HAS 106A ARMY CODING PRACTICE EXAM (SAMPLE)

STUDY GUIDE



SAMPLE STUDY GUIDE WITH LIMITED QUESTIONS

VISIT US ONLINE FOR THE FULL VERSION STUDY GUIDE

https://has106a.examzify.com

Copyright © 2026 by Examzify - A Kaluba Technologies Inc. product.

ALL RIGHTS RESERVED.

No part of this book may be reproduced or transferred in any form or by any means, graphic, electronic, or mechanical, including photocopying, recording, web distribution, taping, or by any information storage retrieval system, without the written permission of the author.

Notice: Examzify makes every reasonable effort to obtain accurate, complete, and timely information about this product from reliable sources.

SAMPLE

Table of Contents

Copyright 1
Table of Contents 2
Introduction 3
How to Use This Guide 4
Questions 5
Answers 8
Explanations 10
Next Steps 15

SAMPLE

Introduction

Preparing for a certification exam can feel overwhelming, but with the right tools, it becomes an opportunity to build confidence, sharpen your skills, and move one step closer to your goals. At Examzify, we believe that effective exam preparation isn't just about memorization, it's about understanding the material, identifying knowledge gaps, and building the test-taking strategies that lead to success.

This guide was designed to help you do exactly that.

Whether you're preparing for a licensing exam, professional certification, or entry-level qualification, this book offers structured practice to reinforce key concepts. You'll find a wide range of multiple-choice questions, each followed by clear explanations to help you understand not just the right answer, but why it's correct.

The content in this guide is based on real-world exam objectives and aligned with the types of questions and topics commonly found on official tests. It's ideal for learners who want to:

- Practice answering questions under realistic conditions,
- Improve accuracy and speed,
- Review explanations to strengthen weak areas, and
- Approach the exam with greater confidence.

We recommend using this book not as a stand-alone study tool, but alongside other resources like flashcards, textbooks, or hands-on training. For best results, we recommend working through each question, reflecting on the explanation provided, and revisiting the topics that challenge you most.

Remember: successful test preparation isn't about getting every question right the first time, it's about learning from your mistakes and improving over time. Stay focused, trust the process, and know that every page you turn brings you closer to success.

Let's begin.

SAMPLE

How to Use This Guide

This guide is designed to help you study more effectively and approach your exam with confidence. Whether you're reviewing for the first time or doing a final refresh, here's how to get the most out of your Examzify study guide:

1. Start with a Diagnostic Review

Skim through the questions to get a sense of what you know and what you need to focus on. Your goal is to identify knowledge gaps early.

2. Study in Short, Focused Sessions

Break your study time into manageable blocks (e.g. 30 – 45 minutes). Review a handful of questions, reflect on the explanations.

3. Learn from the Explanations

After answering a question, always read the explanation, even if you got it right. It reinforces key points, corrects misunderstandings, and teaches subtle distinctions between similar answers.

4. Track Your Progress

Use bookmarks or notes (if reading digitally) to mark difficult questions. Revisit these regularly and track improvements over time.

5. Simulate the Real Exam

Once you're comfortable, try taking a full set of questions without pausing. Set a timer and simulate test-day conditions to build confidence and time management skills.

6. Repeat and Review

Don't just study once, repetition builds retention. Re-attempt questions after a few days and revisit explanations to reinforce learning. Pair this guide with other Examzify tools like flashcards, and digital practice tests to strengthen your preparation across formats.

There's no single right way to study, but consistent, thoughtful effort always wins. Use this guide flexibly, adapt the tips above to fit your pace and learning style. You've got this!

