-system handling; it does not always mean a programming error. A valid page may need loading, while an invalid or prohibited access may lead to a fault signal or termination. Virtual memory does not create unlimited RAM: sustained memory pressure can make a system slow or unable to satisfy requests.

## Lesson 5 | Allocation, locality and replacement

Allocation decides which memory regions are available to programs and how they are tracked. Internal fragmentation is unused space inside an allocated unit; external fragmentation is free space split into separate regions that cannot satisfy a larger contiguous request. Paging addresses some external-contiguity problems but introduces its own costs and internal waste.

Locality explains why caching can work. Temporal locality means recently used data may be used again. Spatial locality means nearby data may be accessed soon. Programs scanning arrays in order often behave differently from programs touching unpredictable addresses.

Worked example: Suppose two frames use FIFO replacement and the reference string is 1,2,1,3,1. Load 1: fault. Load 2: fault. Read 1: hit. Read 3: fault, replacing the oldest loaded page 1. Read 1: fault, replacing page 2. Total: four faults. The hit on page 1 did not refresh its position in FIFO. Under LRU, accesses do

update recency; do not confuse the policies.

## Lesson 6 | Concurrency, races and synchronization

A race occurs when correctness depends on uncontrolled ordering or timing between concurrent operations. Even a simple increment can involve reading, modifying and writing shared state. A critical section is the part that must be coordinated to preserve an invariant. A mutex can ensure only one participating thread enters a protected region at a time.

Worked example: Shared count starts at 0. Thread A reads 0, thread B reads 0, A writes 1, B writes 1. Two increments have produced 1 rather than 2. Protecting the complete read-modify-write operation with the same mutex prevents that interleaving among threads following the protocol.

A lock should protect a clearly stated invariant, not just an arbitrary line. Condition variables help wait for a predicate while releasing and later reacquiring a related mutex. Predicates are generally checked in a loop because wakeups do not guarantee the desired condition still holds. Semaphores track permits and can support resource limits, but using a semaphore does not by itself make a design correct.

## Lesson 7 | Deadlock and progress

Deadlock is a situation where participants cannot proceed because each waits for resources or actions held up by others. The classic necessary conditions are mutual exclusion, hold-and-wait, no preemption and circular wait. Breaking a necessary condition can prevent this class of deadlock, but the method may impose other costs.

Worked example: A holds lock X and waits for Y; B holds Y and waits for X. Both follow their local plans, but neither can progress. A consistent global lock order, such as always acquiring X before Y, prevents that particular cycle when all participants follow the order.

Starvation is different: a participant waits indefinitely while others continue to make progress. Livelock involves activity without useful progress, such as two participants repeatedly yielding to one another. Describe the observed progress pattern before naming the problem.

Practical rule: minimize the resources held while waiting, define ownership clearly, and avoid assuming that adding more locks automatically improves correctness.

## Lesson 8 | Files, permissions and a safe shell lab

A filesystem organizes named data and metadata. A directory maps names to entries; a path identifies a location. An open file descriptor or handle refers to an opened resource. Filesystems and permission models differ, so Linux examples do not imply every operating system has identical behavior.

On Unix-like systems, the common mode bits give read, write and execute permissions for owner, group and others. For directories, execute generally permits traversal, read permits listing names, and write permits changes to directory entries subject to other rules. File deletion depends strongly on the containing directory's permissions, not simply on whether the file itself is writable.

Safe lab: Work only inside a new practice folder. Use pwd to confirm location, mkdir for a directory, printf to create a small text file, cat to read it, and ls -l to inspect permissions. Try chmod 600 on your own practice file. Use ps to inspect your processes without terminating anything. Do not run destructive cleanup commands or change system permissions to complete this lab.

Worked example: Mode 640 is owner read+write (4+2), group read (4), others none (0): rw-r-----. It does not grant execute permission. A script's executable bit and its interpreter are separate considerations.

## Practice set

1. Explain kernel versus shell in two sentences.
2. Can one program correspond to two processes? Give an example.
3. State a resource threads commonly share and one item each thread needs separately.
4. Why does blocked differ from ready?
5. FCFS: A arrives 0, burst 4; B arrives 1, burst 2; C arrives 2, burst 1. Find completion, turnaround and waiting times. Assume no overhead and no I/O.
6. All arrive at 0: A needs 3, B needs 2. Round robin, quantum 1, starting A then B, no overhead. Give the timeline and completion times.
7. With 1024-byte pages, split address 3500 into page number and offset.
8. If that page maps to frame 5, calculate the physical address.
9. Two frames, FIFO, reference string 1,2,1,2,3. Count faults.
10. Give an interleaving that loses one of two increments from an initial count of 4.
11. Explain a lock-order solution for an X/Y deadlock.
12. Translate mode 750 into owner/group/other permissions. Explain why applying it to a directory is not the same as applying it to a regular file.

## Answers and reasoning

1. The kernel manages resources and protection at a privileged level. A shell reads commands and launches programs or performs its own built-in operations.
2. Yes. Opening two separate instances of the same executable can create two processes with different execution states and resources.
3. Threads share the process address space. Each needs its own stack and execution context, including a program-counter state.
4. A blocked task is waiting for an event such as I/O. A ready task can run when the scheduler assigns execution resources.
5. A runs 0-4, B 4-6, C 6-7. Completion A=4,B=6,C=7. Turnaround 4,5,5. Waiting 0,3,4.
6. A 0-1, B 1-2, A 2-3, B 3-4, A 4-5. B completes at 4; A at 5.
7. 3500=3*1024+428, so page 3, offset 428.
8. 5*1024+428=5548. Frame number replaces page number; offset is preserved.
9. Faults on 1,2,3; hits on the repeated 1 and 2. Total three faults. Page 1 is replaced at the final access.
10. Both read 4; A writes 5; B writes 5. Final 5 rather than 6. The update needs coordination.

