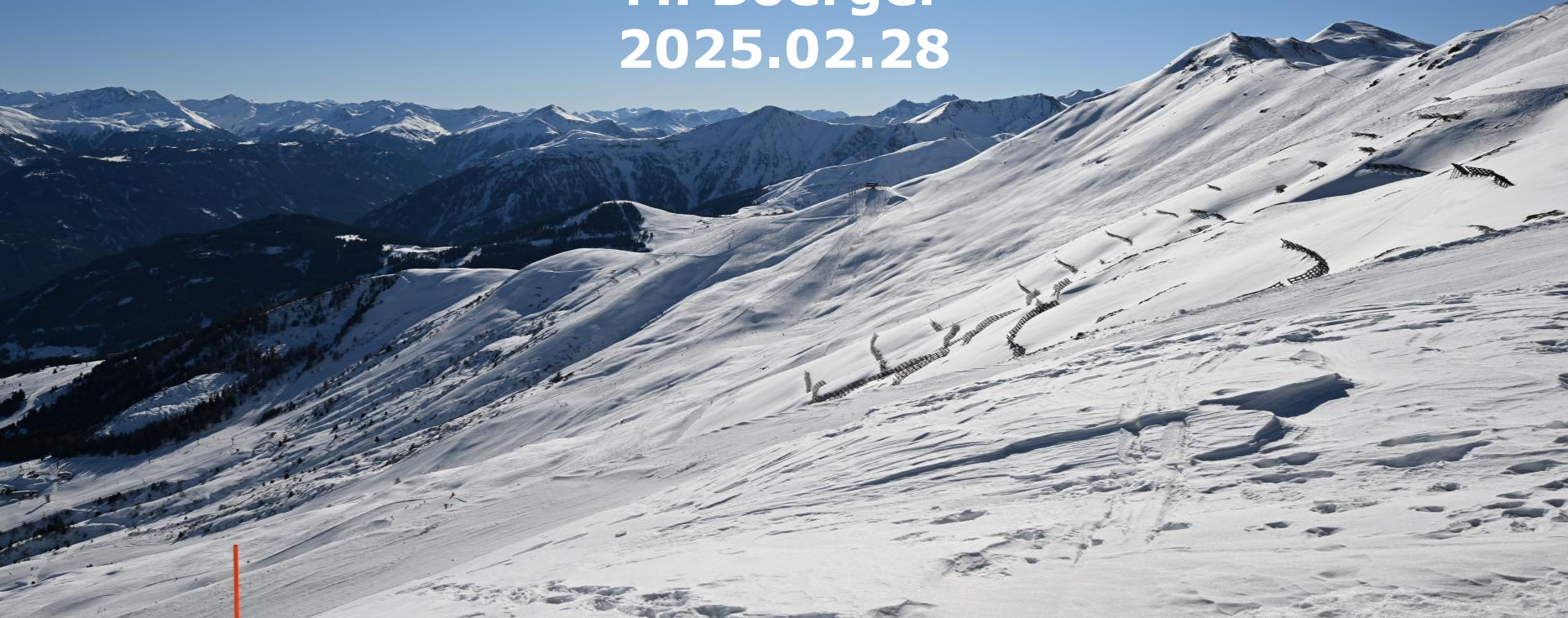


So, C++ is unsafe!

M. Boerger
2025.02.28



So, what?

- Why is C++ unsafe?
- How bad is it.
- How am I affected.
- What else.
- What are the alternatives.
- What now, what next?



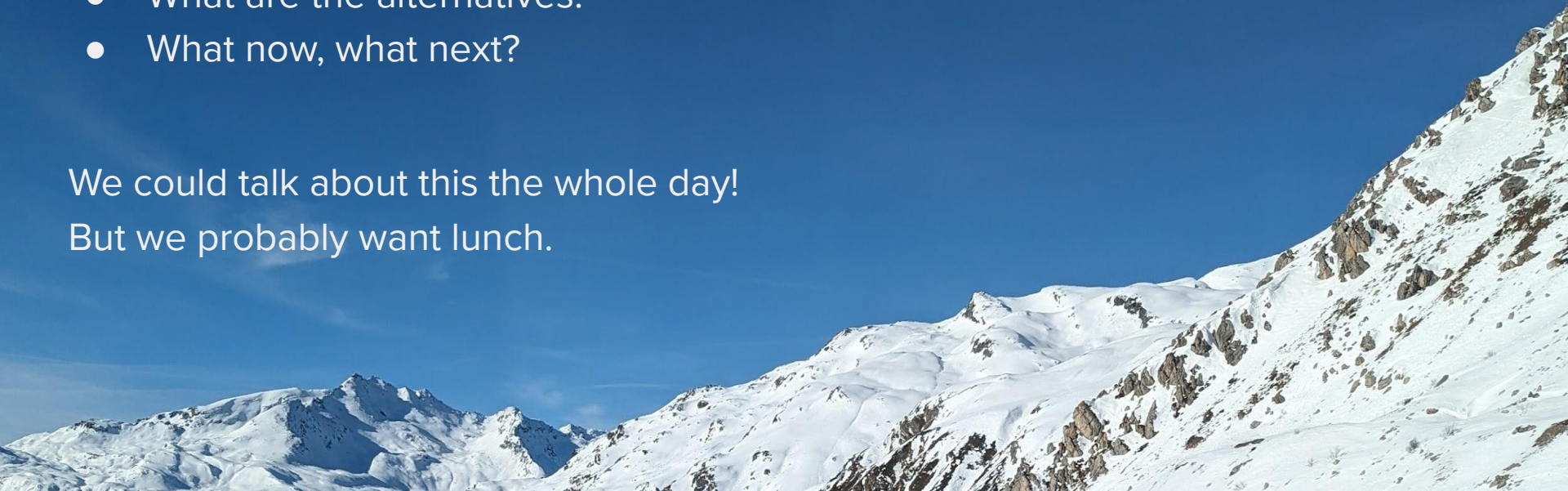
So, what?

