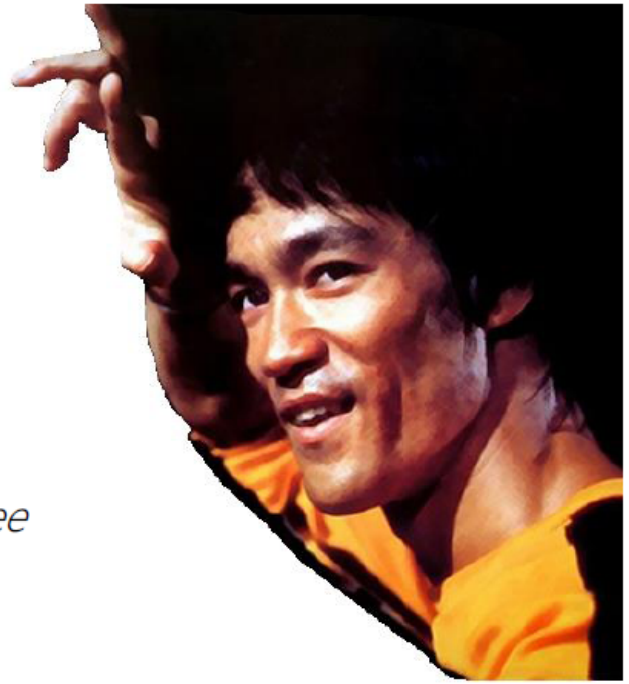
Bruce Lee for C++ programmers

Marco Arena

The best fighter is not a Boxer, Karate or Judo man. The best fighter is someone who can adapt on any style.

Bruce Lee



Secret #4: It's independent of the paradigm

Research your own experience.

Absorb what is useful.

Reject what is useless.

Add what is specifically your own.

Marlowe – 1969

The best fighter can adapt on any style

The best fighter is not a boxer, karate or judo man. The best fighter is someone who can adapt on any style.

There should only be tools to use as effectively as possible. The highest art is no art. The best form is no-form.

The best fighter can adapt on any style

C++ cannot be expressed as a single *style*.

C++ supports many alternative paradigms and tools.

In C++ we do mix *by design*.

Mixing styles and idioms is by design

Procedural

Generic

Functional
(sort of ☺)

```
void rotate_and_draw(vector<Shape*>& vs, int r)
{
    for_each(vs.begin(), vs.end(), [](Shape* p) {
        p->rotate(r);
    });

    for (Shape* p : vs)
        p->draw();
}
```

Object-Oriented

Example from C++20 – ranges

```
auto up = accumulate(zip_with(greater<>{}, tail(low), close), 0);
auto down = accumulate(zip_with(less<>{}, tail(high), close), 0);
cout << up << " " << down;
```

```
up=sum(map(lambda (a,b): a>b, zip(low[1:], close)))
down=sum(map(lambda (a,b): a<b, zip(high[1:], close)))
print("%d %d" % (up, down))
```

Wait!
That's Python ☺

I tried to implement the STL in other languages and failed.

C++ was the only language in which I could do it.

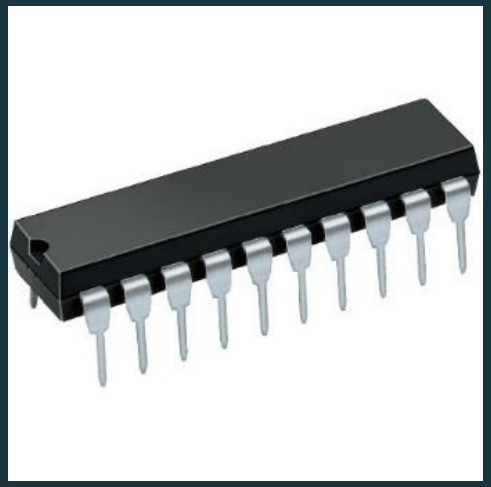
Alexander Stephanov

The best fighter can adapt on any style

The way of the Dragon – 1972

https://youtu.be/YsHKE4LR77Y?t=193

Adapting to new and unique scenarios



Adapting to C++ evolution

Suppose we decide not to use *new* C++ features.

What if our company will do?

What if we change team or boss?

What if we want to change job?