11. Require all code needing both locks to acquire X before Y. A participant cannot hold Y while waiting for X under this protocol, breaking the circular pattern.
12. Owner rwx, group r-x, others --- . On a directory, x means traversal and r means listing; on an ordinary file, x means permission to execute, subject to other requirements.

## Further study and evidence

Use the authors' free OSTEP chapters at https://pages.cs.wisc.edu/~remzi/OSTEP/ for deeper treatment. Consult your local man pages for commands on your machine; Linux and macOS details may differ. Produce one annotated scheduling timeline, one address-translation calculation and one terminal transcript from the sandbox. Original explanations and exercises in this pack are not copied from OSTEP.

# CS 8138 | Programming Concepts in C




## Learning outcomes and route

Learn to turn a problem into a precise algorithm, express it in C, compile with warnings, test boundary cases and explain the result. This foundation course covers types, control flow, functions, arrays, strings, pointers, structures, files and debugging. The supplied practice folder contains standalone C17 programs and a test runner. They use the standard library, not a course-specific library.

Prerequisites: basic arithmetic and a willingness to trace variables on paper. Start with small correct programs; do not begin with a large menu system whose parts you cannot explain.

## Lesson 1 | From problem to executable

An algorithm is a precise sequence of steps that transforms allowed inputs into required outputs. Before coding, state the inputs, output, constraints and exceptional cases. Pseudocode helps separate the logic from language syntax.

A C source file is translated and linked into an executable. Compilation errors differ from runtime failures and wrong answers. Use warnings to expose suspicious code early. In the practice folder, compile with a command such as:

```
clang -std=c17 -Wall -Wextra -Wpedantic average.c -o average
./average
```

Worked example: To find the maximum of three numbers, start with the first as the current maximum; compare the second and update if larger; repeat for the third. Starting at zero would fail if all inputs were negative. A correct initial value should follow from the problem, not from habit.

A minimal program uses #include <stdio.h>, an int main(void) function, statements within braces, and a return value. A semicolon terminates many statements; braces group a block. Keep indentation consistent so the structure can be inspected.

## Lesson 2 | Types, expressions and input

A variable has a type that constrains its representation and operations. int is suitable for many whole-number calculations; double represents floating-point values approximately; char represents a character-sized integer type. Exact widths and ranges depend on the implementation. Do not assume every integer is 32 bits. Use limits from the implementation when range matters.

Integer division is particularly important. If both operands are integers, 7/2 produces 3, not 3.5. Assigning the result to a double does not restore the discarded fraction. Use 7.0/2 or convert an operand before division. The remainder operator % applies to integer types. A zero divisor is invalid and must be handled before division.

Worked example: marks are 70, 71 and 72. Sum is 213, average is 71.0. To reveal an integer-division bug, choose a sum not divisible by the count: 70+71+73=214, and 214/3 as integers produces 71, while 214/3.0 gives about 71.3333. A good test distinguishes the incorrect and correct algorithms.

For interactive input, check whether conversion succeeds and whether the value is in the allowed range. The supplied marks example reads a line with fgets and checks integer conversion with strtol. Do not use gets; it cannot enforce the buffer size.

## Lesson 3 | Selection and Boolean conditions

An if statement chooses a branch when its condition is nonzero. A comparison such as x == 0 tests equality; x = 0 assigns a value. Logical AND is &&, logical OR is ||, and logical NOT is !. These differ from the bitwise operators &, | and ~.

Worked example: A valid mark must satisfy mark >= 0 && mark <= 100. Writing 0 <= mark <= 100 does not express a mathematical chained inequality in C. The first comparison yields 0 or 1, and the second comparison then compares that result with 100. Write both comparisons explicitly.

Check ranges in a sensible order. For classification, validate the input first, then apply the categories. Avoid using overlapping independent if statements when exactly one category should be chosen. An if / else if / else chain expresses mutual choice.

Short-circuit behavior matters. In divisor != 0 && total/divisor > 2, the division is not evaluated when divisor is zero. This is useful, but the condition must still match the intended logic. Never rely on unspecified evaluation order of unrelated expressions.

## Lesson 4 | Loops, invariants and tracing

A for loop makes initialization, continuation condition and update visible together. A while loop checks its condition before every iteration; a do-while checks after the body. Choose the form that makes termination and state changes easiest to understand.

A loop invariant is a property that remains true at the relevant point in each iteration. For summing the first i array elements, the invariant can be: total equals the sum of elements with indices 0 through i-1. Initialize total to zero before the first iteration; add the next element; advance i. At i=n, the invariant yields the required whole-array sum.

Worked example: Sum the integers from 1 to 4. Before the loop, total=0. After i=1, total=1; after 2, total=3; after 3, total=6; after 4, total=10. Trace the actual update order rather than writing only the final answer. A condition i < 4 would omit the last term.

Watch for off-by-one errors, uninitialized accumulators, a condition that never changes, and updating the wrong variable. For very large data, also consider integer overflow and whether the chosen type is adequate.

## Lesson 5 | Functions, scope and contracts

A function packages a coherent operation. Its declaration communicates parameter types and return type. Its contract states what callers must provide and what the function promises. C passes argument values; modifying a parameter variable does not modify the caller's separate variable.

Worked example: int square(int x) returns x*x for values whose square is representable as int. The range qualification matters: a mathematically valid square can overflow the chosen integer type. A function that sums a fixed small array can have a straightforward contract; a production function may need explicit overflow handling.