- Why is C++ unsafe?
- How bad is it.
- How am I affected.
- What else.
- What are the alternatives.
- What now, what next?

We could talk about this the whole day!
But we probably want lunch.

Ideas not Solutions

- Only TWO C++ code example
- I won't explain all issues
- I won't teach you C++



Wait, what about C vs C++

Technically C and C++ are completely different languages.

They share most of the syntax and some of the semantics.

C++ can bind and execute C.

C has no classes and no memory management.

When code bases grow C developers tend to implement C++ manually.

The bottom line is that C is often orders of magnitudes worse than C++.

Worse, we often (have to) refer to (possibly unsafe) C functions.



Why is C++ unsafe?

C++ allows direct memory access which is what makes it fast.

C++ inherits C which has absolutely no safety whatsoever, no matter what you hear.

C++ does not check memory access (for the most), your program and the CPU do.

C++11 has some memory checking (also C++11 is an inconsistent mess).

C++20 has reasonable memory checking and C++23 makes it a round thing.

C++20 brought new favorites.

C++ no matter the version allows C coding - hence C++ is unsafe

unless...?



An Example

An example

At some point there was `Int2Str`.

And things were good.

```
char* Int2Str(int32_t v) {  
    // Maxint value = 2147483648 = 10 digits!  
    char* buf = malloc(10);  
    snprintf(buf, 10, "%d", v);  
    return buf;  
}
```

An example

At some point there was `Int2Str`.

And things were good.

And much later there were crashes.

```
char* Int2Str(int32_t v) {  
    // Maxint value = 2147483648 = 10 digits!  
    char* buf = malloc(10);  
    snprintf(buf, 10, "%d", v);  
    return buf;  
}
```


An example

At some point there was Int2Str.

And things were good - until they were not.

And much later there were crashes.

And so things were improved.

```
char* Int2Str(int32_t v) {  
    #define MAXINT_LEN = 10  
    // 2147483648 = 10 digits!  
    char* buf = malloc(MAXINT_LEN);  
    size_t len = snprintf(buf, MAXINT_LEN, "%d", v);  
    buf[len] = '\0';  
    return buf;  
}
```

An example

At some point there was Int2Str.

And things were bad.

And much later there were crashes.

And so things were improved.

And intentions were great.

```
char* Int2Str(int32_t v) {  
    #define MAXINT_LEN = 10  
    // 2147483648 = 10 digits!  
    char* buf = malloc(MAXINT_LEN);  
    size_t len = snprintf(buf, MAXINT_LEN, "%d", v);  
    buf[len] = '\0';  
    return buf;  
}
```

An example

At some point there was Int2Str.

And things were bad.

And much later there were crashes.

And so things were improved. Kind of.

And intentions were great.

And there were still crashes.

```
char* Int2Str(int32_t v) {  
    #define MAXINT_LEN = 10  
    // 2147483648 = 10 digits!  
    char* buf = malloc(MAXINT_LEN);  
    size_t len = snprintf(buf, MAXINT_LEN, "%d", v);  
    buf[len] = '\0';  
    return buf;  
}
```

An example

At some point there was Int2Str.

And things were bad.

And much later there were crashes.

And so things were improved. Kind of.

And intentions were great.

And there were still crashes.

And now data corruption started to pop up.

```
char* Int2Str(int32_t v) {  
    #define MAXINT_LEN = 10  
    // 2147483648 = 10 digits!  
    char* buf = malloc(MAXINT_LEN);  
    size_t len = snprintf(buf, MAXINT_LEN, "%d", v);  
    buf[len] = '\0';  
    return buf;  
}
```

An example

At some point there was Int2Str.

And things were bad.

And much later there were crashes.

And so things were improved. Kind of.

And intentions were great.

And there were still crashes.

And now data corruption started to pop up.

And the sheer increase in data volume was blamed.

```
char* Int2Str(int32_t v) {  
    #define MAXINT_LEN = 10  
    // 2147483648 = 10 digits!  
    char* buf = malloc(MAXINT_LEN);  
    size_t len = snprintf(buf, MAXINT_LEN, "%d", v);  
    buf[len] = '\0';  
    return buf;  
}
```


An example

This is roughly version 25 of int to string conversion.

