2023

The New Library on the Block

A Strong Library Foundation for Your next Project

Jonathan Müller & Arno Schödl

C++ now

Context

Core library for think-cell.

- < 30 full-time developers
- 20-year-old monorepo
- continuously updated to latest standards
- able to experiment and refactor

fEKxcYeqh

github.com/think-cell/think-cell-library

- Extend/complement/improve on the standard library
- Not monolithic: pick only the parts you like
- No guaranteed backward compatibility

think-cell

Improving C++ as a language

C: A single cast operator

Truncate float to int.

```
(int)3.14;
```

C: A single cast operator

Truncate float to int.

```
(int)3.14;
```

Interpret object as bytes!

```
(char*)&global;
```

C: A single cast operator

Truncate float to int.

```
(int)3.14;
```

Interpret object as bytes!

```
(char*)&global;
```

Pointer to int!?

```
(short)&global;
```

think-cell

C: A single cast operator

Truncate float to int.

```
(int)3.14;
```

Interpret object as bytes!

```
(char*)&global;
```

Pointer to int!?

```
(short)&global;
```

Remove `const`?!

```
(int*)&const_global;
```

think-cell

C++ Dedicated cast operators

Truncate float to int.

```
static_cast<int>(3.14);
```

Interpret object as bytes!

```
reinterpret_cast<char*>(&global);
```

Pointer to int!?

```
static_cast<short>(reinterpret_cast<std::uintptr_t>(&global));
```

Remove `const`?!

```
const_cast<int*>(&const_global);
```

think-cell

C++: A single static_cast operator (1)

Truncate or wrap arithmetic types:

```
static_cast<int>(3.14f);
static_cast<short>(12345678901234567890);
static_cast<float>(3.14);
```

C++: A single `static_cast` operator (1)

Truncate or wrap arithmetic types:

```
static_cast<int>(3.14f);
static_cast<short>(12345678901234567890);
static_cast<float>(3.14);
```

Call constructor:

```
static_cast<std::string>("Hello World!");
```

Call conversion operator:

```
static_cast<std::string_view>(my_string);
```

C++: A single `static_cast` operator (2)

Character types to/from other arithmetic type (including floats!):

```
static_cast<int>('a');
static_cast<char>(65);
```

```
static_cast<float>('a');
static_cast<char>(3.14);
```

C++: A single `static_cast` operator (2)

Character types to/from other arithmetic type (including floats!):

```
static_cast<int>('a');
static_cast<char>(65);
```

```
static_cast<float>('a');
static_cast<char>(3.14);
```

`enum` to/from arithmetic type (including floats!):

```
static_cast<int>(my_enum);
static_cast<my_enum>(42);
```

```
static_cast<float>(my_enum);
static_cast<float>(3.14);
```

think-cell

C++: A single `static_cast` operator (2)

Character types to/from other arithmetic type (including floats!):

```
static_cast<int>('a');
static_cast<char>(65);
```

```
static_cast<float>('a');
static_cast<char>(3.14);
```

`enum` to/from arithmetic type (including floats!):

```
static_cast<int>(my_enum);
static_cast<my_enum>(42);
```

```
static_cast<float>(my_enum);
static_cast<float>(3.14);
```

(Unchecked!) Downcast/upcast in class hierarchy:

```
static_cast<Derived&>(base);
```

```
static_cast<Base&>(derived);
```

think-cell

C++: A single static_cast operator (3)

void* to/from T*:

```
static_cast<T*>(malloc(42));
```

```
static_cast<void*>(ptr);
```

C++: A single static_cast operator (3)

void* to/from T*:

```
static_cast<T*>(malloc(42));
```

```
static_cast<void*>(ptr);
```

Move values:

```
static_cast<T&&>(obj);
```

think-cell

C++: A single `static_cast` operator (3)

`void*` to/from `T*`:

```
static_cast<T*>(malloc(42));
```

```
static_cast<void*>(ptr);
```

Move values:

```
static_cast<T&&>(obj);
```

Discard a value:

```
static_cast<void>(nodiscard_function());
```

think-cell

think-cell: `tc::explicit_cast`

```
tc::explicit_cast<T>(value);
```

- convert classes if conversion is safe
- convert between actual numbers with debug check against loss
- convert between characters with debug check against loss
- convert nullable types to `bool`

And that’s it.

think-cell

`tc::explicit_cast` class conversions

`tc::explicit_cast<ClassT>(args...)` can call:

- user-defined constructor
- user-defined conversion operator
- aggregate initialization
- user-defined customization point

tc::explicit_cast class conversions

`tc::explicit_cast<ClassT>(args...)` can call:

- user-defined constructor
- user-defined conversion operator
- aggregate initialization
- user-defined customization point

```
std::ranges::to<std::vector<std::string>>(rng)
```

think-cell

`tc::explicit_cast` class conversions

`tc::explicit_cast<ClassT>(args...)` can call:

- user-defined constructor
- user-defined conversion operator
- aggregate initialization
- user-defined customization point

```
tc::explicit_cast<std::vector<std::string>>(rng)
```

think-cell

`tc::explicit_cast` safe class conversions

`tc::explicit_cast` class conversions are safe:

- no slicing
- no dangling spans
- no reference to temporary

tc::explicit_cast safe class conversions

`tc::explicit_cast` class conversions are safe:

- no slicing
- no dangling spans
- no reference to temporary

```
template <typename Source, typename Target>
concept safely_convertible_to;

template <typename Target, typename ... Args>
concept safely_constructible_from;
```

Traits to mark conversions as unsafe.

think-cell

think-cell: Dedicated casts

`tc::explicit_cast` is the cast to pick by default.

think-cell: Dedicated casts

`tc::explicit_cast` is the cast to pick by default.