Prefer one responsibility per function. Separate reading input from calculating a result when this makes testing easier. A calculation function can then be checked with known inputs without simulating a keyboard.

Local variables normally exist within their block's scope. Do not return the address of an automatic local variable; its lifetime ends when the function returns. Use meaningful names such as count, capacity and total instead of assigning every variable a single letter.

## Lesson 6 | Arrays, strings and bounds

An array holds a fixed number of elements of the same type. For n elements, valid indices are 0 through n-1. C does not automatically check ordinary array subscripts. Accessing outside the array is not a reliable way to obtain a convenient value; it is a program error with potentially undefined behavior.

A C string is a sequence of characters terminated by a zero byte, '\0'. A buffer must have space for both the visible characters and this terminator. The newline character '\n' is different from the string terminator. A character literal uses single quotes; a string literal uses double quotes.

Worked example: char word[4] = "cat" contains 'c', 'a', 't', '\0'. It has capacity 4 but visible string length 3. Storing four visible letters as a string would require at least five elements. The sizeof operator measures the array object in bytes where its type is still an array; strlen counts visible characters before the terminator.

When arrays are passed to ordinary function parameters, the parameter does not retain the array length automatically. Pass a count or another explicit bound. The supplied array program demonstrates an index loop and a separate count.

## Lesson 7 | Pointers, structures and data ownership

A pointer holds an address of an object or function, or a null pointer value, according to its type. &x obtains the address of x; *p accesses the pointed-to object when p is valid for that access. A pointer does not automatically make the pointed-to object's lifetime longer.

Worked example: int x=4; int *p=&x; *p=9; changes x to 9 because p points to x. In a swap function, pass addresses of the caller's variables and exchange the pointed-to values. Passing copies of the integers and swapping only those copies would leave the originals unchanged.

A struct groups related fields, such as a student identifier and mark. Use . for a structure object and -> for a pointer to a structure. Decide who owns a resource and when it remains valid. Dynamic memory obtained with malloc must be checked for allocation failure and later released with free when no longer needed;

using freed storage or freeing it twice is invalid. This pack's core exercises use bounded arrays to keep introductory practice focused.

Do not memorize pointer syntax without drawing a small memory picture. Label the variable, the pointer and the object it refers to.

## Lesson 8 | Files, testing and debugging

A file operation can fail. Check fopen before using its result, check read/write outcomes where relevant, and close an opened stream. Choose the mode intentionally: opening a file with "w" can truncate an existing file. The supplied file example writes only in a newly created test directory.

Worked example: To write three marks, open a practice file, write each mark on a separate line, check success, close the file, reopen for reading and verify the sum. The program should report failure rather than pretending success when the file cannot be opened. Never point a practice program at an important file.

Test valid, boundary and invalid cases. A mark parser should consider 0,100,-1,101, a nonnumeric string and a numeric string with unwanted trailing characters. For an average, use a total that is not evenly divisible by the count. Explain what each case is intended to expose.

Debug systematically: reproduce the failure, reduce the example, inspect warnings, trace state, change one cause, and rerun both the failing case and earlier passing cases. A program that compiles is not necessarily correct. Sanitizers can help find memory errors when supported; they do not prove correctness.

## Practice set

1. Write an input-output contract for a function finding the minimum of three integers.
2. Predict 9/4, 9%4 and 9/4.0.
3. Why does double mean = sum/count fail when both operands are int and the total is not divisible by the count?
4. Write a correct condition for an integer between 10 and 20 inclusive.
5. Trace a loop adding 2,4,6 to an initially zero total. Show each state.
6. What is wrong with indexing a five-element array using i <= 5 starting at i=0?
7. How many elements are required to store "Mbeya" as a C string?
8. A function receives int *p and performs *p=12. Under what conditions can this change a caller's variable safely?
9. Why is returning the address of a local automatic int invalid for later access?
10. Design six tests for a mark parser accepting integers 0-100.
11. Explain how to avoid accidentally truncating a personal file in a file-writing exercise.
12. Build the supplied C examples, run their tests and modify one program to handle a new requirement. Record your change and a test for it.

## Answers and reasoning

1. Example: accepts three representable int values and returns a value no larger than any of them and equal to at least one input. Ties are valid.

2. 2,1,2.25. The first two use integer arithmetic; the third uses a floating-point operand.

3. Integer division happens before assignment. Use (double)sum/count after ensuring count is nonzero and the sum has not overflowed.

4. x >= 10 && x <= 20. Do not use a chained mathematical comparison.

5. Before iteration 1 total=0; after adding 2 total=2; after 4 total=6; after 6 total=12.

6. It attempts indices 0,1,2,3,4,5. Index 5 is outside the array. Use i < 5 for that range.

7. Six: five letters plus '\0'.

8. p must point to a live, writable int object that the caller allows the function to change. It must not be null, dangling or otherwise invalid.

9. The object's lifetime ends on return. An address value does not preserve that object's validity.

10. 0 and 100 accept; -1 and 101 reject; "abc" rejects; "42x" rejects. Also consider empty input and unusually long input when extending the parser.

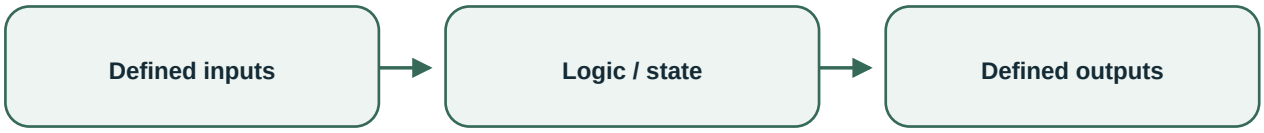
11. Use a fresh sandbox directory and a fixed practice filename. Understand the file mode, check errors and never reuse an important existing path.