Lots of good intentions:

- Mostly dropped C legacy & no malloc!
- The input is restricted in size.
- Siezeof is used to allocate the buffer
- That way there is space even for the zero char.
- And we are in shiny new C++ - so yeah!

```
std::string Int2Str(int32_t v) {  
    char buf[sizeof("2147483648")]; // Maxint value!  
    snprintf(buf, sizeof(buf), "%d", v);  
    return std::string(buf);  
}
```

An example

This is roughly version 25 of int to string conversion.

Lots of good intentions:

- Mostly dropped C legacy & no malloc!
- The input is restricted in size.
- Siezeof is used to allocate the buffer
- That way there is space even for the zero char.
- And we are in shiny new C++ - so yeah!

The maxint is $2^{31} = 2147483647$ - off by 1 :-)

```
std::string Int2Str(int32_t v) {  
    char buf[sizeof("2147483648")]; // Maxint value!  
    snprintf(buf, sizeof(buf), "%d", v);  
    return std::string(buf);  
}
```

An example

This is roughly version 25 of int to string conversion.

Lots of good intentions:

- Mostly dropped C legacy & no malloc!
- The input is restricted in size.
- Siezeof is used to allocate the buffer
- That way there is space even for the zero char.
- And we are in shiny new C++ - so yeah!

The maxint is $2^{31} = 2147483647$ - off by 1 **TWICE** :-)

The 31 should have indicated there are other values.

Turns out $0x80\ 00\ 00\ 00 = -2147483648$.

```
std::string Int2Str(int32_t v) {  
    char buf[sizeof("2147483648")]; // Maxint value!  
    snprintf(buf, sizeof(buf), "%d", v);  
    return std::string(buf);  
}
```

An example

This is roughly version 25 of int to string conversion.

Lots of good intentions:

- Mostly dropped C legacy & no malloc!
- The input is restricted in size.
- Siezeof is used to allocate the buffer
- That way there is space even for the zero char.
- And we are in shiny new C++ - so yeah!

The maxint is $2^{31} = 2147483647$ - off by 1 TWICE :-)

The 31 should have indicated there are other values.

Turns out $0x80\ 00\ 00\ 00 = -2147483648$.

```
std::string Int2Str(int32_t v) {  
    char buf[sizeof("2147483648")]; // Maxint value!  
    snprintf(buf, sizeof(buf), "%d", v);  
    return std::string(buf);  
}
```

Turns out negative numbers need a character for the '-' sign.

An example

This is roughly version 25 of int to string conversion.

Lots of good intentions:

- Mostly dropped C legacy & no malloc!
- The input is restricted in size.
- Siezeof is used to allocate the buffer
- That way there is space even for the zero char.
- And we are in shiny new C++ - so yeah!

The maxint is $2^{31} = 2147483647$ - off by 1 TWICE :-)

The 31 should have indicated there are other values.

Turns out `0x80 00 00 00` = -2147483648.

```
std::string Int2Str(int32_t v) {  
    char buf[sizeof("2147483648")]; // Maxint value!  
    snprintf(buf, sizeof(buf), "%d", v);  
    return std::string(buf);  
}
```

Turns out negative numbers need a character for the '-' sign.

But that is the least problem here.

An example

This is roughly version 25 of int to string conversion.

Lots of good intentions:

- Mostly dropped C legacy & no malloc!
- The input is restricted in size.
- Siezeof is used to allocate the buffer
- That way there is space even for the zero char.
- And we are in shiny new C++ - so yeah!

The maxint is $2^{31} = 2147483647$ - off by 1 TWICE :-)

The 31 should have indicated there are other values.

Turns out `0x80 00 00 00` = -2147483648.

C99 zero terminates - It is still rubbish.

```
std::string Int2Str(int32_t v) {  
    char buf[sizeof("2147483648")]; // Maxint value!  
    snprintf(buf, sizeof(buf), "%d", v);  
    return std::string(buf);  
}
```

Turns out negative numbers need a character for the '-' sign.

Technically C99 always zero terminates.

But that is the least problem here.

An example

Today's correct solution: Use the provided functions.

Works for all standard types.

Can be extended to any other type.

Integrates well with the rest of the language.

Any drawbacks?

- It is not actually a template.
- Instead it is an ADL thing.

```
namespace std {
```

```
// template <typename T>  
// std::string to_string(T&& v);
```

```
std::string to_string(int);  
std::string to_string( long value );  
std::string to_string( long long value );  
std::string to_string( unsigned value );  
std::string to_string( unsigned long value );  
std::string to_string( unsigned long long value );  
std::string to_string( float value );  
std::string to_string( double value );  
std::string to_string( long double value );  
} // namespace std
```

An example

Today's correct solution: Use the provided functions.

Works for all standard types.

Can be extended to any other type.

Integrates well with the rest of the language.

Any drawbacks?

- It is not actually a template.
- Instead it is an ADL thing.
- If only we had thought of it earlier.
- Thousands separators anyone?

```
namespace std {
```

```
// template <typename T>  
// std::string to_string(T&& v);
```

```
std::string to_string(int);  
std::string to_string( long value );  
std::string to_string( long long value );  
std::string to_string( unsigned value );  
std::string to_string( unsigned long value );  
std::string to_string( unsigned long long value );  
std::string to_string( float value );  
std::string to_string( double value );  
std::string to_string( long double value );  
} // namespace std
```

An example - Necessary?

