УДК 004.43:004.415.5 **DOI:10.15330/itee.2025.2.05**

Р. І. ЗАПУХЛЯК, канд. фіз.-мат., наук, В. В. ДОВГИЙ, канд. техн. наук

БЕЗПЕКА ПАМ'ЯТІ У C++: ІНІЦІАТИВИ WG21 (LIFETIME, CONTRACTS, SAFETY PROFILES) ТА ЕМПІРИЧНИЙ АНАЛІЗ OPEN-SOURCE ПРОЄКТІВ

Вступ

C++ залишається ключовою мовою системного програмування завдяки високій продуктивності, контролю над ресурсами та багатим абстракціям. Водночас саме доступність низькорівневих механізмів (вказівники, арифметика адрес, неініціалізована пам'ять, ручне керування буферами) зумовлює ризики порушення memory safety. Уразливості пам'яті лежать в основі значної частини інцидентів безпеки у великих програмних системах, а undefined behavior (UB) [2] (UB) у C/C++ створює додатковий клас помилок: програма може демонструвати коректну поведінку в одних умовах і бути вразливою або некоректною в інших.

Останніми роками в межах WG21 [2] активізовано роботу над «безпечнішою C++» як поєднанням: (1) профілів та контрактів, що дозволяють підвищувати рівень перевірок; (2) формалізації lifetime-аналізу (зокрема на основі Core Guidelines Lifetime profile [3]); (3) практик та інструментів (компіляторні попередження, статичний аналіз, санітайзери [1]). Мета цієї роботи — надати огляд ініціатив WG21 [2] та показати відтворюваний експериментальний зріз для трьох репрезентативних open-source бібліотек.

Завдання дослідження:

1) систематизувати напрями Lifetime Safety [3, 8], Contracts [5], Safety Profiles [4, 6, 7];

2) розробити протокол експерименту (g++ + санітайзери [1]) та отримати вимірювані результати;

3) інтерпретувати результати у контексті ініціатив WG21 [2] та обмежень експерименту.

Дослідження має характер відтворюваного case study та не претендує на статистично репрезентативну оцінку всієї екосистеми C++

Класи помилок і роль undefined behavior (UB)

Під memory safety у контексті C++ зазвичай розуміють запобігання або виявлення принаймні таких класів помилок: out-of-bounds доступи (читання/запис поза межами буфера), use-after-free (доступ після звільнення), double free, dangling references/iterators, некоректне перетворення типів та порушення вирівнювання. Значна частина таких ситуацій класифікується стандартом як UB, що надає компілятору свободу оптимізацій. Як наслідок, UB може мати неочевидні прояви, включно з видаленням «захисних» перевірок під час оптимізації.

RAII та інваріанти часу життя

RAII (Resource Acquisition Is Initialization) є фундаментальним механізмом C++ для керування ресурсами: ресурс набувається під час ініціалізації об'єкта і звільняється в деструкторі. Однак RAII не усуває потребу у правильному керуванні посиланнями/вказівниками на об'єкти, час життя яких може завершитися раніше за час життя посилання. Саме цей розрив (lifetime mismatch) породжує dangling references та use-after-free. Статичні правила (напр., Core Guidelines Lifetime profile [3]) та контракти можуть формалізувати інваріанти та зробити їх перевірку системнішою.

Інструментальний рівень: санітайзери

Санітайзери - це інструменти, що додають інструментування до коду під час компіляції. AddressSanitizer (ASan) [1] виявляє широке коло помилок пам'яті (out-of-bounds, use-after-free тощо) і широко застосовується в індустрії та open-source. UBSan спрямований на виявлення

виконуваних проявів UB (переповнення, некоректні зсуви, порушення вирівнювання, некоректні перетворення типів тощо). У роботі санітайзери [1] застосовано як практичний критерій «виявлення» проблем у тестових сценаріях.

