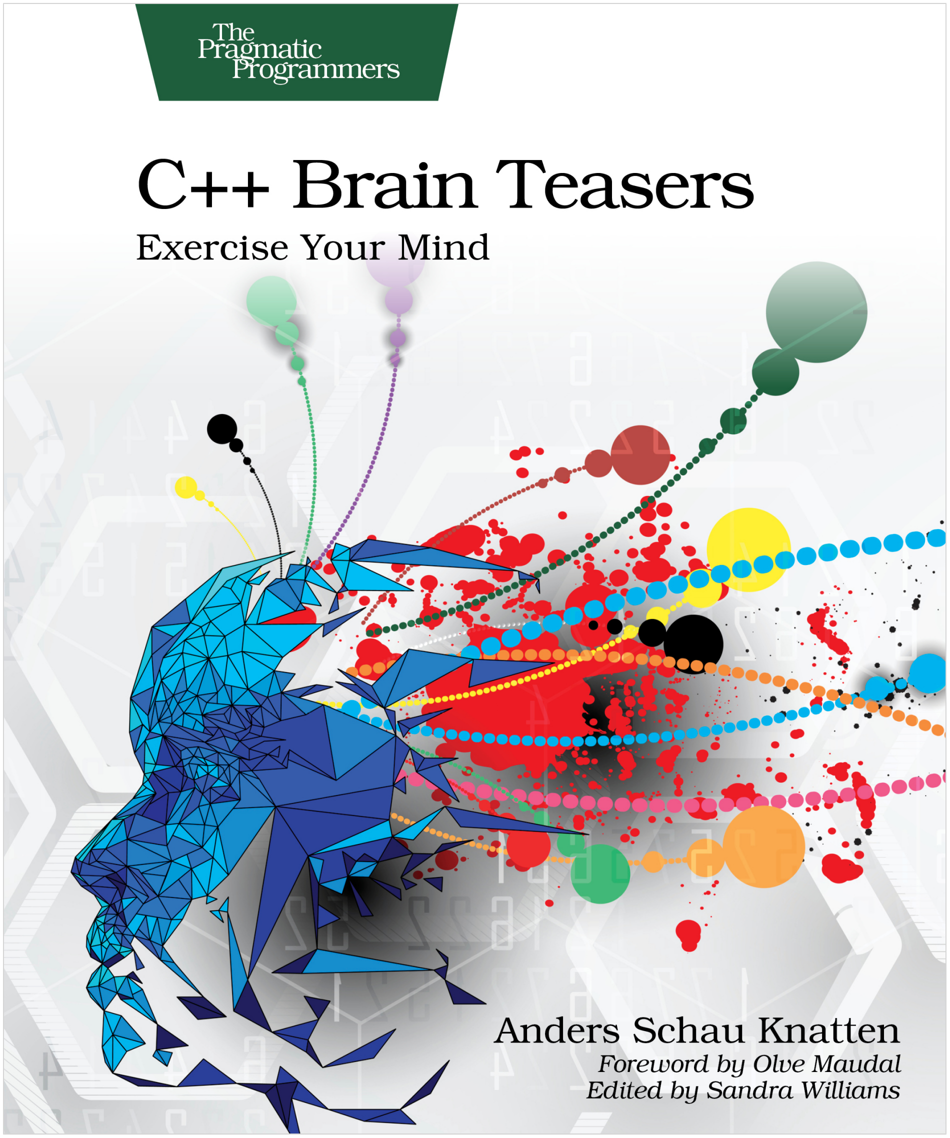


The Pragmatic Programmers

C++ Brain Teasers

Exercise Your Mind

Anders Schau Knatten

Foreword by Olve Maudal

Edited by Sandra Williams

Hack the Planet!

hack-the-planet.cpp

```
#include <iostream>

int getUserId() { return 1337; }

void restrictedTask1()
{
    int id = getUserId();
    if (id == 1337) { std::cout << "did task 1\n"; }
}

void restrictedTask2()
{
    int id;
    if (id == 1337) { std::cout << "did task 2\n"; }
}

int main() {
    restrictedTask1();
    restrictedTask2();
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program has undefined behavior! But it might display the following output:

