Danyliuk Ivan, PhD in Physics and Mathematics, Associate Professor, Associate Professor of the Department of Applied Mathematics and Information Technologies, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine, ORCID: http://orcid.org/0000-0002-6637-9557

APPLICATION OF CONTAINERIZATION FOR UNIFICATION OF THE DEVELOPMENT ENVIRONMENT IN TEACHING THE C LANGUAGE

Abstract

The article addresses the problem of development environment instability when teaching information technology students to detect memory management errors in the C programming language, where heterogeneous operating systems, divergent library versions and platform-specific tooling collectively erode the practical training time available to instructors and students. The main classes of memory management defects encountered in introductory and advanced systems programming courses are characterised – memory leaks, use-after-free, buffer overflow, double-free and use of uninitialised memory – together with modern detection instruments: Valgrind/ Memcheck and AddressSanitizer for dynamic analysis, Cppcheck for static analysis, and the GDB debugger for interactive investigation of failures. It is shown that in real student groups working on a mixed Linux/Windows/macOS fleet, a significant share of study time is spent on resolving technical environment issues rather than on forming target competencies – a phenomenon known as the “works on my machine” problem. A Docker-based containerisation approach is proposed in which the entire tool environment is described declaratively in a Dockerfile stored alongside the course source code under version control, thus realising the “Infrastructure as Code” principle in the educational context. The three-tier architecture of the solution is described: the

host operating system, a Docker image based on debian:bookworm-slim, and integration with Visual Studio Code via the Dev Containers mechanism, with attention to the security parameters required for GDB inside the container. Specifics of deployment on Linux, on Windows via WSL2 and on macOS via OrbStack are discussed separately, including filesystem performance considerations. A comparative analysis of native and Docker approaches across nine educational criteria – including reproducibility, project isolation, environment versioning and assignment verification complexity – is performed. Practical approbation in a Programming course confirms a fivefold reduction in environment-related instructor consultations and an increase from 47% to 72% in the share of students who successfully configure the complete toolchain at the first session. The approach is recommended as a standard for organising practical work in systems programming courses and creates the basis for automated assignment assessment.

Keywords: C language, memory management, memory leaks, Docker, containerization, Valgrind, AddressSanitizer, Dev Containers, environment reproducibility, WSL2.

Problem statement

The C programming language remains one of the cornerstones of training specialists in software engineering, computer science and applied informatics. The classic textbook by Kernighan and Ritchie [1], which has formed the basis of most university curricula for over three decades, clearly formulates the fundamental principle: in C, the programmer bears full responsibility for allocating and freeing dynamic memory. This property gives the language maximum flexibility and efficiency, making it indispensable in systems programming, operating system kernel development, embedded systems, high-performance computing and modern infrastructure software. At the same time, the absence of automatic memory

management opens a wide class of potential defects: memory leaks, use-after-free, buffer overflows, double-free errors and reads from uninitialised memory. According to industry analyses cited by Seacord [2], the majority of critical security vulnerabilities catalogued by the Common Vulnerabilities and Exposures (CVE) database for systems written in C and C++ are directly attributable to errors in dynamic memory management.

Practical instruction in detecting these errors requires a stable and reproducible toolchain: a GCC or Clang compiler, dynamic analysis tools Valgrind/Memcheck [3] and AddressSanitizer [4], the static analyzer Cppcheck, and the GDB debugger. The problem is that in real student groups, students work in heterogeneous environments: different Linux distributions (Debian 11/12, Ubuntu 20.04/22.04/24.04, Fedora, Arch Linux), as well as Windows and macOS. This leads to discrepancies in the output format of analyzers, incompatibilities between library versions (in particular, libasan6 in Ubuntu 20.04 versus libasan8 in Ubuntu 24.04), differences in default compiler flags, and, as a consequence, significant losses of study time on resolving purely environmental problems unrelated to the subject of the course. This phenomenon, known as the “works on my machine” problem, is especially acute in introductory courses, where students lack the experience to diagnose environmental conflicts and cannot distinguish a bug in their own code from a tool malfunction or a packaging idiosyncrasy of their distribution.