- `tc::as_unsigned`/`tc::as_signed`: signed <-> unsigned

think-cell

think-cell: Dedicated casts

`tc::explicit_cast` is the cast to pick by default.

- `tc::as_unsigned`/`tc::as_signed`: signed <-> unsigned
- `tc::to_underlying`/`tc::from_underlying`: enum <-> integer

think-cell

think-cell: Dedicated casts

`tc::explicit_cast` is the cast to pick by default.

- `tc::as_unsigned`/`tc::as_signed`: signed <-> unsigned
- `tc::to_underlying`/`tc::from_underlying`: enum <-> integer
- `tc::base_cast`/`tc::derived_cast`: casts in class hierarchy

think-cell: Dedicated casts

`tc::explicit_cast` is the cast to pick by default.

- `tc::as_unsigned`/`tc::as_signed`: signed <-> unsigned
- `tc::to_underlying`/`tc::from_underlying`: enum <-> integer
- `tc::base_cast`/`tc::derived_cast`: casts in class hierarchy
- `tc::void_cast`: `void*` -> `T*`

think-cell: Dedicated casts

`tc::explicit_cast` is the cast to pick by default.

- `tc::as_unsigned`/`tc::as_signed`: signed <-> unsigned
- `tc::to_underlying`/`tc::from_underlying`: enum <-> integer
- `tc::base_cast`/`tc::derived_cast`: casts in class hierarchy
- `tc::void_cast`: `void* -> T*`
- `tc::discard`: discard a value

think-cell

Aside: What is a number?

C++:

- `signed char`, `short`, `int`, `long`, `long long`
- `unsigned` version of the above
- `char`, `char8_t`, `char16_t`, `char32_t`
- `bool`

think-cell

Aside: What is a number?

C++:

- `signed char`, `short`, `int`, `long`, `long long`
- `unsigned` version of the above
- `char`, `char8_t`, `char16_t`, `char32_t`
- `bool`

think-cell:

- `tc::char_type`: `char`, `char8_t`, `char16_t`, `char32_t`
- `tc::actual_integer`: `std::integral` without `tc::char_type` and without `bool`
- `tc::actual_arithmetic`: `tc::actual_integer` and `std::floating_point`

Aside: What is an enum?

- Actual `enum` and `enum class`.

Aside: What is an enum?

- Actual `enum` and `enum class`.
- `bool` with underlying type `unsigned char`

Aside: What is an enum?

- Actual `enum` and `enum class`.
- `bool` with underlying type `unsigned char`
- `char` with underlying type `unsigned char`
- `char8_t` with underlying type `std::uint8_t`
- `char16_t` with underlying type `std::uint16_t`
- `char32_t` with underlying type `std::uint32_t`

Casts in return

```
SomeVeryVeryLongAndAnnoyingType foo() {
  …
  if (…)
      return tc::explicit_cast<SomeVeryVeryLongAndAnnoyingType>(…);
  …
  return tc::explicit_cast<SomeVeryVeryLongAndAnnoyingType>(…);
}
```

think-cell

Casts in return

```
SomeVeryVeryLongAndAnnoyingType foo() {
  …
  if (…)
      return tc_return_cast(…);
  …
  return tc_return_cast(…);
}
```

Implementing tc_return_cast: First attempt

34f7Y1P53

```cpp
template <typename Source>
struct return_cast_impl {
    Source source;

    template <typename T>
    operator T() {
        return T(source);
    }
};

template <typename Source>
auto return_cast(Source source) {
    return return_cast_impl<Source>{source};
}
```

think-cell

Implementing tc_return_cast: First attempt

```
char foo() {
    return return_cast(3.14);
}
```

Implementing tc_return_cast: First attempt

```
char foo() {
    return return_cast(3.14);
}
```

```
auto foo() {
    return return_cast(3.14);
}
```

think-cell

Implementing `tc_return_cast`: Handling `auto`

oTzKcbf9f

Idea: prevent returning from function

think-cell

Implementing tc_return_cast: Handling auto

oTzKcbf9f

Idea: prevent returning from function

```
template <typename Source>
struct return_cast_impl {
    …

    return_cast_impl(return_cast_impl const&) = delete;
    return_cast_impl& operator=(return_cast_impl const&) = delete;

    …
};
```

```
auto foo() {
    return return_cast(3.14);
}
```

think-cell

Implementing `tc_return_cast`: Preventing copy-elision

na3hT9Maj

Idea: create an xvalue, not a prvalue

Implementing tc_return_cast: Preventing copy-elision

na3hT9Maj

Idea: create an xvalue, not a prvalue

```
template <typename Source>
struct return_cast_impl {
    …

    return_cast_impl const&& operator+() const {
        return static_cast<return_cast_impl const&&>(*this);
    }

    …
};
```

```
auto foo() {
    return +return_cast(3.14);
}
```

Implementing tc_return_cast: A macro

```
#define tc_return_cast +tc::return_cast_detail::return_cast
```

```
auto foo() {
    return tc_return_cast(3.14);
}
```

Aside: Macros

Macros are evil, right?

