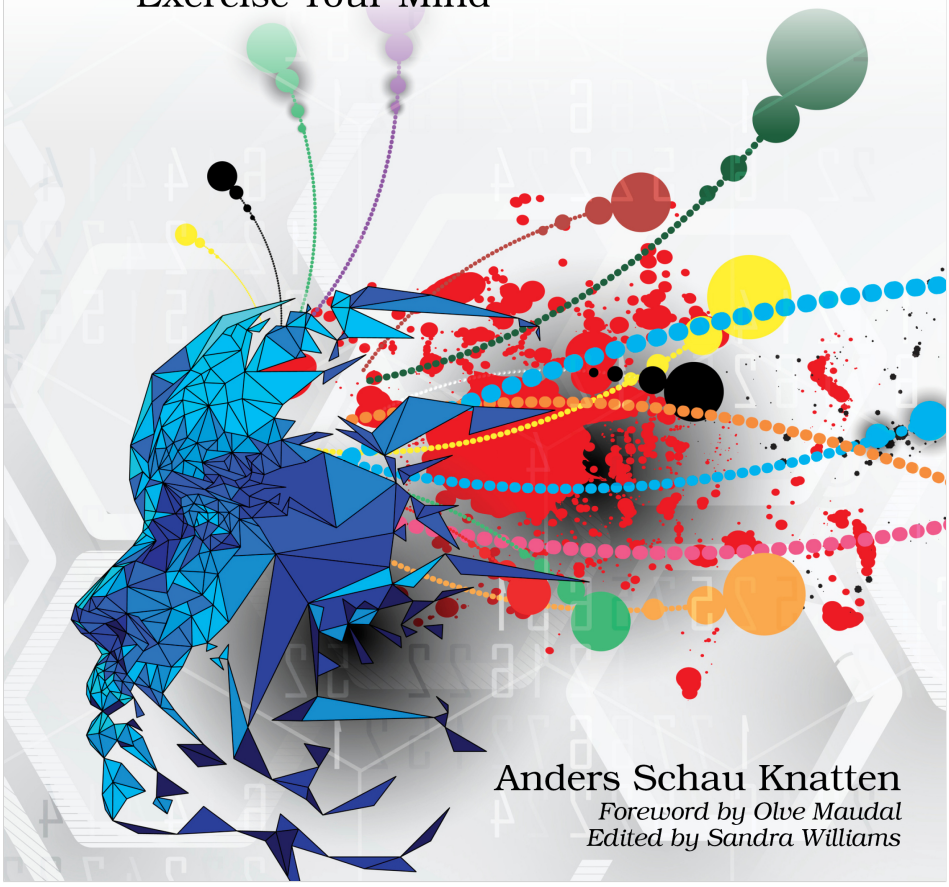
The Pragmatic Programmers

C++ Brain Teasers

Exercise Your Mind



Anders Schau Knatten

Foreword by Olve Maudal

Edited by Sandra Williams

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Preface

C++ is one of the largest and oldest programming languages in common use. It's famous for getting all the default behaviors wrong and for, as is the famous example in the C++ community, making demons fly out of your nose.[1] You couldn't have picked a better language for a programming brain teasers book!

Through 25 puzzles we'll explore how C++ works under the hood, including some of its important quirks. To get the most out of the book, you should have some experience writing C++ and be familiar with the basics of the language, including simple object-oriented programming and templates.

After finishing the book, you'll have a deeper understanding of topics like initialization, lifetimes, overload resolution, implicit conversions, inheritance, undefined behavior, and more. But more importantly, I hope you'll have gained a sense of curiosity about how C++ really works, even if only a few of those answers fit in a single book.

How to Use This Book

The book contains 25 C++ puzzles with answers and explanations. Most will be well-formed programs, which the C++ standard guarantees the output of. Some might, however, have a compilation error, and some might have undefined or unspecified behavior. Your task is to figure out what happens when you compile and run the program in each puzzle on a conforming C++ implementation.