Ініціативи WG21: Lifetime, Contracts, Safety Profiles

Під Lifetime Safety [3, 8] у рамках WG21 [2]/SG23 часто мають на увазі розвиток підходів, що дозволяють компілятору/аналізатору виявляти типові випадки dangling (use-after-free) на основі локального аналізу, не вимагаючи повноцінного borrow-checker на кшталт Rust. Репрезентативним документом є опис Core Guidelines Lifetime profile [3], де формалізовано набір правил і підхід до діагностики типових помилок часу життя (зокрема через анотації/атрибути та аналіз повернених значень).

Контракти дозволяють задавати преумови, постумови та твердження (assertions) як частину інтерфейсу. Це дає можливість: (1) документувати інваріанти; (2) виконувати перевірки з різними режимами (від вимкнених до жорстких); (3) інтегрувати контракти з профілями безпеки та hardening стандартної бібліотеки. У контексті memory safety контракти корисні для позначення припущень щодо розмірів, непорожності, допустимих діапазонів, валідності вказівників/ітераторів, що зменшує ймовірність UB.

Safety Profiles [4, 6, 7] розглядаються як механізм «конфігурованих» обмежень і перевірок, що можуть застосовуватися до коду без повної зміни мови. Ідея полягає у визначенні профілів (напр., lifetime/type профілі) як сукупності заборон/вимог та способів їх перевірки (компілятор, бібліотека, статичний аналіз, санітайзери [1]). Профілі можуть бути адаптовані до домену (embedded, safety-critical тощо) і співіснувати з існуючим C++ кодом.

Методологія експерименту

Об’єктами дослідження обрано три популярні C++ бібліотеки різного масштабу: fmt (форматування), spdlog (логування) та nlohmann/json (JSON). Мета — отримати відтворювані метрики на реальному коді: (1) кількість попереджень компілятора у конфігурації «warnings-lite»; (2) кількість унікальних спрацювань ASan під час виконання тестів; (3) кількість унікальних спрацювань UBSan (рядки `runtime error:`) під час тестів. Додатково фіксуються commit-ідентифікатори та LOC (cloc).

Середовище виконання

Опис середовища виконання наведено в табл. 1.

Таблиця 1.

Характеристика середовища виконання

Параметр	Значення
OS	Ubuntu (Linux kernel 6.8.0-101-generic, x86_64)
CPU	16 logical cores (nproc=16)
Compiler	g++ (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0
CMake	3.28.3
Make	GNU Make 4.3

Протокол збірки та тестування (g++)

Для забезпечення відтворюваності застосовано три незалежні конфігурації збірки для кожного проєкту. Усі команди виконувалися в окремих директоріях збірки. Паралелізм складання: `make -j4` (стабільний режим), тести запускалися через `ctest --output-on-failure`.

Конфігурація A: попередження (warnings-lite):

```
cmake -DCMAKE_CXX_COMPILER=g++ -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_CXX_FLAGS="-Wall -Wextra -Wpedantic -Wshadow -Wconversion -Wsign-conversion -
Wnull-dereference -Wdouble-promotion -Wformat=2 -Wduplicated-cond -Wduplicated-branches
-Wlogical-op -Wuseless-cast" ..
make -j4 > warnings_build_lite.log 2>&1
grep -E "warning:|warning\(" warnings_build_lite.log | wc -l
Конфігурація B: AddressSanitizer (ASan) [1].
cmake -DCMAKE_CXX_COMPILER=g++ -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_CXX_FLAGS="-fsanitize=address -g -fno-omit-frame-pointer" \
  -DCMAKE_EXE_LINKER_FLAGS="-fsanitize=address" ..
make -j4 > asan_build.log 2>&1
ctest --output-on-failure 2>&1 | tee asan_ctest.log
grep -oE "SUMMARY: AddressSanitizer: [A-Za-z0-9_-]+" asan_ctest.log | sort | uniq -c
Конфігурація C: ASan + UBSan.
cmake -DCMAKE_CXX_COMPILER=g++ -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_CXX_FLAGS="-fsanitize=address,undefined -g -fno-omit-frame-pointer" \
  -DCMAKE_EXE_LINKER_FLAGS="-fsanitize=address,undefined" ..
make -j4 > asan_ubsan_build.log 2>&1
ctest --output-on-failure 2>&1 | tee asan_ubsan_ctest.log
grep -oE "runtime error: [^:]*" asan_ubsan_ctest.log | sort | uniq -c
```

Для порівняння проєктів різного розміру використано нормалізацію на 10 000 рядків коду (LOC):

Отже, основні результати наведені в табл. 2.