Aside: Macros

Macros are evil, right?

> *Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer.*

Bjarne Stroustrup, The C++ programming language

Aside: Macros

Macros are evil, right?

> *Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer.*

Bjarne Stroustrup, The C++ programming language

This is a flaw in the programming language.

Aside: Function-like macros

```
#define tc_return_cast +tc::return_cast_detail::return_cast
```

NOT

```
#define tc_return_cast(...) +tc::return_cast_detail::return_cast(__VA_ARGS__)
```

Aside: Function-like macros

```
#define tc_return_cast +tc::return_cast_detail::return_cast
```

NOT

```
#define tc_return_cast(...) +tc::return_cast_detail::return_cast(__VA_ARGS__)
```

```
auto foo() {
    return tc_return_cast(some_long(expression,
                split, over,
                mulitple, lines));
}
```

std::move

```
template <typename T>
std::remove_reference_t<T>&& move(T&& t)
{
    return static_cast<std::remove_reference_t<T>&&>(t);
}
```

std::move

```
template <typename T>
std::remove_reference_t<T>&& move(T&& t)
{
    return static_cast<std::remove_reference_t<T>&&>(t);
}
```

Problem 1:

```
template <typename T>
void foo(T const obj) {
    sink(std::move(obj));
}
```

std::move

```
template <typename T>
std::remove_reference_t<T>&& move(T&& t)
{
    return static_cast<std::remove_reference_t<T>&&>(t);
}
```

Problem 2:

```
template <typename T>
void foo(T& obj) {
    sink(std::move(obj));
}
```

think-cell

think-cell: tc_move

tc_move(expr): move but assert non-const, not lvalue-reference

think-cell: `tc_move`

`tc_move(expr)`: move but assert non-const, not lvalue-reference

- Steal from lvalue reference: `tc_move_always(expr)`
- Keep as lvalue reference: `tc_move_if_owned(expr)`

think-cell

Temporary lifetime extension

```
std::vector<std::string> get_strings();
```

think-cell

Temporary lifetime extension

```
std::vector<std::string> get_strings();
```

```
auto const& strs = get_strings();
```

Temporary lifetime extension

```
std::vector<std::string> get_strings();
```

```
auto const& strs = get_strings();
```

```
auto const& str  = get_strings()[0];
```

think-cell

Temporary lifetime extension requires temporaries

Lifetime of a temporary object can be extended when bound to a reference.

```
auto const& strs = get_strings();
```

Temporary lifetime extension requires temporaries

Lifetime of a temporary object can be extended when bound to a reference.

```
auto const& strs = get_strings();
```

```
T const& std::vector<T>::operator[](std::size_t idx) const;

auto const& str = get_strings()[0];
```

Temporary lifetime extension lies about the type

```
decltype(auto) foo() {
    auto const& strs = get_strings();

    …

    return strs;
}
```

`decltype(auto)` is `auto const&`, which dangles!

think-cell

Guideline

Do not use temporary lifetime extension.

Guideline

Do not use temporary lifetime extension.

Always use `auto const`?

Do not use temporary lifetime extension.

Always use `auto const`?

```
std::vector<std::string> const& get_strings_from_somewhere_else();
```

```
auto const strs = get_strings_from_somewhere_else();
```

think-cell: `tc_auto_cref`

The idea:

- lvalue-reference: `auto const&`
- value, rvalue-reference: `auto const`

think-cell

think-cell: `tc_auto_cref`

The idea:

- lvalue-reference: `auto const&`
- value, rvalue-reference: `auto const`

```
tc_auto_cref(strs, get_strings());
```

Value, so `auto const`

think-cell: `tc_auto_cref`

The idea:

- lvalue-reference: `auto const&`
- value, rvalue-reference: `auto const`

```
tc_auto_cref(strs, get_strings());
```

Value, so `auto const`

```
tc_auto_cref(strs, get_strings_from_somewhere_else());
```

Lvalue-reference, so `auto const&`.

think-cell

think-cell: `tc_auto_cref`

The idea:

- lvalue-reference: `auto const&`
- value, rvalue-reference: `auto const`

```
tc_auto_cref(strs, get_strings());
```

Value, so `auto const`

```
tc_auto_cref(strs, get_strings_from_somewhere_else());
```

Lvalue-reference, so `auto const&`.

```
tc_auto_cref(str, get_strings()[0]);
```

Lvalue-reference, so `auto const&`?!!

think-cell

Return the correct type

```
template <typename T>
T const& std::vector<T>::operator[](std::size_t idx) const&;

template <typename T>
T&& std::vector<T>::operator[](std::size_t idx) &&;
```

think-cell

Function pointers aren't that great

Function pointers aren't that great

```
void foo(int i);
void foo(std::string const& str);
auto ptr = &foo;
```

think-cell

Function pointers aren't that great

```
void foo(int i);
void foo(std::string const& str);
auto ptr = &foo;
```

```
std::all_of(begin, end, &std::islower);
```

