
2022
Contractual Disappointment in C++
John McFarlane

Contractual Disappointment in C++

johnmcfarlane.github.io/cpponsea-2022

2022-07-07

C++ on Sea

About Me

John McFarlane

Software Engineer, Ennis, Co Clare

Background

Background

Work: games, servers, automotive

Background

Work: games, servers, automotive

Fun: numerics, workflow, word games

Background

Work: games, servers, automotive

Fun: numerics, workflow, word games

C++: low latency, numerics, contracts

Useful Links

- slides: johnmcfarlane.github.io/cpponsea-2022
- code: github.com/johnmcfarlane/wss
- talk: cpponsea.uk/2022/sessions/contractual-disappointment-in-cpp.html
- essay: github.com/johnmcfarlane/papers/blob/main/cpp
- rant: twitter.com/JSAMcFarlane


2022
Contractual Disappointment
in C++
John McFarlane

Definitions

Contracts

Contract Programming in C++(20)

Alisdair Meredith, CppCon 2018

A contract is an exchange of promises between a client and a provider.

Disappointment

P0157R0: Handling Disappointment in C++

Lawrence Crowl, 2015

When a function fails to do what we want, we are disappointed. How do we report that disappointment to callers? How do we handle that disappointment in the caller?

Bugs and Errors

P0709R2: Zero-overhead deterministic exceptions: Throwing values

Herb Sutter, 2018

Programming bugs (e.g., out-of-bounds access, null dereference) and abstract machine corruption (e.g., stack overflow) cause a corrupted state that cannot be recovered from programmatically, and so they should never be reported to the calling code as errors that code could somehow handle.

Contracts

Contracts

Contracts

Types

Contracts

Types

- C++ API Contracts

Contracts

Types

- C++ API Contracts
- C++ Standard

Contracts

Types

- C++ API Contracts
- C++ Standard
- End User Contract

Contracts

Types

- C++ API Contracts
- C++ Standard
- End User Contract
- Test User Contract

Contracts

Types

- C++ API Contracts
- C++ Standard
- End User Contract
- Test User Contract

Attributes

Contracts

Types

- C++ API Contracts
- C++ Standard
- End User Contract
- Test User Contract

Attributes

- Agreement

Contracts

Types

- C++ API Contracts
- C++ Standard
- End User Contract
- Test User Contract

Attributes

- Agreement
- Client

Contracts

Types

- C++ API Contracts
- C++ Standard
- End User Contract
- Test User Contract

Attributes

- Agreement
- Client
- Provider

Contracts

Types

- C++ API Contracts
- C++ Standard
- End User Contract
- Test User Contract

Attributes

- Agreement
- Client
- Provider
- (Client) Violation

Contract Attributes

	C++ API	standard	end user	test user
agreement				
client				
provider				
violation				

Contract Attributes

	C++ API	**standard**	**end user**	**test user**
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

Contract Attributes

	C++ API	standard	end user	test user
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

Contract Attributes

	C++ API	standard	end user	**test user**
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

Contract Attributes

	C++ API	standard	end user	test user
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

Contract Attributes

	end user
client	user

End User Contract

End User Contract

End User Contract

- The exchange of promises between the user and developer of a software product.

End User Contract

- The exchange of promises between the user and developer of a software product.
- It's expected that the user may violate the contract.

End User Contract

- The exchange of promises between the user and developer of a software product.
- It's expected that the user may violate the contract.
  - All people make mistakes.

End User Contract

- The exchange of promises between the user and developer of a software product.
- It's expected that the user may violate the contract.
  - All people make mistakes.
  - Some people are naughty!

End User Contract

- The exchange of promises between the user and developer of a software product.
- It's expected that the user may violate the contract.
  - All people make mistakes.
  - Some people are naughty!
- Such violations are *errors*.

End User Contract

- The exchange of promises between the user and developer of a software product.
- It's expected that the user may violate the contract.
  - All people make mistakes.
  - Some people are naughty!
- Such violations are *errors*.
- Errors should be handled by the program.

Errors

Errors

- are imperfections modelled within the system

Errors

- are imperfections modelled within the system
- arise from real-world unpredictability/unreliability

Errors

- are imperfections modelled within the system
- arise from real-world unpredictability/unreliability
- are caused by real-world phenomena (such as humans)

Errors

- are imperfections modelled within the system
- arise from real-world unpredictability/unreliability
- are caused by real-world phenomena (such as humans)
- Input is a major source of errors:

Errors

- are imperfections modelled within the system
- arise from real-world unpredictability/unreliability
- are caused by real-world phenomena (such as humans)
- Input is a major source of errors:
  - command line, network traffic, files, input devices.

Errors

- are imperfections modelled within the system
- arise from real-world unpredictability/unreliability
- are caused by real-world phenomena (such as humans)
- Input is a major source of errors:
  - command line, network traffic, files, input devices.
- are introduced through interfaces with the real world, e.g.:

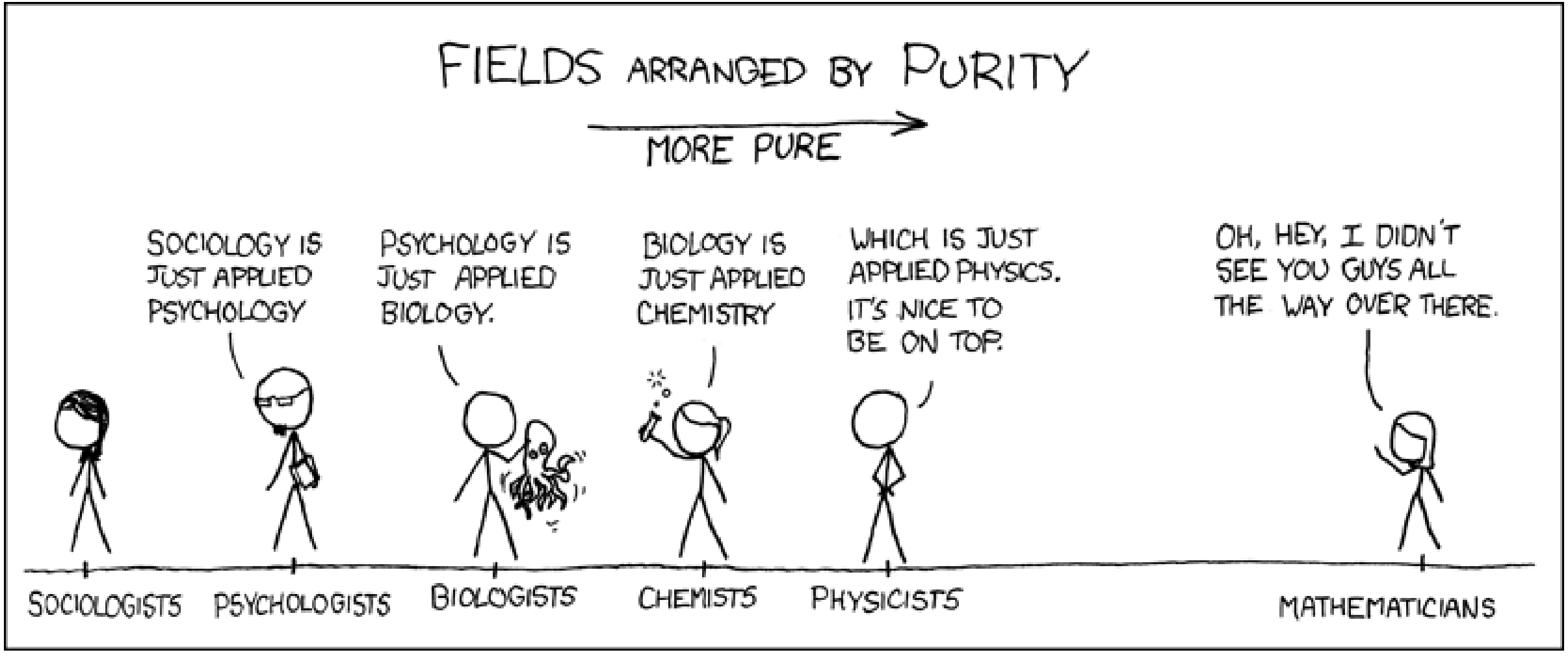
Errors

- are imperfections modelled within the system
- arise from real-world unpredictability/unreliability
- are caused by real-world phenomena (such as humans)
- Input is a major source of errors:
  - command line, network traffic, files, input devices.
- are introduced through interfaces with the real world, e.g.:
  - `std::filesystem` and `std::string` are UI elements

Errors

- are imperfections modelled within the system
- arise from real-world unpredictability/unreliability
- are caused by real-world phenomena (such as humans)
- Input is a major source of errors:
  - command line, network traffic, files, input devices.
- are introduced through interfaces with the real world, e.g.:
  - `std::filesystem` and `std::string` are UI elements
  - `std::chrono` models the real world and similarly 'messy'

xkcd.com/435



errors errors errors errors errors bugs

Errors are things that can go wrong

- even in perfect programs.

Examples of Errors

Examples of Errors

resource

Examples of Errors

resource

ill-formed input

- file too short
- file doesn't conform to format, e.g. JSON
- parameter is out of range
- unexpected device type
- unexpected network packet size

Error Handling

Program should handle *client violations* of the *End User Contract*

Error Handling

Program should handle *client violations* of the *End User Contract*

halt

- When an error occurs, the *request* should fail.
- Attempting to continue 'helpfully' is unwise.
- The failure should be prominent (e.g. `std::abort()`).

Error Handling

Program should handle *client violations* of the *End User Contract*

halt

- When an error occurs, the *request* should fail.
- Attempting to continue 'helpfully' is unwise.
- The failure should be prominent (e.g. `std::abort()`).

help

- Explain what failed.
- Explain how/why.
- Be thoughtful about details.
- What will help the user succeed next time?

$1,000,000 Question

How does your program handle errors?

$1,000,000 Answer

It depends.

It depends on the program

It depends on the program

- Is your program batch or steady-state?