12. Evidence must include compiler output, expected/actual results and your changed source. A pass from the provided test runner verifies its listed cases, not every possible input or platform.

## Further study

Harvard CS50 C materials: https://cs50.harvard.edu/x/weeks/1/ . C language reference for precise semantics: WG14 N1570 committee draft, https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf . N1570 is a C11-era draft, not the latest language edition; these examples intentionally use the widely supported C17 mode and standard-library foundations. The practice files and explanations are original.

# CS 8102 | Basics in Digital Circuit




## Learning outcomes and route

Digital circuits use physical signals to represent discrete logical values. Learn to convert representations, reason with Boolean algebra, build combinational functions and explain how sequential circuits store state. This foundation module has eight lessons, original examples and a practice set. Use paper or a simulator unless your instructor supervises appropriate hardware work.

Prerequisites: powers of two, basic algebra and careful table reading. A logical 1 is an abstraction, not a universal voltage value. Valid voltage ranges depend on the logic family and supply.

## Lesson 1 | Number systems and representation

In base b, a digit's place value is a power of b. Binary uses 0 and 1; hexadecimal uses 0-9 and A-F. Hexadecimal is convenient because one hex digit corresponds exactly to four bits. Leading zeroes do not change an unsigned value, but the chosen width matters for arithmetic and signed interpretation.

Worked example: 45 decimal = 32+8+4+1, so its six-bit binary form is 101101. Group into four-bit blocks with leading zeroes: 0010 1101, giving hex 2D. Checking back, 2*16+13=45.

To convert a fractional binary value, use negative powers of two. 0.101 binary = 1/2 + 0/4 + 1/8 = 0.625 decimal. Not every decimal fraction has a finite binary representation. The stored bit pattern and its interpretation must be specified; the same bits can represent an unsigned integer, a signed integer or part of another encoding.

## Lesson 2 | Binary arithmetic and two's complement

Binary addition follows carry rules: 1+1 gives sum bit 0 and carry 1. For an n-bit unsigned value, the representable range is 0 through 2^n-1. An n-bit two's-complement value represents -2^(n-1) through 2^(n-1)-1. Interpret the most significant bit with negative weight when reading two's complement.

Worked example: In 8-bit two's complement, -5 is found by inverting 00000101 to 11111010 and adding 1, giving 11111011. Adding this to 00001001 (9) gives 1 00000100. Discarding the carry beyond 8 bits yields 00000100, which is 4.

Carry-out and signed overflow are different concepts. Adding 01111111 and 00000001 gives 10000000 in 8 bits: an unsigned result of 128 fits, but the signed mathematical result 128 is outside the two's-complement range, so signed overflow occurred. Always state width and interpretation before judging overflow.

## Lesson 3 | Gates and truth tables

The basic logical operations include NOT, AND, OR and XOR. AND outputs 1 only when both inputs are 1. OR outputs 1 when at least one input is 1. XOR outputs 1 when exactly one of two inputs is 1. NAND negates AND; NOR negates OR; XNOR negates XOR.

| A | B | AND | OR | XOR | NAND |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Worked example: An alarm activates when a door is open AND the system is armed. Let D=door-open and A=armed. The specification is Y=D AND A. A truth table prevents confusing this with OR, which would activate the alarm whenever the system was armed, even with a closed door.

Write the meaning of 0 and 1 for each input before building the table. A label such as "door" alone is ambiguous: does 1 mean open or closed?

## Lesson 4 | Boolean algebra and equivalent circuits

Boolean algebra manipulates logic functions. Common laws include X+0=X, X*1=X, X+X=X, X*X=X, X+X'=1 and X*X'=0. Here + means OR, * means AND and ' means NOT; these symbols are not ordinary arithmetic in this context.

De Morgan's laws are (AB)'=A'+B' and (A+B)'=A'B'. They help convert between gate structures and reason about negated conditions. NAND and NOR are each functionally complete: suitable combinations can implement any Boolean function.

Worked example: Y=AB+AB'. Factor A to obtain A(B+B')=A*1=A. Four truth-table rows confirm that the output depends only on A. The simplified implementation needs only a connection carrying A, rather than redundant gates.

Be careful with distribution and negation. (A+B)' is not A'+B'. Write every complement over its intended expression, using parentheses in plain-text notes so the scope is unambiguous.

## Lesson 5 | From truth tables to simplified logic

A sum-of-products expression ORs together terms that AND literals. To build one from a truth table, create a minterm for every row whose output is 1. For a row A=0,B=1,C=0, the minterm is A'BC'. Then simplify with algebra or a Karnaugh map when appropriate.

A Karnaugh map places adjacent cells in Gray-code order, so neighboring cells differ by one input bit. Group adjacent 1s into rectangular power-of-two groups, allowing wraparound where the map structure permits it. Larger groups remove more changing variables. A don't-care entry can be used only when the specification genuinely allows either output for that input.

Worked example: For inputs A,B,C, suppose Y=1 for minterms 1,3,5,7 under the binary index order ABC. These are 001,011,101,111. C is 1 in every row, while A and B vary over all combinations. The simplified function is Y=C.

A simplified expression is meaningful only if it still matches every required row. Verify it rather than trusting the visual shape of a group.

## Lesson 6 | Combinational building blocks

A combinational circuit's outputs depend on its current inputs, allowing for propagation delay. A multiplexer selects an input according to control bits. A decoder maps an encoded input to selected outputs. Adders combine bits and carries. These blocks compose into larger datapaths.

