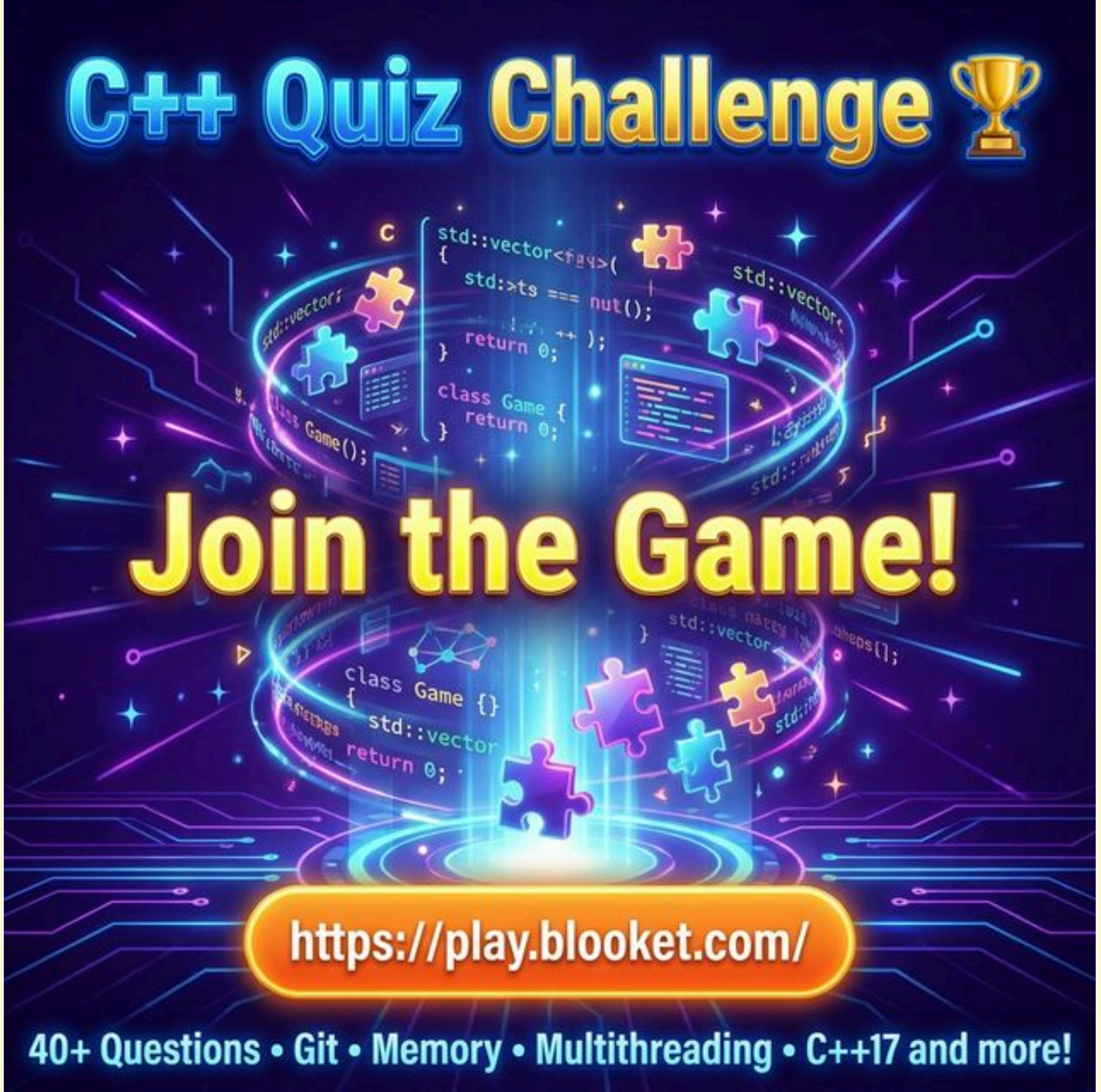
C++ Quiz Challenge
Join the Game!
https://play.blooket.com/
40+ Questions • Git • Memory • Multithreading • C++17 and more!

```
$ git stash
```

Q1. 🐙 What does 'git stash' do?