Таблиця 2.

Результати виконання експерименту

Проєкт	Commit	LOC	Попередження	ASan (унік.)	UBSan (унік.)
fmt	82553a7a5b78e446f7f559d7b968f606652ee5d2	18740	18	1	0
spdlog	0f7562a0f9273cfc71fddc6ae52ebff7a490fa04	3636	0	0	0
json (nlohmann/ json)	8167d2f64122c3ef7e5a0d9cab239fdba37aa54c	48347	0	0	0

Нормалізовані метрики експерименту продемонстровано в табл. 3.

Таблиця 3.

Нормалізовані метрики

Проєкт	warnings / 10k LOC	ASan / 10k LOC	UBSan / 10k LOC
fmt	9.61	0.53	0.00
spdlog	0.00	0.00	0.00
json (nlohmann/json)	0.00	0.00	0.00

На рисунках 1-4 продемонстровано діаграми результатів при експериментів та виявлення різних чинників.

Під час запуску тестового набору fmt в конфігурації ASan/ASan+UBSan зафіксовано один унікальний інцидент: `SUMMARY: AddressSanitizer: allocation-size-too-big`. Локалізація вказувала на тест `util_test.format_system_error` (файл `fmt/test/format-test.cc`, рядок 278). ASan перервав виконання, оскільки запитаний розмір виділення пам’яті перевищував максимальний підтримуваний ліміт. Важливо, що це не обов’язково вказує на експлуатовану уразливість у бібліотеці як такій – у тестах можуть бути негативні сценарії та перевірки граничних умов. Однак сам факт виявлення некоректного запиту демонструє корисність санітайзерів як інструменту для раннього виявлення помилок керування ресурсами або некоректної обробки даних.

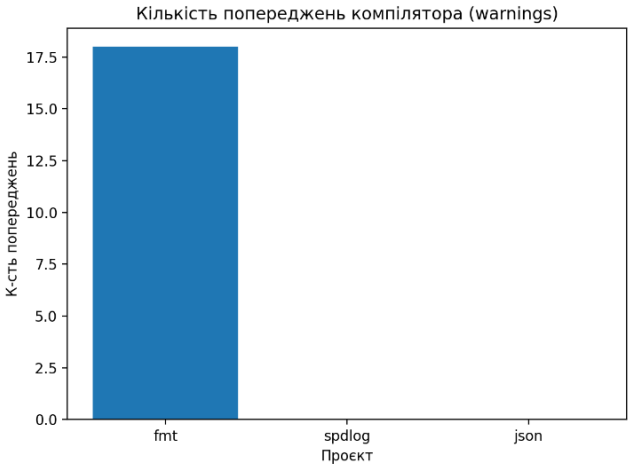

Рис. 1. Кількість попереджень компілятора (warnings)