```
did task 1
did task 2
```

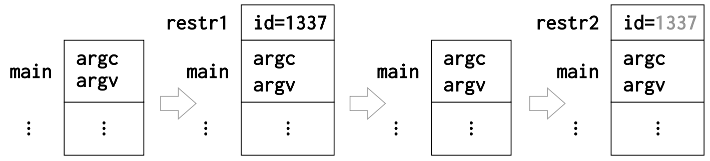
Discussion

The variable id in function restrictedTask2 has not been initialized and has an indeterminate value. To use its value is undefined behavior. Anything can happen when a program runs into undefined behavior; the C++ standard makes no guarantees. Even the part of the program that happened *before* we took id's value is undefined!

However, if you run this program on your own computer, it will probably print both did task 1 and did task 2, at least if you compile without optimizations. So the value 1337 magically teleported from restrictedTask1 to restrictedTask2! How could this happen?

Most systems use a stack for local variables. restrictedTask1 has a local variable id, which it sets aside space for in its stack frame. As it happens, restrictedTask2 has the same number and types of local variables (one int), so its stack frame layout will be identical to restrictedTask1's.

When in main, the stack will have grown to a certain point—see the leftmost illustration in the following figure. We then call restrictedTask1, the stack grows, restrictedTask1 sets aside space in its stack frame for id, and initializes id to 1337. Then control returns to main and the stack shrinks again.



Next, we call restrictedTask2, the stack grows again, and restrictedTask2 sets aside space in its stack frame for id but doesn't initialize it. However, restrictedTask2's stack frame ends up in the exact same place as restrictedTask1's stack frame. The stack isn't cleared between function calls, so the contents of restrictedTask2's stack frame will be exactly as restrictedTask1 left it, including the value 1337 in the position of the local variable id. This causes id to be 1337 in both functions.

If you turn on optimizations, though, restrictedTask1 and restrictedTask2 are likely to be inlined into main, and the stack is not used for these calls. You can try this yourself by compiling with -O2 (GCC/Clang) or /O2 (MSVC). On my x86_64 Linux machine, GCC prints did task 1 with -O2, whereas Clang segfaults. What happens on your machine?

Avoiding Uninitialized Variables

To avoid nasal demons, security holes like this one, and garbage data in general, you should always initialize your variables before using them. It's easy to slip up, but help is to be had! First of all, always compile with warnings. GCC, Clang, and MSVC will warn you about this particular case, as will tools like clang-tidy. But that's only because it's straightforward to prove that id is uninitialized when it's used. Often, we can't know this until runtime. This is where sanitizers come in.

Different sanitizers exist for different purposes, but all monitor your code at runtime in various ways to detect problems that can't be detected at compile time. One such sanitizer is MemorySanitizer, which is designed to catch usages of uninitialized memory. If we run this program with MemorySanitizer (pass -fsanitize=memory as a compiler option to Clang), it'll print something like this:

```
==1416660==WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x5603d43af5eb in restrictedTask2() example.cpp:14:9
#1 0x5603d43af64d in main example.cpp:19:5
```

It tells you that you're using uninitialized memory and gives you a stack trace to where it happened.

Sanitizers can make your program much slower, so you typically want separate build configurations with sanitizers. A sanitizer can only detect an issue if that issue actually occurs at runtime, though, so make sure to run as much of your test suite as possible with this build.

Recommendations

- Always enable warnings. -Wall -Wextra -Wpedantic is a good start on GCC/Clang, and /W4 on MSVC.
- If your IDE supports integrations with linting tools like clang-tidy, turn them on. They can sometimes report issues like this one before you even compile the code.
- Enable warnings as errors, at least in your Continuous Integration jobs, so you don't overlook any warnings.
- Set up sanitizer builds for as many sanitizers as you can, and run your tests with these builds.

Further Reading

Undefined Behavior

https://en.cppreference.com/w/cpp/language/ub

Deep C (and C++) by Olve Maudal and Jon Jagger

https://www.slideshare.net/olvemaudal/deep-c

List of Sanitizers

https://github.com/google/sanitizers