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?
- Does your program respond through:

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?
- Does your program respond through:
  - a console,

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?
- Does your program respond through:
  - a console,
  - a GUI,

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?
- Does your program respond through:
  - a console,
  - a GUI,
  - a RESTful API,

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?
- Does your program respond through:
  - a console,
  - a GUI,
  - a RESTful API,
  - something else, or

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?
- Does your program respond through:
  - a console,
  - a GUI,
  - a RESTful API,
  - something else, or
  - nothing at all?

It depends on the program

- Is your program batch or steady-state?
- Does your program have realtime constraints?
- Does your program respond through:
  - a console,
  - a GUI,
  - a RESTful API,
  - something else, or
  - nothing at all?
- Is your program even a program, or reusable library?

Choices, choices!

Choices, choices!

C++ has too many error-handling facilities.

Choices, choices!

C++ has too many error-handling facilities.

But part of the problem is its versatility.

Choices, choices!

C++ has too many error-handling facilities.

But part of the problem is its versatility.

An important consideration is to allow for versatility.

If you’re lucky

If you're lucky

```
 1 namespace acme {
 2   // everything needed for program to do its thing;
 3   // well-formed and error-free
 4   struct sanitized_input {
 5     // ...
 6   };
 7
 8   // safety boundary; untrusted input; trusted output
 9   std::optional<sanitized_input> digest_input(std::span<char const* const> arg
10
11   // free from error handling
12   std::string do_the_thing(sanitized_input in);
13 }
14
15 int main(int argc, char const* const* argv)
16 {
```

If you're lucky

```
2   // everything needed for program to do its thing;
3   // well-formed and error-free
4   struct sanitized_input {
5     // ...
6   };
7
8   // safety boundary; untrusted input; trusted output
9   std::optional<sanitized_input> digest_input(std::span<char const* const> arg
10
11  // free from error handling
12  std::string do_the_thing(sanitized_input in);
13 }
14
15 int main(int argc, char const* const* argv)
16 {
```

If you're lucky

```
 8   // safety boundary; untrusted input; trusted output
 9   std::optional<sanitized_input> digest_input(std::span<char const* const> arg
10
11   // free from error handling
12   std::string do_the_thing(sanitized_input in);
13 }
14
15 int main(int argc, char const* const* argv)
16 {
17   // variable binding; type safety FTW!
18   auto const args{std::span{argv, argv+argc}};
19
20   auto const input{acme::digest_input(args)};
21   if (!input) {
22     return EXIT_FAILURE;
```

If you're lucky

```
13 }
14
15 int main(int argc, char const* const* argv)
16 {
17   // variable binding; type safety FTW!
18   auto const args{std::span{argv, argv+argc}};
19
20   auto const input{acme::digest_input(args)};
21   if (!input) {
22     return EXIT_FAILURE;
23   }
24
25   std::cout << acme::do_the_thing(*input);
26   return EXIT_SUCCESS;
27 }
```

If you're lucky

```
13 }
14
15 int main(int argc, char const* const* argv)
16 {
17   // variable binding; type safety FTW!
18   auto const args{std::span{argv, argv+argc}};
19
20   auto const input{acme::digest_input(args)};
21   if (!input) {
22     return EXIT_FAILURE;
23   }
24
25   std::cout << acme::do_the_thing(*input);
26   return EXIT_SUCCESS;
27 }
```

If you're lucky

```
13 }
14
15 int main(int argc, char const* const* argv)
16 {
17   // variable binding; type safety FTW!
18   auto const args{std::span{argv, argv+argc}};
19
20   auto const input{acme::digest_input(args)};
21   if (!input) {
22     return EXIT_FAILURE;
23   }
24
25   std::cout << acme::do_the_thing(*input);
26   return EXIT_SUCCESS;
27 }
```

If you're lucky

```
13 }
14
15 int main(int argc, char const* const* argv)
16 {
17   // variable binding; type safety FTW!
18   auto const args{std::span{argv, argv+argc}};
19
20   auto const input{acme::digest_input(args)};
21   if (!input) {
22     return EXIT_FAILURE;
23   }
24
25   std::cout << acme::do_the_thing(*input);
26   return EXIT_SUCCESS;
27 }
```

No Slides Complete Without...

No Slides Complete Without...

```
auto main(int argc, char const* const* argv) -> int
{
    auto const cl{command_line{argv, static_cast<unsigned>(argc)}};
    auto const result{wordle::run(cl)};

    if (auto const exit_status{std::get_if<int>(&result)}) {
        return *exit_status;
    }

    auto const& suggestions{std::get<wordle::words>(result)};
    print(suggestions);
}
```

github.com/johnmcfarlane/wss/blob/main/src/wordle/main.cpp

Some Techniques for Simple Programs

Some Techniques for Simple Programs

- Reporting:

Some Techniques for Simple Programs

- Reporting:
  - Log, e.g. print something helpful to `stderr`

Some Techniques for Simple Programs

- Reporting:
  - Log, e.g. print something helpful to `stderr`
- Control Flow (Sad Path):

Some Techniques for Simple Programs

- Reporting:
  - Log, e.g. print something helpful to `stderr`
- Control Flow (Sad Path):
  - Exceptions

Some Techniques for Simple Programs

- Reporting:
  - Log, e.g. print something helpful to `stderr`
- Control Flow (Sad Path):
  - Exceptions
  - Return values

Some Techniques for Simple Programs

- Reporting:
  - Log, e.g. print something helpful to `stderr`
- Control Flow (Sad Path):
  - Exceptions
  - Return values
  - Abnormal program termination

Some Techniques for Simple Programs

- Reporting:
  - Log, e.g. print something helpful to `stderr`
- Control Flow (Sad Path):
  - Exceptions
  - Return values
  - Abnormal program termination

Example 1: Print result, return success, log details

Example 1: Print result, return success, log details

```
1  // print file's size or return false
2  auto print_file_size(char const* filename)
3  {
4    std::ifstream in(filename, std::ios::binary | std::ios::ate);
5    if (!in) {
6      std::cerr << std::format("failed to open file \"{}\"\n", filename);
7      return false;
8    }
9
10   std::cout << std::format("{}\n", in.tellg());
11   return true;
12 }
13
14 auto print_config_file_size()
15 {
16   if (!print_file_size("default.cfg")) {
```

Example 1: Print result, return success, log details

```
 5    if (!in) {
 6      std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7      return false;
 8    }
 9
10    std::cout << std::format("{}\n", in.tellg());
11    return true;
12  }
13
14  auto print_config_file_size()
15  {
16    if (!print_file_size("default.cfg")) {
17      // in this function, we know the nature of the file
18      std::cerr << "failed to print the size of the config file\n";
19    }
20  }
```

Example 1: Print result, return success, log details

```
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     return false;
 8   }
 9
10   std::cout << std::format("{}\n", in.tellg());
11   return true;
12 }
13
14 auto print_config_file_size()
15 {
16   if (!print_file_size("default.cfg")) {
17     // in this function, we know the nature of the file
```

Example 1: Print result, return success, log details

```cpp
// print file's size or return false
auto print_file_size(char const* filename)
{
  std::ifstream in(filename, std::ios::binary | std::ios::ate);
  if (!in) {
    std::cerr << std::format("failed to open file \"{}\"\n", filename);
    return false;
  }

  std::cout << std::format("{}\n", in.tellg());
  return true;
}

auto print_config_file_size()
{
  if (!print_file_size("default.cfg")) {
```

Example 1: Print result, return success, log details

```
 5     if (!in) {
 6       std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7       return false;
 8     }
 9
10     std::cout << std::format("{}\n", in.tellg());
11     return true;
12   }
13
14   auto print_config_file_size()
15   {
16     if (!print_file_size("default.cfg")) {
17       // in this function, we know the nature of the file
18       std::cerr << "failed to print the size of the config file\n";
19     }
20   }
```

Example 2: Return result, ??? success, log details

Example 2: Return result, ??? success, log details

```
 1 // return file's size
 2 auto file_size(char const* filename)
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     // how is the disappointment returned now?
 8   }
 9
10   return in.tellg();
11 }
```

Example 2: Return result, ??? success, log details

```
 1 // return file's size
 2 auto file_size(char const* filename)
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     // how is the disappointment returned now?
 8   }
 9
10   return in.tellg();
11 }
```

Example 2: Return result, ??? success, log details

```
1  // return file's size
2  auto file_size(char const* filename)
3  {
4    std::ifstream in(filename, std::ios::binary | std::ios::ate);
5    if (!in) {
6      std::cerr << std::format("failed to open file \"{}\"\n", filename);
7      // how is the disappointment returned now?
8    }
9
10   return in.tellg();
11 }
```

Example 2: Return result, ??? success, log details

```
1  // return file's size
2  auto file_size(char const* filename)
3  {
4    std::ifstream in(filename, std::ios::binary | std::ios::ate);
5    if (!in) {
6      std::cerr << std::format("failed to open file \"{}\"\n", filename);
7      // how is the disappointment returned now?
8    }
9
10   return in.tellg();
11 }
```

Example 3: Return result *or* failure, log details

Example 3: Return result *or* failure, log details

```
 1 auto file_size(char const* filename)
 2 -> std::optional<std::ifstream::pos_type>
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     return std::nullopt;
 8   }
 9
10   return in.tellg();
11 }
```

Example 3: Return result *or* failure, log details

```
 1 auto file_size(char const* filename)
 2 -> std::optional<std::ifstream::pos_type>
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     return std::nullopt;
 8   }
 9
10   return in.tellg();
11 }
```

Example 3: Return result *or* failure, log details

```
 1 auto file_size(char const* filename)
 2 -> std::optional<std::ifstream::pos_type>
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     return std::nullopt;
 8   }
 9
10   return in.tellg();
11 }
```

Example 3: Return result *or* failure, log details

```
 1 auto file_size(char const* filename)
 2 -> std::optional<std::ifstream::pos_type>
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     return std::nullopt;
 8   }
 9
10   return in.tellg();
11 }
```

Example 3: Return result *or* failure, log details

```
 1 auto file_size(char const* filename)
 2 -> std::optional<std::ifstream::pos_type>
 3 {
 4   std::ifstream in(filename, std::ios::binary | std::ios::ate);
 5   if (!in) {
 6     std::cerr << std::format("failed to open file \"{}\"\n", filename);
 7     return std::nullopt;
 8   }
 9
10   return in.tellg();
11 }
```

