Extracted from:

C++ Brain Teasers

Exercise Your Mind

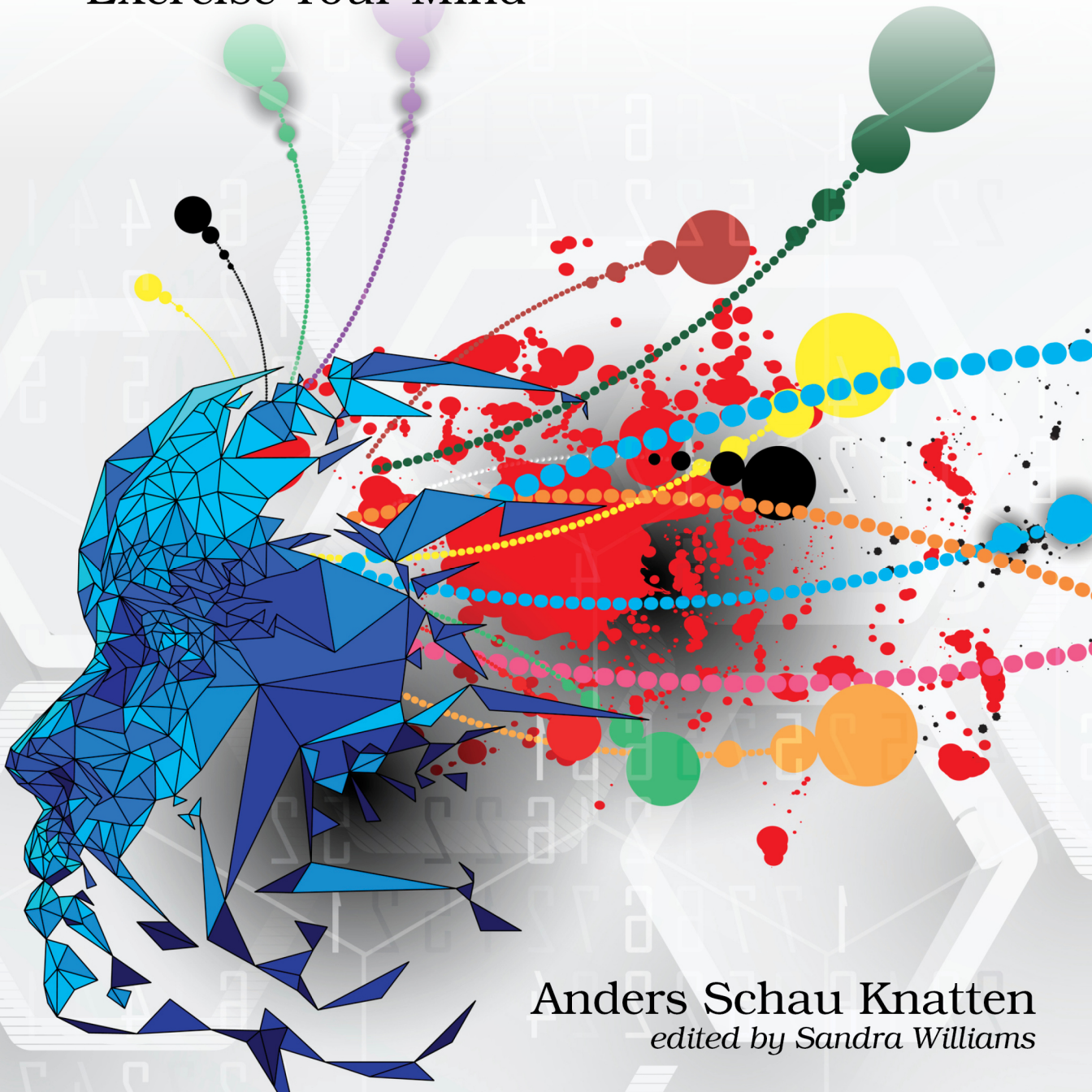
The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

C++ Brain Teasers

Exercise Your Mind



Anders Schau Knatten

edited by Sandra Williams

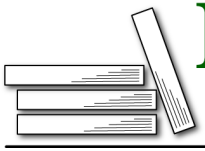
C++ Brain Teasers

Exercise Your Mind

Anders Schau Knatten

The Pragmatic Bookshelf

Dallas, Texas



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*.

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 2

String Theory

string-theory.cpp

```
#include <iostream>
#include <string>

void serialize(const void*) { std::cout << "const void*"; }

void serialize(const std::string&) { std::cout << "const string&"; }

int main()
{
    serialize("hello world");
}
```

Guess the Output

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

The program displays the following output:

```
const void*
```

Discussion

Why does passing a string to serialize cause the overload taking a void pointer to be called, rather than the overload taking a string?