Take this imaginary puzzle as an example. It's a complete C++ program with a `main` function:

```
#include <iostream>

int main()
{
    std::cout << (1 < 2);
}
```

1. http://catb.org/jargon/html/N/nasal-demons.html

Your task will be to read through the code and try to guess what the output will be when the program is compiled and run. Always make sure you give it a proper go before turning the page to look at the answer!

Please note a few technicalities:

- For brevity, I declare `main` with no parameters (no `argc` / `argv`) and don't explicitly return a value. These are all optional, and omitting them makes the puzzles slightly easier to read.
- I use `struct` instead of `class` for all the puzzles. There's no semantic difference—structs just have `public` rather than `private` as the default visibility for members, so we don't have to put `public:` everywhere.
- As always in C++, `bool`s are printed as `1` and `0` by default, not `true` and `false`.

The answer to the puzzle is that `1` is less than `2`, so the program prints `1` (representing `true`).

Getting the answers right is only half the fun, though. The other half is understanding *why* it works the way it does. The puzzles are excuses to learn more about how C++ works under the hood. I encourage you to read the explanations thoroughly enough to understand them.

Undefined Behavior

Some puzzles might have undefined behavior. Undefined behavior is the term used when something bad happens during the execution of a program, which the compiler is unable to (or, more specifically, not required to) detect. For instance, we might access an element past the end of an array, or an arithmetic expression with signed integers might overflow. In these cases, the C++ standard imposes no restrictions on the implementation, and anything can happen, including nasal demons. If a puzzle has undefined behavior, your task is to identify the undefined behavior but also to guess what happens in practice on a typical system. Does it actually make demons fly out of your nose, or does something specific happen?

Again, let's look at an example:

```
#include <iostream>
#include <limits>

int main()
{
    std::cout << std::numeric_limits<int>::max() + 1;
}
```

Signed integer overflow in arithmetic is undefined behavior, so if you identified that, you're halfway there!

Making assumptions about what will happen in the case of undefined behavior is a bad idea. So let's do that next! For any puzzles with undefined behavior, the other half of the puzzle is to figure out what would happen if you ran the program on your computer. My computer uses two's complement for signed integers (as do all conforming implementations since C++20), and my CPU doesn't generate an exception when I overflow. So when I add `1` to the largest positive integer, the value simply wraps around to the smallest negative integer. Since my system uses 32-bit `int`s, the program prints `-2147483648`. You don't need to know that exact value, but if you guessed it would print the smallest negative integer, you solved it!

Don't Do This at Home

Guessing or testing what happens in the case of undefined behavior is an interesting exercise that can teach you more about how C++ works on your platform. It might also enable you to recognize certain error patterns when you see them happening in real programs. But do not make any assumptions about your real programs based on what you find! Your assumptions might be untrue on other computers, after upgrading your compiler, or when you compile with different optimization settings. The compiler is even allowed to remove error checking from your code if it can prove that there's undefined behavior!

Unspecified and Implementation-Defined Behavior

The C++ standard doesn't specify everything strictly; it leaves some freedom to the implementation. These are some examples:

- The specific sizes of integer types
- The order of evaluation of function arguments
- The order of initialization of global variables

This allows each implementation to make choices that make the most sense on that particular system.

Most programs have some unspecified or implementation-defined behavior; this is not a bug. And contrary to undefined behavior, demons will not fly out of your nose. It's just that different implementations might behave a bit differently within a set of allowed behaviors.

If a puzzle has unspecified or implementation-defined behavior, try to also guess what a typical behavior would be.

Playing Around with the Code

The most important part of learning anything programming related is to play around with it yourself. The code from this book can be found on the book's home page at The Pragmatic Bookshelf.[2] You can build it locally by opening CMakeLists.txt in your favorite IDE or on the command line:

```
mkdir build
cd build
cmake ..
cmake --build .
```

The project contains one .cpp file per puzzle, each resulting in one binary, both named the same as the corresponding puzzle in the book.

You can also try the code directly in your browser by pasting it into an online compiler. I highly recommend Compiler Explorer,[3] where you can choose from and compare different compiler versions and architectures, try out different optimization levels, and add other compiler flags, sanitizer options, and so forth.

Let's get started! Oh, and beware of the demons.

2. https://pragprog.com/titles/akbrain
3. https://godbolt.org