Function pointers aren't that great

```cpp
void foo(int i);
void foo(std::string const& str);
auto ptr = &foo;
```

```cpp
std::all_of(begin, end, &std::islower);
```

```cpp
bool my_less(const foo& lhs, const foo& rhs);
std::map map(begin, end, &my_less);
```

think-cell: Function pointers are banned

```
#define tc_fn(...) \
  [](auto&&... args) noexcept -> decltype(...) { \
      return __VA_ARGS__(tc_move_if_owned(args)...); \
  }
```

think-cell

think-cell: Function pointers are banned

```
#define tc_fn(...) \
  [](auto&&... args) noexcept -> decltype(...) { \
      return __VA_ARGS__(tc_move_if_owned(args)...); \
  }
```

```
void foo(int i);
void foo(std::string const& str);
auto ptr = tc_fn(foo);
```

```
std::all_of(begin, end, tc_fn(std::islower));
```

```
bool my_less(const foo& lhs, const foo& rhs);
std::map map(begin, end, tc_fn(my_less));
```

think-cell

Member pointers aren't that great

```
auto fn_ptr = &foo;
fn_ptr(obj, arg);
```

```
auto mem_ptr = &Type::foo;
(mem_ptr.*obj)(arg);
```

think-cell: tc_member/tc_mem_fn

```
#define tc_member(...) \
  [](auto&& obj) -> decltype(...) { \
      return tc_move_if_owned(obj)__VA_ARGS__;  \
  }

#define tc_mem_fn(...) \
  [](auto&& obj, auto&&... args) -> decltype(...) { \
      return tc_move_if_owned(obj)__VA_ARGS__(tc_move_if_owned(args)...);  \
  }
```

```
auto get_size = tc_mem_fn(.size);

get_size(std::string("hello"));
get_size(std::vector{1, 2, 3});
```

think-cell: `tc::invoke` as generalized call

Similar to `std::invoke`, but:

- `static_assert`'s against function pointers
- `static_assert`'s against member pointers
- automatic expansion of tuple-like objects

think-cell: `tc::invoke` as generalized call

Similar to `std::invoke`, but:

- `static_assert`'s against function pointers
- `static_assert`'s against member pointers
- automatic expansion of tuple-like objects

```
auto add = [](auto lhs, auto rhs) { return lhs + rhs; };

tc::invoke(add, 1, 2);

tc::invoke(add, std::pair(1, 2));
tc::invoke(add, std::tuple(1, 2));
tc::invoke(add, std::array{1, 2});
```

think-cell

think-cell: `tc::invoke` as generalized call

Similar to `std::invoke`, but:

- `static_assert`'s against function pointers
- `static_assert`'s against member pointers
- automatic expansion of tuple-like objects

```
auto add = [](auto lhs, auto rhs) { return lhs + rhs; };

tc::invoke(add, 1, 2);

tc::invoke(add, std::pair(1, 2));
tc::invoke(add, std::tuple(1, 2));
tc::invoke(add, std::array{1, 2});
```

```
tc::for_each(tc::zip(rng1, rng2), [](int a, int b) { … });
```

Small utilities for fluent code

We love std::exchange

```
template <typename Var, typename Value>
T exchange(Var& var, Value&& value);
```

```
template <typename T>
class my_smart_ptr {
    T* _ptr;

public:
    my_smart_ptr(my_smart_ptr&& other)  noexcept
    : _ptr(other._ptr)
    {
        other._ptr = nullptr;
    }
};
```

We love std::exchange

```
template <typename Var, typename Value>
T exchange(Var& var, Value&& value);
```

```
template <typename T>
class my_smart_ptr {
    T* _ptr;

public:
    my_smart_ptr(my_smart_ptr&& other)  noexcept
    : _ptr(std::exchange(other._ptr, nullptr))
    {}
};
```

think-cell

`tc::change`: Update a value if different

```cpp
void tc::optional<T>::reset() {
    if (_has_value) {
        _has_value = false;
        value().~T();
    }
}
```

think-cell

tc::change: Update a value if different

```
void tc::optional<T>::reset() {
    if (tc::change(_has_value, false)) {
        value().~T();
    }
}
```

Aside: Reentrance vs exception-safety

```
void foo() {
    …
    if (dirty) {
        dirty = false;
        clean();
    }
    …
}
```

- Problematic if `clean()` throws.
- Save if `clean()` calls `foo()` again.

