UCLA COMPUTER SCIENCE FIELD NOTES

CS 31

Introduction to Computer Science I

C++ programming, decomposition, memory, recursion, and testable design

How these notes are written. I wrote each chapter the way I wish the material had been explained the first time: the real idea, the point where I usually get stuck, the simplest mental model that still stays honest, and a worked decision instead of a polished answer appearing from nowhere. Every chapter closes with practice and a fully written solution. If a sentence sounds impressive but does not help me solve or explain something, it does not belong here.

Daniel Mastick

Curriculum map, working notes, practice, and solutions

Course map

Week	Core thread	What should be solid by the end
1	**From source file to running program**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
2	**Control flow and loop invariants**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
3	**Functions, contracts, and decomposition**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
4	**Arrays, strings, and index discipline**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
5	**Pointers, ownership, and lifetime**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
6	**Classes and abstract data types**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
7	**Recursion and the call stack**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
8	**Debugging, testing, and program design**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
9	**Integration workshop**	Connect at least three chapters in one end-to-end problem and explain where their contracts meet.
10	**Synthesis and project review**	Audit assumptions and explain a complete solution from interface to failure handling.

Scope anchor: https://catalog.registrar.ucla.edu/course/2024/COMSCI31. The sequence is aligned to the official catalog description and expanded into a practical ten-week study plan.

How I would use this packet

Treat each numbered section as one chapter. Read the formal explanation, pause at the student interpretation, and see whether the easy version still says the same thing. Rework the example without looking, then cover the solution in the chapter practice box. The cumulative set at the end is deliberately mixed: knowledge becomes durable when I have to choose the tool instead of being told which chapter I am in.

Contents

CHAPTER 1 From source file to running program

The idea

C++ source passes through preprocessing, compilation, assembly, linking, loading, and execution. Types constrain values and operations; diagnostics usually point near the first place the program stopped matching the language.

Rebuild the chapter from first principles

The claim I need to own is this: C++ source passes through preprocessing, compilation, assembly, linking, loading, and execution. Types constrain values and operations; diagnostics usually point near the first place the program stopped matching the language. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of from source file to running program. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach from source file to running program on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from the course foundations to control flow and loop invariants. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** C++ source passes through preprocessing, compilation, assembly, linking, loading, and execution.
- **Mechanism.** Types constrain values and operations; diagnostics usually point near the first place the program stopped matching the language.
- **Boundary.** The useful test is whether I can apply from source file to running program to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when from source file to running program is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I treat the compiler as my first reviewer and fix the first real error before reading the cascade below it.

The easy version

Code is a precise recipe: compilation checks and translates it, linking connects its pieces, and execution finally runs it.

Worked example

Compare integer 7/2 with 7.0/2. The first truncates before assignment; the second produces 3.5. The worked question below makes that reasoning concrete: Why does double x=7/2 store 3.0? The conclusion is Both operands are integers, so integer division happens first. Use 7.0/2 or cast

one operand before division.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why does double x=7/2 store 3.0?
Solution. Both operands are integers, so integer division happens first. Use 7.0/2 or cast one operand before division.

Practice 2. Give a short oral explanation of from source file to running program that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct from source file to running program from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a from source file to running program problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 2 Control flow and loop invariants

The idea

Conditionals select a path; loops repeat a state transition. A loop invariant states what is true before and after every iteration and explains initialization, progress, and termination.

Rebuild the chapter from first principles

The claim I need to own is this: Conditionals select a path; loops repeat a state transition. A loop invariant states what is true before and after every iteration and explains initialization, progress, and

termination. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of control flow and loop invariants. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach control flow and loop invariants on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from from source file to running program to functions, contracts, and decomposition. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Conditionals select a path; loops repeat a state transition.
- **Mechanism.** A loop invariant states what is true before and after every iteration and explains initialization, progress, and termination.
- **Boundary.** The useful test is whether I can apply control flow and loop invariants to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when control flow and loop invariants is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I write the invariant before the loop body; it catches off-by-one mistakes faster than another trace.

The easy version

A loop is a tiny machine that improves a known state until a stopping test is true.

Worked example

For a running maximum, best is the largest value in the prefix already examined. Each iteration extends that prefix by one. The worked question below makes that reasoning concrete: Give an invariant for summing a[0] through a[n-1]. The conclusion is Before iteration i, sum equals a[0]+...+a[i-1]. Adding a[i] establishes the claim for i+1; i<n includes the last index n-1.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Give an invariant for summing a[0] through a[n-1].
Solution. Before iteration i, sum equals a[0]+...+a[i-1]. Adding a[i] establishes the claim for i+1; i<n includes the last index n-1.

Practice 2. Give a short oral explanation of control flow and loop invariants that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison

according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct control flow and loop invariants from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a control flow and loop invariants problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 3 Functions, contracts, and decomposition

The idea

A function owns one coherent responsibility. Parameter types, return type, preconditions, postconditions, and side effects form its contract. Value copies; reference aliases; const reference reads efficiently without mutation.