The connection between this problem and important practical and scientific tasks is twofold. First, ensuring the reproducibility of the development environment is a precondition for the reliability of empirical results obtained from any computational experiment, including educational ones – a requirement explicitly recognised in the modern software quality standard ISO/IEC 25010:2023 [5] and, more specifically, in the IEEE 2675-2021 Standard for DevOps [15]. Second, the formation of competencies in handling memory errors

directly affects national cybersecurity, since the bulk of critical infrastructure software, embedded controllers and operating systems continues to be written in C and its derivatives.

Analysis of recent research and publications

Detection and prevention of memory errors in C programs is a research domain with a long and well-developed tradition. The seminal work of Nethercote and Seward [3] on Valgrind described the architecture of a heavyweight dynamic binary instrumentation framework based on shadow memory, which to this day remains the de facto reference instrument for detecting subtle memory errors, including reads from uninitialised memory. The team led by Serebryany [4] from Google proposed AddressSanitizer (ASan) – a fundamentally different approach implemented at the level of compiler instrumentation, providing a slowdown of approximately 2× relative to native execution and immediate diagnostic output at the moment of access violation. Both tools are deeply integrated into modern compilers (GCC and Clang) and are widely used both in industry and in academic instruction. Seacord [2], in a comprehensive treatment of secure coding in C and C++, classifies typical memory management defects from the standpoint of exploitability and provides systematic recommendations for their elimination, which can be directly transferred into the educational process.

The economic perspective on defect detection was systematised by Boehm [6] in the foundational work on software engineering economics. He demonstrated that the cost of fixing defects grows nonlinearly depending on the stage at which they are discovered. This principle applies directly to the educational process: environmental problems identified at the very first session cost far less – in time, motivation and attention – than those that accumulate throughout the semester and degrade the student's perception of the course as a whole. Wilson and colleagues [7], in a study of best practices in scientific computing, separately identify the reproducibility of the computational environment as a key condition for quality

work with code, listing it alongside version control, automated testing and code review – a conclusion that fully applies to the educational context, where the instructor effectively performs the role of “principal investigator” of the educational “experiment”.

The reproducibility problem in computational research has acquired particular acuity in the past decade in connection with the widespread adoption of containerisation technologies. Boettiger [8] proposed Docker as a generic solution to the reproducibility crisis in computational science, demonstrating that container images stored in public registries enable bit-exact reproduction of computational pipelines years after the original publication. Cito and Gall [9] empirically investigated the use of Docker in software engineering research and confirmed its effectiveness as a tool for delivering reproducible experimental environments. Merkel [10], in the original Linux Journal article that effectively introduced Docker to a wide professional audience, described the technology as a means of packaging applications together with their dependencies into lightweight containers that run identically on different operating systems. The Open Container Initiative [11] later formalised the resulting image format and runtime specification, which guarantees long-term portability of the approach.

In the educational context, integration of containerised environments with development tools is supported by the Development Containers specification [12], which standardises the description of the editor-side environment and is implemented by Visual Studio Code, JetBrains products and a number of cloud platforms. The methodological foundations of competency-based curriculum design in computing, including expectations regarding systems-level skills such as memory analysis, are summarised in the joint ACM/IEEE Computing Curricula 2020 framework [13], which explicitly highlights the ability to operate within reproducible engineering environments as a transferable competency.

Despite the considerable body of literature on individual elements of the proposed solution – Valgrind, AddressSanitizer, Docker, Dev Containers – there remains a clearly unsolved part of the general problem: the absence of a holistic, methodically substantiated approach that combines containerisation with editor integration specifically for the task of teaching C memory analysis in a heterogeneous student environment, together with quantitative confirmation of its educational effectiveness. The present article is devoted to filling this gap.

Aim of the article

The aim of this work is to substantiate and describe a Docker-based containerisation approach as a means of unifying the tool environment when teaching memory analysis in C, as well as to provide a comparative analysis of this approach against native installation of the toolchain, to specify the architecture of the proposed solution at the level of an editor configuration and a Dockerfile, to describe the platform-specific aspects of deployment on Linux, Windows and macOS, and to present the results of practical approbation in a real third-year student group. A subsidiary aim is to formulate recommendations for the construction of similar environments for other systems-programming courses and to outline directions for the automation of assignment assessment based on a shared container image.