C/C++ started out long ago.

It was meant to be efficient and became widely used.

Like anything C/C++ had/s bugs and bad decisions.

Understanding the behavior would have prevented it.

```
namespace std {  
  
// template <typename T>  
// std::string to_string(T&& v);  
  
std::string to_string(int);  
std::string to_string( long value );  
std::string to_string( long long value );  
std::string to_string( unsigned value );  
std::string to_string( unsigned long value );  
std::string to_string( unsigned long long value );  
std::string to_string( float value );  
std::string to_string( double value );  
std::string to_string( long double value );  
} // namespace std
```

An example - Necessary?

C/C++ started out long ago.

It was meant to be efficient and became widely used.

Like anything C/C++ had/s bugs and bad decisions.

Understanding the behavior would have prevented it.

Still:

- C99 forces zero termination in snprintf.
- Around 30 *******printf functions.
- None addresses the allocation problem.

```
namespace std {
```

```
// template <typename T>  
// std::string to_string(T&& v);
```

```
std::string to_string(int);  
std::string to_string( long value );  
std::string to_string( long long value );  
std::string to_string( unsigned value );  
std::string to_string( unsigned long value );  
std::string to_string( unsigned long long value );  
std::string to_string( float value );  
std::string to_string( double value );  
std::string to_string( long double value );  
} // namespace std
```


An example - Necessary?


C/C++ started out long ago.

It was meant to be efficient and became widely used.

Like anything C/C++ had/s bugs and bad decisions.

Understanding the behavior would have prevented it.

Still:

- C99 forces zero termination in `sprintf`.
- Around 30 `***printf` functions.
- None addresses the allocation problem.
- 24 years ago I implemented [spprintf](#)  for PHP.

```
namespace std {
```

```
// template <typename T>  
// std::string to_string(T&& v);
```

```
std::string to_string(int);  
std::string to_string( long value );  
std::string to_string( long long value );  
std::string to_string( unsigned value );  
std::string to_string( unsigned long value );  
std::string to_string( unsigned long long value );  
std::string to_string( float value );  
std::string to_string( double value );  
std::string to_string( long double value );  
} // namespace std
```

An example - Necessary?


C/C++ started out long ago.

It was meant to be efficient and became widely used.

Like anything C/C++ had/s bugs and bad decisions.

Understanding the behavior would have prevented it.

Still:

- C99 forces zero termination in `sprintf`.
- Around 30 `***printf` functions.
- None addresses the allocation problem.
- 24 years ago I implemented [spprintf](#)  for PHP.
- This bug is mostly a knowledge & tooling issue.

```
namespace std {
```

```
// template <typename T>
```

```
// std::string to_string(T&& v);
```

```
std::string to_string(int);
```

```
std::string to_string( long value );
```

```
std::string to_string( long long value );
```

```
std::string to_string( unsigned value );
```

```
std::string to_string( unsigned long value );
```

```
std::string to_string( unsigned long long value );
```

```
std::string to_string( float value );
```

```
std::string to_string( double value );
```

```
std::string to_string( long double value );
```

```
} // namespace std
```

Another Example

Another example

C++ does not initialize everything - today!

People think it is important :-/

What does the program on the right return?

```
struct Test {  
    int value;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local;  
    ret += local;  
    return ret;  
}
```

Another example

C++ does not initialize everything - today!

People think it is important :-/

What does the program on the right return?

Nothing - It does not compile :-)

error: variable 'local' is **uninitialized** when used here
[-Werror,-Wuninitialized]

```
struct Test {  
    int value;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local;  
    ret += local;  
    return ret;  
}
```


Another example

C++ does not initialize everything - today!

People think it is important :-/

What does the program on the right return?

```
struct Test {  
    int value;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local = 0;  
    ret += local;  
    return ret;  
}
```

Another example

C++ does not initialize everything - today!

People think it is important :-/

What does the program on the right return?

Now it returns some arbitrary number :-)

```
struct Test {  
    int value;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local = 0;  
    ret += local;  
    return ret;  
}
```

Another example

C++ does not initialize everything - today!

People think it is important :-/

What does the program on the right return?

```
struct Test {  
    int value = 0;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local = 0;  
    ret += local;  
    return ret;  
}
```

Another example

C++ does not initialize everything - today!

People think it is important :-/

What does the program on the right return?

It finally returns zero.

Because **our** configuration initializes the global data.

```
struct Test {  
    int value = 0;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local = 0;  
    ret += local;  
    return ret;  
}
```

Another example - Necessary?

The compiler or linter should always warn about this.

The developer should enable all warnings as errors.

The linter should prevent merging.

But is it really bad?

```
struct Test {  
    int value = 0;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local = 0;  
    ret += local;  
    return ret;  
}
```

Another example - Necessary?

The compiler or linter should always warn about this.

The developer should enable all warnings as errors.

The linter should prevent merging.

But is it really bad?

Yes: the uninitialized value could be a buffer index.

```
struct Test {  
    int value = 0;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local = 0;  
    ret += local;  
    return ret;  
}
```


Another example - Necessary?