SAMPLE

Questions

SAMPLE

1. How can you detect and fix off-by-one errors in loops?

A. Inspect loop bounds, use consistent inclusive/exclusive logic, and add tests that exercise boundary indices.

B. Ignore boundary cases and rely on compiler checks.

C. Double the loop bounds to avoid underflow.

D. Use random test values and hope one hits the boundary.

2. MASCAL stands for?

A. Mass Casualty

B. Mass Casualty Incident

C. Medical Assistance for Casualties

D. Multi-Agency Safety for Casualties

3. Why is direct equality comparison problematic for floating-point numbers?

A. Floating point uses finite precision; a small epsilon is unnecessary.

B. Floating point uses finite precision; comparisons should be exact.

C. Floating point uses finite precision; avoid direct equality checks; compare with an epsilon or use integer arithmetic when possible.

D. Floating point numbers are always exact for integers.

4. What is a key benefit of using inline functions over macros in C?

A. Inline functions introduce more overhead than macros.

B. Inline functions provide type safety and avoid some side effects that macros have.

C. Inline functions can replace all macros without caveats.

D. Inline functions are not supported by modern compilers.

5. Explain the concept of time-space trade-offs with a simple example.

A. Precomputing results (faster runtime, more memory) versus computing on demand (slower runtime, less memory)

B. Always minimizing both time and space

C. Preferring space over time without limits

D. Recomputing everything every time to ensure freshness

SAMPLE

6. Where should MASCAL patients be built in the facility's records system?

A. HER

B. BED Status Board

C. Security Desk

D. Transportation Log

7. What issue can arise from mixed signed/unsigned comparisons in C, and how can you prevent it?

A. Signed and unsigned comparisons are always safe with no care.

B. Casting a signed value to unsigned is always harmless.

C. Cast operands to a common unsigned type to avoid surprises.

D. Negative values become large unsigned when cast; cast operands to a common signed type or carefully check ranges.

8. Which form number corresponds to the Death Tags?

A. SF 523

B. DA 3910

C. SF 523-A

D. DA 3894

9. Which statement best describes the first character in coding?

A. The first character is always alpha.

B. The first character is always numeric.

C. The first character is always punctuation.

D. The first character is never alpha.

10. Where are automatic local variables defined within a function typically allocated and deallocated?

A. On the stack, with lifetime tied to function calls.

B. On the heap.

C. In static storage.

D. In registers only.

SAMPLE

Answers

SAMPLE

1. A
2. A
3. C
4. B
5. A
6. A
7. D
8. B
9. A
10. A

SAMPLE

Explanations

SAMPLE

1. How can you detect and fix off-by-one errors in loops?

A. Inspect loop bounds, use consistent inclusive/exclusive logic, and add tests that exercise boundary indices.

B. Ignore boundary cases and rely on compiler checks.

C. Double the loop bounds to avoid underflow.

D. Use random test values and hope one hits the boundary.

Off-by-one errors come from how you set up the loop's bounds and termination condition. The safest way to avoid them is to pin down the bounds precisely and use a consistent boundary convention, then test the boundary cases. Using a half-open range is a common and reliable approach: start at the first index (usually 0) and continue while the index is less than the end value. For an array of length n, iterating with i starting at 0 and i < n touches exactly indices 0 through n-1. This simple rule keeps the loop from accidentally including an extra element or skipping one, because the upper bound n is not included in the loop. Make the logic uniform: pick either an exclusive upper bound or an inclusive one and stick with it throughout the code. If you choose an inclusive upper bound, you must carefully set the condition (for example, i <= lastIndex) to avoid off-by-one mistakes; with an exclusive upper bound, the condition i < end is typically less error-prone. Support this with tests that exercise boundary indices: verify behavior for an empty collection, a single-element collection, and collections at the typical and near-boundary sizes. Tests that touch the first and last elements, as well as one past the end, help reveal off-by-one problems, and will guide you to the correct bounds. Other approaches don't directly address the issue: ignoring boundary cases or relying on compiler checks won't catch logical mistakes, doubling the loop bounds can mask or create errors, and random values are unlikely to land on the exact boundary you need to test.