Presentation of the main material

1. Classification of memory management errors in C

For the purposes of the educational process, we identify the main classes of memory management defects in C, ordered roughly by the difficulty of their diagnosis without specialised tools.

Memory leaks – a situation in which allocated dynamic memory is not freed after its use is complete. During prolonged program execution, this leads to gradual exhaustion of the address space, swapping, and ultimately to the termination of the process by the operating system's out-of-memory killer. This is

the most common class of error among novice programmers, since a program may continue to "work" for a considerable time despite the presence of a leak – short test runs do not expose the defect, and even relatively long executions may complete normally if the process exits before resources are exhausted. Pedagogically, this means that a simple manual test cannot detect a leak; a specialised instrument such as Valgrind/Memcheck is required.

Use-after-free (UAF) – accessing a block of memory after a call to free(). This error is a potential security vulnerability that may allow arbitrary code execution if the freed region is subsequently re-allocated and overwritten by attacker-controlled data, and at minimum leads to unpredictable program behaviour. UAF errors are notoriously difficult to reproduce, since the visible consequences depend on the exact sequence of subsequent allocations.

Buffer overflow / underflow – writing or reading beyond the boundaries of an allocated memory block. This is one of the most dangerous vulnerabilities in system software, with consequences ranging from silent data corruption in adjacent allocations to remote code execution in the case of stack-based overflows. From the pedagogical perspective, the importance of this class is reinforced by the fact that classic exploitation techniques (return-into-libc, return-oriented programming) are built on top of buffer overflow primitives.

Double free – a repeated call to free() for an already freed block, which corrupts the memory allocator's internal data structures and can result either in an immediate program crash or in an exploitable vulnerability of the heap metadata.

Use of uninitialised memory – reading values from variables or memory blocks before their explicit initialisation, which produces undefined behaviour and is the source of intermittent, non-reproducible bugs that depend on the previous content of the memory region.

2. Modern tools for memory analysis

Valgrind/Memcheck. Valgrind is a framework for heavyweight dynamic binary instrumentation, described in detail by Nethercote and Seward [3]. Its key tool, Memcheck, implements the “shadow memory” technique: for each byte of the program's memory, an additional status byte is maintained that tracks the addressability and initialisation state of each bit. Running Valgrind requires only that the program be compiled with the -g -O0 flags:

valgrind --leak-check=full --show-leak-kinds=all ./program

Valgrind detects all five of the mentioned error classes, provides a detailed call stack at the point of each error, and produces a summary HEAP SUMMARY report. An execution slowdown of 10–50× is acceptable for educational programs of moderate size. The pedagogical advantage of Valgrind is its human-readable output, which a student can interpret directly without additional knowledge of the compiler's internals, and its ability to detect uninitialised reads – a class of errors that is invisible to AddressSanitizer.

AddressSanitizer (ASan), developed by the Google team [4], is implemented as compiler instrumentation: when compiled with the -fsanitize=address flag, checks are inserted before every memory access. This results in a slowdown of only ~2× compared to the 10–50× of Valgrind. An important pedagogical advantage of ASan is that the error message appears at the moment the error occurs, indicating the exact source file and line number – in line with the principle of immediate feedback in learning. ASan is particularly effective for buffer overflows and use-after-free, where the prompt and precise diagnostic message significantly accelerates the formation of mental models of memory layout in students.

Both tools are complementary: ASan is more effective for detecting access errors and rapid checking during development, while Valgrind additionally tracks uninitialised values and multithreaded access issues via its Helgrind tool. In the

proposed methodology students learn to use both, switching between them depending on the diagnostic task at hand.

Static analysis: Cppcheck. Cppcheck detects potential errors without executing the program. In the educational context it serves as the first line of code review before the testing stage, allowing obvious errors to be caught at the writing stage. Static analysis is particularly valuable for catching defects on infrequently executed code paths that may escape dynamic analysis if the corresponding test inputs are not exercised. Standards such as MISRA C:2012 [14], although primarily aimed at safety-critical industrial software, illustrate the same general principle of relying on static rule checking to enforce a defined memory and aliasing discipline.