Worked example: A two-input multiplexer with select S has Y=S'A+SB. If S=0, the expression reduces to A. If S=1, it reduces to B. This derivation is a clearer specification than memorizing a symbol without knowing which input each select value chooses.

A half adder has sum A XOR B and carry AB. A full adder also includes an incoming carry. One expression for its carry is AB + C_in(A XOR B); the sum is A XOR B XOR C_in. Test the all-ones row: 1+1+1=3, so sum=1 and carry=1.

In a ripple-carry adder, carries pass between stages. Correct Boolean outputs do not imply instantaneous operation; the worst-case carry path affects timing.

## Lesson 7 | State, latches, flip-flops and counters

A sequential circuit stores state, so its outputs or future behavior can depend on past inputs. A latch is level-sensitive under its control conditions; an edge-triggered flip-flop captures according to a clock edge and its timing requirements. A register groups storage elements, while a counter updates its stored value according to a rule.

Worked example: A two-bit binary counter cycles 00 -> 01 -> 10 -> 11 -> 00. For a synchronous next-state description, the low bit toggles each step; the high bit toggles when the old low bit is 1. Use the old state consistently when deriving next-state equations.

Reset behavior must be specified. Does reset act synchronously with the clock or asynchronously? Which state does it establish? Without these choices, two circuit descriptions may behave differently even if their normal counting sequence agrees.

Do not describe a flip-flop as "always copying the input." Its sampling behavior and clock conditions are the point of the abstraction.

## Lesson 8 | State machines, timing and verification

A finite-state controller connects current state and inputs to next state, and possibly to outputs. In a Moore-style description, outputs depend on state; in a Mealy-style description, outputs can depend on state and current input. Specify which model you are using.

Worked example: A traffic-light teaching model has states Red, Green and Amber, with timer events moving Red -> Green -> Amber -> Red. Outputs are encoded by state. Real road-control systems need far more safety and timing logic; this simplified model is for understanding state transitions, not for

deployment.

Propagation delay measures the time before an input change affects an output. Setup and hold requirements constrain when data must be stable around a sampling edge. A circuit can be logically correct but fail when timing constraints are violated. Hazards and glitches can occur when paths have different delays.

Verification routine: declare signal meanings, enumerate required input combinations or state transitions, predict outputs, simulate, compare and explain discrepancies. Keep a waveform or truth table as evidence. Changing a wire until the simulator appears quiet is not a substitute for a specification.

## Practice set

1. Convert decimal 26 to binary and hex.
2. Interpret 110010 as an unsigned binary value.
3. Represent -12 in 8-bit two's complement.
4. Add 0111 and 0001 as 4-bit numbers. Discuss unsigned and signed interpretation.
5. Complete the XOR truth table from memory.
6. Simplify A + AB and explain the law used.
7. Apply De Morgan's law to (A+B+C)'.
8. Build the sum-of-products expression for a two-input function that is 1 only for inputs 01 and 10.
9. Evaluate a 2:1 mux for A=0,B=1, first with S=0 and then S=1.
10. For A=1,B=0,C_in=1, find full-adder sum and carry.
11. List six successive states of a 2-bit counter starting from 10.
12. Explain why a truth table alone does not prove a clocked circuit meets timing requirements.

## Answers and reasoning

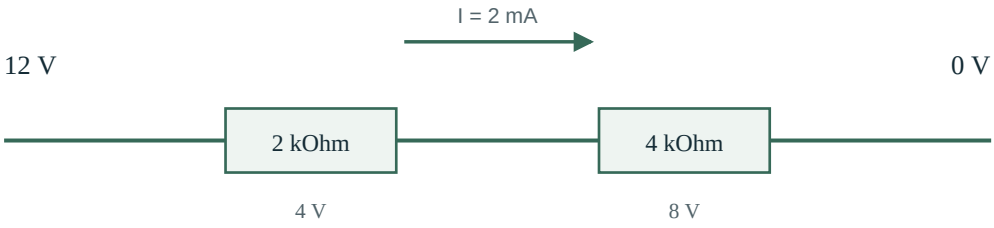
1. 26=16+8+2, so binary 11010; pad to 0001 1010 for hex 1A.
2. 32+16+2=50.
3. +12 is 00001100; invert to 11110011, add 1: 11110100.
4. The bit result is 1000. Unsigned 7+1=8 fits. Signed 4-bit values range -8 to 7, so adding +7 and +1 overflows and the stored pattern reads as -8.
5. For inputs 00,01,10,11 the outputs are 0,1,1,0.
6. A+AB=A by absorption. If A=1 the output is already 1; if A=0 both terms are 0.
7. A'B'C'. All OR terms become complemented AND terms.
8. A'B+AB', which is XOR. Include the complement on the input that is 0 in each selected row.
9. S=0 selects A, giving 0. S=1 selects B, giving 1.

10. 1+0+1=2, whose two-bit form is 10: sum=0, carry=1.
11. 10,11,00,01,10,11. State whether the starting value counts as the first listed state, as done here.
12. A truth table describes stable logical relationships. It does not quantify propagation delay or show whether data meets setup and hold requirements around clock edges.

## Further study and evidence

MIT Computation Structures has topic videos and worksheets: https://ocw.mit.edu/courses/6-004-computation-structures-spring-2017/ . Use it as a supporting course, not as proof that every topic appears in your current module. Produce one verified truth table, one simplified Boolean function and one counter waveform. The examples in this pack are original.

# CS 8103 | Electrical Fundamentals




## Learning outcomes and route

This module builds the language needed to reason about voltage, current, resistance, power, simple circuits and elementary time-dependent behavior. Work through eight lessons and practise drawing a circuit before selecting equations. These are foundation notes based on the module title; your laboratory instructions and current syllabus take priority for assessed work.