The compiler or linter should always warn about this.

The developer should enable all warnings as errors.

The linter should prevent merging.

But is it really bad?

Yes: the uninitialized value could be a buffer index.

Still: This is yet another knowledge and tooling issue.

```
struct Test {  
    int value = 0;  
};  
  
int global_value;  
  
int main() {  
    int ret = 0;  
    Test test;  
    ret += test.value;  
    ret += global_value;  
    int local = 0;  
    ret += local;  
    return ret;  
}
```

Top favorites

Personal favorites

C++

```
std::auto_ptr<T>
```

Personal favorites

C++

C < C99

```
std::auto_ptr<T>
```

```
snprintf()
```

Well, it's gone for good.

Personal favorites

C++

C < C99

C++

```
std::auto_ptr<T>
```

```
snprintf()
```

```
~MyClass() { if (!ptr_) throw Error(); }
```

Well, it's gone for good.

No zero termination guarantee.

Personal favorites

C++

```
std::auto_ptr<T>
```

C < C99

```
snprintf()
```

C++

```
~MyClass() { if (!ptr_) throw Error(); }
```

C++11 in 2013

anything really

Well, it's gone for good.

No zero termination guarantee.

UB!

Personal favorites

C++

```
std::auto_ptr<T>
```

C < C99

```
snprintf()
```

C++

```
~MyClass() { if (!ptr_) throw Error(); }
```

C++11 in 2013

anything really

C++11

```
std::move()
```

Well, it's gone for good.

No zero termination guarantee.

UB!

If only C++11 was ABI compatible.

Personal favorites

C++

`std::auto_ptr<T>`

C < C99

`snprintf()`

C++

`~MyClass() { if (!ptr_) throw Error(); }`

C++11 in 2013

anything really

C++11

`std::move()`

C++20

`std::range`

Well, it's gone for good.

No zero termination guarantee.

UB!

If only C++11 was ABI compatible.

Using moved-out data!

Personal favorites

C++

`std::auto_ptr<T>`

C < C99

`snprintf()`

C++

`~MyClass() { if (!ptr_) throw Error(); }`

C++11 in 2013

anything really

C++11

`std::move()`

C++20

`std::range`

Well, it's gone for good.

No zero termination guarantee.

UB!

If only C++11 was ABI compatible.

Using moved-out data!

Possibly unsafe in a range based for loop.

How bad is it?

How bad is it?

In theory any program with uncontrolled memory access allows to do anything.

Example:

- Get a program to receive large **strange data**.

- Get the program to violate its memory model and execute that data.

- Organize the data so that if executed it will give you system access.

- Determine the OS.

- Rely on OS issues and repeat until you have admin access.



How bad is it?

Roughly 1 TRILLION lines of C/C++ code exist.

C/C++ is pretty much used for everything.



How am I affected?

All relevant operating systems are C/C++ based.

That includes:

- All Unix derivatives - which means Linux.

- Communication with the OS through simple C structs, files and MMAP.

- All Windows versions.

- Communication with the OS handled via versioned structures.



How am I affected?

Almost all embedded systems are C/C++.

Almost all higher level languages are developed in/with C++ (Java, PHP, Python, ...)

Most daily tools are written in C/C++ but:

- newer languages are overtaking in popularity.
- and we trust (sadly) in new stuff that everyone is using.
- we rely on there being enough security researchers finding issues.
- we hope they find issues faster than bad actors.



Just the other day

High-Severity OpenSSL Vulnerability Found by Apple Allows MitM Attacks


OpenSSL has patched CVE-2024-12797, a high-severity vulnerability found by Apple that can allow man-in-the-middle attacks.



Just the other day

High-Severity OpenSSL Vulnerability Found by Apple Allows MitM Attacks

OpenSSL has patched CVE-2024-12797, a high-severity vulnerability found by Apple that can allow man-in-the-middle attacks.

 **snyk** | SECURITY

▼ Red Hat

7.4 HIGH

<u>Attack Vector (AV)</u>	Network	<u>Scope (S)</u>	Unchanged	<u>Confidentiality (C)</u>	High
<u>Attack Complexity (AC)</u>	High			<u>Integrity (I)</u>	High
<u>Privileges Required (PR)</u>	None			<u>Availability (A)</u>	None
<u>User Interaction (UI)</u>	None				

Just the other day

What to do?

- Apply rigorous security checks! Continuously! Always!
- Ensure your systems can and will be updated.
- Updates can cause issues in themselves.
- Review and limit your dependencies.



What are the alternatives?

Python?

Go?

TypeScript?

Rust?

Java?

C++29?



What are the alternatives?

C++29?

- Why replace a language if you can improve it.
- There are roughly 1 Trillion lines of C/C++ code (give or take).
- Memory safety and other issues are being addressed (over time) constantly.
- Presumably with C++29 we will have profiles that prevent memory safety.



What are the alternatives?

C++29, C++32, C++35, C++38, C++41?

- Why replace a language if you can improve it.
- There are roughly 1 Trillion lines of C/C++ code (give or take).
- Memory safety and other issues are being addressed (over time) constantly.
- Presumably with C++29 we will have profiles that prevent memory safety.
- You can use linting and other tooling to accomplish much the same **today**.
- C++ will still support unsafe C functions.
- C++ code will still interface with unsafe libraries, tools and the OS.