Interactive debugging: GDB. The GNU Debugger remains the principal interactive instrument for stepping through execution, inspecting memory and locating the precise origin of a fault. Modern integrated development environments embed GDB as a back-end and provide a graphical interface over its core functionality. As discussed below, the requirement to use GDB inside a Docker container imposes specific demands on container security configuration, which the proposed methodology takes into account.

3. The reproducibility problem in heterogeneous educational environments

The current edition of ISO/IEC 25010:2023 [5] defines Reliability as one of nine top-level product quality characteristics; its four subcharacteristics – faultlessness, availability, fault tolerance, and recoverability – collectively describe the dependability of a software product without explicitly naming reproducibility as a distinct term. Environment reproducibility is, however, formalised as a first-class engineering requirement by IEEE 2675-2021 [15], the IEEE Standard for DevOps, which mandates that build, package, and deployment environments be defined in a deterministic and traceable manner – a principle that

extends directly to educational toolchains, where the instructor's environment must behave identically on every student machine. Among the key changes in the new edition are the expansion of the quality model to nine characteristics (Safety was added), the renaming of Usability to Interaction Capability, and Portability to Flexibility. In the context of the educational process, reproducibility takes on a practical meaning: the learning environment must deliver the same analysis result for the same source code regardless of which machine the work is performed on – whether belonging to the student, the instructor, or an automated assessment system.

In practice, achieving this property through native installation is complicated by several factors. First, different Linux distribution versions are simultaneously present in a single student group, with non-trivial differences in default compiler flags, glibc versions and packaging conventions. Second, the libasan library – the runtime support component of AddressSanitizer – exhibits noticeable behavioural changes between versions (libasan6 on Ubuntu 20.04 versus libasan8 on Ubuntu 24.04), including the format of error reports and the set of detected sub-classes. Third, the fundamental differences among Linux, Windows and macOS – the three platforms simultaneously present in a typical student group – manifest in incompatible toolchain ecosystems: Valgrind, for instance, has historically lacked first-class support on macOS for recent kernel versions, and on native Windows altogether.

The Wilson et al. study of best practices in scientific computing [7] explicitly recommends pinning the entire computational environment to specific versions and treating it as a versioned artefact. Boettiger [8] further showed that this recommendation, when combined with container technology, allows for reproduction of computational results by independent researchers years after publication. In the educational context, the same principles imply that the

instructor should be able to fix the environment once and have students reuse exactly that environment without redoing the configuration work.

4. Docker as an environment encapsulation mechanism

Merkel [10] described Docker as a technology that allows applications to be packaged together with their dependencies into lightweight containers that run identically on different operating systems. For our purposes, the key property is a deterministic runtime environment: a container image described in a Dockerfile always contains exactly the same packages at exactly the same versions, regardless of the host system. The container image format and runtime are now standardised by the Open Container Initiative [11], which protects long-term investment in containerised tooling against vendor lock-in.

A Dockerfile is a declarative description of the environment – a text file stored in the version control system alongside the project code. This implements the “Infrastructure as Code” principle with respect to the learning environment: the instructor fixes the exact toolset once, and all students receive an identical environment. Critically for the educational use case, this same Dockerfile becomes a self-documenting artefact: any student who reads it can see precisely which compiler version, sanitiser library and analysis tool will be used, eliminating the “hidden curriculum” of undocumented environment expectations.

5. Architecture of the proposed Docker-based learning environment

The proposed solution consists of three tiers (Fig. 1).

Tier 1. Host system – any OS: Linux, Windows 10/11, or macOS. Requires only Docker Engine (or a compatible runtime) and Visual Studio Code. The host system retains its own user-level tools, browsers and document editors; Docker introduces no global modifications to the host's package manager state.

Tier 2. Docker image based on debian:bookworm-slim with a fixed toolset: GCC 12, GDB 13, Valgrind 3.19, AddressSanitizer (libasan8), Cppcheck 2.10.

The image is described by a Dockerfile stored in the .devcontainer/ directory of the course repository, alongside the source code of practical assignments.