Adapting to C++ evolution – *Vigilant Approach*

Know about new feature / give a try / study

Be ready to use them

Know how to learn them in deep

Evaluate if they can give you some *value*

Adapting to C++ evolution – *Vigilant Approach*

```
struct Aggregate
{
        Aggregate() = delete;
};

Aggregate a{}; // Ok in C+17 :O
```

Adapting to C++ evolution – *Vigilant Approach*

```
struct Aggregate
{
        Aggregate() = delete;
};

Aggregate a{}; // won’t compile in C++20
```

Adapting to C++ evolution – *Vigilant Approach*

```
void MightThrow() throw()
{
    // ...
}
```

Adapting to C++ evolution – *Vigilant Approach*

```
void MightThrow() throw()
{                 // won’t compile in C++20
    // ...
}
```

Guidelines might not work forever

Examples:

Do not use STL algorithms, they are hard to customize

Use auto_ptr to handle dynamic allocations

Guidelines might not work forever

Examples:

~~*Do not use STL algorithms, they are hard to customize*~~

Consider lambdas to generate in-place callable objects

~~*Use auto_ptr to handle dynamic allocations*~~

Consider unique_ptr instead of auto_ptr

Empty your cup

Empty your cup

The way of the Dragon – 1972

ITA: https://youtu.be/E59E0koivmY?t=24

ENG: https://youtu.be/Hsqw9r8aqo0?t=1388

In C++, we are constantly exposed to alternatives,

options, trade-offs.

We are even exposed to new things and changes,

when a new standard is officialized.

We should see them as *opportunities*, with *opennes*.

Example – *Error handling*

How many ways we can handle errors in C++?

- Exceptions
- Error codes/flags
- Observers/callbacks
- Globals
- …

C++17: *std::optional*

```
std::optional<int> try_parse_int(const std::string& s)
{
    //try to parse an int from the given string,
    //and return "nothing" if you fail
}

// optional arguments
std::vector<std::pair<std::string, double>> search(
    std::string query,
    std::optional<int> max_count,
    std::optional<double> min_match_score);
```

C++17: *std::optional*

```
auto maybeInt = try_parse_int("10");
cout << *maybeInt; // 10

if (auto notInt = try_parse_int("abc"); maybeInt) {
        // ...
}
else
{
        // ...
}
```

C++17: *std::optional*

```
auto Process(const string& input) {
    auto opt1 = Func1(input);
    if (opt1) {
        auto opt2 = Func2(*opt2);
        if (opt2) {
                return Func3(*opt2);
        }
    }
    return std::nullopt;
}
```

Boilerplate…

Composition with *std::optional*

```
auto Process(const string& input) {
    return Func1(input) ||
           Func2 ||
           Func3;
}
```

```
template<typename T, typename F>
auto operator||(std::optional<T> opt, F f)
{
        return opt ? f(opt.value()) : std::nullopt;
}
```

Composition with *std::optional*

```
optional<UrlInfo> ClickShortUrl(const string& url)
{
        return GetShortUrl(url)
            || IfNotExpired
            || IfNotPrivate
            || Click;
}
```

Who failed?

Second try: *expected*

```
expected<UrlInfo, ErrorType> ClickShortUrl(const string& url)
{
        return GetShortUrl(url)
            || IfNotExpired
            || IfNotPrivate
            || Click;
}

template<typename T, typename F>
auto operator||(expected<T, ErrorType> ex, F f)
{
        return ex ? f(ex.value()) : ex.error();
}
```

Other possible problems:

- return values can be ignored (exceptions cannot)
- composition is *by hand*
- every function is explicitly polluted with *ADTs*

Again, let's empty our cup:

What other languages do?

Conversations with other *masters*

Swift

C++ Master: *How do you encapsulate errors?*

Swift Master: *We use exceptions.*

C++ Master: *Are you happy with them?*

Swift Master: Well, let me show you some code…

Swift

```
func mightThrow() throws -> String

func cannotThrow() -> String


// call
result = try mightThrow();
result2 = cannotThrow();
```

Rust

C++ Master: *How do you encapsulate errors?*

Rust Master: *We use something like **expected**.*

C++ Master: what about the boilerplate?