A) 🗑️ Deletes untracked files
B) 📦 Commits all staged changes
C) ⏪ Resets to last commit
D) 💾 Saves uncommitted changes, cleans working directory

Q2. 🔄 What is the state of myfile.cpp after these two commands?

```
$ git add myfile.cpp
$ git reset HEAD -- myfile.cpp
```

A) 🌿 Committed to branch
B) ❌ Deleted from repo
C) 📝 Modified, unstaged
D) 📬 Staged, ready to commit

```
# .gitattributes
my_rules.csv  binary
```

Q3. ⚙️ What does 'my_rules.csv binary' in .gitattributes do?

A) 🪟 Forces CRLF on checkout
B) 🗜️ Compresses the file
C) 👻 Ignores the file in commits
D) 🚫 git won't try to merge it

Git

Version control: init, add, commit, push

Branching: one branch = one task

Stash: save work without committing

Pre-commit hooks: automate checks

CRLF: always use LF (eol=lf)

CI/CD: compile on isolated machines

```
$ g++ -E  simple.cpp -o simple.i
$ g++ -S  simple.i  -o simple.s
$ g++ -c  simple.s  -o simple.o
$ g++     simple.o  -o simple
```

Q5. 🔧 What is the correct order of these 4 steps?

A) Compile → Preprocess → Assemble → Link
B) Assemble → Compile → Preprocess → Link
C) Preprocess → Compile → Assemble → Link
D) Preprocess → Assemble → Compile → Link

```
$ ldd simple
  libstdc++.so.6 => /usr/lib/...
  libc.so.6      => /lib/...
  linux-vdso.so.1 (0x...)
```

Q6. 🔍 What does 'ldd simple' show?

A) 🏷️ All symbols defined in the binary

B) 🖥️ System calls used at runtime

C) 🗃️ Static libraries linked into the binary

D) 📚 Shared libraries the binary depends on

```
# CMakeLists.txt
set(CMAKE_SYSTEM_NAME  Linux)
set(CMAKE_SYSTEM_PROCESSOR aarch64)
set(CMAKE_CXX_COMPILER
    aarch64-linux-gnu-g++)
```

Q7. 🌍 What does CMAKE_SYSTEM_NAME set in a CMake toolchain file?

A) 🏠 The host machine's OS name
B) ⚙️ The build system generator
C) 🏭 The compiler vendor name
D) 🎯 The target OS for cross-compilation

Toolchain & Build

Toolchain: compiler, linker, assembler, debugger

Compilation: Preprocess -> Compile -> Assemble -> Link

ldd: inspect shared library dependencies

glibc: tightly coupled to Linux kernel

Static vs dynamic linking trade-offs

Cross-compilation: CMAKE_SYSTEM_NAME / PROCESSOR

```
// myheader.h
#pragma once

int foo();
```

Q9. 🛡️ What does '#pragma once' do in a header file?

A) 🔕 Disables all warnings in the file

B) 🏷️ Marks the file as system header

C) 📌 Prevents the file from being included more than once

D) 🚀 Forces the file to compile before all others

```
#include <algorithm>
#include <Windows.h>

int k = std::min(3,4);
```

Q10. 💥 This code fails to compile on Windows. What is the cause?

A) 🚫 algorithm is not available on Windows

B) 📂 Missing #include for <windows.h>

C) 🪟 Windows.h defines min/max as macros

D) 📐 std::min needs two identical types

```
#define SQR(x)  x*x

std::cout << SQR(4+1);
```

Q11. 🤯 #define SQR(x) x*x What is the output of SQR(4+1)?

A) 9
B) 21
C) 25
D) 1

Preprocessor

#include: text substitution, watch search dirs

#pragma once: prevent multiple inclusion

#define: avoid macros, prefer constexpr

Windows.h: define NOMINMAX before including

g++ -E: dump preprocessed output to debug

clang-tidy: detect include issues

```
uint64_t foo() {
  const int N = 1024;
  uint64_t total = 0;
  for(int i=0;i<N;++i)
    total += i*i;
  return total;
}  // g++ -O3
```

Q13. 🚀 What does g++ -O3 return for this function?

A) 💥 A compiler error: loop is infinite

B) 0️⃣ Zero, loop body is optimized away

C) 🧮 A single constant (constant folding)

D) 🔁 The loop runs 1024 times at runtime

```cpp
#include <climits>
int main() {
  int max = INT_MAX;
  max++;  // ???
}
```

Q14. 💀 What is the behavior of 'max++' when max == INT_MAX?