Practical boundary: do the exercises on paper or in a circuit simulator. Physical experiments require appropriate instructor-approved low-voltage equipment and supervision. Do not use mains electricity, open power supplies or charge/discharge unknown capacitors as a self-study experiment. Measurement instructions here explain concepts, not authorization to work on live equipment.

## Lesson 1 | Quantities, units and physical meaning

Charge Q is measured in coulombs (C). Current I is the rate of charge flow, measured in amperes (A). Voltage V is energy per unit charge, measured in volts (V). Resistance R relates voltage and current for an ohmic resistor under the applicable conditions, measured in ohms. Power P is the rate of energy transfer, measured in watts (W).

Prefixes matter: milli means 10^-3, micro means 10^-6, kilo means 10^3 and mega means 10^6. Convert consistently before substituting values. A factor-of-one-thousand error can make a neat-looking answer physically wrong.

Worked example: If 0.024 C passes a point in 0.006 s at a steady average rate, I=Q/t=4 A. If current is 4 mA instead, that is 0.004 A. The unit is part of the value, not optional decoration.

Conventional current direction follows the direction positive charge would move. Electron drift in a metal is opposite to that conventional direction. You can solve a circuit using either consistent convention, but do not change direction definitions midway through a calculation.

## Lesson 2 | Ohm's law and power

For an ohmic resistor in its valid operating conditions, V=IR. Rearranging gives I=V/R and R=V/I. This is not a universal relationship for every component: diodes, for example, do not behave like a constant resistance across all operating points.

Power transferred with the passive sign convention is P=VI, positive when the element absorbs power under the chosen voltage/current reference directions. For a resistor, combine Ohm's law to obtain P=I^2R=V^2/R. Energy over a constant-power interval is E=Pt.

Worked example: A 1.0 kOhm resistor has 5.0 V across it. I=5/1000=0.005 A=5 mA. Its dissipated power is 5*0.005=0.025 W=25 mW. The same result follows from 25/1000. Checking with a second expression can reveal a unit error.

A practical component has a power rating and may have temperature and voltage limits. A calculated current alone is not a complete component-selection procedure. In a paper exercise, state when you assume ideal components and constant temperature.

## Lesson 3 | Series circuits and voltage division

Resistors are in series when the same branch current must pass through each without an intervening branch carrying away current. Equivalent resistance is the sum. The voltage across each resistor is its current times its resistance, so the larger resistance receives the larger voltage drop for the same current.

Worked example: A 12 V ideal source drives 2 kOhm and 4 kOhm in series. Equivalent resistance is 6 kOhm, giving I=12/6000=2 mA. Voltage drops are 4 V and 8 V. Their sum is the source voltage.

The unloaded divider relation is V_out=V_in*R_bottom/(R_top+R_bottom), when output is measured across the bottom resistor and no extra load changes the circuit. Adding a load in parallel with the bottom resistor changes the effective bottom resistance. Do not use the unloaded formula unchanged after attaching a significant load.

A circuit's appearance on a drawing can be misleading. Identify nodes and connections rather than deciding "series" because two components are drawn next to each other.

## Lesson 4 | Parallel circuits and current division

Components are in parallel when they share both connection nodes and therefore have the same voltage across them. For resistors, 1/R_eq = 1/R1 + 1/R2 + ... . Two resistors give R_eq=R1R2/(R1+R2). For positive finite resistances, the equivalent is smaller than the smallest branch resistance.

Worked example: Put 3 kOhm and 6 kOhm across 12 V. Their currents are 4 mA and 2 mA. Total current is 6 mA, so equivalent resistance is 12/0.006=2 kOhm. The parallel formula gives the same result.

In a two-branch resistive current divider, a smaller resistance receives a larger current. If you forget the formula, calculate the common voltage or reason from I=V/R rather than guessing which resistance belongs in the numerator.

For a mixed series-parallel circuit, simplify an identifiable group, redraw the reduced circuit, and repeat. If the topology is not reducible by those patterns, use a more general method such as nodal analysis.

## Lesson 5 | Kirchhoff's laws and nodal analysis

Kirchhoff's current law (KCL) balances charge at a node: with consistent signs, currents entering and leaving sum to zero in the lumped-circuit model. Kirchhoff's voltage law (KVL) balances voltage changes around a closed loop under the conditions of that circuit model. Reference directions can be chosen; a negative answer often means the actual direction opposes the chosen reference.

Worked example: A node V connects through 2 kOhm to an ideal 10 V node and through 3 kOhm to ground. Taking currents leaving the unknown node gives (V-10)/2000 + V/3000 = 0. Multiply by 6000: 3(V-10)+2V=0, so 5V=30 and V=6 V. Current from the 10 V node is (10-6)/2000=2 mA, equal to 6/3000

leaving to ground.

A nodal-analysis routine: choose a reference node, label unknown node voltages, write each resistor current as a voltage difference divided by resistance, apply KCL, solve, then check currents and power. The reference node defines zero potential for the calculation; it does not imply zero current everywhere connected to it.

## Lesson 6 | Capacitors and RC transients

A capacitor stores energy in an electric field. For the ideal capacitor relation Q=CV, capacitance C is measured in farads. Current is i=C*dv/dt under a consistent reference convention. An ideal capacitor's voltage cannot jump instantaneously without an impulse of current in the idealized model.

For a simple RC step response, the time constant is tau=RC. Charging from 0 toward a constant source V_s through R gives v_C(t)=V_s*(1-exp(-t/tau)). Discharging from V_0 through R gives v_C(t)=V_0*exp(-t/tau), for the appropriate simple circuit and initial conditions.