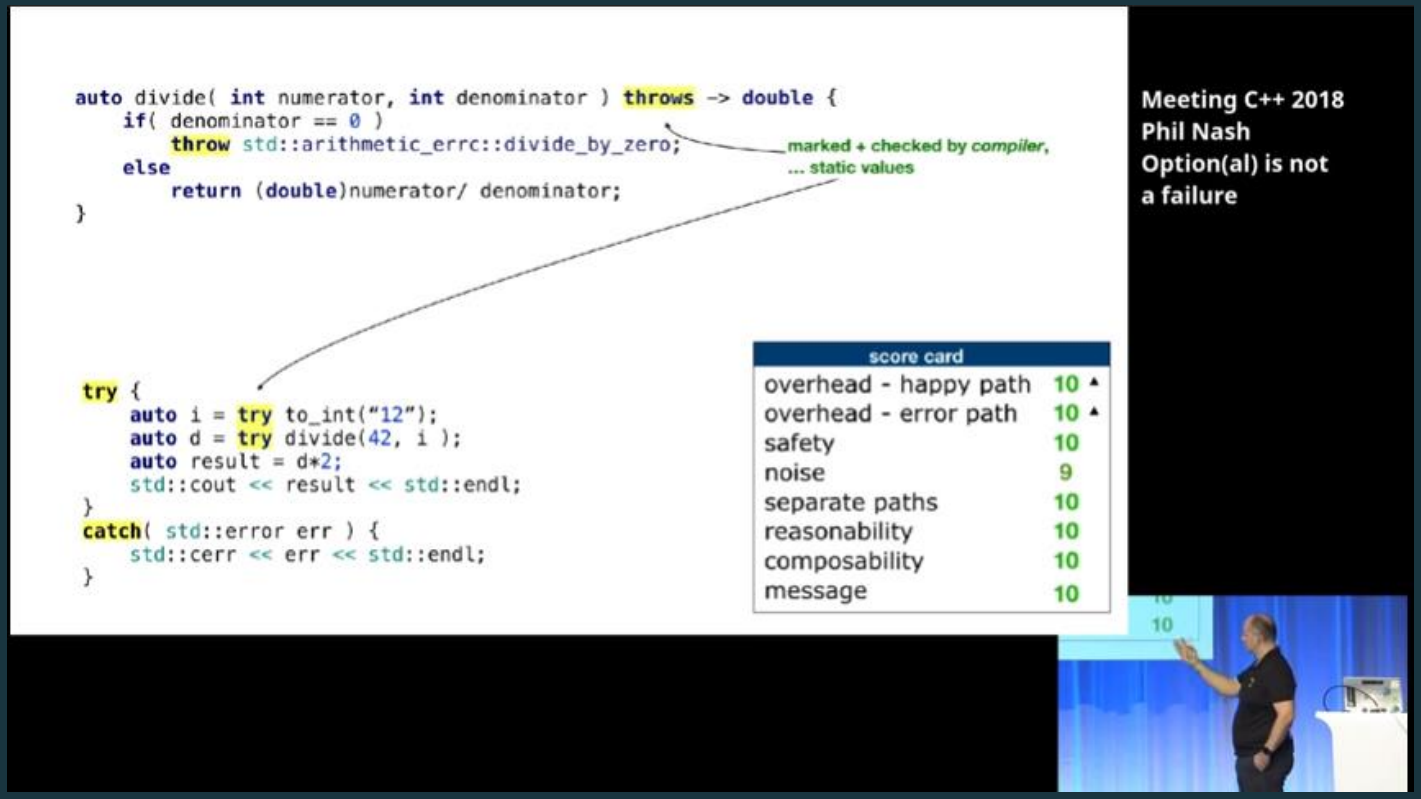
Rust Master: Well, let me show you some code…

Rust

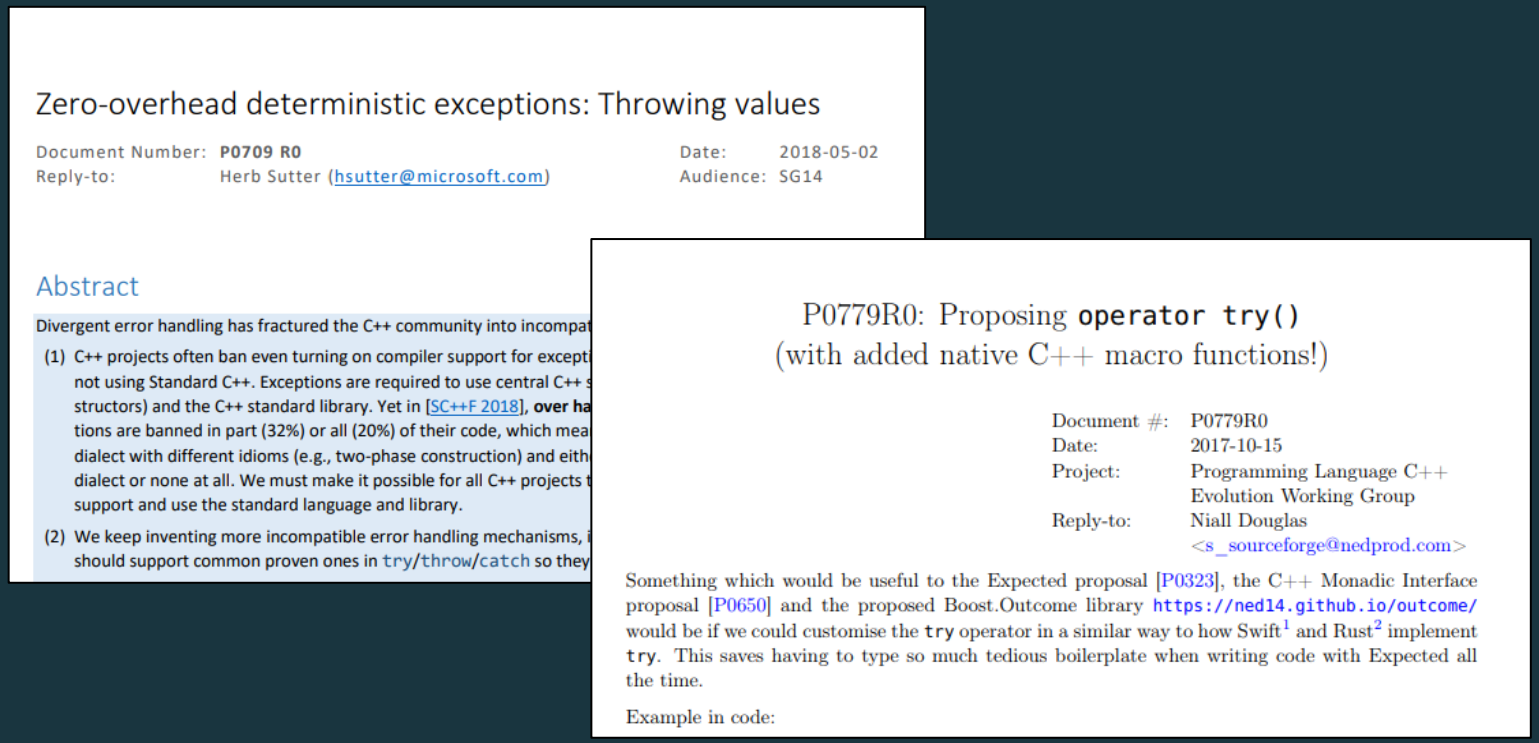
```
result = foo();
if (!result)
  return result.error();
// result.value()


result = foo()?; // early return or continue
result2 = foo2()?; // early return or continue
```