Aside: Reentrance vs exception-safety

```
void foo() {
    …
    if (dirty) {
        clean();
        dirty = false;
    }
    …
}
```

- Save if `clean()` throws.
- Problematic if `clean()` calls `foo()` again.

think-cell

Aside: Reentrance vs exception-safety

```
void foo() {
    …
    if (tc::change(dirty, false)) {
        try {
            clean();
        } catch (...) {
            dirty = true;
            throw;
        }
    }
    …
}
```

- Save if `clean()` throws.
- Save if `clean()` calls `foo()` again.

think-cell

tc::assign_better

```
template <typename Better, typename Var, typename Value>
bool assign_better(Better better, Var&& var, Value&& value)
{
    if (better(value, var)) {
        std::forward<Var>(var) = std::forward<Value>(value);
        return true;
    } else {
        return false;
    }
}
```

- `tc::change`: `better` is `value != var`
- `tc::assign_max`: `better` is `value > var`
- `tc::assign_min`: `better` is `value < var`

think-cell

Actions and transformations

Transformation:

```
T transformation(T const& obj);
```

Action:

```
void action(T& obj);
```

think-cell

Actions and transformations

Transformation:

```
T transformation(T const& obj);
```

```
void modify(auto& obj, auto f)
{
    obj = f(obj);
}
```

Action:

```
void action(T& obj);
```

```
auto modified(auto&& obj, auto f)
{
  if constexpr (is_lvalue_or_const) {
      auto copy = obj;
      f(copy);
      return copy;
  } else {
      f(obj);
      return tc_move(obj);
  }
}
```

Guideline: Define only actions

```cpp
template <typename Container>
void sort(Container& container); // okay

template <typename Container>
Container sorted(Container const& container); // no
```

Guideline: Define only actions

```cpp
template <typename Container>
void sort(Container& container); // okay

template <typename Container>
Container sorted(Container const& container); // no
```

```cpp
iterator& iterator::operator++(); // okay

iterator next(iterator iter); // no
```

think-cell

think-cell: tc_modified

```
#define tc_modified(obj, ...)  \
  modified(obj, [&](auto& _) -> void { __VA_ARGS__; })
```

```
auto sorted = tc_modified(container, sort(_));
```

```
auto next = tc_modified(iter, ++_);
```

think-cell

think-cell: tc_lazy

```
auto opt_result = compute_result();
auto result     = opt_result ? *opt_result : compute_fallback();
```

think-cell

think-cell: tc_lazy

```
auto opt_result = compute_result();
auto result     = opt_result.value_or(compute_fallback());
```

think-cell: tc_lazy

```
auto opt_result = compute_result();
auto result     = opt_result.value_or(tc_lazy(compute_fallback()));
```

think-cell

Idea: Leverage implicit conversion

```
T optional<T>::value_or(auto&& fallback) {
    if (*this)
        return value();
    else
        return fallback;
}
```

think-cell

tc_lazy Implementation

```
template <typename Fn>
struct make_lazy : Fn {
    operator auto() const {
        return (*this)();
    }
};

#define tc_lazy(...) \
    make_lazy([&] -> decltype(auto) { return __VA_ARGS__; })
```

See also: www.foonathan.net/2017/06/lazy-evaluation

think-cell

.or_else()

```
auto opt_result = compute_result();
auto result     = opt_result.value_or(tc_lazy(compute_fallback()));
```

.or_else()

```
auto opt_result = compute_result();
auto result     = opt_result.or_else(compute_fallback);
```

Monadic operations

`std::optional<T>` monadic operations:

- `opt.value_or(fallback)`:
  - `*opt` or `fallback`
  - `fallback` convertible to `T`
- `opt.or_else(f)`:
  - `*opt` or `f()`
  - `f` returns type convertible to `T`
- `opt.and_then(f)`:
  - `f(*opt)` or `std::nullopt`
  - `f` returns `std::optional<U>`
- `opt.transform(f)`:
  - `std::optional(f(*opt))` or `std::nullopt`
  - `f` returns `U`

think-cell

Monadic operations

"optional-like" monadic operations:

- `tc::value_or(opt, fallback)`:
  - `*opt` or `fallback`
  - `fallback` convertible to `*opt`
- `tc::value_or(opt, tc_lazy(f()))`:
  - `*opt` or `f()`
  - `f` returns type convertible to `*opt`
- `tc::and_then(opt, f)`:
  - `f(*opt)` or `decltype(f(*opt)){}`
  - `f` returns default-constructible type
- `tc::and_then(opt, tc::chained(tc::fn_make_optional{}, f))`:
  - `std::make_optional(f(*opt))` or `std::nullopt`
  - `f` returns `U`

think-cell

Monadic tc::and_then

oPj7zEddh

```
void unsubscribe_from_mailing_list(UserID id) {
    auto user = lookup_user(id);
    auto user_email = user ? user->email : std::optional<EMail>();
    if (user_email) {
        if (subscriber_list.remove(*user_email)) {
            subscriber_list_changed();
        }
    }
}
```