2. MASCAL stands for?

A. Mass Casualty

B. Mass Casualty Incident

C. Medical Assistance for Casualties

D. Multi-Agency Safety for Casualties

MASCAL is used to quickly signal a situation with a large number of injured people requiring urgent, coordinated response. The expansion Mass Casualty fits this purpose because it directly describes the scale of the event and the need to mobilize triage and additional resources. While you might hear Mass Casualty Incident in some contexts, this material uses Mass Casualty as the standard expansion to trigger the appropriate response actions. The other options don't reflect the recognized wording used to indicate a large-scale casualty event.

3. Why is direct equality comparison problematic for floating-point numbers?

A. Floating point uses finite precision; a small epsilon is unnecessary.

B. Floating point uses finite precision; comparisons should be exact.

C. Floating point uses finite precision; avoid direct equality checks; compare with an epsilon or use integer arithmetic when possible.

D. Floating point numbers are always exact for integers.

Floating-point numbers are stored with finite precision, so many real values can't be represented exactly. When you perform calculations, rounding happens along the way, and two values that should be the same in math can end up slightly different in code. Because of this, using direct equality checks can fail even for values that are effectively equal. The best approach is to compare with a small tolerance (an epsilon) or use integer arithmetic when possible. For example, 0.1 + 0.2 often isn't exactly 0.3 in floating-point, so checking if it equals 0.3 can fail; instead, check if the absolute difference is below a tiny threshold, like abs(a - b) < 1e-9. If you can represent the values as integers (fixed-point or scaled integers), you can achieve exact equality.

SAMPLE

4. What is a key benefit of using inline functions over macros in C?

A. Inline functions introduce more overhead than macros.

B. Inline functions provide type safety and avoid some side effects that macros have.

C. Inline functions can replace all macros without caveats.

D. Inline functions are not supported by modern compilers.

Inline functions give the compiler a real function to work with, so the code is checked for correct types and proper use of parameters. This type safety means the compiler will catch mismatched argument types and other mistakes, something macros cannot guarantee because they are simple textual substitutions without any type checking. Macros substitute code directly as written, which can lead to unintended behavior if arguments have side effects or aren't evaluated in a single, well-defined way. A classic example is a macro that multiplies its argument by itself: if you pass an expression with increment operators, the expression may be evaluated more than once, changing program behavior in surprising ways. An inline function evaluates its arguments once, preserving the intended semantics and avoiding that kind of surprise. Inline functions also respect scopes and names more naturally than macros, reducing accidental name clashes and other confusion. While macros have their own valid uses, the safety and predictability you get with type-checked inlining is the key benefit in most cases.

5. Explain the concept of time-space trade-offs with a simple example.

A. Precomputing results (faster runtime, more memory) versus computing on demand (slower runtime, less memory)

B. Always minimizing both time and space

C. Preferring space over time without limits

D. Recomputing everything every time to ensure freshness

Time-space trade-offs are about balancing how fast your program runs with how much memory it uses. A simple way to see it is to compare precomputing results to computing them on demand. If you precompute and store answers (like a list of primes up to a limit), you can answer each query very quickly, but you pay in memory to hold all those results. If you instead skip precomputation and calculate each answer as needed, you save memory but your runtime gets longer because you're doing the work again each time. In practice you choose the balance based on constraints like available memory and how many queries you expect. This isn't about always minimizing both time and space. Resources are finite, so making something faster typically uses more memory, and trying to keep memory extremely low can slow things down. Likewise, aiming to save memory by recomputing everything every time can degrade performance, even if it keeps memory usage low. The key idea is to pick a reasonable balance that fits the situation.

SAMPLE

6. Where should MASCAL patients be built in the facility's records system?

A. HER

B. BED Status Board

C. Security Desk

D. Transportation Log

In a mass casualty situation, the priority is to create a single, verifiable patient record quickly so all care providers reference the same information. The Hospital Electronic Registry (HER) is the place to build a new patient entry because it is designed to generate a unique patient identifier, capture demographics, triage category, arrival details, and initial injury information, and then link this record to all clinical modules (labs, imaging, notes). This ensures data integrity and a consistent flow of information for the patient's ongoing care. The other options serve different operational needs: the Bed Status Board tracks bed occupancy and room locations, not medical records; the Security Desk handles access and safety; and the Transportation Log records patient movements but does not store or consolidate the medical record.