Worked example: R=10 kOhm and C=100 microfarads give tau=10000*0.0001=1 s. Charging toward 5 V from zero gives about 3.16 V at 1 s, because 1-exp(-1) is about 0.632. It approaches 5 V asymptotically; it does not jump to 5 V at exactly one time constant.

Stored energy is E=0.5*C*V^2. A disconnected capacitor can retain energy. That is why unknown stored-energy components are outside unsupervised self-study practice.

## Lesson 7 | Inductors and AC foundations

An ideal inductor stores energy in a magnetic field. Its voltage is v=L*di/dt, with inductance L in henries. Its current cannot change instantaneously without an impulse of voltage in the ideal model. Energy is E=0.5*L*I^2. These relations are different from the resistor's constant proportional relationship.

A sinusoidal voltage can be written v(t)=V_peak*sin(2*pi*f*t+phi), where frequency f is in hertz and period T=1/f. For a sine wave with no DC offset, V_rms=V_peak/sqrt(2). RMS is useful for comparing resistive heating; it is not the same as the arithmetic mean of the waveform over a whole cycle.

Worked example: A 50 Hz sine wave has period 1/50=0.02 s=20 ms. If its peak is 10 V, RMS is about 7.07 V. Across a pure 100 Ohm resistor, average power is V_rms^2/R=0.5 W.

In AC analysis, reactance introduces frequency dependence and phase. For simple sinusoidal steady-state models, X_L=2*pi*f*L and X_C=1/(2*pi*f*C). Do not apply a purely resistive formula blindly to mixed AC loads. Phasor methods are a next topic if required by your lecturer.

## Lesson 8 | Measurement, uncertainty and checking answers

An ideal voltmeter measures across two points and has infinite input resistance. An ideal ammeter measures branch current in series and has zero resistance. Real instruments differ from these ideals and can alter the circuit. Connecting a current-measurement input across a voltage source is not a valid substitute for measuring voltage and may create a dangerous short circuit.

For safe self-study, use a simulator to practise instrument placement. For physical measurements, follow the supervised lab's range, connection and equipment procedures. A continuity or resistance measurement is ordinarily performed on appropriately isolated, unpowered circuitry under those

procedures, not on a live unknown circuit.

Worked example: A nominal 1000 Ohm resistor measures 980 Ohm. The difference from nominal is -20 Ohm; its magnitude relative to nominal is 2%. Compare this with specified tolerance and instrument uncertainty before judging it. Do not call every difference an experimental mistake.

Check every calculation in three ways: dimensions, magnitude and conservation. A parallel resistance greater than both positive branches is suspicious. In a simple resistor loop, voltage drops should balance the source. Source power supplied should match total power absorbed under your sign convention, within the model and rounding.

## Practice set

1. Convert 4.7 kOhm, 220 microfarads and 15 mA to base units.
2. Find current through 2 kOhm with 6 V across it.
3. Find power dissipated in that resistor using two formulas.
4. An ideal 9 V source drives 1 kOhm and 2 kOhm in series. Find current and both voltage drops.
5. Find the equivalent of 1 kOhm and 1 kOhm in parallel.
6. Across 10 V, parallel branches are 2 kOhm and 5 kOhm. Find both currents and total current.
7. An unloaded divider has R_top=3 kOhm, R_bottom=1 kOhm and V_in=8 V. Find V_out.
8. Write KCL for a node connected to 12 V through 2 kOhm and to ground through 4 kOhm. Solve for the node voltage.
9. Find tau for R=5 kOhm and C=200 microfarads.
10. A capacitor charges from zero toward 10 V. Approximately what voltage is reached at one time constant?
11. Find the period and RMS voltage of a 100 Hz sine wave with peak 8 V.
12. Explain correct conceptual placement of a voltmeter and ammeter, and why "it has numbers on the display" does not prove the measurement is valid.

## Answers and reasoning

1. 4700 Ohm; 0.000220 F; 0.015 A.
2. I=6/2000=0.003 A=3 mA.
3. VI=6*0.003=0.018 W. V^2/R=36/2000=0.018 W, or 18 mW.
4. Total R=3000 Ohm; I=3 mA. Drops: 3 V across 1 kOhm and 6 V across 2 kOhm; their sum is 9 V.
5. Equal branches give half either resistance: 500 Ohm.
6. I1=10/2000=5 mA; I2=10/5000=2 mA; total 7 mA.
7. 8*1000/(3000+1000)=2 V. This assumes no loading at the output.

8. (V-12)/2000+V/4000=0. Multiply by 4000: 2V-24+V=0, so V=8 V. Both branch-current magnitudes are 2 mA.
9. 5000*0.0002=1 s.
10. About 6.32 V, using 10*(1-exp(-1)).
11. T=0.01 s=10 ms. V_rms=8/sqrt(2), approximately 5.66 V.
12. A voltmeter connects across the voltage of interest; an ammeter belongs in series with the branch. Correct mode, range, connections and instrument characteristics matter. A displayed value can reflect a wrong or unsafe setup.

## Further study and evidence

MIT Circuits and Electronics: https://ocw.mit.edu/courses/6-002-circuits-and-electronics-spring-2007/ . OpenStax University Physics Volume 2, current chapter: https://openstax.org/books/university-physics-volume-2/pages/9-1-electrical-current . The examples above were created for this pack. Submit a labeled circuit sketch, a complete calculation with units and a short explanation of your checks for each problem you solve.