A near future?



A near future?



Zero-overhead deterministic exceptions: Throwing values

Document Number: P0709 R0 Date: 2018-05-02
Reply-to: Herb Sutter (hsutter@microsoft.com) Audience: SG14

Abstract

Divergent error handling has fractured the C++ community into incompat

(1) C++ projects often ban even turning on compiler support for excepti not using Standard C++. Exceptions are required to use central C++ s structors) and the C++ standard library. Yet in [SC++F 2018], **over ha** tions are banned in part (32%) or all (20%) of their code, which mea dialect with different idioms (e.g., two-phase construction) and eith dialect or none at all. We must make it possible for all C++ projects t support and use the standard language and library.

(2) We keep inventing more incompatible error handling mechanisms, i should support common proven ones in `try`/`throw`/`catch` so they

P0779R0: Proposing `operator try()`
(with added native C++ macro functions!)

Document #: P0779R0
Date: 2017-10-15
Project: Programming Language C++ Evolution Working Group
Reply-to: Niall Douglas <s_sourceforge@nedprod.com>

Something which would be useful to the Expected proposal [P0323], the C++ Monadic Interface proposal [P0650] and the proposed Boost.Outcome library `https://ned14.github.io/outcome/` would be if we could customise the `try` operator in a similar way to how Swift[1] and Rust[2] implement `try`. This saves having to type so much tedious boilerplate when writing code with Expected all the time.

Example in code:

Hack away the unessential

It is not daily increase but daily decrease, hack away the unessential.

True refinement seeks simplicity.

C++ Lifetime Patterns

```
{ // automatic lifetime
      Foo foo; // constructor
      // ... exceptions possible ...
} // foo destructed and deallocated here


// dynamic lifetime
int* arr = new int[10]{}; // dynamic buffer
// in case of exceptions... :(
delete [] arr;
```

C++ Lifetime Patterns

```
// Example of classical RAII wrapper

struct Handler
{
        Handler(resource* res) : m_res(res){}

        ~Handler() { delete m_res; }

        void Use()
        {
            // use m_res...
        }

private:
    resource* m_res;
};
```

Rule of Zero – Example

Classes that have custom destructors, copy/move constructors or copy/move assignment operators should **deal exclusively with ownership**. Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.

(application of the *Single Responsibility Principle*)

Rule of Zero – Example

```
struct ResourceWrapper
{
        Handler(std::unique_ptr<resource> res) : m_res(std::move(res)){}

        void Use()
        {
            // use m_res...
        }

private:
    std::unique_ptr<resource> m_res;
};
```

Rule of Zero – Some tools

- **smart pointers** – general-purpose resource managers
- **containers** – data structures
- **scope guards** – anonymous destructors
- *your own* wrapper

Pointers headache

```
void Func(Foo* p);

// Is p an owner?
// can be p be null?
// p is one or more instances?
// ...
```

One syntax,

several semantics

Pointers headache – Alternatives

```
void Func(unique_ptr<Foo> p);
void Func(another_ptr<Foo> p);
void Func(owner<Foo> p); *

// Owners

(*)     template<typename T>
        using owner = T*;
```

Pointers headache – Alternatives

```
void Func(Foo& p);
void Func(reference_wrapper<Foo> p);
void Func(not_null<Foo> p);

// Non-nullable references
```