Example 4: Return result, abort on failure, log details

Example 4: Return result, abort on failure, log details

```
// error handler function
template <typename... args>
[[noreturn]] void fatal(args&&... parameters)
{
  std::cerr << std::format(std::forward<args>(parameters)...);
  std::abort();
}

int main(int argc, char* argv[])
{
  auto const expected_num_params{3};
  if (argc != expected_num_params) {
    fatal(
        "Wrong number of arguments provided. Expected={}; Actual={}\n",
        expected_num_params, argc);
    return EXIT_FAILURE;
```

Example 4: Return result, abort on failure, log details

```
 3 [[noreturn]] void fatal(args&&... parameters)
 4 {
 5   std::cerr << std::format(std::forward<args>(parameters)...);
 6   std::abort();
 7 }
 8
 9 int main(int argc, char* argv[])
10 {
11   auto const expected_num_params{3};
12   if (argc != expected_num_params) {
13     fatal(
14         "Wrong number of arguments provided. Expected={}; Actual={}\n",
15         expected_num_params, argc);
16     return EXIT_FAILURE;
17   }
18 }
```

Example 4: Return result, abort on failure, log details

```
 3 [[noreturn]] void fatal(args&&... parameters)
 4 {
 5   std::cerr << std::format(std::forward<args>(parameters)...);
 6   std::abort();
 7 }
 8
 9 int main(int argc, char* argv[])
10 {
11   auto const expected_num_params{3};
12   if (argc != expected_num_params) {
13     fatal(
14         "Wrong number of arguments provided. Expected={}; Actual={}\n",
15         expected_num_params, argc);
16     return EXIT_FAILURE;
17   }
18 }
```

Example 4: Return result, abort on failure, log details

```
 1 // error handler function
 2 template <typename... args>
 3 [[noreturn]] void fatal(args&&... parameters)
 4 {
 5   std::cerr << std::format(std::forward<args>(parameters)...);
 6   std::abort();
 7 }
 8
 9 int main(int argc, char* argv[])
10 {
11   auto const expected_num_params{3};
12   if (argc != expected_num_params) {
13     fatal(
14         "Wrong number of arguments provided. Expected={}; Actual={}\n",
15         expected_num_params, argc);
16     return EXIT_FAILURE;
```

Functions Are a Track Event

Functions Are a Track Event

There are zero or more obstacles and one finish line.

Functions Are a Track Event

There are zero or more obstacles and one finish line.

```
auto do_something(auto param)
{
  // hurdle 1
  auto intermediate_thing1 = get_a_thing(param)
  if (!intermediate_thing1) {
    return failure;
  }

  // hurdle 2
  auto intermediate_thing2 = get_another_thing(intermediate_thing1)
  if (!intermediate_thing2) {
    return failure;
  }

  // finish line
  return intermediate_thing2;
```

Functions Are a Track Event

There are zero or more obstacles and one finish line.

```
auto do_something(auto param)
{
  // hurdle 1
  auto intermediate_thing1 = get_a_thing(param)
  if (!intermediate_thing1) {
    return failure;
  }

  // hurdle 2
  auto intermediate_thing2 = get_another_thing(intermediate_thing1)
  if (!intermediate_thing2) {
    return failure;
  }

  // finish line
  return intermediate_thing2;
```

Functions Are a Track Event

There are zero or more obstacles and one finish line.

```
{
  // hurdle 1
  auto intermediate_thing1 = get_a_thing(param)
  if (!intermediate_thing1) {
    return failure;
  }

  // hurdle 2
  auto intermediate_thing2 = get_another_thing(intermediate_thing1)
  if (!intermediate_thing2) {
    return failure;
  }

  // finish line
  return intermediate_thing2;
}
```

Functions Are a Track Event

There are zero or more obstacles and one finish line.

```
{
  // hurdle 1
  auto intermediate_thing1 = get_a_thing(param)
  if (!intermediate_thing1) {
    return failure;
  }

  // hurdle 2
  auto intermediate_thing2 = get_another_thing(intermediate_thing1)
  if (!intermediate_thing2) {
    return failure;
  }

  // finish line
  return intermediate_thing2;
}
```

Exceptions

Exceptions

- Pros:
  - versatile/scalable
  - very efficient normal path
  - hide control flow

Exceptions

- Pros:
  - versatile/scalable
  - very efficient normal path
  - hide control flow
- Cons:
  - exceedingly slow in exceptional path
  - not always optimal in normal path
  - hide control flow

Contract Attributes

	C++ API	standard	end user	test user
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

Contract Attributes

	C++ API	**standard**	end user	test user
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

C++ Standard

C++ Standard as a Contract

- The exchange of promises between C++ developers and C++ implementers.
- The authors of the contract are WG21 - not necessarily the providers.
- Client violations are *bugs*.
- Violation is UB*.
- Fixing bugs is as important as fixing compile-time errors.†

* In a talk about run-time disappointment

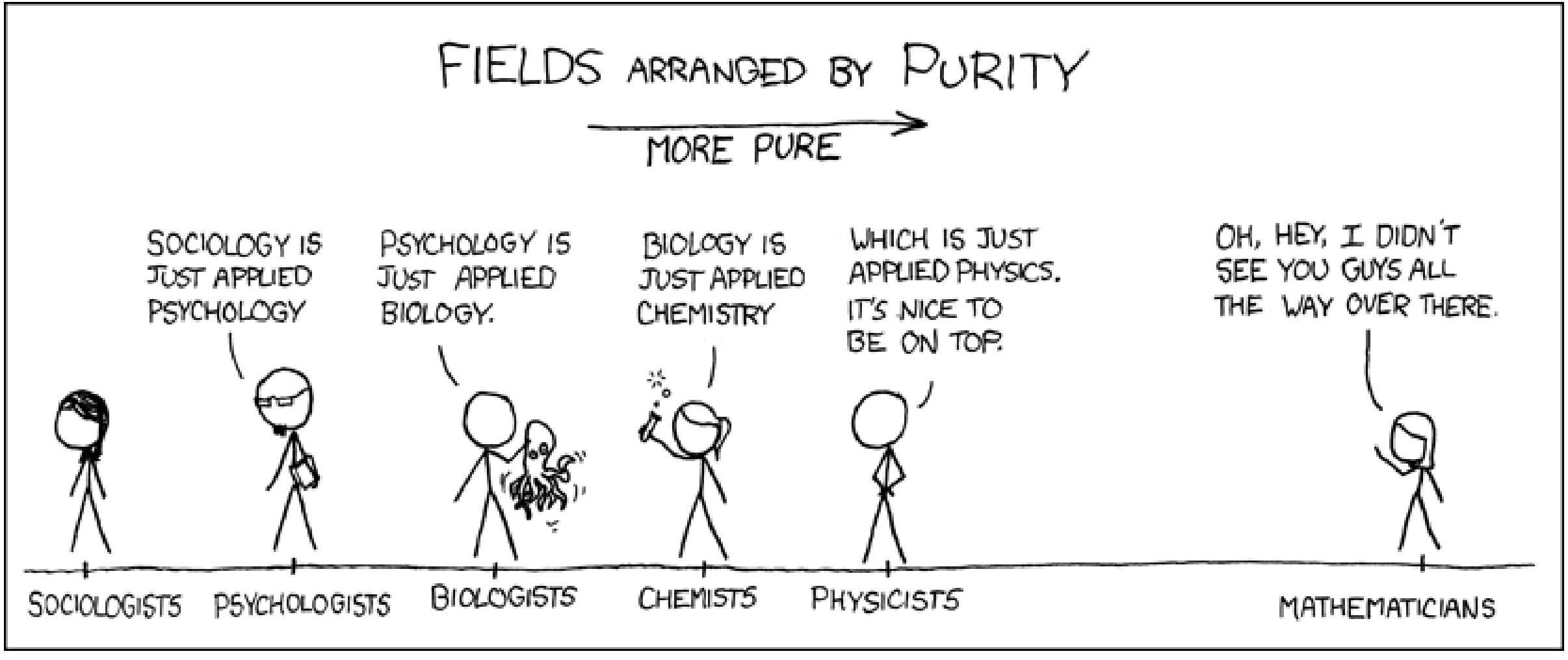
† Contentious

Bugs!

A program with a bug:

- is incorrect
- contains undefined behaviour
- is vulnerable
- violates the *End User Contract*.

xkcd.com/435



errors errors errors errors errors bugs

Bugs!

Prominent C++ Standard contract violation bugs fall into two main categories

Bugs!

Prominent C++ Standard contract violation bugs fall into two main categories

arithmetic

- divide-by-zero
- overflow

Bugs!