A) 🛑 Compiler error: overflow detected
B) 🔄 Wraps to INT_MIN (guaranteed)
C) 0️⃣ Result is 0
D) 💣 Undefined behavior (signed overflow)

```
$ g++ -fsanitize=address \
      -fsanitize=undefined \
      main.cpp
```

Q15. 🩺 What do sanitizer flags like -fsanitize=address do?

A) ✂️ Strip debug symbols from the binary
B) 🔕 Disable compiler optimizations
C) ⚡ Speed up execution by removing bounds checks
D) 🔎 Add runtime checks that detect memory errors

Compiler

-O0/-O2/-O3: optimization levels

Constant folding: loops -> single instruction

Undefined behavior: silent data corruption

Sanitizers: -fsanitize=address,undefined

-Wall -Wextra -Werror: enable all warnings

clang vs MSVC: error message quality

```
# Which is correct on Linux?
g++ main.o -lpow -lmul -ladd
g++ main.o -ladd -lmul -lpow
```

Q17. 🔗 Which link command is correct on Linux?

A) 🛠️ Use --start-group for all cases
B) g++ main.o -ladd -lmul -lpow
C) g++ main.o -lpow -lmul -ladd
D) 🤷 Order does not matter on Linux

```
$ nm -C libmul.a
U add(int, int)
T mul(int, int)
```

Q18. 🔍 In nm output, what does 'T' mean next to a symbol?

A) 📊 In a data section
B) 📋 Template function
C) ❓ Undefined, must be linked
D) 💡 Defined in the text (code) section

```
// add.cpp  (Windows DLL)
int add(int a, int b) {
    return a + b;
}
```

Q19. 🪟 A Windows DLL is built but no .lib file is created. Why?

A) 🚫 No __declspec(dllexport) symbols in the code
B) ⚙️ CMake forgot to set SHARED keyword
C) 🔑 Wrong architecture (x86 vs x64)
D) 🏗️ Missing /MD compiler flag

Linker

Links .o files into executable / .so / .dll

Linux: library order matters (left to right)

nm: inspect symbols (T=defined, U=undefined)

Static .a vs dynamic .so/.dll trade-offs

Windows: __declspec(dllexport) to export

Linux: -fvisibility=hidden + __attribute__

```cpp
#include <iostream>
int main() {
  std::cout << sizeof(long);
}
// 64-bit Linux vs Windows?
```

Q21. 📏 sizeof(long) on 64-bit Linux vs 64-bit Windows?

A) Both: 4
B) Linux: 8, Windows: 4
C) Both: 8
D) 🪟 Linux: 4, Windows: 8

```
char c = 200;
if (c > 0)
    std::cout << "Positive";
else
    std::cout << "Negative";
// x86 GCC vs ARM GCC?
```

Q22. 🤔 char c = 200 — what is the output on x86 GCC vs ARM GCC?

A) ➖ Both print Negative
B) 💀 Undefined behavior on both platforms
C) ➕ Both print Positive
D) 🔀 Different — char signedness is implementation-defined

```
#pragma pack(push, 1)
struct S {
    char  a;   // 1 byte
    int   b;   // 4 bytes
    short c;   // 2 bytes
};
#pragma pack(pop)
```

Q23. 📦 What is sizeof(S) after #pragma pack(push,1)?

A) 12
B) 4
C) 7
D) 8

ABI

ABI = Application Binary Interface

Rule 1: Use POD types only

Rule 2: Use fixed-width types (int32_t...)

Rule 3: #pragma pack(push,1) for structs

Rule 4: Export only needed symbols

Rule 5: extern "C" to prevent name mangling

```
$ g++ -g -O2 main.cpp -o app
$ strip app
```

Q25. ✂️ What does 'strip app' do to the binary?

A) 🗑️ Deletes the binary from disk
B) 🔒 Encrypts the binary for distribution
C) 🔄 Removes optimizations, adds debug info
D) 📉 Removes debug symbols, shrinks binary size

```
(gdb) watch var
(gdb) rwatch var
(gdb) awatch var
```

Q26. 👀 What does 'watch var' do in gdb?

A) ⏸️ Stops execution when var is written

B) 📖 Stops execution when var is read

C) 🖨️ Prints var value every step

D) 0️⃣ Sets var to zero and continues

```
$ ulimit -c unlimited
$ echo "/tmp/core.%e.%p" |
  sudo tee /proc/sys/kernel/core_pattern
```

Q27. 💥 What do these two commands enable on Linux?