Pointers headache – Alternatives

```
void Func(std::span<Foo> seq);
void Func(Foo* seq, int len);
void Func(Foo* seq, size_t len);
void Func(const array<Foo, len>& seq);
void Func(const vector<Foo>& seq);

// Sequences
```

Pointers headache – Alternatives

```
void Func(Foo* nullableReference);

// nullable-references
```

Unpractical complexity

A recent example from a famous C++ blog:

Implementing Default Parameters That Depend on Other Parameters in C++

Unpractical complexity

```
void f(double x, double y, DefaultedF<double, GetDefaultAmount> z)
{

}
```

```
template<typename T, typename GetDefaultValue>
class DefaultedF
{
public:
    DefaultedF(T const& t) : value_(t){}
    DefaultedF(DefaultValue) : value_(std::nullopt) {}

// ...

private:
   std::optional<T> value_;
};
```

```
template<typename... Args>
T get_or_default(Args&&... args)
{
    if (value_)
    {
        return *value_;
    }
    else
    {
        return GetDefaultValue::get(std::forward<Args>(args)...);
    }
}
```

Unpractical complexity

```
void f(double x, double y, double z)
{
    //...
}

void f(double x, double y)
{
    f(x, y, x+y);
}
```

Adding Enables Removing – Kate Gregory

New standard = opportunities to ditch custom code

Requires a *vigilant* and *responsible* approach

Adding Enables Removing – Example C++98

```
std::vector<Customer> c = ...;

struct NameAndSurnameMatcher
{
   NameAndSurnameMatcher(const string& name, const string& surname)
           : m_name(name), m_surname(surname)  { }

   bool operator()(const Customer& c) const {
           return c.Name == m_name && c.Surname == m_surname;
   }
private:
   string m_name, m_surname;
};

std::vector<Customer>::iterator it = std::find_if(c.begin(), c.end(),

NameAndSurnameMatcher(name, surname));
```

Adding Enables Removing – Example C++11

```
std::vector<Customer> c = ...;

struct NameAndSurnameMatcher
{
   NameAndSurnameMatcher(const string& name, const string& surname)
           : m_name(name), m_surname(surname)  { }

   bool operator()(const Customer& c) const {
           return c.Name == m_name && c.Surname == m_surname;
   }
private:
   string m_name, m_surname;
};

auto it = std::find_if(c.begin(), c.end(),
                            [&](const Customer& c) {
                                        return c.Name == name && c.Surname == surname;
                            });
```

At Google, we do not use exceptions

[…]

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project.

[…]

https://google.github.io/styleguide/cppguide.html#Exceptions

Examples of what we can *hack away*:

- *responsibility* from classes
- *utility code* from business code
- multiple *semantics* from types
- *custom* code in favour of standard code
- *generalizations/complexity* when not strictly needed
- *features*, if they do not contribute expressing *your own* C++
- *…many more…*

A punch is just a punch

Before I studied the art, a punch was just a punch, a kick was just a kick. After I learned the art, a punch was no longer a punch, a kick was no longer a kick. Now that I've understood the art, a punch is just a punch, a kick is just a kick.

The three stages of cultivation

Pupil: *Master, what is a string?*

Master: *Just a sequence of characters.*

Pupil: *And what about const char*, std::string, CString, QString, System::String?*

Master: *What is a string, after all?*

C++17: `std::string_view`

A string is just a sequence of characters.

C++17: `std::string_view`

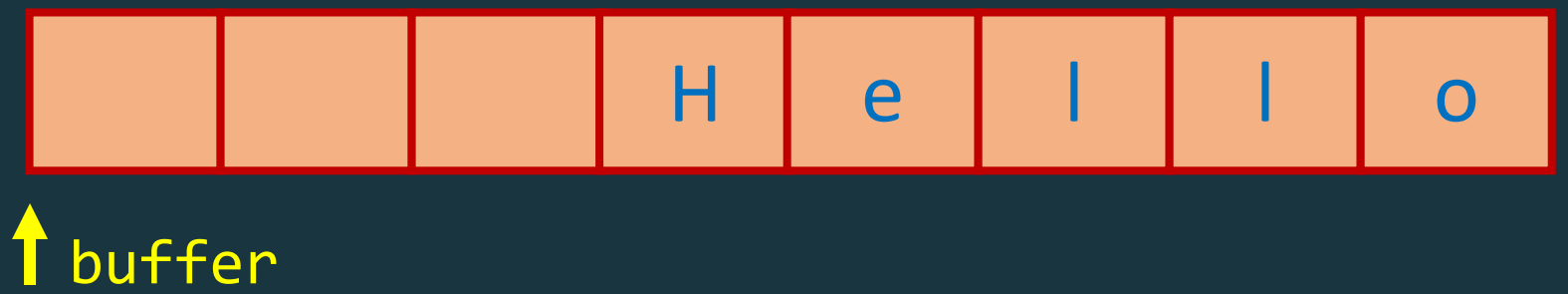
Internally, it's just like:

```
const char* buffer; // immutable
size_t length;
```