Monadic tc::and_then

oPj7zEddh

```
void unsubscribe_from_mailing_list(UserID id) {
    tc::and_then(lookup_user(id),
        [&](User const& user) {
            return std::make_optional(user.email);
        },
        [&](std::string const& email) -> bool {
            return subscriber_list.remove(email);
        },
        [&] {
            subscriber_list_changed();
        });
}
```

think-cell

think-cell: `tc_scope_exit`

```
auto hfile = …;
…
CloseHandle(hfile);
```

think-cell: tc_scope_exit

```
auto hfile = …;
try {
    …
    CloseHandle(hfile);
} catch (...) {
    CloseHandle(hfile);
    throw;
}
```

think-cell

think-cell: tc_scope_exit

```
auto hfile = …;
auto close = std::experimental::scope_exit([&]{ CloseHandle(file); });
```

think-cell

think-cell: tc_scope_exit

```
auto hfile = …;
tc_scope_exit { CloseHandle(hfile); };
```

tc_scope_exit Implementation

```
template <typename Fn>
struct scope_exit_impl : Fn {
    ~scope_exit_impl() {
        (*this)();
    }
};

#define tc_scope_exit(...) \
    auto TC_UNIQUE_IDENTIFIER = tc::scope_exit([&]{ __VA_ARGS__ })
```

think-cell

tc_scope_exit Implementation

```
template <typename Fn>
struct scope_exit_impl : Fn {
    ~scope_exit_impl() {
        (*this)();
    }
};

#define tc_scope_exit(...) \
    auto TC_UNIQUE_IDENTIFIER = tc::scope_exit([&]{ __VA_ARGS__ })
```

```
auto hfile = …;
tc_scope_exit(CloseHandle(hfile););
```

think-cell

Actual tc_scope_exit implementation

```
template <typename Fn>
struct scope_exit_impl { … };

struct make_scope_exit_impl {
    template <typename Fn>
    auto operator->*(Fn const& fn) const {
        return scope_exit_impl(fn);
    }
};

#define tc_scope_exit \
    auto TC_UNIQUE_IDENTIFIER = tc::make_scope_exit_impl{} ->* [&]
```

think-cell

think-cell Ranges

think-cell

Example

fjf14eExa

```
auto ints = stdv::iota(1, 20);
auto even_ints = ints | stdv::filter([](int i) { return i % 2 == 0; });
auto squared_ints = even_ints | stdv::transform([](int i) { return i * i; });

for (int i : squared_ints)
  std::printf("%d\n", i);
```

think-cell

Example

x7MasYvWc

```
auto ints = tc::iota(1, 20);
auto even_ints = tc::filter(ints, [](int i) { return i % 2 == 0; });
auto squared_ints = tc::transform(even_ints, [](int i) { return i * i; });

tc::for_each(squared_ints,
    [](int i) {
        std::printf("%d\n", i);
    });
```

What is a range?

- Standard: iterator range

think-cell

What is a range?

- Standard: iterator range
- think-cell: generator range
- think-cell: index range

Pythagorean triples: Iterators

```
auto pythagorean_triples() {
  return for_each(iota(1), [](int z) {
        return for_each(iota(1, z+1), [=](int x) {
            return for_each(iota(x, z+1), [=](int y) {
                return yield_if(x*x + y*y == z*z, make_tuple(x, y, z));
            });
        });
  });
}
```

ericniebler.com/2018/12/05/standard-ranges/

think-cell

External vs. internal iteration

External iteration

- Caller controls the iteration.
- Loops in iterator need to be awkwardly split, build a state machine.

Internal iteration

- Iterator controls the iteration.
- Iterator can just write a loop.

think-cell

External vs. internal iteration

External iteration

- Caller controls the iteration.
- Loops in iterator need to be awkwardly split, build a state machine.

Internal iteration

- Iterator controls the iteration.
- Iterator can just write a loop.

Coroutines: write internal iteration with the control of external iteration.

Pythagorean triples: Coroutines

```
std::generator<std::tuple<int, int, int>> pythagorean_triples() {
    for (auto z = 1; true; ++z)
        for (auto x = 1; x <= z; ++x)
            for (auto y = x; y <= z; ++y)
                if (x*x + y*y == z*z)
                    co_yield make_tuple(x, y, z);
}
```

think-cell

Pythagorean triples: Coroutines

```
std::generator<std::tuple<int, int, int>> pythagorean_triples() {
    for (auto z = 1; true; ++z)
        for (auto x = 1; x <= z; ++x)
            for (auto y = x; y <= z; ++y)
                if (x*x + y*y == z*z)
                    co_yield make_tuple(x, y, z);
}
```

But:

- heap allocation
- opaque for optimizer
- requires coroutines in the entire call stack.

think-cell

Pythagorean triples: Generator ranges

```cpp
auto pythagorean_triples() {
    return [](auto sink) {
        for (auto z = 1; true; ++z)
            for (auto x = 1; x <= z; ++x)
                for (auto y = x; y <= z; ++y)
                    if (x*x + y*y == z*z)
                        tc_yield(sink, make_tuple(x, y, z));
    };
}
```

think-cell

Using generator ranges

s8Tjz8TMz

```
tc::for_each(pythagorean_triples(),
    [](int x, int y, int z) {
        std::printf("%d, %d, %d\n", x, y, z);
    });
```

think-cell

Using generator ranges

s8Tjz8TMz

```
auto count = 0;
tc::for_each(pythagorean_triples(),
    [&](int x, int y, int z) {
        std::printf("%d, %d, %d\n", x, y, z);

        if (++count == 10)
            return tc::break_;
        else
            return tc::continue_;
    });
```

think-cell

Implementing tc_yield

```
enum break_or_continue {
    break_,
    continue_,
};

template <typename Sink, typename ... Args>
auto continue_if_not_break(Sink&& sink, Args&&... args) {
    using result_type = decltype(sink(args)...));
    if constexpr (std::is_same_v<result_type, break_or_continue> {
        return sink(args...);
    } else {
        sink(args...);
        return tc::continue_;
    }
}
```