# Study plan and practical work

This is a proposed supplement to classes, not an official university timetable or estimate of the total workload. Adapt around employment, travel, rest, lecturer deadlines and your actual weak areas. All seven modules receive equal dedicated practice in the starting rhythm.

## Session method

• 5 minutes: retrieve the last lesson without opening notes.

• 10 minutes: read one section and inspect the worked example.

• 20 minutes: solve or build something independently.

• 5 minutes: check against the answer, compiler, truth table or dimensional calculation.

• 5 minutes: record one error and write the next retrieval question.

Do not count a copied solution as independent success. Reattempt a related example on the next study day, about a week later, and later again as needed. These are adjustable review intervals, not a guaranteed retention formula.

## Starting weekly rhythm

Monday: maths 45 minutes and C 45 minutes. Tuesday: communication 45. Wednesday: theory 45. Thursday: OS 45. Friday: digital circuits 45. Saturday: electrical fundamentals 45 and mixed review 90. Sunday: rest or genuinely necessary catch-up. Total planned self-study is 6 hours 45 minutes, additional to classes. If this is too much, use 25-minute sessions and retain all seven subjects rather than silently dropping the difficult ones.

## The twelve-week route

Weeks 1-8: take the matching numbered lesson in each of the seven modules. For example, Week 3 covers matrices, paragraphs, DFA state meaning, scheduling, C selection, logic gates and series circuits. The sequencing is for these notes; if your lecturer teaches another order, follow the class sequence and use the corresponding lesson here.

Week 9: complete all 12 questions per module without the answer sections. Divide them across the week. Record which answers required a hint. Review the underlying lesson before repeating the missed skill with different numbers or inputs.

Week 10: work on the integrated mini-project below, and explain it in writing. Keep ordinary topic revision active rather than devoting the entire week to cosmetic presentation.

Week 11: make your own mixed mock assessment: 20 minutes maths, 20 C, 20 theory, 20 OS, 20 digital, 20 electrical and 20 communication. These are practice intervals, not predictions about the university's examination pattern. Mark with the included answer guides and retain the work.

Week 12: repair the three weakest skills across the diagnostic results. Re-solve a fresh version from a blank page. Explain the correction aloud, and update a one-page formula/concept sheet using only items you can justify.

## Integrated project | A small study-session logger

Use synthetic data, not personal records. Start with the supplied C study_logger.c. Then add one clearly scoped feature: a subject filter, validated entry input, or a report written to a sandbox file.

• Maths: calculate total and mean minutes; explain integer versus real-valued division.

• Communication: write a one-page README and a short test report with audience and purpose.

• Theory: draw a small state machine for input states such as waiting, valid entry and error. State what your model abstracts away.

• OS: explain where the program executes as a process, which files it opens, and the permissions needed.

• C: separate data, calculation and presentation; test the chosen feature.

• Digital: draw a truth table for a status indicator based on two defined Boolean inputs, such as target-met and invalid-entry. Explain the intended output.

• Electrical: design a paper-only resistor calculation for a hypothetical low-voltage indicator load using explicit component assumptions. Do not connect hardware as part of this self-study project.

Deliverables: source code, test results, a labeled diagram, a calculation page, a one-page report and a three-minute explanation. No fabricated measurements or claims of external deployment.

## What counts as progress

Use five levels: not attempted; can follow; can solve with notes; can solve independently; can explain and transfer. A topic is not mastered because its video has played to the end. Keep a record of the independent artifact that supports your current level.

# Resources and provenance

## Scope and provenance

The names and codes were checked against the supplied MUST Prospectus 2025/2026, printed page 173. That prospectus is older than the 2026/2027 intake. Module titles do not specify every assessed topic, so this pack is a substantial foundation course, not an official MUST textbook, current syllabus or promise of examination coverage. Match it to your lecturer's outlines, timetable and assignments when available.

This edition focuses its detailed lessons on the seven first-semester modules. It does not present later-year modules as completed notes. Use the current course outlines to identify any additional topics or assessment requirements.

The package contains original notes, exercises and code. External textbooks and original university documents are not redistributed. Public companion articles use original educational examples.

## Learning resources

• Maths: MIT 18.06, https://ocw.mit.edu/courses/18-06-linear-algebra-spring-2010/ ; OpenStax Calculus, https://openstax.org/books/calculus-volume-1/pages/preface .

• Communication: Google Technical Writing One, https://developers.google.com/tech-writing/one ; Purdue OWL, https://owl.purdue.edu/owl/general_writing/academic_writing/paragraphs_and_paragraphing/index.html .

• Theory: MIT 6.045J, https://ocw.mit.edu/courses/6-045j-automata-computability-and-complexity-spring-2011/ .

• OS: the authors' OSTEP site, https://pages.cs.wisc.edu/~remzi/OSTEP/ .

• C: Harvard CS50, https://cs50.harvard.edu/x/weeks/1/ ; WG14 C11-era committee draft N1570, https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf . The examples use C17 mode; N1570 is a precision reference for shared foundations, not a current-edition tutorial.

• Digital: MIT Computation Structures, https://ocw.mit.edu/courses/6-004-computation-structures-spring-2017/ .

• Electrical: MIT Circuits, https://ocw.mit.edu/courses/6-002-circuits-and-electronics-spring-2007/ ; OpenStax Physics, https://openstax.org/books/university-physics-volume-2/pages/9-1-electrical-current .

Links checked 18 November 2026. Availability and course editions can change. Download third-party materials from their own sites subject to their terms; this package does not relicense them.

## Practical program checks

The six original C17 examples were compiled with Clang using -Wall -Wextra -Wpedantic -Werror. All 15 supplied execution cases passed. The test runner checks its stated inputs and outputs, not every possible behavior. Run it again after changing a program.