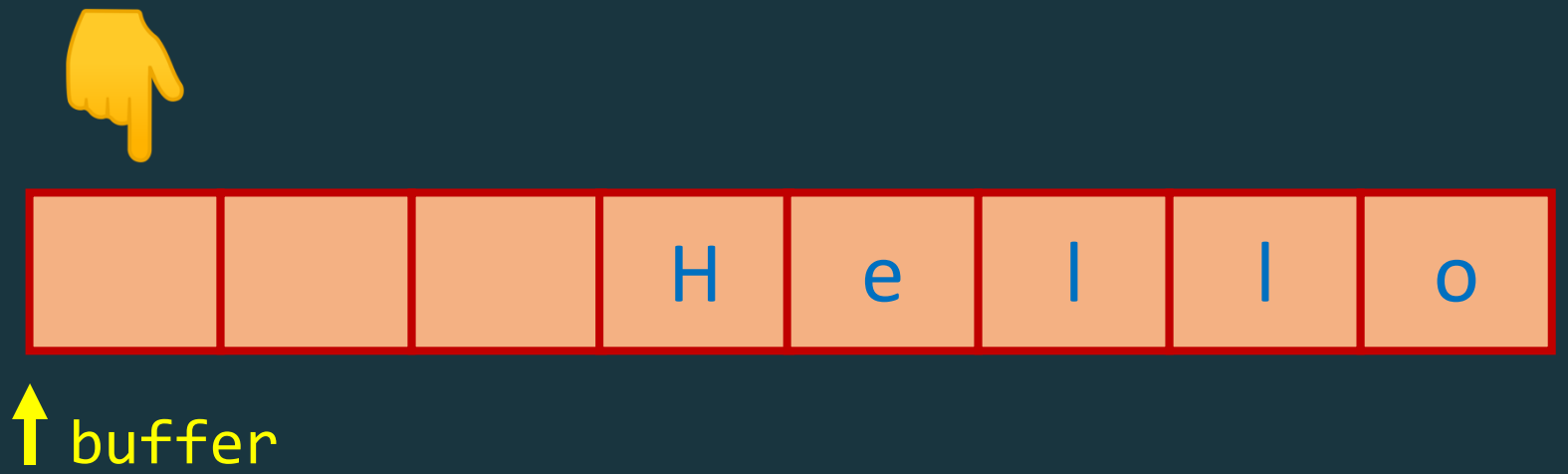
Copying is just as cheap as copying two 64bit ints

(on 64bit applications).