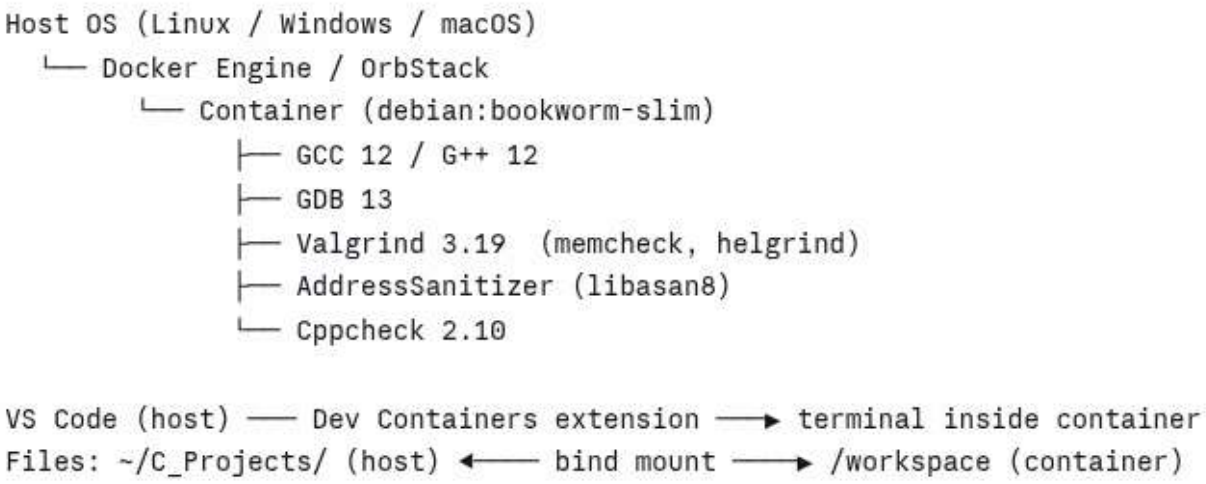


Figure 1 – Architecture of the Docker-based learning environment

The choice of debian:bookworm-slim as a base is motivated by its small footprint (under 80 MB before tool installation), its long support cycle, and its precise correspondence to the Debian/Ubuntu family that students most commonly use natively.

Tier 3. Development environment. VS Code connects to the container via the Dev Containers extension [12], providing the student with the familiar editor interface with all tools running inside the container. Project files are stored in the home directory of the host system (~/C_Projects/) and mounted into the container via the bind mount mechanism, ensuring data persistence between sessions and maximum file I/O performance – bind mount throughput within a single Linux kernel reaches 97–99% of native speed.

The key configuration file is .devcontainer/devcontainer.json, which describes the container launch parameters, VS Code extension installation, and a post-creation verification command. In particular, the parameters --cap-add=SYS_PTRACE and --security-opt seccomp=unconfined are required for the

GDB debugger to function correctly inside the container, since GDB uses the ptrace system call, which is blocked by Docker's default security profile. These parameters relax security in a controlled manner: they affect only the dedicated learning container and do not influence the host system or other containers. The complete VS Code configuration for both environment variants – native and Docker – with detailed setup instructions is available in the open repository [16].

6. Platform-specific deployment considerations

An important advantage of the Docker approach is its platform independence, which directly solves the problem of heterogeneous operating systems in student groups. Nevertheless, each platform requires specific attention to filesystem layout and runtime selection.

On ***Linux***, Docker Engine is installed natively via the package manager without any additional tools, providing maximum performance and the simplest possible setup. There is no virtualisation layer between the container and the host kernel, so I/O latency is minimal and process startup is virtually instantaneous.

For ***Windows 10/11***, the recommended option is to install Docker Engine inside the WSL2 subsystem (Windows Subsystem for Linux 2) without Docker Desktop, which provides native Linux kernel performance and full support for all memory analysis tools, including Valgrind. Project files should be stored in the WSL2 Linux filesystem (~/C_Projects/), not on the Windows drive (/mnt/c/) – this delivers 97–99% of native speed versus ~40% when working with the Windows filesystem due to the file-by-file translation through the 9P protocol. Educating students about this filesystem distinction is itself a useful pedagogical side-effect of the proposed methodology.

On ***macOS***, Docker Engine cannot run without a Linux kernel as a substrate. The recommended free alternative is OrbStack [17], a fully compatible replacement for Docker Desktop that requires no commercial license for personal use and automatically suspends the VM when Docker is idle, minimising battery

impact on portable devices. OrbStack uses native Apple Virtualization Framework, which provides near-native I/O performance for the bind-mounted source tree.