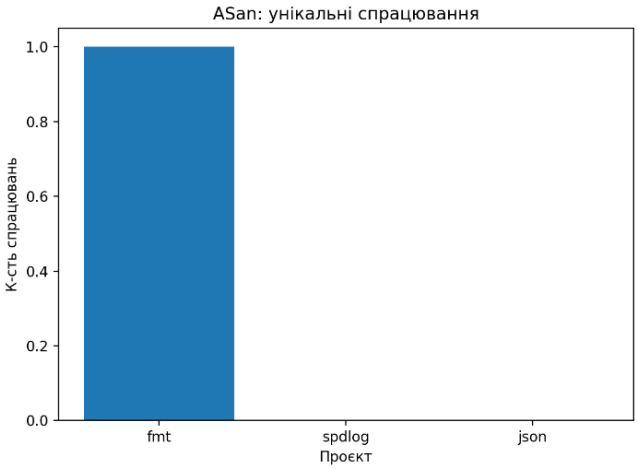

Рис. 2. Унікальні спрацювання ASan

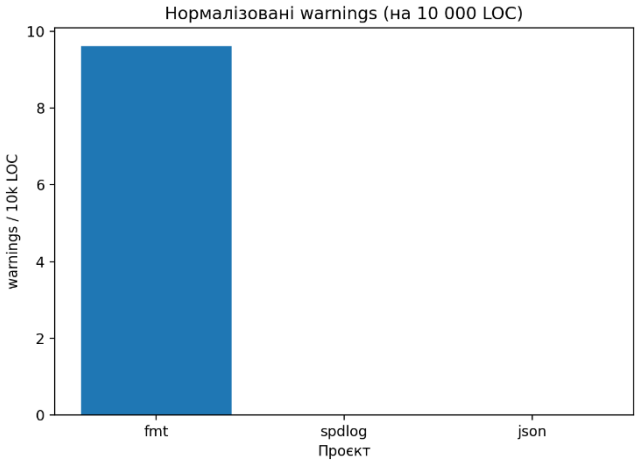

Рис. 3. Нормалізовані warnings (на 10 000 LOC)

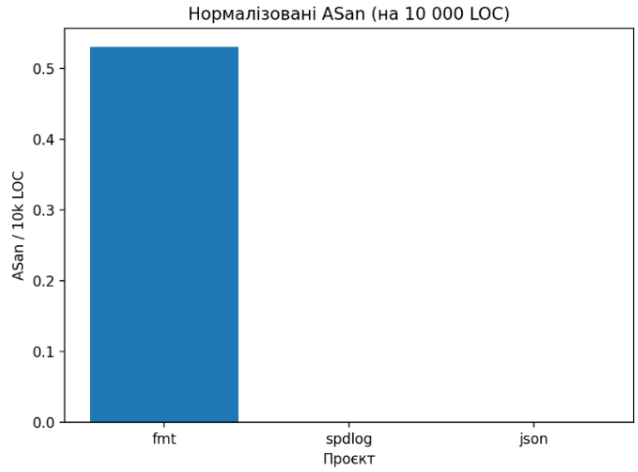

Рис. 4. Нормалізовані warnings (на 10 000 LOC)

Обґрунтування результатів у контексті ініціатив WG21

У конфігурації warnings-lite попередження з'явилися лише в fmt (18), тоді як spdlog та nlohmann/json зібралися без попереджень у заданому наборі прапорів. Це ілюструє, що «чистота» щодо попереджень є властивістю не лише якості коду, а й стилю проєкту, специфіки тестів та компіляторних налаштувань. Нормалізований показник для fmt становить 9,61 warnings/10k LOC.

ASan виявив один унікальний інцидент у fmt та не виявив спрацювань у spdlog і nlohmann/json. UBSan не зафіксував жодного `runtime error:` у тестових сценаріях усіх трьох проєктів. Це може означати: (1) відсутність UB у покритих тестами шляхах; (2) наявність UB, яке не активувалося в межах тестів; (3) UB, що проявляється лише під іншими компіляторами/прапорцями/платформами. Таким чином, результати слід інтерпретувати як «виявлено за умов конкретного експерименту», а не як абсолютний доказ відсутності UB.

Відповідність класів проблем напрямам WG21

Фіксація інциденту, пов'язаного з некоректним виділенням пам'яті, корелює з мотивацією напрямів WG21 [2]: lifetime/type профілі та контракти націлені на те, щоб або забороняти небезпечні патерни, або робити їх перевірюваними та керованими. Зокрема:

• Lifetime Safety [3, 8] може зменшувати ризики dangling/use-after-free через аналіз походження посилань та повернених значень;

• Contracts [5] дозволяють виражати передумови (наприклад, допустимі розміри, діапазони, валідність аргументів) і керувати семантикою перевірок;

• Safety Profiles [4, 6, 7] надають механізм оголошення та застосування наборів обмежень (наприклад, заборона reinterpret_cast або обмеження сирих вказівників) відповідно до доменних вимог.

Обмеження експерименту:

1) Вибірка з трьох бібліотек не репрезентує всю екосистему C++.

2) Виявлення санітайзерів залежить від тестового покриття та умов виконання.