Prominent C++ Standard contract violation bugs fall into two main categories

arithmetic

- divide-by-zero
- overflow

object lifetime

- null pointer dereference
- dangling pointer dereference (use after free)
- out-of-bounds sequence lookup (e.g. buffer overflow)
- double-deletion
- ~~leaks~~

Example 1: Arithmetic

The problem

```
1  int main()
2  {
3    return 1/0;
4  }
```

Example 1: Arithmetic

The problem

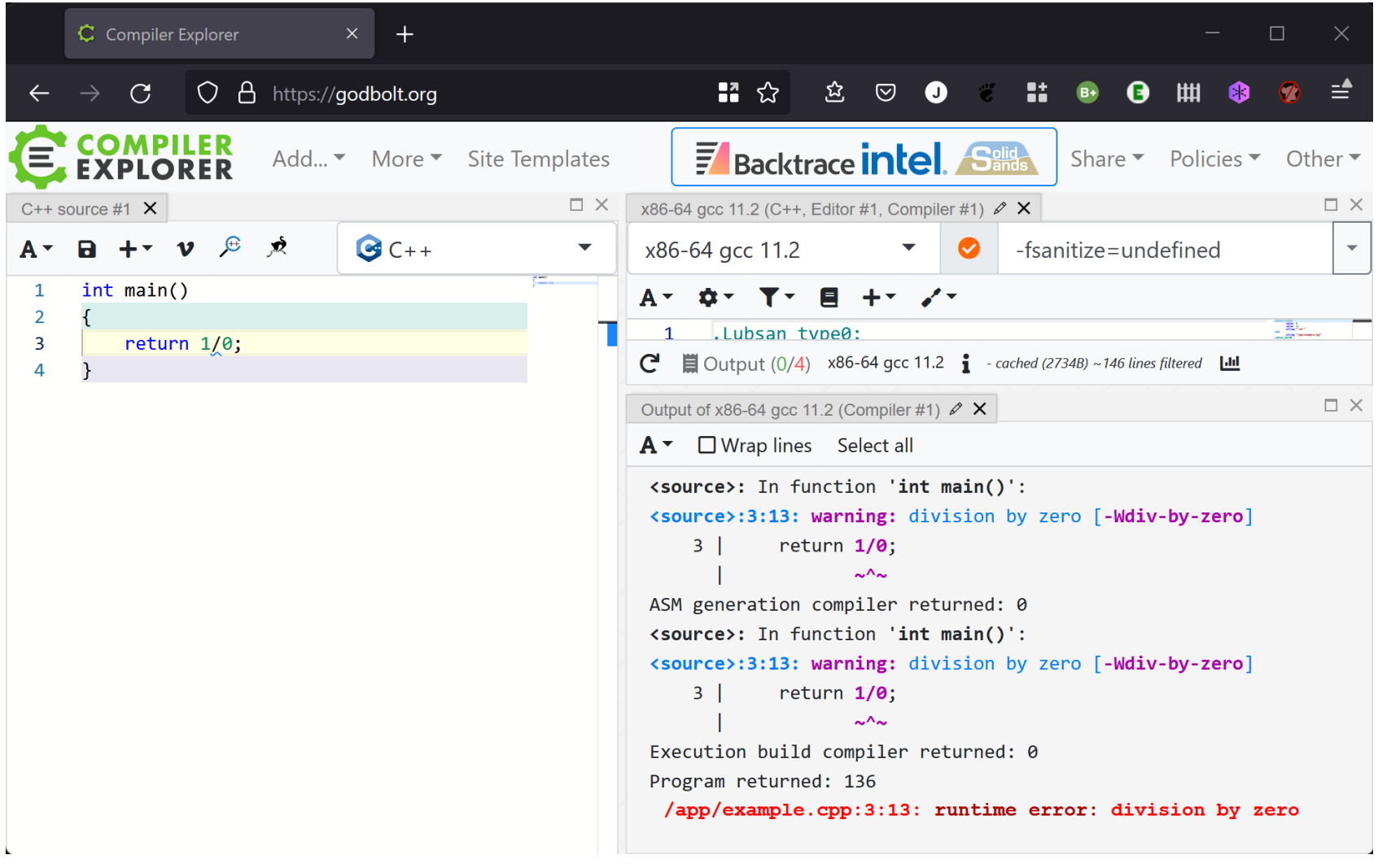
```
1  int main()
2  {
3    return 1/0;
4  }
```

Example 1: Arithmetic

The solution

flag	effect	when
`-Werror`	treat warnings as errors	compile
`-Wall -Wextra -pedantic`	gimme more warnings!	compile
`-fsanitize=undefined`	trap UB	execution

Undefined Behavior Sanitizer (UBSan)



Example 2

The problem

```
1 int main()
2 {
3     auto v{std::vector{0, 1}};
4     v.push_back(2);
5     fmt::print("{}\n", v[3]);
6 }
```

Example 2

The problem

```
1 int main()
2 {
3     auto v{std::vector{0, 1}};
4     v.push_back(2);
5     fmt::print("{}\n", v[3]);
6 }
```

Example 2

The problem

```
1 int main()
2 {
3     auto v{std::vector{0, 1}};
4     v.push_back(2);
5     fmt::print("{}\n", v[3]);
6 }
```

Example 2

The problem

```
1 int main()
2 {
3     auto v{std::vector{0, 1}};
4     v.push_back(2);
5     fmt::print("{}\n", v[3]);
6 }
```

Example 2

The solution

libc++: *-D_LIBCPP_DEBUG*

libstdc++: *-D_GLIBCXX_ASSERTIONS*

MSVC: */D_ITERATOR_DEBUG_LEVEL=1*

More Flags

flag or `intrinsic`	Clang	GCC	MSVC	Description
-Werror	✓	✓		turn warnings into errors
/WX			✓	turn warnings into errors
-Wall, *-Wconversion*, *-Wextra* and *-Wpedantic*	✓	✓		enable many warnings
/W4			✓	enable many warnings
-D_LIBCPP_ENABLE_NODISCARD	✓			enable some warnings
-fsanitize=undefined,address etc.	✓	✓		flag C++ Standard user contract violations[†]
-fno-sanitize-recover=all	✓	✓		trap bugs flagged with *-fsanitize=*
-fsanitize-recover=all etc.	✓	✓		report bugs flagged with *-fsanitize=*, then continue
-ftrapv	✓	✓		avoid; broken on GCC
-D_LIBCPP_DEBUG=1	✓			trap Standard Library user contract violations
-D_GLIBCXX_ASSERTIONS		✓		trap Standard Library user contract violations
-D_GLIBCXX_DEBUG or *-D_GLIBCXX_DEBUG_PEDANTIC*		✓		enable libstdc++ debug mode
/D_ITERATOR_DEBUG_LEVEL=2			✓	trap Standard Library user contract violations
`__builtin_unreachable()`	✓	✓		flag Unambiguous Bugs to compiler[‡]
`__assume(false)`			✓	flag Unambiguous Bugs to compiler[‡]
-DNDEBUG	✓	✓	✓	disable `assert` macro
-O0	✓	✓		disable optimisations[*]
/Od			✓	disable optimisations[*]
-fwrapv	✓	✓		disable signed integer overflow
-O, *-O1*, *-O2*, *-O3*, *-Os*, *-Ofast* or *-Og*	✓	✓		optimise code
/O1, */O2*, */Os*, */Ot* or */Ox*			✓	optimise code

https://github.com/johnmcfarlane/papers/blob/main/cpp/contractual-disappointment.md#appendix-a---toolchain-specific-recommendations

Contract Attributes

	C++ API	**standard**	end user	test user
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

Contract Attributes

	C++ API	**standard**	**end user**	**test user**
agreement	docs	ISO/IEC 14882	docs	docs
client	dev	dev	user	dev
provider	dev	implementer	dev	implementer
violation	bug	bug	error	error

C++ API Contracts

C++ API Contracts

- The exchange of promises between the developer(s) using and implementing a C++ API.
- Again, violations are *bugs*.
- Again, violation is UB.

Example of Client C++ API Contract Violation #1

PID Controller

PID controller

A **proportional–integral–derivative controller** (**PID controller** or **three-term controller**) is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an *error value* $e(t)$ as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (denoted *P*, *I*, and *D* respectively), hence the name.

In practical terms, PID automatically applies an accurate and responsive correction to a control function. An everyday example is the cruise control on a car, where ascending a hill would lower speed if constant engine power were applied. The controller's PID algorithm restores the measured speed to the desired speed with minimal delay and overshoot by increasing the power output of the engine in a controlled manner.

The first theoretical analysis and practical application of PID was in the field of automatic steering systems for ships, developed from the early 1920s onwards. It was then used for automatic process control in the manufacturing industry, where it was widely implemented in at first pneumatic and then electronic controllers. Today the PID concept is used universally in applications requiring accurate and optimized automatic control.

Contents

Kp=.1, Ki=.5, Kd=.01, setpoint=10, pv=30

Contract from PID

Mathematical form [edit]

The overall control function $u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t},$

where K_p, K_i, and K_d, all non-negative, denote the coefficients for the proportional, integral, and derivative terms respectively (sometimes denoted *P*, *I*, and *D*).

In the *standard form* of the equation (see later in article), K_i and K_d are respectively replaced by $K_\mathrm{p}/T_\mathrm{i}$ and $K_\mathrm{p}T_\mathrm{d}$; the advantage of this being that T_i and T_d have some understandable physical meaning, as they represent an integration time and a derivative time respectively. $K_\mathrm{p}T_\mathrm{d}$ is the time constant with which the controller will attempt to approach the set point. $K_\mathrm{p}/T_\mathrm{i}$ determines how long the controller will tolerate the error being consistently above or below the set point.

$$u(t) = K_\mathrm{p} \left(e(t) + \frac{1}{T_\mathrm{i}} \int_0^t e(\tau)\,\mathrm{d}\tau + T_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t} \right),$$

en.wikipedia.org/wiki/PID_controller#Mathematical_form

Contract from PID

Mathematical form [edit]

