Extracted from:

C++ Brain Teasers

Exercise Your Mind

This PDF file contains pages extracted from *C++ Brain Teasers*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

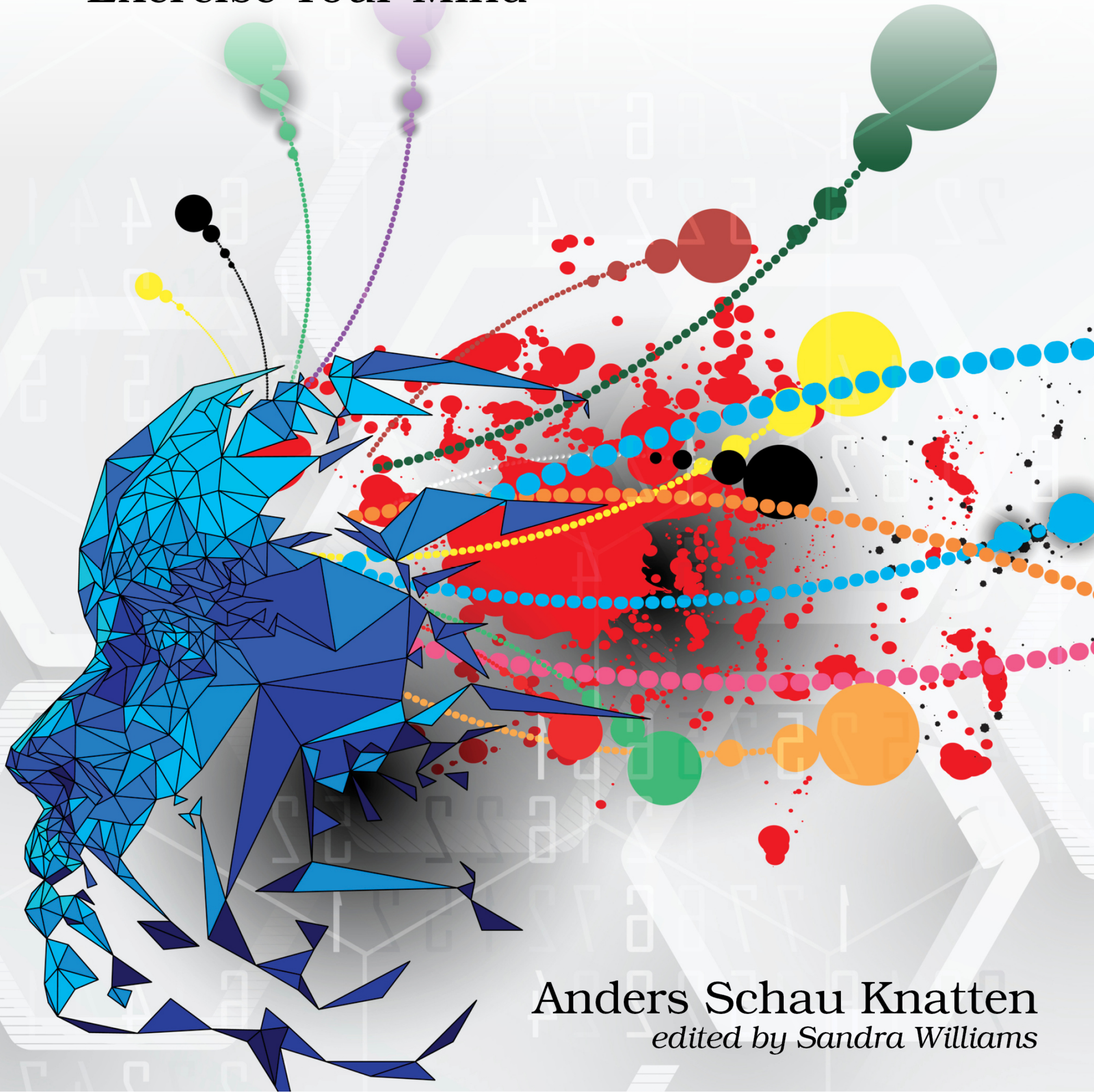
The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

C++ Brain Teasers

Exercise Your Mind



Anders Schau Knatten

edited by Sandra Williams

C++ Brain Teasers

Exercise Your Mind

Anders Schau Knatten

The Pragmatic Bookshelf
Dallas, Texas

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 979-8-88865-051-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—November 8, 2023

Puzzle 3

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.

• Click HERE to purchase this book now. discuss

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

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

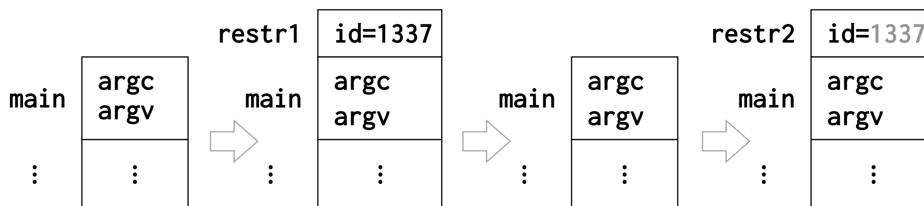
Discussion

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, restrictedTask2 sets aside space in its stack frame for id, but does not initialize it. However, restrictedTask2's stack frame ends up in the exact same place as restrictedTask1's stack frame. The stack is not 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 1337 to be printed in both functions.

• Click HERE to purchase this book now. discuss

If you turn on optimizations, however, 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 is easy to slip up, but there is help to be had! First of all, always compile with warnings. Both GCC, Clang, and MSVC will warn you about this particular case, as will tools like clang-tidy. But that is only because it is straightforward to prove that id is uninitialized when it is used. Often, we can't know this until run-time. This is where sanitizers come in.

Different sanitizers exist for different purposes, but they all monitor your code at run-time 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 will 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.

• Click HERE to purchase this book now. discuss

Recommendations
• 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