3) У тестових наборах можуть бути «негативні» тести (наприклад, assert-test у fmt), що навмисно призводять до аварійної зупинки.

4) Результати отримано для g++ 13.3; інші компілятори або налаштування можуть дати інші спрацювання.

5) Показники warnings залежать від набору прапорів і політики попереджень у проєкті.

Порівняння з попередніми емпіричними дослідженнями

Отримані в роботі результати демонструють низьку частоту runtime-виявлень (ASan/UBSan) у трьох зрілих open-source бібліотеках за умов виконання штатних тестових наборів. Такі спостереження узгоджуються з висновками низки попередніх емпіричних досліджень, присвячених проблематиці undefined behavior (UB) та безпеки пам’яті в мовах сімейства C/C++.

Зокрема, у роботі Yang та ін. [11] показано, що навіть за наявності формально коректного коду компіляторні оптимізації можуть індукувати некоректну поведінку за умов UB, що ускладнює його виявлення. Це підкреслює, що відсутність runtime-спрацювань у конкретному експерименті не еквівалентна відсутності потенційно небезпечних конструкцій у кодовій базі. Натомість це свідчить про те, що в межах протестованих сценаріїв такі конструкції або відсутні, або не активуються.

Дослідження, присвячені аналізу вразливостей у прикладах C/C++-коду з відкритих джерел [12], демонструють, що значна частина memory-related помилок виникає у фрагментах з недостатнім тестуванням або у навчальних/демонстраційних прикладах. На цьому тлі низька кількість спрацювань у бібліотеках fmt, spdlog та nlohmann/json може бути пояснена високим рівнем зрілості цих проєктів, наявністю розвинених CI-пайплайнів та регулярного використання інструментів статичного й динамічного аналізу.

У роботах, присвячених Rust та memory-safe мовам [13], підкреслюється роль статичних механізмів перевірки часу життя та власності як способу радикального зменшення класів помилок use-after-free і dangling references. У цьому контексті ініціативи WG21 щодо Lifetime Safety [3, 8] та Safety Profiles [4, 6, 7] можна розглядати як еволюційний підхід до часткової імплементації подібних гарантій у межах існуючої парадигми C++. Отримані експериментальні результати опосередковано підтверджують доцільність такого поетапного підсилення гарантій: навіть без повної зміни мовної моделі поєднання тестування та санітайзерів дозволяє знизити прояви memory-related дефектів у виконуваних сценаріях.

Огляд сучасних технік забезпечення memory integrity для мов без повної статичної безпеки пам’яті [14] також наголошує на багаторівневому характері захисту: поєднанні мовних обмежень, профілів безпеки, компіляторних перевірок та runtime-інструментування. Результати цього дослідження узгоджуються з таким багаторівневим підходом: використання ASan/UBSan у поєднанні з дисципліною розробки та тестуванням у зрілих бібліотеках демонструє практичну ефективність інструментального рівня, тоді як ініціативи WG21 можуть надати додаткові статичні гарантії на рівні мови та стандартної бібліотеки.

Таким чином, результати проведеного case study не суперечать попереднім емпіричним спостереженням, а доповнюють їх у прикладному аспекті, демонструючи реальний стан

memory-safety у конкретних популярних C++-проєктах за умов сучасних інструментів аналізу. Водночас вони підтверджують, що для повнішої картини необхідні ширші вибірки, різні компіляторні конфігурації та додаткові методи аналізу, що визначає напрями подальших досліджень.

Наукова новизна роботи полягає у поєднанні систематизованого аналізу сучасних ініціатив WG21 у сфері підвищення безпеки C++ (Lifetime Safety, Contracts, Safety Profiles) з відтворюваним емпіричним дослідженням реальних open-source проєктів на основі стандартизованого протоколу збірки та тестування з використанням санітайзерів (ASan, UBSan). На відміну від суто концептуальних оглядів або ізольованих експериментальних звітів, у роботі запропоновано інтегрований підхід, що поєднує нормативно-стандартизаційний контекст (WG21) з практичними метриками (кількість попереджень, унікальні runtime-спрацювання, нормалізація на 10 000 LOC). Отримані результати дозволяють конкретизувати реальний рівень проявів memory-related дефектів у зрілих C++ бібліотеках за умов сучасних інструментів аналізу та формують емпіричну основу для оцінки доцільності подальшого розвитку профілів безпеки та механізмів формалізації часу життя в стандарті C++.