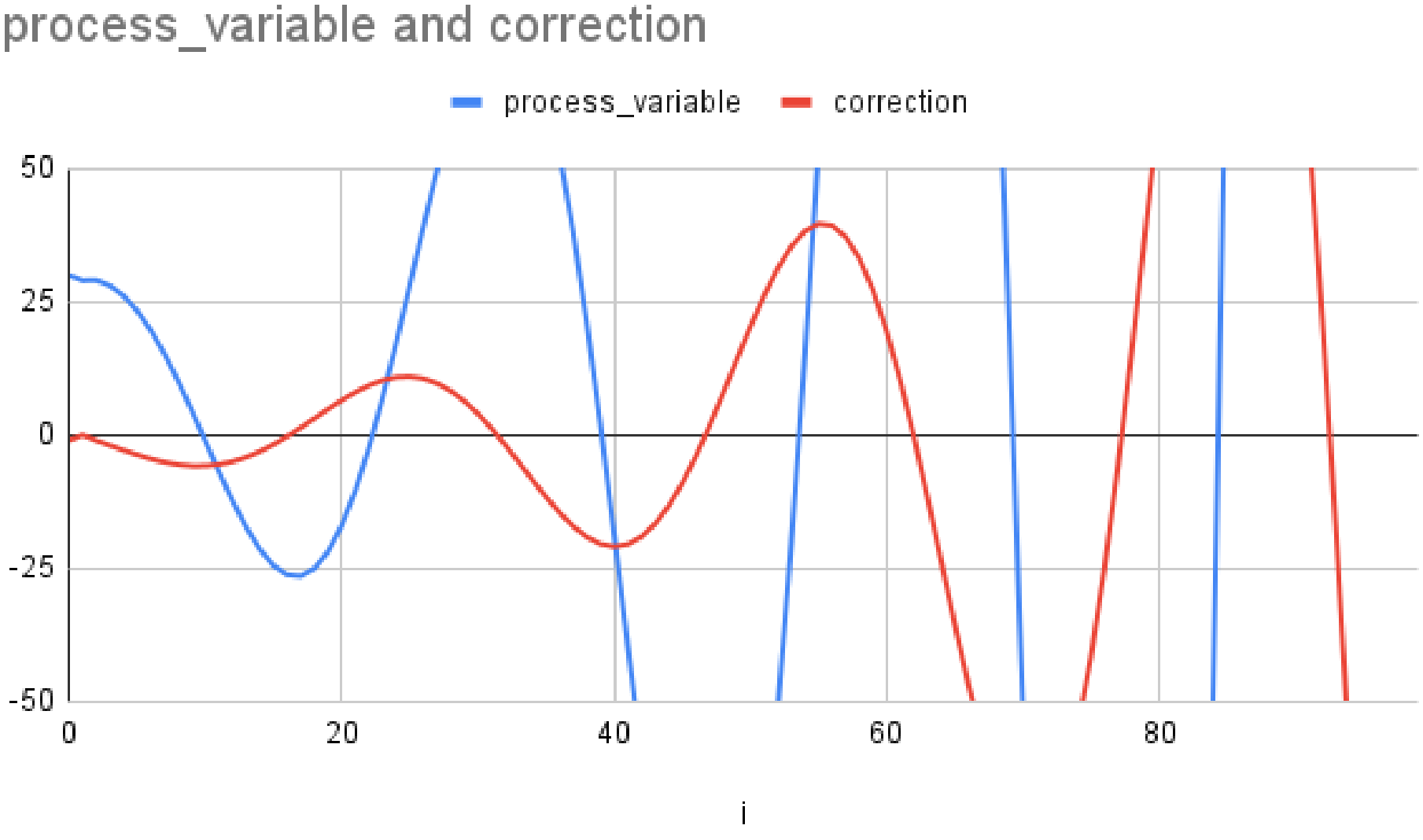
The overall control function $u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t}$,

where K_p, K_i, and K_d, all non-negative, denote the coefficients for the proportional, integral, and derivative terms respectively (sometimes denoted *P*, *I*, and *D*).

In the *standard form* of the equation (see later in article), K_i and K_d are respectively replaced by $K_\mathrm{p}/T_\mathrm{i}$ and $K_\mathrm{p}T_\mathrm{d}$; the advantage of this being that T_i and T_d have some understandable physical meaning, as they represent an integration time and a derivative time respectively. $K_\mathrm{p}T_\mathrm{d}$ is the time constant with which the controller will attempt to approach the set point. $K_\mathrm{p}/T_\mathrm{i}$ determines how long the controller will tolerate the error being consistently above or below the set point.

$$u(t) = K_\mathrm{p}\left(e(t) + \frac{1}{T_\mathrm{i}}\int_0^t e(\tau)\,\mathrm{d}\tau + T_\mathrm{d}\frac{\mathrm{d}e(t)}{\mathrm{d}t}\right),$$

en.wikipedia.org/wiki/PID_controller#Mathematical_form



Kp=-.1, Ki=.5, Kd=.01, setpoint=10, pv=30

PID Controller (interface)

```
 1 namespace pid {
 2   struct components {
 3     double proportional;
 4     double integral;
 5     double derivative;
 6   };
 7
 8   // values kept constant throughout operation of a controller
 9   struct parameters {
10     // non-negative factors used to generate PID terms
11     components k;
12
13     double dt;
14   };
15
16   struct state {
```

PID Controller (interface)

```
 1 namespace pid {
 2   struct components {
 3     double proportional;
 4     double integral;
 5     double derivative;
 6   };
 7
 8   // values kept constant throughout operation of a controller
 9   struct parameters {
10     // non-negative factors used to generate PID terms
11     components k;
12
13     double dt;
14   };
15
16   struct state {
```

PID Controller (interface)

```
 7
 8   // values kept constant throughout operation of a controller
 9   struct parameters {
10     // non-negative factors used to generate PID terms
11     components k;
12
13     double dt;
14   };
15
16   struct state {
17     double integral;
18     double error;
19   };
20

21   struct input {
```

PID Controller (interface)

```
 6   };
 7
 8   // values kept constant throughout operation of a controller
 9   struct parameters {
10     // non-negative factors used to generate PID terms
11     components k;
12
13     double dt;
14   };
15
16   struct state {
17     double integral;
18     double error;
19   };
20
```

PID Controller (interface)

```
20
21    struct input {
22      // desired value
23      double setpoint;
24
25      // actual value
26      double process_variable;
27    };
28
29    struct result {
30      // corrective value to apply to system
31      double correction;
32
33      // to pass in to next iteration as input::previous state
34      state current;
```

PID Controller (interface)

```
24
25      // actual value
26      double process_variable;
27    };
28
29    struct result {
30      // corrective value to apply to system
31      double correction;
32
33      // to pass in to next iteration as input::previous state
34      state current;
35    };
36
37    [[nodiscard]] auto calculate(parameters params, state previous, input in)
38        -> result;
39  }
```

PID Controller (interface)

```
24
25      // actual value
26      double process_variable;
27    };
28
29    struct result {
30      // corrective value to apply to system
31      double correction;
32
33      // to pass in to next iteration as input::previous state
34      state current;
35    };
36
37    [[nodiscard]] auto calculate(parameters params, state previous, input in)
38        -> result;
39  }
```

PID Controller (interface)

```
25      // actual value
26      double process_variable;
27    };
28
29    struct result {
30      // corrective value to apply to system
31      double correction;
32
33      // to pass in to next iteration as input::previous state
34      state current;
35    };
36
37    [[nodiscard]] auto calculate(parameters params, state previous, input in)
38        -> result;
39  }
```

PID Controller (implementation)

```cpp
#include "pid.h"

#include "pid_assert.h"

[[nodiscard]] auto pid::calculate(parameters params, state previous, input in)
    -> result
{
  PID_ASSERT(params.k.proportional >= 0);
  PID_ASSERT(params.k.integral >= 0);
  PID_ASSERT(params.k.derivative >= 0);
  PID_ASSERT(params.dt > 0);

  auto const error = in.setpoint - in.process_variable;
  auto const next_integral{previous.integral + error * params.dt};
  auto const derivative = (error - previous.error) / params.dt;

```

PID Controller (implementation)

```
 3 #include "pid_assert.h"
 4
 5 [[nodiscard]] auto pid::calculate(parameters params, state previous, input in)
 6     -> result
 7 {
 8   PID_ASSERT(params.k.proportional >= 0);
 9   PID_ASSERT(params.k.integral >= 0);
10   PID_ASSERT(params.k.derivative >= 0);
11   PID_ASSERT(params.dt > 0);
12
13   auto const error = in.setpoint - in.process_variable;
14   auto const next_integral{previous.integral + error * params.dt};
15   auto const derivative = (error - previous.error) / params.dt;
16
17   auto const terms{components{
18     .proportional = params.k.proportional * error,
```

PID Controller (implementation)

```
 7 {
 8   PID_ASSERT(params.k.proportional >= 0);
 9   PID_ASSERT(params.k.integral >= 0);
10   PID_ASSERT(params.k.derivative >= 0);
11   PID_ASSERT(params.dt > 0);
12
13   auto const error = in.setpoint - in.process_variable;
14   auto const next_integral{previous.integral + error * params.dt};
15   auto const derivative = (error - previous.error) / params.dt;
16
17   auto const terms{components{
18       .proportional = params.k.proportional * error,
19       .integral = params.k.integral * next_integral,
20       .derivative = params.k.derivative * derivative}};

21
```

PID Controller (implementation)

```
13    auto const error = in.setpoint - in.process_variable;
14    auto const next_integral{previous.integral + error * params.dt};
15    auto const derivative = (error - previous.error) / params.dt;
16
17    auto const terms{components{
18        .proportional = params.k.proportional * error,
19        .integral = params.k.integral * next_integral,
20        .derivative = params.k.derivative * derivative}};

21
22    auto const output = terms.proportional + terms.integral + terms.derivative;
23
24    return result{
25        output,
26        state{next_integral, error}};
27 }
```

PID Controller (implementation)

```
13    auto const error = in.setpoint - in.process_variable;
14    auto const next_integral{previous.integral + error * params.dt};
15    auto const derivative = (error - previous.error) / params.dt;
16
17    auto const terms{components{
18        .proportional = params.k.proportional * error,
19        .integral = params.k.integral * next_integral,
20        .derivative = params.k.derivative * derivative}};

21
22    auto const output = terms.proportional + terms.integral + terms.derivative;
23
24    return result{
25        output,
26        state{next_integral, error}};
27 }
```

PID Controller (implementation)

```
13    auto const error = in.setpoint - in.process_variable;
14    auto const next_integral{previous.integral + error * params.dt};
15    auto const derivative = (error - previous.error) / params.dt;
16
17    auto const terms{components{
18        .proportional = params.k.proportional * error,
19        .integral = params.k.integral * next_integral,
20        .derivative = params.k.derivative * derivative}};

21
22    auto const output = terms.proportional + terms.integral + terms.derivative;
23
24    return result{
25        output,
26        state{next_integral, error}};
27 }
```

PID Controller (implementation)

```
13    auto const error = in.setpoint - in.process_variable;
14    auto const next_integral{previous.integral + error * params.dt};
15    auto const derivative = (error - previous.error) / params.dt;
16
17    auto const terms{components{
18        .proportional = params.k.proportional * error,
19        .integral = params.k.integral * next_integral,
20        .derivative = params.k.derivative * derivative}};

21
22    auto const output = terms.proportional + terms.integral + terms.derivative;
23
24    return result{
25        output,
26        state{next_integral, error}};
27 }
```