Rebuild the chapter from first principles

The claim I need to own is this: A function owns one coherent responsibility. Parameter types, return type, preconditions, postconditions, and side effects form its contract. Value copies; reference aliases; const reference reads efficiently without mutation. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of functions, contracts, and decomposition. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.

4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach functions, contracts, and decomposition on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from control flow and loop invariants to arrays, strings, and index discipline. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** A function owns one coherent responsibility.
- **Mechanism.** Parameter types, return type, preconditions, postconditions, and side effects form its contract.
- **Boundary.** Value copies; reference aliases; const reference reads efficiently without mutation.
- **Decision test.** I should be able to say when functions, contracts, and decomposition is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

If a function is hard to name, I assume it is doing more than one job.

The easy version

A function is a small agreement: valid inputs buy a promised result and controlled side effects.

Worked example

Split a grade program into readScores, mean, and letterGrade so each promise can be tested independently. The worked question below makes that reasoning concrete: How should a large read-only string parameter be passed? The conclusion is Use const string reference. It avoids a copy and prevents mutation that the contract does not permit.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. How should a large read-only string parameter be passed?
Solution. Use const string reference. It avoids a copy and prevents mutation that the contract does not permit.

Practice 2. Give a short oral explanation of functions, contracts, and decomposition that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct functions, contracts, and decomposition from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a functions, contracts, and decomposition problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 4 Arrays, strings, and index discipline

The idea

Arrays store contiguous elements; an index is an offset. Valid indices are zero through n-1. Half-open ranges [begin,end) represent empty ranges naturally and compose without double-counting boundaries.

Rebuild the chapter from first principles

The claim I need to own is this: Arrays store contiguous elements; an index is an offset. Valid indices are zero through n-1. Half-open ranges [begin,end) represent empty ranges naturally and compose without double-counting boundaries. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of arrays, strings, and index discipline. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach arrays, strings, and index discipline on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from functions, contracts, and decomposition to pointers, ownership, and lifetime. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Arrays store contiguous elements; an index is an offset.
- **Mechanism.** Valid indices are zero through n-1.
- **Boundary.** Half-open ranges [begin,end) represent empty ranges naturally and compose without double-counting boundaries.
- **Decision test.** I should be able to say when arrays, strings, and index discipline is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I write the intended range next to every loop, especially when shifting or scanning backward.

The easy version

The first item is index zero because it is zero steps from the beginning.

Worked example

Deleting index k shifts k+1 through n-1 left, then decreases logical size; capacity does not change. The worked question below makes that reasoning concrete: Give bounds for in-place array reversal. The conclusion is For i from 0 while i<n/2, swap a[i] and a[n-1-i]. Continuing past the midpoint would undo earlier swaps.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Give bounds for in-place array reversal.
Solution. For i from 0 while i<n/2, swap a[i] and a[n-1-i]. Continuing past the midpoint would undo earlier swaps.

Practice 2. Give a short oral explanation of arrays, strings, and index discipline that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct arrays, strings, and index discipline from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a arrays, strings, and index discipline problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 5 Pointers, ownership, and lifetime

The idea

A pointer stores an address. Correctness depends on which object lives there, how long it remains alive, who owns it, and whether aliases outlive it. Null, dangling, leaked, and double-freed pointers are distinct failures.

Rebuild the chapter from first principles

The claim I need to own is this: A pointer stores an address. Correctness depends on which object lives there, how long it remains alive, who owns it, and whether aliases outlive it. Null, dangling, leaked, and double-freed pointers are distinct failures. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule

from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of pointers, ownership, and lifetime. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach pointers, ownership, and lifetime on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from arrays, strings, and index discipline to classes and abstract data types. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method

valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** A pointer stores an address.
- **Mechanism.** Correctness depends on which object lives there, how long it remains alive, who owns it, and whether aliases outlive it.
- **Boundary.** Null, dangling, leaked, and double-freed pointers are distinct failures.
- **Decision test.** I should be able to say when pointers, ownership, and lifetime is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I draw objects as boxes and pointers as arrows; if I cannot name the owner, the design is not finished.

The easy version

A pointer is a note containing an address, useful only while the object still lives there.

Worked example

Returning the address of a local variable dangles because the stack object dies at return; returning a value is safe. The worked question below makes that reasoning concrete: Why is deleting one pointer twice undefined? The conclusion is The first delete ends the allocation lifetime. The address remains in the pointer but no longer names live owned storage; prefer automatic objects or unique ownership.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why is deleting one pointer twice undefined?
Solution. The first delete ends the allocation lifetime. The address remains in the pointer but no longer names live owned storage; prefer automatic objects or unique ownership.

Practice 2. Give a short oral explanation of pointers, ownership, and lifetime that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct pointers, ownership, and lifetime from a blank page. Include

the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a pointers, ownership, and lifetime problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 6 Classes and abstract data types

The idea

A class packages representation with operations that preserve an invariant. Public methods state what clients may do; private fields record how. Constructors establish validity and representation independence permits implementation changes.