`std::string_view` – Example

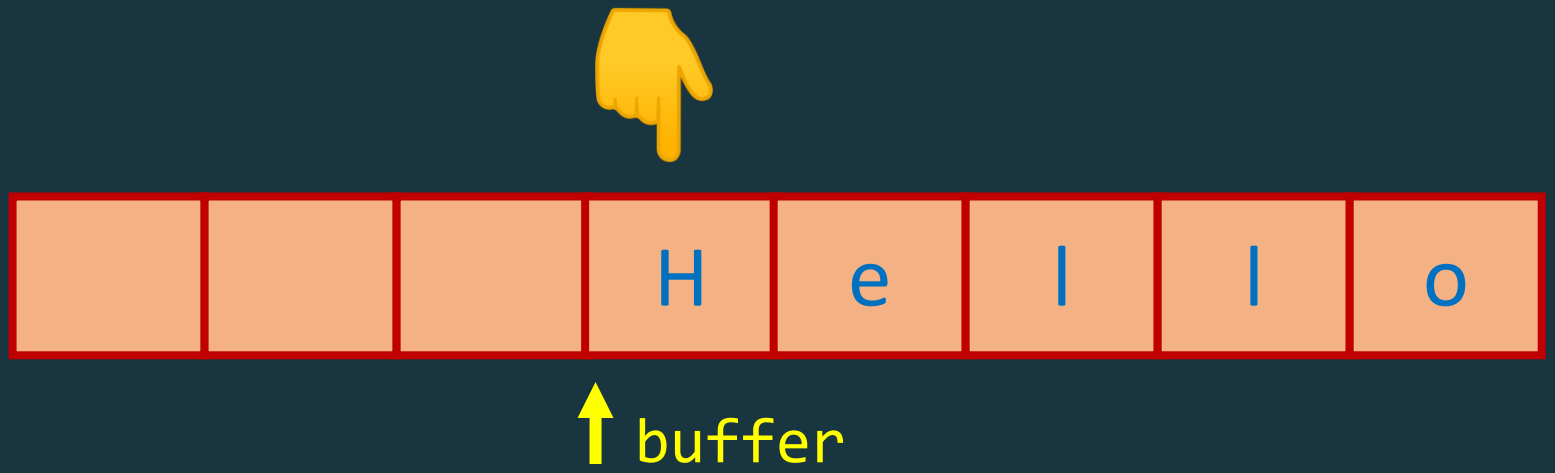


std::string_view – Example



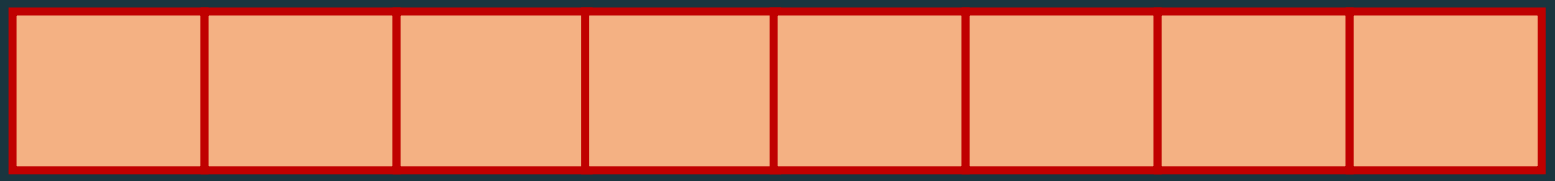

```
sv.find_first_not_of(" ")
```

std::string_view – Example




```
sv.remove_prefix(    sv.find_first_not_of(" ")          );
```

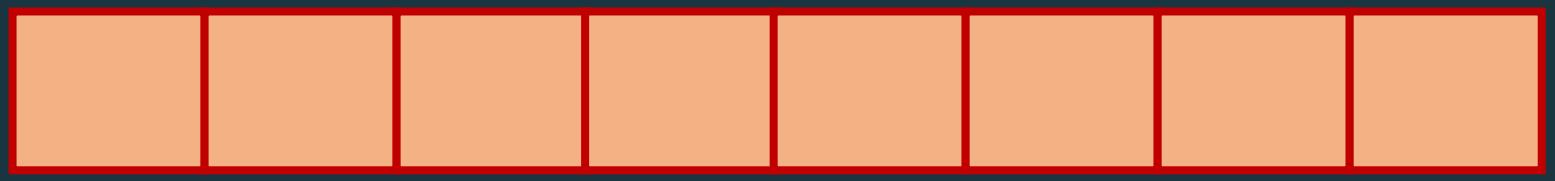
std::string_view – Example



buffer

```
sv.remove_prefix(    sv.find_first_not_of(" ")          );
```

`std::string_view` – Example



```
sv.remove_prefix(min(sv.find_first_not_of(" "), sv.size()));
```

std::string_view – Example

```
string_view trim_left(string_view str)
{
        sv.remove_prefix(
                std::min(sv.find_first_not_of(" "), sv.size()));
}
```

`std::string_view` – One type to rule them all

```
void businessCode(const char* str);
void businessCode(const string& str);
void businessCode(const QString& str);
void businessCode(const CString& str);
//...

void businessCode(std::string_view str);
```

`std::string_view` – One type to rule them all

```
string_view sv = cStr; // const char* (null-terminated)

string_view sv {cStr, len}; // const char* (general)

string_view sv = stdStr; // std::string

string_view sv = qStr.toLocal8Bit().constData(); // QString

string_view sv = atlCString.GetString(); // CString
```

std::string_view – Warning!

```
void businessCode(std::string_view str)
{
        // are you sure BusinessImpl does not expect \0 at the end?
        ExternalLibrary::BusinessImpl(str.data());
}
```

Adding string_view into an existing codebase is not always the right answer: changing parameters to pass by string_view can be inefficient if those are then passed to a function requiring a string or a NUL-terminated const char*. It is best to adopt string_view starting at the utility code and working upward, or with complete consistency when starting a new project.

https://abseil.io/tips/1

A punch is just a punch

Some scenarios `string_view` does not fit in:

- need to guarantee the sequence is null-terminated

- need to modify the sequence

- need to handle the memory of the sequence

C++20: `std::span`

```
void Func(std::span<Foo> seq);
```

It’s basically "high level systems programming"

How to use Span<T> and Memory<T>

Antão Almada Follow

Mar 12, 2018 · 6 min read

*(Updated to **.NET Core 2.1** official release version)*

Other examples of a *punch is just a punch* in C++:

- iterators

A punch is just a punch

Other examples of a *punch is just a punch* in C++:

- iterators

- ranges

Other examples of a *punch is just a punch* in C++:

- iterators

- ranges

- tuples

std::tuple as "structured data lingua franca"

```
struct Foo
{
        std::string m_name;
        int m_age;
};
```

```
struct Bar
{
        std::string m_name;
        int m_age;
};
```

```
std::tuple<string, int>
```

Research your own experience

Research your own experience.

Absorb what is useful.

Reject what is useless.

Add what is specifically your own.