Example of Client C++ API Contract Violation #2

(Anecdotal) UID vs Bitmap

```
1  typedef uid = std::uint32_t;
2  constexpr auto invalid_id{uid{-1}};
3  ...
4  class bitset {
5  public:
6    bool get(std::size_t index) const {
7      if (index >= size()) {
8        resize(index+1);
9      }
10   ...
11   }
12   ...
13 };
```

Example of Client C++ API Contract Violation #2

(Anecdotal) UID vs Bitmap

```
1  typedef uid = std::uint32_t;
2  constexpr auto invalid_id{uid{-1}};
3  ...
4  class bitset {
5  public:
6    bool get(std::size_t index) const {
7      if (index >= size()) {
8        resize(index+1);
9      }
10   ...
11   }
12   ...
13 };
```

Example of Client C++ API Contract Violation #2

(Anecdotal) UID vs Bitmap

```
1  typedef uid = std::uint32_t;
2  constexpr auto invalid_id{uid{-1}};
3  ...
4  class bitset {
5  public:
6    bool get(std::size_t index) const {
7      if (index >= size()) {
8        resize(index+1);
9      }
10   ...
11   }
12   ...
13 };
```

Example of Client C++ API Contract Violation #2

(Anecdotal) UID vs Bitmap

```
1  typedef uid = std::uint32_t;
2  constexpr auto invalid_id{uid{-1}};
3  ...
4  class bitset {
5  public:
6    bool get(std::size_t index) const {
7      if (index >= size()) {
8        resize(index+1);
9      }
10   ...
11   }
12   ...
13 };
```

Example of Client C++ API Contract Violation #2

(Anecdotal) UID vs Bitmap

```
typedef uid = std::uint32_t;
constexpr auto invalid_id{uid{-1}};
...
class bitset {
public:
  bool get(std::size_t index) const {
    if (index >= size()) {
      resize(index+1);
    }
  ...
  }
  ...
};
```

Example of Client C++ API Contract Violation #2

(Anecdotal) UID vs Bitmap

```
1  typedef uid = std::uint32_t;
2  constexpr auto invalid_id{uid{-1}};
3  ...
4  class bitset {
5  public:
6    bool get(std::size_t index) const {
7      if (index >= size()) {
8        resize(index+1);
9      }
10   ...
11   }
12   ...
13 };
```

UID vs Bitmap

Observations

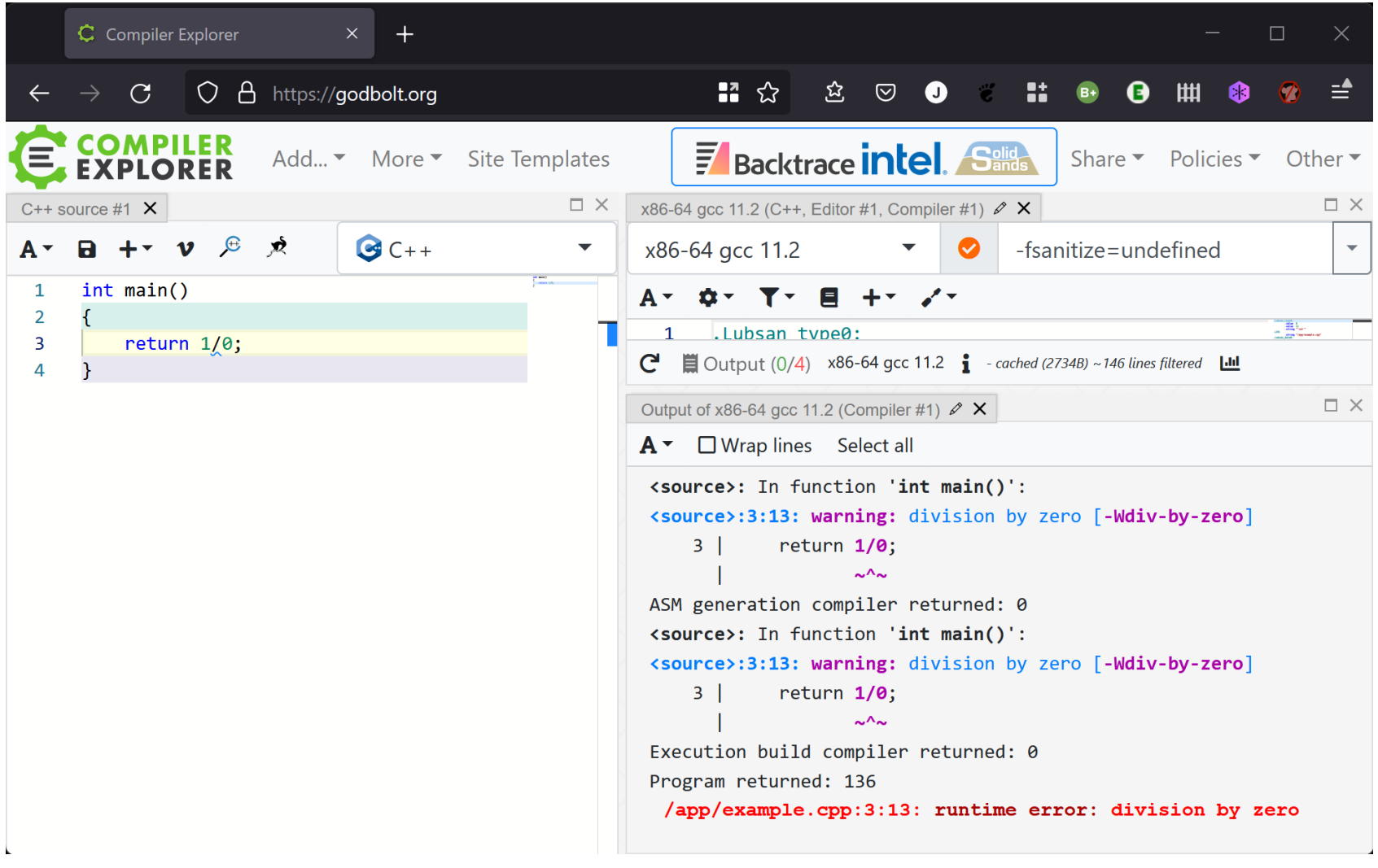
- Sentinel values, e.g. `invalid_id`, are trouble!
- *Defensive* or *helpful* code is unwelcome complexity.
- Trap bugs as they hatch.
- Don't rely on `unsigned` types to help you.

Test User Contract

The Test User Contract

- An exchange of promises between C++ developers and C++ tools providers.
- One such tool is `assert`.
- Sanitizers are newer such tools.
- Authors are implementers, not the committee.
- API/standard violations are ~~bugs~~ *errors*.
- These errors arise at the point where a bug is discovered.
- The user is a dev in need of feedback about correctness.

Remember Example 1?



Trigger Warning: This Assertion Triggers UB!

```
 1 // For testing coverage, assertions are not necessarily a concern.
 2 #if defined(PID_DISABLE_ASSERTS)
 3 #define PID_ASSERT(cond)
 4
 5 // In debug builds, fail fast and loud when an assertion is challenged.
 6 #elif !defined(NDEBUG)
 7 #define PID_ASSERT(cond) ((cond) ? static_cast<void>(0) : std::terminate())
 8
 9 // In optimised GCC builds, optimise/sanitize accordingly.
10 #elif defined(__GNUC__)
11 // NOLINTNEXTLINE(cppcoreguidelines-macro-usage)
12 #define PID_ASSERT(cond) ((cond) ? static_cast<void>(0) : __builtin_unreachabl
13
14 // In optimised MSVC builds, optimise/sanitize accordingly.
15 #elif defined(_MSC_VER)
16 #define PID_ASSERT(cond) __assume(cond)
```

Trigger Warning: This Assertion Triggers UB!

```
 1 // For testing coverage, assertions are not necessarily a concern.
 2 #if defined(PID_DISABLE_ASSERTS)
 3 #define PID_ASSERT(cond)
 4
 5 // In debug builds, fail fast and loud when an assertion is challenged.
 6 #elif !defined(NDEBUG)
 7 #define PID_ASSERT(cond) ((cond) ? static_cast<void>(0) : std::terminate())
 8
 9 // In optimised GCC builds, optimise/sanitize accordingly.
10 #elif defined(__GNUC__)
11 // NOLINTNEXTLINE(cppcoreguidelines-macro-usage)
12 #define PID_ASSERT(cond) ((cond) ? static_cast<void>(0) : __builtin_unreachabl
13
14 // In optimised MSVC builds, optimise/sanitize accordingly.
15 #elif defined(_MSC_VER)
16 #define PID_ASSERT(cond) __assume(cond)
```

Trigger Warning: This Assertion Triggers UB!

```
 6 #elif !defined(NDEBUG)
 7 #define PID_ASSERT(cond) ((cond) ? static_cast<void>(0) : std::terminate())
 8
 9 // In optimised GCC builds, optimise/sanitize accordingly.
10 #elif defined(__GNUC__)
11 // NOLINTNEXTLINE(cppcoreguidelines-macro-usage)
12 #define PID_ASSERT(cond) ((cond) ? static_cast<void>(0) : __builtin_unreachabl
13
14 // In optimised MSVC builds, optimise/sanitize accordingly.
15 #elif defined(_MSC_VER)
16 #define PID_ASSERT(cond) __assume(cond)
17
18 // In other optimised builds assume code is correct.
19 #else
20 #define PID_ASSERT(cond)
```

Trigger Warning: This Assertion Triggers UB!

```
 8
 9 // In optimised GCC builds, optimise/sanitize accordingly.
10 #elif defined(__GNUC__)
11 // NOLINTNEXTLINE(cppcoreguidelines-macro-usage)
12 #define PID_ASSERT(cond) ((cond) ? static_cast<void>(0) : __builtin_unreachabl
13
14 // In optimised MSVC builds, optimise/sanitize accordingly.
15 #elif defined(_MSC_VER)
16 #define PID_ASSERT(cond) __assume(cond)
17
18 // In other optimised builds assume code is correct.
19 #else
20 #define PID_ASSERT(cond)
21
22 #endif
```

Simplicity, Uniformity, Versatility

Simplicity, Uniformity, Versatility

- The choice of how to handle bugs lies in the hands of the developer using the code.