Висновки

У роботі систематизовано сучасні ініціативи WG21 [2] у сфері підвищення безпеки C++ та проведено відтворюваний експеримент на трьох open-source бібліотеках. Емпіричні результати показали відсутність UBSan-спрацювань у тестових сценаріях та один унікальний інцидент ASan у fmt, а також відмінності у кількості компіляторних попереджень. Це підтверджує, що поєднання інструментального рівня (санітайзери [1], попередження) з еволюцією мови (контракти, профілі, lifetime-аналіз) є практично обґрунтованою стратегією. Подальша робота може включати розширення вибірки, порівняння компіляторів (Clang/LLVM) та дослідження впливу різних профілів/режимів контрактів на реальні кодові бази.

Список літератури:

1. Serebryany K., Bruening D., Potapenko A., Vyukov D. AddressSanitizer: A Fast Address Sanity Checker. *Proceedings of the 2012 USENIX Annual Technical Conference* (*USENIX ATC '12*). 2012. URL: https://dl.acm.org/doi/10.5555/2342821.2342849
2. ISO/IEC JTC1/SC22/WG21. Working Draft, Standard for Programming Language C++. URL: https://open-std.org
3. Sutter H. Lifetime safety: Preventing common dangling. P1179R1. WG21, 2019. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf
4. Stroustrup B., Dos Reis G. Design Alternatives for Type-and-Resource Safe C++. P2687R0. WG21, 2022. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2687r0.pdf
5. Doumler T. *at al*. Contracts for C++. P2900R13. WG21, 2025. URL: https://open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2900r13.pdf
6. Sutter H. Core safety profiles for C++26. P3081R2. WG21, 2025. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3081r2.pdf
7. Berne J., Lakos J. Prevent Undefined Behavior By Default. P3558R0. WG21, 2025. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3558r0.pdf
8. Dos Reis G. Reference checking. P2878R1. WG21, 2023. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2878r1.html
9. Sutter H. Making Safe C++ Happen. P3700R0. WG21, 2025. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3700r0.html
10. Du J. X. K. L. Z. *at al*. A Study of Compiler-Introduced Security Bugs. 2023. URL: https://nebelwelt.net/files/23SEC4.pdf
11. X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," *ACM SIGPLAN Not.*, vol. 46, no. 6, pp. 283–294, Jun. 2011. doi: https://doi.org/10.1145/1993316.1993532
12. Verdi M. *at al*. An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples. 2019. URL: https://arxiv.org/pdf/1910.01321
13. Xu H. та ін. Memory-Safety Challenge Considered Solved? An In-Depth Study of Rust. 2020. URL: https://arxiv.org/pdf/2003.03296

14.Moghadam V. E. Memory Integrity Techniques for Memory-Unsafe Languages: A Survey. 2024. URL: https://iris.santannapisa.it/retrieve/9ebf4c27-86fb-401c-827b-6fad063f1d08/Memory_Integrity_Techniques_for_Memory-Unsafe_Languages_A_Survey.pdf
15.nlohmann. JSON for Modern C++. GitHub repository. URL: https://github.com/nlohmann/json

Надійшла до редколегії 27.05.2025

Відомості про авторів:
Запухляк Руслан Ігорович – кандидат фізико-математичних наук, професор кафедри комп'ютерної інженерії та електроніки, Прикарпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Precarpathian National University, Україна; email: ruslan.zapukhlyak@pnu.edu.ua, ORCID: https://orcid.org/0000-0002-4204-8235
Довгий Віктор Володимирович – канд. техн. наук, старший викладач кафедри комп'ютерної інженерії та електроніки, Прикарпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Precarpathian National University, Україна; email: viktor.dovhyi@pnu.edu.ua ORCID: https://orcid.org/0009-0009-7158-6938