In all three cases the student uses the same unmodified Dockerfile, which serves as the single entry point for student groups with a mixed operating system fleet. The instructor is freed from the obligation to maintain three parallel sets of platform-specific instructions and can concentrate on the substantive content of the course.

7. Comparative analysis of native and Docker approaches

A comparison of two approaches for a Linux system – native tool installation and Docker containerisation – was performed. Criteria relevant to the educational process were evaluated (Table 1).

Table 1 – Comparative characteristics of the native and Docker approaches

Criterion	Native	Docker
Environment reproducibility	Low	High
Initial setup time	10–15 min	20–30 min
Dependency on distribution	High	None
Compilation speed	100%	97–99%
Project isolation	No	Yes
Environment versioning	Not possible	Dockerfile in git
Assignment verification complexity	High	Low
GDB support	Full	Requires --cap-add
Fixed libasan version	OS-dependent	Guaranteed

Analysis of the table shows that the only criterion by which the Docker approach is inferior to the native one is initial setup time. However, this difference is offset over the course of the semester: a one-time investment of 10–15 additional minutes at the outset is compensated by the elimination of

environmental problems at every subsequent session. The performance of both approaches is practically identical when project files are correctly located in the Linux filesystem.

Particularly important for the educational process is the assignment verification complexity criterion. With the native approach, the output format of Valgrind and ASan may differ between versions, making automated verification impossible and complicating peer review among students. With the Docker approach, the output is identical for both student and instructor, opening the possibility of automated assessment of assignments using a shared container image. In conjunction with continuous integration pipelines on platforms such as GitHub Actions or GitLab CI, this enables a fully reproducible grading workflow consistent with the principles of reproducible research articulated in [7] and [8].

8. Approbation in a real student group

The proposed approach was trialled in the Programming course for IT students at Yuriy Fedkovych Chernivtsi National University. The student group consisted of 24 students using various operating systems: Ubuntu 20.04/22.04/24.04, Debian 11/12, Windows 11 (via WSL2), and macOS. The composition of the group, in terms of the host operating system, was approximately representative of the broader student population at the faculty.

Under the native approach (previous semester), the number of instructor consultations regarding environment problems during the first month of the course was 31. Only 47% of students were able to independently set up the full toolchain at the first session. Typical issues included: incompatibility of libasan versions across distributions, incorrect GDB configuration on macOS, the absence of Valgrind support on Windows without WSL, and conflicts between system and user package managers (in particular, the use of pip/Homebrew to install tools that conflicted with system-provided versions).

Under the Docker approach (current semester), the number of environment-related consultations dropped to 6 – a fivefold reduction. The share of students who independently configured the complete environment at the first session rose from 47% to 72%. The nature of the questions also changed qualitatively: students moved from technical environment issues to substantive questions about interpreting Valgrind and ASan output, which is direct evidence of improved learning effectiveness. Anonymous end-of-semester student feedback confirmed that the unified environment was perceived as a positive feature of the course, and several students reported subsequently reusing the same Dockerfile as a starting template in other courses and personal projects.

The results are consistent with the theoretical argument of Boehm [6] regarding the nonlinear growth of defect-correction costs depending on the stage of detection: resolving environmental problems at the start of the course is far less costly than their accumulation across the semester.

9. Pedagogical implications and limitations

Beyond the immediate quantitative outcomes, the proposed methodology has several pedagogical implications worth highlighting. First, by making the environment itself a versioned, inspectable artefact, the course implicitly trains students in the habit of treating infrastructure as code – a transferable skill explicitly listed in CC2020 [13] among the competencies expected of graduates of computing programmes. Second, the consistency of tool output between the student's machine, the instructor's machine and any automated grader removes a class of conflicts that would otherwise consume teacher-student communication bandwidth; the freed bandwidth can be redirected to substantive feedback on memory analysis. Third, the use of Docker creates a low-friction path for students to take their environment into subsequent courses, internships and graduate work, providing a long-term return on the up-front configuration effort.