Implementing tc_yield

```
#define tc_return_if_break(...) \
do \
{ \
    if ((__VA_ARGS__) == tc::break_) \
        return tc::break_; \
} while (0)

#define tc_yield(...) \
  tc_return_if_break(tc::continue_if_not_break(__VA_ARGS__))
```

think-cell

Generator ranges and adapters

Iterator to generator:

```
[rng](auto sink) {
  for (auto&& elem : rng)
      tc_yield(sink, tc_move_if_owned(elem));
}
```

think-cell

Generator ranges and adapters

Iterator to generator:

```
[rng](auto sink) {
  for (auto&& elem : rng)
      tc_yield(sink, tc_move_if_owned(elem));
}
```

Generator to iterator: n/a

think-cell

Generator ranges and adapters

Iterator to generator:

```
[rng](auto sink) {
  for (auto&& elem : rng)
      tc_yield(sink, tc_move_if_owned(elem));
}
```

Generator to iterator: n/a

Adapters:

- generator and iterator interface
- algorithms prefer generator interface

think-cell

Generator adapters are trivial

```
auto filter(auto rng, auto predicate) {
    return [=](auto sink) {
        return tc::for_each(rng, [&](auto&& item) {
            if (predicate(item))
                tc_yield(sink, tc_move_if_owned(item));
        });
    };
}
```

think-cell

Generator adapters are trivial

```
auto filter(auto rng, auto predicate) {
    return [=](auto sink) {
        return tc::for_each(rng, [&](auto&& item) {
            if (predicate(item))
                tc_yield(sink, tc_move_if_owned(item));
        });
    };
}
```

```
auto concat(auto ... rngs) {
    return [=](auto sink) {
        return tc::for_each(std::make_tuple(rngs...), [&](auto rng) {
            return tc::for_each(rng, sink);
        });
    };
}
```

Iterator range

```
struct range
{
    struct sentinel {}; // empty

    struct iterator // stores state
    {
        T& operator*() const;
        iterator& operator++();
        bool operator==(sentinel) const;
    };

    iterator begin() const;
    sentinel end() const;
};
```

Iterator has state and logic.

think-cell

Index range

```
struct range
{
    struct tc_index {}; // stores state

    tc_index begin_index() const;

    T& dereference_index(const tc_index& idx);
    void increment_index(tc_index& idx) const;
    bool at_end_index(const tc_index& idx) const;
};
```

Iterator has state only, range has logic.

think-cell

Duality

```
struct iterator_from_index {
    index_range* _range;
    index_range::tc_index _idx;

    T& operator*() { return _range->dereference_index(_idx); }
    …
};
```

```
struct index_range_from_iterator {
    using tc_index = iterator;

    T& dereference_index(tc_index idx) { return *idx; }


    …
};
```

Advantage over index ranges

- Iterators can dangle, indices cannot.
- Indices are less likely to be invalidated.
- We can do efficient bounds checking.
- Space efficient when nesting adapters.

think-cell

std::filter_view

```
template <typename View, typename Predicate>
struct filter_view {
    View _base;
    Predicate _pred;

    struct iterator {
        filter_view* _parent;
        ranges::iterator_t<View> _current;

        auto& operator*() const { return *_current; }
        iterator& operator++() {
            do {
                ++_current;
            } while (_current != _parent->end() && !_parent->_pred(*_current));
        }
    };
```

tc::filter_adaptor

```
template <typename Rng, typename Predicate>
struct filter_adaptor {
    Rng _base;
    Predicate _pred;

    using tc_index = tc::index_t<Rng>;
    auto& dereference_index(const tc_index& idx) {
        return _base.dereference_index(idx);
    }
    void increment_index(tc_index& idx) const {
        do {
            _base.increment_index(idx);
        } while (!_base.at_end_index(idx) && !_pred(dereference_index(idx)));
    }
};
```

Nested filters

```
auto view = stdv::filter(stdv::filter(stdv::filter(input), p), p, p);
```

- `view` stores `input` and three copies of `p`
- `decltype(view)::iterator` stores `decltype(input)::iterator` and three pointers to `filter_view`

Nested filters

```
auto rng = tc::filter(tc::filter(tc::filter(input), p), p, p);
```

- `rng` stores `input` and three copies of `pred`
- `decltype(rng)::tc_index` stores `decltype(input)::tc_index` and nothing else
- `tc::make_iterator(rng, view.begin_index())` stores `decltype(input)::tc_index` and pointer to `rng`

think-cell

Output iterators

Iterators that are write-only.

Output iterators

Iterators that are write-only.

```
std::vector<int> vec = …;
std::ranges::copy(vec, ptr);
```

Calls `std::memcpy`.