Rebuild the chapter from first principles

The claim I need to own is this: A class packages representation with operations that preserve an invariant. Public methods state what clients may do; private fields record how. Constructors establish validity and representation independence permits implementation changes. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of classes and abstract data types. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach classes and abstract data types on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from pointers, ownership, and lifetime to recursion and the call stack. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** A class packages representation with operations that preserve an invariant.
- **Mechanism.** Public methods state what clients may do; private fields record how.
- **Boundary.** Constructors establish validity and representation independence permits implementation changes.
- **Decision test.** I should be able to say when classes and abstract data types is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I write the invariant before the fields because it tells every method what it must protect.

The easy version

Clients should learn the buttons on the machine, not depend on its gears.

Worked example

A Stack exposes push, pop, top, empty, and size while keeping its array, size, and capacity private. The worked question below makes that reasoning concrete: What invariant connects stack size and capacity? The conclusion is Maintain 0<=size<=capacity, with live elements in indices 0 through size-1. Every public method must preserve it.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. What invariant connects stack size and capacity?
Solution. Maintain 0<=size<=capacity, with live elements in indices 0 through size-1. Every public method must preserve it.

Practice 2. Give a short oral explanation of classes and abstract data types that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct classes and abstract data types from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a classes and abstract data types problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 7 Recursion and the call stack

The idea

Recursion needs a base case and a recursive case that strictly reduces a measure. Each call owns a separate stack frame. Correctness follows an inductive promise: trust the smaller call, then justify the current step.

Rebuild the chapter from first principles

The claim I need to own is this: Recursion needs a base case and a recursive case that strictly reduces a measure. Each call owns a separate stack frame. Correctness follows an inductive promise: trust the smaller call, then justify the current step. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of recursion and the call stack. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach recursion and the call stack on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from classes and abstract data types to debugging, testing, and program design. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Recursion needs a base case and a recursive case that strictly reduces a measure.
- **Mechanism.** Each call owns a separate stack frame.
- **Boundary.** Correctness follows an inductive promise: trust the smaller call, then justify the current step.
- **Decision test.** I should be able to say when recursion and the call stack is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I draw separate frames; recursion stops looking magical when each frame has one small job.

The easy version

Solve a problem by trusting a smaller version, then perform the one remaining step.

Worked example

Palindrome recursion compares the ends, then calls itself on the shorter interior substring. The worked question below makes that reasoning concrete: State the decreasing measure for recursive palindrome checking. The conclusion is The active substring length decreases by two. When left>=right the length is zero or one, which is the base case.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. State the decreasing measure for recursive palindrome checking.
Solution. The active substring length decreases by two. When left>=right the length is zero or one, which is the base case.

Practice 2. Give a short oral explanation of recursion and the call stack that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct recursion and the call stack from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a recursion and the call stack problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 8 Debugging, testing, and program design

The idea

Tests sample behavior; they do not prove absence of defects. Strong suites cover ordinary, boundary, empty, singleton, invalid, and interacting cases. Debugging should reproduce, minimize, hypothesize, observe, and change one cause at a time.

Rebuild the chapter from first principles

The claim I need to own is this: Tests sample behavior; they do not prove absence of defects. Strong suites cover ordinary, boundary, empty, singleton, invalid, and interacting cases. Debugging should reproduce, minimize, hypothesize, observe, and change one cause at a time. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what

stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of debugging, testing, and program design. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach debugging, testing, and program design on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from recursion and the call stack to the cumulative course model. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method

valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Tests sample behavior; they do not prove absence of defects.
- **Mechanism.** Strong suites cover ordinary, boundary, empty, singleton, invalid, and interacting cases.
- **Boundary.** Debugging should reproduce, minimize, hypothesize, observe, and change one cause at a time.
- **Decision test.** I should be able to say when debugging, testing, and program design is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

Random edits erase evidence, so I keep the failing input fixed and ask where the first wrong fact appears.

The easy version

A test is a controlled question; debugging narrows the earliest mismatch between expected and actual state.

Worked example

Test array search on empty input, one item, first and last positions, absence, duplicates, and out-of-range targets. The worked question below makes that reasoning concrete: Why are ten passing examples not a proof of correctness? The conclusion is They cover only ten executions. A defect may live in an untested state or boundary; systematic partitions and invariants provide broader evidence.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why are ten passing examples not a proof of correctness?
Solution. They cover only ten executions. A defect may live in an untested state or boundary; systematic partitions and invariants provide broader evidence.

Practice 2. Give a short oral explanation of debugging, testing, and program design that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct debugging, testing, and program design from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a debugging, testing, and program design problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

Final study check

I should be able to close this packet and reconstruct the core models, not merely recognize their names. My final check is to explain one complete problem aloud: what the inputs mean, which invariant or contract controls the work, why the method is legal, what it costs, how it can fail, and what evidence would convince another engineer.