When we're calling a function with multiple overloads, the compiler uses a process called *overload resolution* to figure out which one is the best fit. The compiler does this by attempting to convert each function argument to the corresponding parameter type for each overload. Some conversions are better than others, and the best conversion is if the argument is already of the correct type.

All the overloads where all arguments can be successfully converted are added to a set of *viable functions*. Then the compiler needs to figure out which overload to select from this set. If an overload has a better conversion for at least one argument and not a worse one for any of the other arguments, this overload is deemed to be the *best viable function* and selected by overload resolution. If no overload is better than all the others, the call is ill-formed and fails to compile.

For example:

```
serialize(int, int); // 1
serialize(float, int); // 2
```

Given these two overloads, when you call serialize like this:

```
serialize(1, 2);
```

Both overloads of serialize are viable. But the first overload has a better conversion for the first argument (int → int is better than int → float) and not a worse conversion for the second argument (int → int for both overloads), so it is selected by overload resolution as the best viable function.

The puzzle is a bit simpler than this example since both overloads of serialize only have one parameter. The first takes a const void * and the second takes a const std::string&. What does the conversion look like for each of the overloads?

std::string is a class in the standard library. It will typically allocate memory on the heap (unless the string is very small) and allows the string to grow or be otherwise modified at run-time.

However, the string "hello world" is not a std::string, but a simple string literal. String literals are plain C-style arrays of chars which get baked into your binary by the linker and can not be modified at run-time. A string literal has the type "array of n const char." "hello world" has 11 characters plus a terminating \0, so its type is "array of 12 const char."

Now we need to figure out which overload of serialize, if any, is the best viable function. Since the argument "hello world" is neither a const void* nor a std::string, but an "array of 12 const char," the compiler must first figure out which of the two overloads are at all viable. If an implicit conversion exists from the argument to the parameter type, that overload is added to the set of viable functions. Otherwise, the overload is ignored.

Let's examine the first overload, and see if "array of 12 const char" can be implicitly converted to const void *. The first thing that happens is that the array gets converted to a pointer. Any "array of N T" can be converted to a "pointer to T" pointing to the first element. So now our "array of 12 const char" has turned into a "pointer to const char."

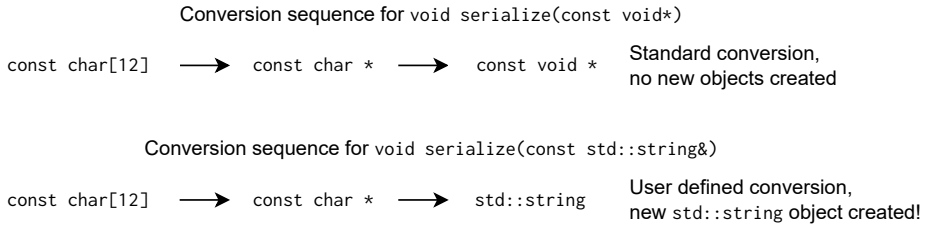
Next, any "pointer to *cv* T" (where *cv* means const, volatile, const volatile or neither) can be converted to "pointer to *cv* void." So now our "pointer to const char" has turned into a "pointer to const void," which is exactly what the first overload expects.

Notice that no constructors or conversion functions were involved in this conversion sequence. This means it's a *standard conversion sequence* and not a *user-defined conversion sequence*. That gets important later.

Let's now examine the second overload, and see if our "array of 12 const char" can be converted to a "reference to const std::string." std::string has a constructor std::string(const char* s), which we can use. First, we convert the "array of 12 const char" to a "pointer to const char" as we did above. Then, we pass this to the std::string constructor and get a std::string back, containing a copy of the string literal. The const std::string& parameter can bind directly to our std::string argument.

Notice that we had to use a constructor for this. This means it's a user-defined conversion sequence and not a standard conversion sequence. It does not matter that std::string is a standard library type; it still counts as user-defined. The rules are the same for you and the standard library.

Now the compiler has found a valid conversion sequence from our "array of 12 const char" to the parameter type of each overload and has to figure out which sequence is best:



To call the const void* overload, we only require a standard conversion sequence (top). To call the std::string overload, we need a user defined conversion sequence (bottom), which involves creating a new temporary std::string object. A standard conversion sequence is always *better* than a user-defined conversion sequence, so the first overload gets called, and const void* is printed.

Further Reading

Overload Resolution

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

std::string

https://en.cppreference.com/w/cpp/string/basic_string

String Literal

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