Simplicity, Uniformity, Versatility

- The choice of how to handle bugs lies in the hands of the developer using the code.
- UB is a strong indicator of bugs.

Simplicity, Uniformity, Versatility

- The choice of how to handle bugs lies in the hands of the developer using the code.
- UB is a strong indicator of bugs.
- All bugs stink.

Simplicity, Uniformity, Versatility

- The choice of how to handle bugs lies in the hands of the developer using the code.
- UB is a strong indicator of bugs.
- All bugs stink.
- If you are unsure about correctness (which you should be) you are taking a risk by releasing your product to the client.

Simplicity, Uniformity, Versatility

- The choice of how to handle bugs lies in the hands of the developer using the code.
- UB is a strong indicator of bugs.
- All bugs stink.
- If you are unsure about correctness (which you should be) you are taking a risk by releasing your product to the client.
- If you are unsure about correctness (which you should be) you are taking a risk by enabling optimisations.

Simplicity, Uniformity, Versatility

- The choice of how to handle bugs lies in the hands of the developer using the code.
- UB is a strong indicator of bugs.
- All bugs stink.
- If you are unsure about correctness (which you should be) you are taking a risk by releasing your product to the client.
- If you are unsure about correctness (which you should be) you are taking a risk by enabling optimisations.
- The distinction between 'user bugs', 'language UB', 'hard UB', 'time travel UB' etc. is false.

Bugs are Bugs

```
1 // precondition: number is in range [1..26]
2 constexpr auto number_to_letter(int number)
3 {
4   return char(number - 1 + 'A');
5 }
6
7 // signed integer overflow violates C++ Standard, is already UB
8 number_to_letter(0x7fffffff);
```

Bugs are Bugs

```
1 // precondition: number is in range [1..26]
2 constexpr auto number_to_letter(int number)
3 {
4   return char(number - 1 + 'A');
5 }
6
7 // signed integer overflow violates C++ Standard, is already UB
8 number_to_letter(0x7fffffff);
```

Bugs are Bugs

```
1 // precondition: number is in range [1..26]
2 constexpr auto number_to_letter(int number)
3 {
4   return char(number - 1 + 'A');
5 }
6
7 // signed integer overflow violates C++ Standard, is already UB
8 number_to_letter(0x7fffffff);
```

Contracts Protect Implementation Freedom

```
1 constexpr auto number_to_letter(int number)
2 {
3   constexpr auto lookup_table = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
4   return lookup_table[number - 1];
5 }
6
7 // signed integer overflow violates C++ Standard, is already UB
8 number_to_letter(0);
```

Contracts Protect Implementation Freedom

```
1 constexpr auto number_to_letter(int number)
2 {
3   constexpr auto lookup_table = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
4   return lookup_table[number - 1];
5 }
6
7 // signed integer overflow violates C++ Standard, is already UB
8 number_to_letter(0);
```

Strategies

- Trap Enforcement Strategy - Bugs are Fatal
- Non-enforcement Strategy - Struggle on
- Log-And-Continue Strategy - Bugs happen
- Prevention Enforcement Strategy - Bugs, what bugs?

Perceived Trade-offs

	correct	safe	efficient	defensive
'unsafe' languages, untested/legacy	😞	😞	😊	🙂
'safe' languages	🙂	🙂	🙂	😊
'unsafe' languages, tested/modern	😊	😊	😊	😃

Mars Code, Gerard J. Holzmann, 2014

- Mars Science Laboratory, written in C
- four static analysers run nightly
- used dynamic thread analysis tool
- warnings enabled and enforced in compiler
- all mission-critical code
  - had to be 2% assertions
  - had to remain enabled after testing

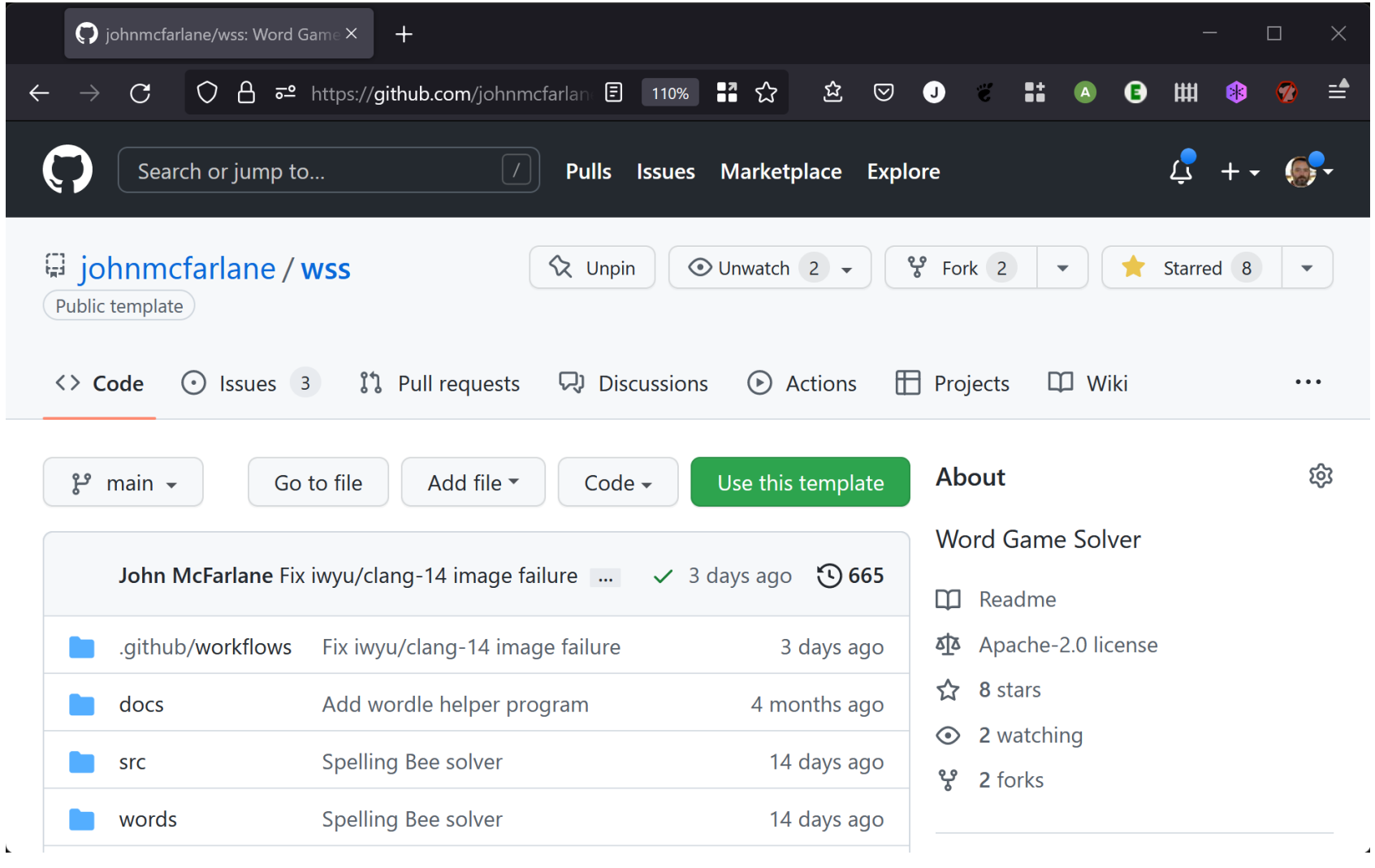
Mars Code, Gerard J. Holzmann, 2014

A failing assertion is now tied in with the fault-protection system and by default places the spacecraft into a predefined safe state where the cause of the failure can be diagnosed carefully before normal operation is resumed.

Don’t Optimise Until You Sanitize!

- Test your code before you release it.
- Make sure it's all tested (coverage).
- Make sure it's all *really* tested (fuzzing).
- Get your 9's.

WSS - Template for C++ Projects



Thank You

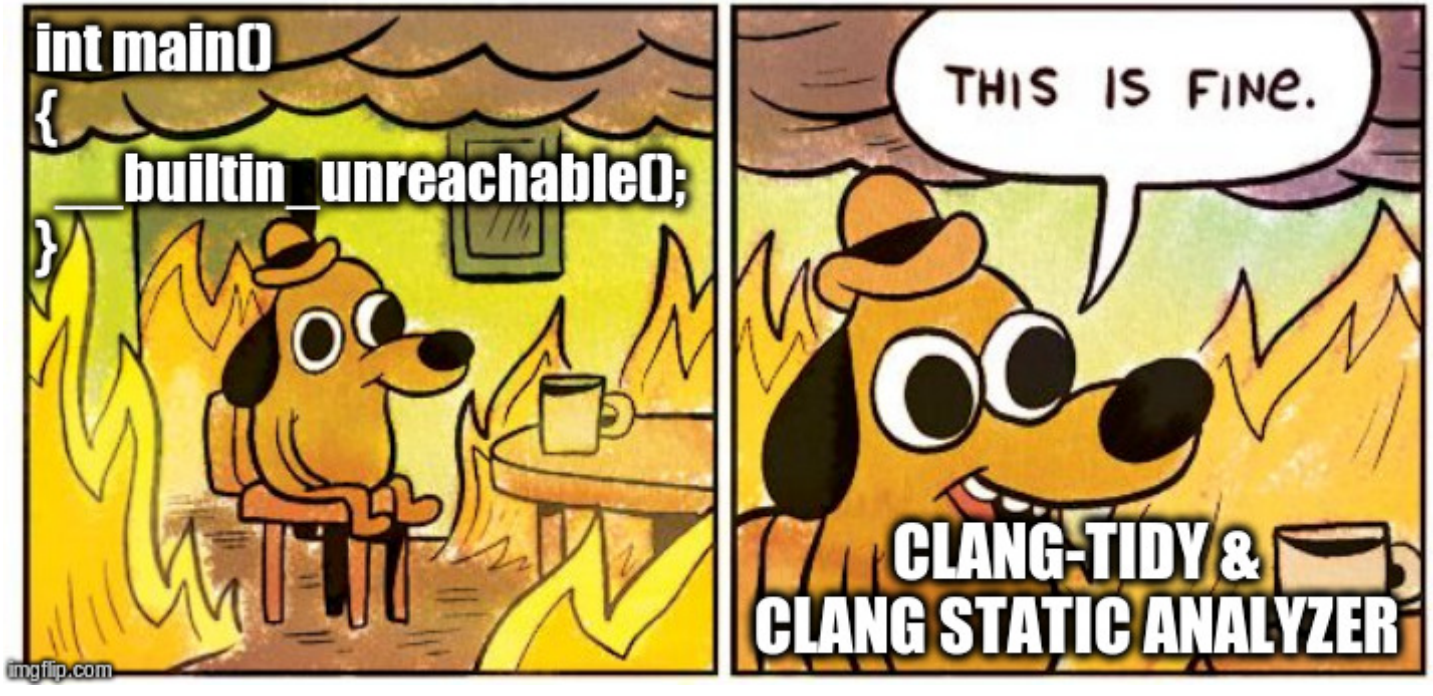
John McFarlane

github.com/johnmcfarlane/wss

twitter.com/JSAMcFarlane

johnmcfarlane.github.io/cpponsea-2022

Clang-Tidy Avoids Unreachable Paths



godbolt.org/z/oWjPfrKds

"Doesn't look like anything to me"