The principal limitations of the approach are also worth recording. The Docker environment introduces a small but non-zero abstraction barrier between the student and the host operating system; students who require deep familiarity with a specific Linux distribution's packaging system will eventually need supplementary native exercises. On older or low-end hardware (notably Windows machines with fewer than 8 GB of RAM and HDD storage), the WSL2 VM may impose a noticeable boot-time and memory-pressure overhead, which can be partially mitigated by limiting WSL2 resources via the .wslconfig file. Finally, the methodology requires that the host has reliable Internet access for the initial image pull; in environments where this assumption fails, the instructor should distribute a pre-built image as an offline tar archive.

Conclusions

The application of Docker containerisation for organising the learning environment in a course on C memory error detection has proved to be a substantiated and practically effective approach. The pedagogical characterisation of memory management error classes and their detection tools – Valgrind/Memcheck and AddressSanitizer as dynamic analysers, Cppcheck for static analysis, and GDB for interactive debugging – provided a coherent foundation for designing the environment around the actual diagnostic workflow. A three-tier architecture was proposed in which the host operating system, a Docker image based on debian:bookworm-slim, and Visual Studio Code connected through the Dev Containers specification jointly deliver an identical toolchain on Linux, Windows and macOS through a single Dockerfile.

A comparative analysis across nine educational criteria confirmed the advantages of containerisation in reproducibility, project isolation, environment versioning and automated assignment verification; the only concession – a slightly longer initial setup – is fully amortised within a semester. Approbation in a IT student group confirmed a fivefold reduction in environment-related consultations

and a rise in independent first-session setup from 47% to 72%, accompanied by a qualitative shift of student questions towards the substantive interpretation of analyser output. The principal limitations – the abstraction barrier over the host OS, WSL2 resource overhead on low-end hardware, and the dependence on Internet access for the initial image pull – are explicitly formulated for future deployments.

Further research directions include automated assignment assessment based on the shared Docker image, evaluation of cloud environments such as GitHub Codespaces and GitPod as alternatives for resource-constrained students, and extension of the methodology to adjacent systems-programming courses.

References

1. Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (2nd ed.). Prentice Hall.

2. Seacord, R. C. (2013). *Secure coding in C and C++* (2nd ed.). Addison-Wesley Professional.

3. Nethercote, N., & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)* (pp. 89–100). ACM. https://doi.org/10.1145/1250734.1250746

4. Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)* (pp. 309–318). USENIX Association. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

5. ISO/IEC 25010:2023. (2023). *Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – Product quality model*. ISO. https://www.iso.org/standard/78176.html

6. Boehm, B. W. (1981). *Software engineering economics.* Prentice-Hall.

7. Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., & Wilson, P. (2014). Best practices for scientific computing. *PLOS Biology*, *12*(1), Article e1001745. https://doi.org/10.1371/journal.pbio.1001745

8. Boettiger, C. (2015). An introduction to Docker for reproducible research, with examples from the R environment. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882

9. Cito, J., & Gall, H. C. (2016). Using Docker containers to improve reproducibility in software engineering research. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)* (pp. 906–907). ACM. https://doi.org/10.1145/2889160.2891057

10. Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), Article 2. https://doi.org/10.5555/2600239.2600241

11. Open Container Initiative. (2023). *OCI image format specification, v1.1.0*. https://github.com/opencontainers/image-spec

12. Microsoft Corporation. (2023). *Development containers specification*. https://containers.dev/

13. CC2020 Task Force. (2020). *Computing curricula 2020: Paradigms for global computing education*. Association for Computing Machinery. https://doi.org/10.1145/3467967

14. MISRA. (2012). *MISRA C: 2012 – Guidelines for the use of the C language in critical systems*. Motor Industry Software Reliability Association.

15. IEEE. (2021). *IEEE standard for DevOps: Building reliable and secure systems including application build, package, and deployment* (IEEE Std 2675-2021). IEEE. https://doi.org/10.1109/IEEESTD.2021.9415476

16. Danyliuk, I. (2026). *VS Code configuration for C development on Linux: Native and Docker approaches* [Online resource]. https://github.com/uixcoder/c_utils /blob/main/memory_leaks/linux/vscode_config_linux_c_dev.md

17. Danyliuk, I. (2026). *C development environment setup via Docker* [Online resource]. https://github.com/uixcoder/c_utils/blob/main/memory_leaks/crossplatform/ vscode_config_docker_crossplatform.md