Vigilant Approach

Adaptability

Responsibility

Openness

Research your own experience

- *In C++, one size does not fit all* – ***by design***
- Mixing styles and idioms is normal
- *You* create *your* own **C++**
- *You* may create *your* own guidelines

Absorb what is useful

- Don’t reinvent the wheel
- We have very good and mature idioms
- Use the standard as much as possible
- Consider the Ecosystem

Reject what is useless

- Using every standard feature is optional!
- Ban features, if needed
- C++ is very complex. Keep it as simple as possible

Add what is specifically your own

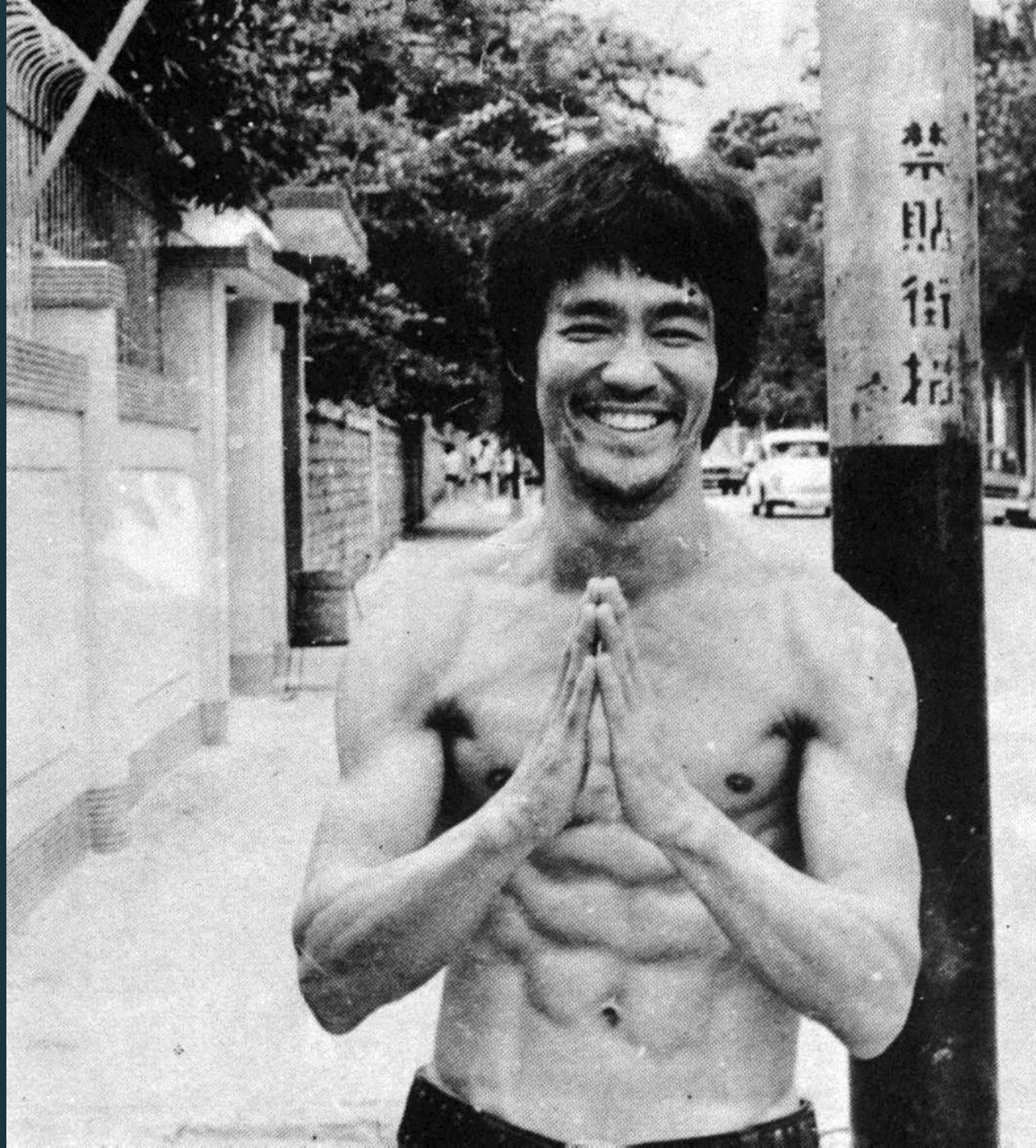
- *You* will have unique needs
- Exploit the C++ flexibility, when needed

Be flexible, responsible and open

- What works now might not work forever
- Guidelines should evolve
- Consider new things as opportunities – be vigilant!

Enter the Dragon – 1973

Thank you!



Bruce Lee tickled your curiosity?

- *Artist of Life – edited by John Little*
- *Striking Thoughts – edited by John Little*
- *The Warrior Within – written by John Little*
- *Bruce Lee Podcast at* brucelee.com/podcast-blog