’Bonus’ Material

Another (maybe) hour of poorly proofed slides

Naming

- Names matter to contracts
- If the meaning of an element changes, consider changing the name

Bug or Error?

```
 1 int f(int const* p, int a, int b)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = a; i <= b; i ++)
 6   {
 7     r += p[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return f(p, -1, 1);
16 }
```

Bug or Error?

```
 1 int f(int const* p, int a, int b)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = a; i <= b; i ++)
 6   {
 7     r += p[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return f(p, -1, 1);
16 }
```

Bug or Error?

```
 1 int f(int const* p, int a, int b)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = a; i <= b; i ++)
 6   {
 7     r += p[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return f(p, -1, 1);
16 }
```

Bug or Error?

```
 1 int f(int const* p, int a, int b)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = a; i <= b; i ++)
 6   {
 7     r += p[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return f(p, -1, 1);
16 }
```

Bug or Error?

```
1  int f(int const* p, int a, int b)
2  {
3    // Are we good?
4    int r = 0;
5    for (int i = a; i <= b; i ++)
6    {
7      r += p[i];
8    }
9    return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return f(p, -1, 1);
16 }
```

Bug or Error?

```
 1 int f(int const* p, int a, int b)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = a; i <= b; i ++)
 6   {
 7     r += p[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return f(p, -1, 1);
16 }
```

maybe a bug, maybe not

Bug?

```
 1 int accumulate(int const* numbers, int first, int last)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += numbers[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return accumulate(p, -1, 1);
16 }
```

Bug?

```
 1 int accumulate(int const* numbers, int first, int last)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += numbers[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return accumulate(p, -1, 1);
16 }
```

Bug?

```
 1 int accumulate(int const* numbers, int first, int last)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += numbers[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return accumulate(p, -1, 1);
16 }
```

it's a bug!

Bug!

```
 1 int accumulate(int const* numbers, int first, int last)
 2 {
 3   assert(first >= 0);
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += numbers[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Bug: -1 isn't in sequence that starts with p
15   return accumulate(p, -1, 1);
16 }
```

Bug!

```
 1 int accumulate(int const* numbers, int first, int last)
 2 {
 3   assert(first >= 0);
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += numbers[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Bug: -1 isn't in sequence that starts with p
15   return accumulate(p, -1, 1);
16 }
```

Bug!

```
 1 int accumulate(int const* numbers, int first, int last)
 2 {
 3   assert(first >= 0);
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += numbers[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Bug: -1 isn't in sequence that starts with p
15   return accumulate(p, -1, 1);
16 }
```

Bug!

```
 1 int accumulate(int const* numbers, int first, int last)
 2 {
 3   assert(first >= 0);
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += numbers[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Bug: -1 isn't in sequence that starts with p
15   return accumulate(p, -1, 1);
16 }
```

but...

Bug?

```
 1 int sample(int const* center, int first, int last)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += center[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return sample(p, -1, 1);
16 }
```

Bug?

```
 1 int sample(int const* center, int first, int last)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += center[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return sample(p, -1, 1);
16 }
```

Bug?

```
 1 int sample(int const* center, int first, int last)
 2 {
 3   // Are we good?
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += center[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // Is this OK?
15   return sample(p, -1, 1);
16 }
```

what about now?

No Bug!

```
 1 int sample(int const* center, int first, int last)
 2 {
 3   // First might be anything.
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += center[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // center is not necessarily the start of the sequence.
15   return sample(p, -1, 1);
16 }
```

No Bug!

```
 1 int sample(int const* center, int first, int last)
 2 {
 3   // First might be anything.
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += center[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // center is not necessarily the start of the sequence.
15   return sample(p, -1, 1);
16 }
```

No Bug!

```
 1 int sample(int const* center, int first, int last)
 2 {
 3   // First might be anything.
 4   int r = 0;
 5   for (int i = first; i <= last; i ++)
 6   {
 7     r += center[i];
 8   }
 9   return r;
10 }
11
12 int g(int const* p)
13 {
14   // center is not necessarily the start of the sequence.
15   return sample(p, -1, 1);
16 }
```

Naming

- Problem:
  - Two functions use the same algorithm
  - But they have different contracts
  - How do you test different contracts from the same function?
- Solution:
  - Different functions?

Naming

```
 1 int accumulate_neighborhood(int const* position, int offset_first, int offset_
 2 {
 3   int r = 0;
 4   for (int i = offset_first; i <= offset_last; i ++)
 5   {
 6     r += position[i];
 7   }
 8   return r;
 9 }
10
11 int sample(int const* center, int first, int last)
12 {
13   return accumulate_neighborhood(center, first, last);
14 }
15
16 int accumulate_subrange(int const* numbers, int first, int last)
```

Naming

```
 6     r += position[i];
 7   }
 8   return r;
 9 }
10
11 int sample(int const* center, int first, int last)
12 {
13   return accumulate_neighborhood(center, first, last);
14 }
15
16 int accumulate_subrange(int const* numbers, int first, int last)
17 {
18   assert(first >= 0);
19   return accumulate_neighborhood(numbers, first, last);
20 }
```

Naming

```
 6     r += position[i];
 7   }
 8   return r;
 9 }
10
11 int sample(int const* center, int first, int last)
12 {
13   return accumulate_neighborhood(center, first, last);
14 }
15
16 int accumulate_subrange(int const* numbers, int first, int last)
17 {
18   assert(first >= 0);
19   return accumulate_neighborhood(numbers, first, last);
20 }
```

Naming

```
 6     r += position[i];
 7   }
 8   return r;
 9 }
10
11 int sample(int const* center, int first, int last)
12 {
13   return accumulate_neighborhood(center, first, last);
14 }
15
16 int accumulate_subrange(int const* numbers, int first, int last)
17 {
18   assert(first >= 0);
19   return accumulate_neighborhood(numbers, first, last);
20 }
```

Naming

```
 6     r += position[i];
 7   }
 8   return r;
 9 }
10
11 int sample(int const* center, int first, int last)
12 {
13   return accumulate_neighborhood(center, first, last);
14 }
15
16 int accumulate_subrange(int const* numbers, int first, int last)
17 {
18   assert(first >= 0);
19   return accumulate_neighborhood(numbers, first, last);
20 }
```

Naming

```
 1 int accumulate_neighborhood(int const* position, int offset_first, int offset_
 2 {
 3   int r = 0;
 4   for (int i = offset_first; i <= offset_last; i ++)
 5   {
 6     r += position[i];
 7   }
 8   return r;
 9 }
10
11 int sample(int const* center, int first, int last)
12 {
13   return accumulate_neighborhood(center, first, last);
14 }
15
16 int accumulate_subrange(int const* numbers, int first, int last)
```

what about now?

Surgery is Now Open

- Q: My project doesn't use analysis tools or modern, quality toolchains.
- A: Sorry about that. Consider running tests against nice tools.

Surgery is Now Open

- Q: A million things would break if I enabled checks.
- A: Disable checks and exclude all files. Then slowly fix things one check/file at a time until all the checks you want are applied to all files.

Surgery is Now Open

- Q: My project doesn't test the code.
- A: You're problems are beyond the specialty of this doctor.

Surgery is Now Open

- Q: My dependencies trigger warnings/errors
- A: Think about the contract between you and your dependency provider; try `-isystem`.

Surgery is Now Open

- Q: This stuff gets hard in big, old projects maintained by big, young teams
- A: Agreed. There is no silver bullet.

Surgery is Now Open

- Q: My project doesn't need to be safe/secure. I don't need to worry about this stuff, right?
- A: ...

On Correctness

- Correctness is a consequence of generally-good practices:
  - using modern features (std::print, std::optional, std::vector, owning pointers)
  - testing code
  - using tools
  - healthy team dynamics (mentoring, pairing, reviewing)
  - avoiding accidental complexity

On Correctness

- Correctness gives you
  - quality - your software works better sooner
  - productivity - less time wasted testing changes, debugging, fixing
  - knowledge - tools teach you how to avoid mistakes
  - safety & security guarantees

In Defence of Simplicity

- Keep all your software simple and correct, including:
  - Functional (production) code
  - Automated tests
  - Documentation
  - Build system
- Avoid control flow, especially `if` statements
- Don't over-engineer or write code you don't need (YAGNI)

Coding Standards

- Commit to modern practices and conventions, e.g.:
  - C++ Core Guidelines
  - Modern CMake
  - Linux-flavour Git commit descriptions
- Enforce with tools, tools, tools!

Keep Your Friends Close; Keep Your Errors Closer

- Minimise distance (in space and time) between bug location (source code that needs fixing) and point of failure (crash, trap, unwanted behaviour)
- Being explicit and strict about C++ API Contracts helps this enormously
- Accordingly assertions help. Language feature will help too.