What are the alternatives?

C++29, C++32, C++35, C++38, C++41?

- Why replace a language if you can improve it.
- There are roughly 1 Trillion lines of C/C++ code (give or take).
- Memory safety and other issues are being addressed (over time) constantly.
- Presumably with C++29 we will have profiles that prevent memory safety.
- You can use linting and other tooling to accomplish much the same today.
- C++ will still support unsafe C functions.
- C++ code will still interface with unsafe libraries, tools and the OS.
- Thousands of pages of standard.
- I am personally learning C++ for 30+ years!
- Insane complexity.



What are the alternatives?

C++29, C++32, C++35, C++38, C++41?

- Why replace a language if you can improve it.
- There are roughly 1 Trillion lines of C/C++ code (give or take).
- Memory safety and other issues are being addressed (over time) constantly.
- Presumably with C++29 we will have profiles that prevent memory safety.
- You can use linting and other tooling to accomplish much the same today.
- C++ will still support unsafe C functions.
- C++ code will still interface with unsafe libraries, tools and the OS.
- Thousands of pages of standard.
- I am personally learning C++ for 30+ years!
- Insane complexity (C++23: 2124p).



What are the alternatives?

C++29, C++32, C++35, C++38, C++41?

- Why replace a language if you can improve it.
- There are roughly 1 Trillion lines of C/C++ code (give or take).
- Memory safety and other issues are being addressed (over time) constantly.
- Presumably with C++29 we will have profiles that prevent memory safety.
- You can use linting and other tooling to accomplish much the same today.
- C++ will still support unsafe C functions.
- C++ code will still interface with unsafe libraries, tools and the OS.
- Thousands of pages of standard.
- I am personally learning C++ for 30+ years!
- Insane complexity (C++23: 2124p) -> Use the available tools - all of them!



What now, what next?

Still using C/C++?

- Drop C as fast as you can and avoid it like the holy water!
 - It does not scale.
 - It is fundamentally insecure.



What now, what next?

Still using C/C++?

- Drop C as fast as you can and avoid it like the holy water!
 - It does not scale.
 - It is fundamentally insecure.
- If you still use C++, then:
 - Apply modern coding practices and style guides.
 - You almost certainly do not test enough by a long shot.
 - Ensure positive and constructive fast reviews that aim at learning/understanding.
 - Use modern C++, so for now that is C++20 or maybe already C++23.
 - Enable all warnings as errors: -Wall -Wextra -Wpedantic -Werror.
 - Use linters, e.g. clang-tidy.
 - Use ASan (and MSan if you can).
 - Use TSan, UBSan, etc..
 - Use static analyzers and other security tools!



Does memory safety matter?

Given easy enough protection, memory safety is (insanely?) overrated?

More problematic and completely independent of the language:

- Leaking passwords and other secrets
- Unprotected system access & Backdoors
- SQL Injection
- CSRF / XSRF, D/DOS, MitM,
- Bad open source or internal actors: creating any security issue
- Spoofing / Phishing
- Services upgraded into incompatibility
- Many other widespread attack patterns



Does memory safety matter?

Given easy enough protection, memory safety is (insanely?) overrated?

More problematic and completely independent of the language:

- Leaking passwords and other secrets
- Unprotected system access & Backdoors
- SQL Injection
- CSRF / XSRF, D/DOS, MitM,
- Bad open source or internal actors: creating any security issue
- Spoofing / Phishing
- Services upgraded into incompatibility
- Many other widespread attack patterns

Above all:
There is no security in
obscurity!



What else?

Many years back Memory safety solutions were exotic:

- layered isolation,
- running code in a sandbox,
- static analysis.
- And many many more mechanisms, tools and techniques for added safety.

Today everyone uses (or could use) the above.

Using containers has become the norm in larger setups.