7. What issue can arise from mixed signed/unsigned comparisons in C, and how can you prevent it?

A. Signed and unsigned comparisons are always safe with no care.

B. Casting a signed value to unsigned is always harmless.

C. Cast operands to a common unsigned type to avoid surprises.

D. Negative values become large unsigned when cast; cast operands to a common signed type or carefully check ranges.

When you compare signed and unsigned values in C, the operands are converted to a common type, and that conversion can flip the result if a negative signed value is involved. If the unsigned type can represent all values of the signed type, the signed value is converted to unsigned. That means a negative number becomes a large unsigned value, which can make comparisons behave in ways you don't expect. This is why casting to a common signed type is recommended: you avoid the wraparound that happens when negatives are treated as large unsigned numbers. Casting to a common unsigned type is risky because negative values turn into large positives, producing surprising results. An alternative is to carefully check ranges (e.g., ensure the signed value is nonnegative or cast both operands to a sufficiently wide signed type) before comparing. For example, comparing -1 with 1 as signed values yields true, but if -1 is cast to unsigned, it becomes a large number and the comparison with 1 becomes false. Casting both to a common signed type (or ensuring values are within representable ranges) prevents that trap.

8. Which form number corresponds to the Death Tags?

A. SF 523

B. DA 3910

C. SF 523-A

D. DA 3894

The form used for documenting death tags is a Department of the Army document specifically designated for recording and tracking the tag data and the casualty handling steps. This form is chosen because it is the official record that captures all necessary identifying information and ensures proper accountability for the remains. The other forms shown are intended for different administrative tasks, such as medical records or other casualty-related processes, and do not pertain to the designation and handling of death tags. Therefore, the DA form numbered for death tag documentation is the best fit.

SAMPLE

9. Which statement best describes the first character in coding?

A. The first character is always alpha.

B. The first character is always numeric.

C. The first character is always punctuation.

D. The first character is never alpha.

In programming, names for variables, functions, and other identifiers follow naming rules that help the compiler recognize them correctly. A common rule is that the first character must be a letter. This avoids confusion with numeric literals, which start with digits, and with punctuation that is used for operators and separators. After the first character, identifiers can usually include letters, digits, or symbols like underscores depending on the language. Because of this, the statement that the first character is always a letter best matches typical practice, making it the most accurate choice. The other options clash with how identifiers are normally defined: starting with a digit would create ambiguity with numbers, starting with punctuation would break tokenization, and saying the first character is never a letter contradicts the standard convention used in most languages.

10. Where are automatic local variables defined within a function typically allocated and deallocated?

A. On the stack, with lifetime tied to function calls.

B. On the heap.

C. In static storage.

D. In registers only.

Automatic local variables have automatic storage duration: they come into existence when the function is entered and disappear when the function returns. They're typically allocated in the function's stack frame, so their lifetime is tied to the function call. They aren't placed on the heap (that would be dynamic allocation), nor in static storage (which lasts for the entire program). While a compiler may keep some in registers for efficiency, you can't rely on them being restricted to registers—the usual behavior is stack allocation with lifetime tied to the function.

SAMPLE

Next Steps

Congratulations on reaching the final section of this guide. You've taken a meaningful step toward passing your certification exam and advancing your career.

As you continue preparing, remember that consistent practice, review, and self-reflection are key to success. Make time to revisit difficult topics, simulate exam conditions, and track your progress along the way.

If you need help, have suggestions, or want to share feedback, we'd love to hear from you. Reach out to our team at hello@examzify.com.

Or visit your dedicated course page for more study tools and resources:

https://has106a.examzify.com

We wish you the very best on your exam journey. You've got this!

SAMPLE