A) 💾 Core dump creation when a process crashes

B) 🔍 Debug symbol loading from /proc

C) 🌐 Remote gdbserver on port 2000

D) 📝 Full memory logging to /tmp

Debugging

Debug info: -g (Linux DWARF) / /Zi (Windows PDB)

RelWithDebInfo: -O2 + -g (best of both worlds)

strip: remove debug info from binary

gdb: break, next, step, bt, watch, p

Core dumps: ulimit -c unlimited

Remote: gdbserver :2000 ./app

```
void foo(int a) {
    int x = 42;
    bar(x);
}
```

Q29. 🧱 What memory region holds local variables like 'int x'?

A) 📚 Stack
B) 💾 ROM
C) 🏛️ Global/Static
D) 📦 Heap

```
int& get_ref() {
    int x = 42;
    return x;
}
int main() {
    int& r = get_ref();
    return r;
}
```

Q30. 💀 What is wrong with this get_ref() function?

A) 📦 x is allocated on the heap
B) 🔁 Infinite recursion
C) 🔕 Missing return type
D) 🔥 Returns a dangling reference to a local variable

```
void leak() {
    int* p = new int[100];
    if (!OpenFile())
        return;
    delete[] p;
}
```

Q31. 🪣 What bug does this leak() function contain?

A) 📏 Buffer overflow
B) 👻 Use-after-free
C) 💧 Memory leak when OpenFile() returns false
D) 🔥 Double free of p

```
void f() {
    int* p = new int[100];
    delete[] p;
    delete[] p;
}
```

Q32. 💥 What is wrong with deleting p twice?

A) 📦 p becomes nullptr automatically
B) 💀 Double free: undefined behavior
C) ✂️ Extra memory is freed safely
D) 🔕 Compiler catches it at build time

```
std::vector<int> v;
v.push_back(1);
int* p = &v[0];
v.push_back(2);
*p = 99;
```

Q33. 🔄 What happens when you dereference 'p' after push_back?

A) 📝 p still points to v[0]
B) 📏 Buffer overflow on the stack
C) 😮 Use-after-free: push_back may reallocate the vector
D) 🛑 Compile error: p is const

```
struct Foo { int x; Foo(){} };
Foo g_foo;
int main() {
    Foo s_foo;
    // g_foo.x vs s_foo.x ?
}
```

Q34. 🏛️ What is g_foo.x vs s_foo.x before any assignment?

A) g_foo.x is undefined, s_foo.x is 0
B) Both are undefined
C) Both are 0 (always zero-init)
D) g_foo.x is 0, s_foo.x is undefined

Memory

Stack: fast, auto, scope-bound (grows down)

Heap: dynamic, explicit new/delete/malloc/free

Global/Static: zero-init, lives for program lifetime

Stack bugs: overflow, dangling ref, buffer overrun

Heap bugs: leak, double-free, use-after-free

Modern C++: prefer RAII + smart pointers

```
$ ./my_tests
    --catch_system_errors=yes

// What does this flag do?
```

Q36. 🔴 Boost.Test: what does --catch_system_errors=yes do?

A) Lets segfaults produce a core dump
B) Only catches C++ exceptions
C) Catches segfaults/signals and reports them as test failures
D) Disables all error checking

Q37. 🧪 How do you make a new feature easy to unit test?

```
// Adding a new feature...
// How to make it unit-testable?

class NewFeature {
    // ???
};
```

A) Write the tests after the full feature is done
B) Add it directly inside main()
C) Implement it as a standalone class
D) Use global variables for shared state

```
// Which framework has the
// strongest mocks ecosystem?

// A) Boost.Test
// B) GoogleTest (gMock)
// C) Catch2
// D) All are equal
```

Q38. 🎭 Which C++ test framework has the strongest mocks ecosystem?

A) GoogleTest (gMock)
B) Boost.Test
C) Catch2
D) All are equal

🧪 Testing

Unit test: one class/function, fast (ms)

Component test: full algorithm + data replay

System test: whole project, threads, IO, mocks

TDD: Red -> Green -> Refactor

Frameworks: GTest / Boost.Test / Catch2

CI: build -> test -> package, fail fast

```
int counter = 0;
void run() {
    for(int i=0;i<100000;++i)
        counter++;
}
// Two threads call run(). Output?
```

Q40. 🏎️ Two threads call run(). What is the output of counter?