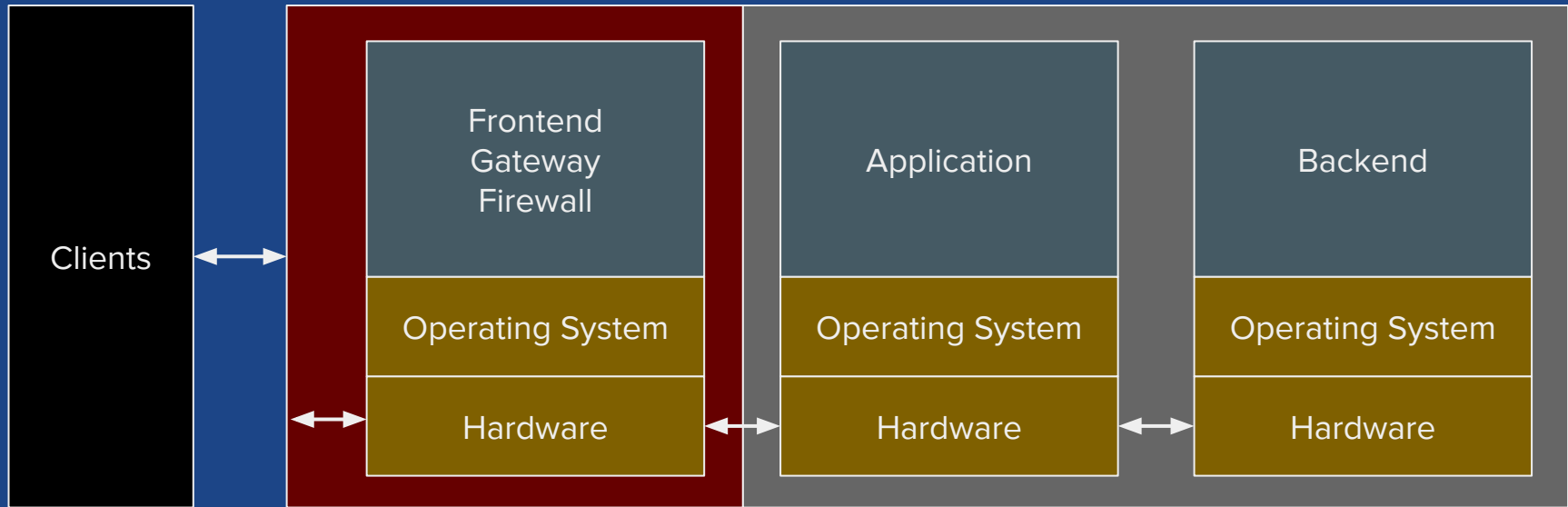


**Two common
and one
underrated
Practices and
Techniques**

Layered Isolation

Clients can only talk to certain resources.

Isolation deals with security!?



Layered Isolation - Good/Bad?

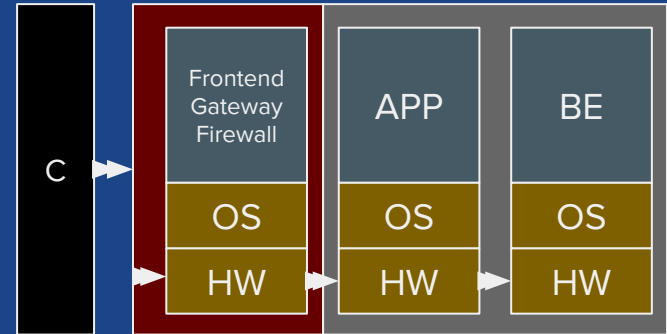
Layered Isolation only gets you so far.

Do you understand your whole setup down to the hardware?

Can the clients truly only reach your frontends?

Bad data can still make it downstream.

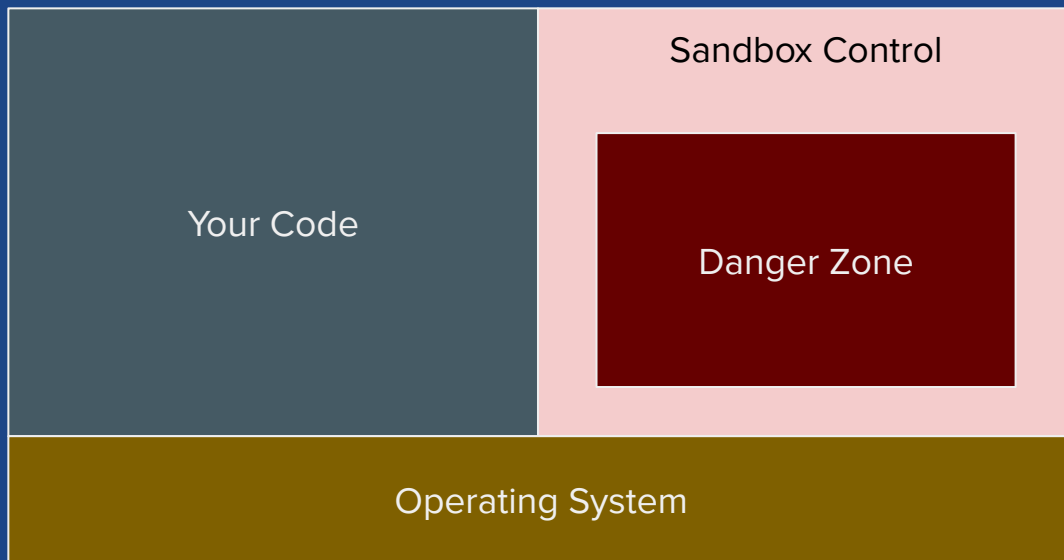
How do you reach the Apps and the Backends?



Sandbox

Your code can talk to the OS, other libraries and the sandbox control.

The Sandboxed code is isolated to the degree that is configured.



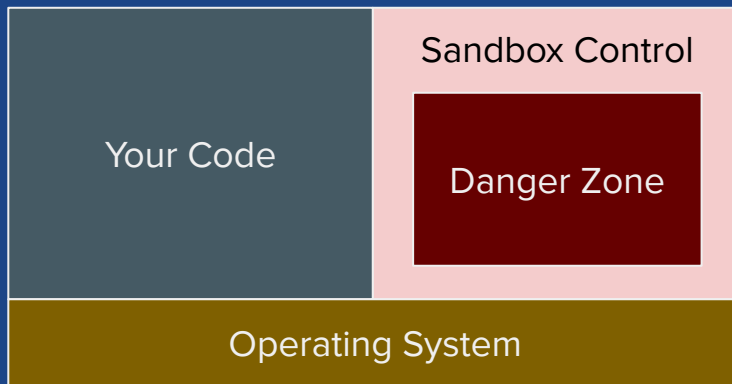
Sandbox

Your code can talk to the OS, other libraries and the sandbox control.

The Sandboxed code is isolated to the degree that is configured.

Example:

- github.com/google/sandboxed-api
- SAPI implements an RPC interface.
- SAPI interjects all OS calls.
- Use the RPC interface as a replacement.



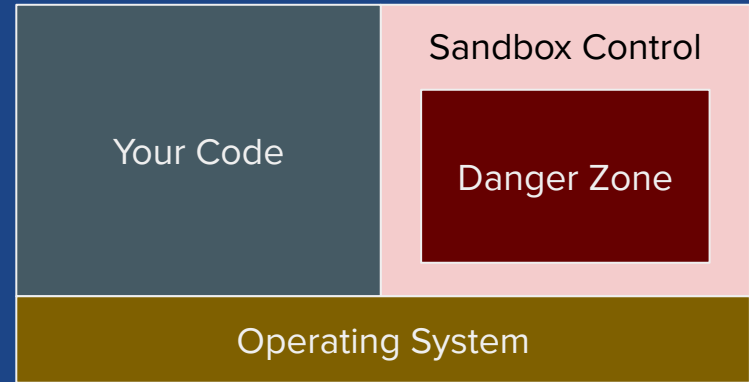
Sandbox - Good/Bad?

Fairly secure and simple.

Until it is no longer simple.

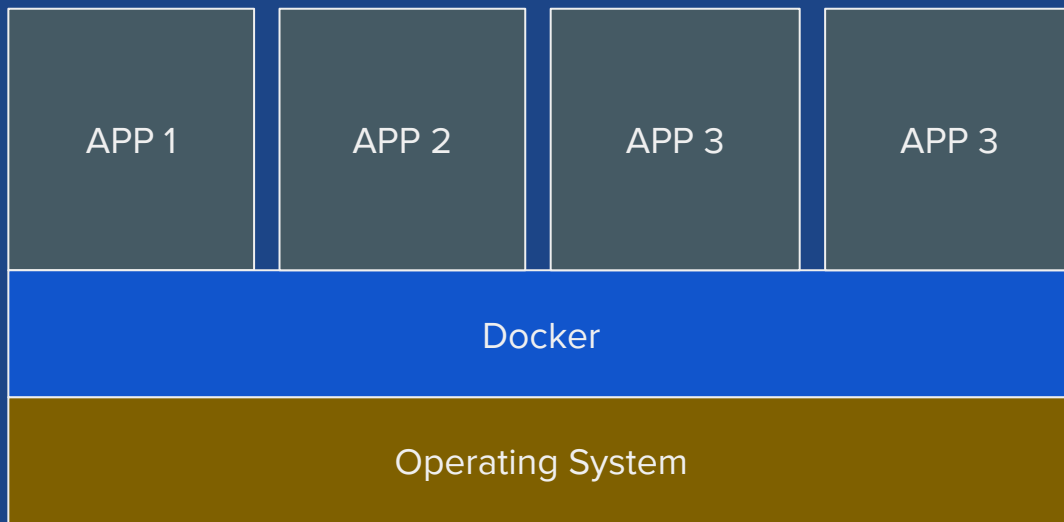
Too easy to open up things that should not.

Bad data can still be processed.



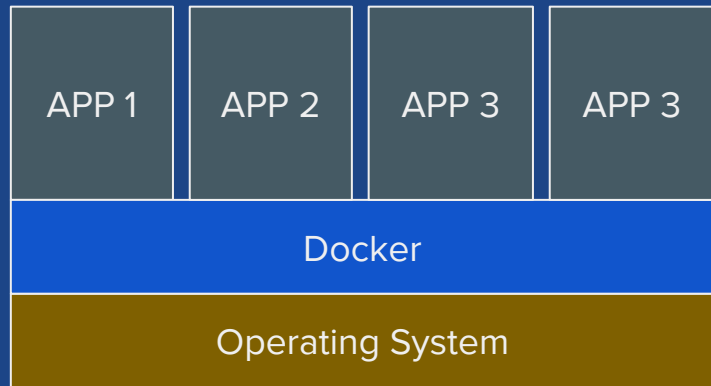
Containers

- You build your code and pack it as an archive of sorts.
- You combine it with an everything you need.
- The container execution is an OS level sandbox!



Containers - Bad?

- Bad data can still reach your apps.
- Docker often too simplistic.
- K8S has very high complexity.
- You can still perform low level access?



Finishing up

Take away

C++ can (mostly) be used in a (memory) safe manner.

C++ will become memory safe - thanks to the wakeup call from the White House.

Memory safety is likely (or at least should be) the least of your problems.

So should you use C++?

It's complicated.

For many: Probably not.

No matter the language: **Test, Analyze & Stop Debugging.**





Thanks

Questions?

Test, Analyze & Stop Debugging!