A) 0 (both threads cancel each other out)
B) Unpredictable (data race, could be anything)
C) A compile error
D) Exactly 200000 (always)

```
std::condition_variable cv;
std::mutex mtx;
// Consumer:
std::unique_lock<std::mutex> lk(mtx);
cv.wait(lk);  // bug?
process();
```

Q41. 😮 What is the bug in this condition_variable usage?

A) Missing predicate: susceptible to spurious wakeups
B) mtx must be a recursive_mutex
C) process() must be called before wait()
D) cv.wait() needs a timeout argument

```
auto ptr = std::make_shared<int>(0);
// Thread A:
*ptr = 20;
// Thread B (concurrent):
*ptr = 30;
```

Q42. 🧵 Is writing *ptr from two threads simultaneously safe?

A) No: data race on the managed object
B) Yes: shared_ptr is fully thread-safe
C) Yes: atomic ref-count protects the value
D) Only unsafe on ARM

Multithreading

Thread: OS resource sharing address space

Data race: two threads, one write -> UB

Mutex: use lock_guard, never lock() manually

CV: always use a predicate (spurious wakeups!)

atomic<T>: lock-free only for small types

Priority inversion: low holds lock, high blocks

Q44. 🔢 What does this code print?

```
double a = 0.1 + 0.2;
if (a == 0.3)
    std::cout << "Equal";
else
    std::cout << "Not Equal";
```

A) Equal (IEEE 754 is exact)
B) Undefined behavior
C) Not Equal (0.1+0.2 has rounding error)
D) Compile error: == invalid for doubles

Q45. ♾️ What does 'double x = 1.0/0.0' print?

```
double x = 1.0 / 0.0;
std::cout << x;
// What is printed?
```

A) Crash: divide by zero
B) 0
C) +inf
D) NaN

```
double n = std::sqrt(-1.0);
if (n == n)
    std::cout << "equal";
else
    std::cout << "not equal";
// What prints?
```

Q46. 🤯 What does this NaN comparison print?

A) equal (NaN is unique)
B) Undefined behavior
C) Compile error
D) not equal (NaN != NaN is always true)

```
// Compiled with g++ -ffast-math
double a = 1e16;
double b = 1.0;
std::cout << (a + b) - a;
// Expected vs actual output?
```

Q47. ⚡ What does -ffast-math do to floating-point code?

A) Only speeds up integers, floats unchanged
B) Allows non-IEEE optimizations, may change results
C) Forces strict IEEE 754 compliance
D) Disables SIMD instructions

Numbers

IEEE 754: sign + exponent + mantissa

0.1 + 0.2 != 0.3 (finite binary precision)

NaN != NaN (always false, use std::isnan)

1.0/0.0 = +inf, 0.0/0.0 = NaN (no exception!)

-ffast-math: fast but non-IEEE, non-deterministic

ULP: smallest difference between two floats

```cpp
std::tuple<int, std::string>
get_user_data() {
    return {1, "Alice"};
}

auto [id, name] = get_user_data();
```

Q49. 1 7 What C++17 feature does 'auto [id, name] = ...' use?

A) Structured bindings: unpack tuple/struct
B) std::pair destructuring (C++11 feature)
C) Template parameter pack
D) Move semantics on return value

Q50. 🔍 After the closing brace, is 'it' accessible?

```
if (auto it = m.find("key");
    it != m.end()) {
    use(it->second);
}
// 'it' after the closing brace?
```

A) Yes: 'it' is a regular local variable
B) No: 'it' is scoped to the if block
C) Only if the if condition was true
D) Only in the else branch

```
template<typename T>
void info(T val) {
    if constexpr (std::is_integral_v<T>)
        std::cout << val;
    else
        std::cout << val.size();
}
// info(42);  -- would .size() compile?
```

Q51. ⚙ Why does info(42) compile even though 'val.size()' is in the code?

A) Template instantiation defers all checks

B) The compiler ignores unreachable code paths

C) int has a .size() method in C++17

D) constexpr if discards the else branch at compile time for int

17 C++17

Structured bindings: auto [a, b] = tuple;

if/switch with initializer: if (auto x=f(); x>0)

std::optional<T>: value or std::nullopt

std::filesystem: portable path, directory_iterator

constexpr if: compile-time branch (no SFINAE!)

std::string_view: non-owning string reference