Output iterators

Iterators that are write-only.

```
std::vector<int> vec = …;
std::ranges::copy(vec, ptr);
```

Calls `std::memcpy`.

```
std::deque<int> deque = …;
std::ranges::copy(deque, ptr);
```

Copies element-by-element, even though chunks are contiguous.

think-cell

Output iterators

Iterators that are write-only.

```
std::vector<int> vec = …;
std::ranges::copy(vec, ptr);
```

Calls `std::memcpy`.

```
std::deque<int> deque = …;
std::ranges::copy(deque, ptr);
```

Copies element-by-element, even though chunks are contiguous.

```
std::vector<int> vec = …;
std::ranges::copy(vec, std::back_inserter(other_vec));
```

Calls `.push_back()` for each element, not `.insert(other_vec.end(), vec.begin(), vec.end())`.

think-cell: Appender instead of output iterators

```
struct Appender {
    template <typename T>
    void operator()(T&& single);

    template <typename Rng>
    void chunk(Rng&& rng);
};
```

Opportunity to append entire ranges at once.

think-cell: `tc::append`

```
tc::append(container, rng1, rng2, rng3)
```

- Uses `tc::appender(container)` CPO.
- Does `tc::explicit_cast` automatically

think-cell

tc::for_each

```
tc::for_each(rng, sink)
```

Tries in order:

1. `sink.chunk(rng)` for appender
2. ADL-based customization point `for_each_impl(rng, sink)`
3. `rng(sink)` for generator ranges
4. Index based iteration.
5. Iterator based iteration.

think-cell

Guideline: Do not use range-based for loop

Goal: No raw loops.

- Simple range based for loops tend to grow over time.
- Range-based for incompatible with generator ranges.

think-cell: Convenience algorithms

- More container-based algorithms:
  - `tc::filter_inplace(container, predicate)` (erase-remove_if-idiom)
  - `tc::take_first_inplace(container, truncated_size)`
  - `tc::sort_unique_inplace(container)`
  - ...

think-cell

think-cell: Convenience algorithms

- More container-based algorithms:
  - `tc::filter_inplace(container, predicate)` (erase-remove_if-idiom)
  - `tc::take_first_inplace(container, truncated_size)`
  - `tc::sort_unique_inplace(container)`
  - ...
- Control what find returns:

```
tc::find_first_if<Return>(rng, predicate)
```

  - `tc::return_value_or_none`: `std::optional<T>`
  - `tc::return_bool`: bool
  - `tc::return_element_index_or_none`: `std::optional<std::size_t>`
  - `tc::return_take_before_or_all`: subrange `[begin, pos)`
  - ...

think-cell

Strings

think-cell

Philosophy

Strings are just ranges.

Encoding

- UTF-8 by default.
- UTF-16 for interaction with the WinAPI.
- Many ASCII string literals.

Encoding

- UTF-8 by default.
- UTF-16 for interaction with the WinAPI.
- Many ASCII string literals.

Ideally:

- `char8_t` for UTF-8
- `char16_t` for UTF-16
- `char` for ASCII

think-cell

Encoding

- UTF-8 by default.
- UTF-16 for interaction with the WinAPI.
- Many ASCII string literals.

Reality:

- `char` for UTF-8
- `char16_t` for UTF-16, but `wchar_t` on Windows
- ??? for ASCII

think-cell

think-cell: ASCII-only character type

```
using char_ascii = tc::value_restrictive<char, '\0', '\x7f'>;
```

- Strong typedef for a `char` restricted between `0x00` and `0x7F`
- Documents and asserts ASCII content
- Enables optimized overloads

String literals have the wrong type

```
std::ranges::size("abc") // 4!
```

- type of string literal is `CharT const (&)[N]`
- `end()` includes null terminator
- no distinction between static string literals and temporary arrays
- value is no longer statically known

think-cell

String literals: _tc UDL

Not yet public.

```
template <tc::string_template_param String>
consteval auto operator""_tc() -> tc::string_literal<…> {
    return {};
}
```

- type of string literal is `tc::string_iteral`
- `end()` does not include null terminator
- value is encoded into type, pointer range generated on-demand

think-cell

String literals: _tc UDL

Not yet public.

```
template <tc::string_template_param String>
consteval auto operator""_tc() -> tc::string_literal<…> {
    return {};
}
```

- type of string literal is `tc::string_iteral`
- `end()` does not include null terminator
- value is encoded into type, pointer range generated on-demand

Character type:

```
u8"hello"_tc    // char
 u"hello"_tc    // char16_t or wchar_t
  "hello"_tc    // tc::char_ascii
```

think-cell

std::string has special comparisons

bhK4r3q3Y

```
std::string lhs = {'a'};
std::string rhs = {char(0xC3), char(0xA4)}; // ä

assert(lhs < rhs);
```

think-cell

std::string has special comparisons

bhK4r3q3Y

```
std::vector<char> lhs = {'a'};
std::vector<char> rhs = {char(0xC3), char(0xA4)}; // ä

assert(lhs < rhs);
```

think-cell

String storage

```
template <typename Char>
using string = std::basic_string<Char, select_char_traits<Char>>;
```

- `std::basic_string` for SSO
- `tc::string<char>` uses “correct” comparison
- `tc::string<char16_t>` is the same as `std::u16string`

think-cell

String storage

```
template <typename Char>
using string = std::basic_string<Char, select_char_traits<Char>>;
```

- `std::basic_string` for SSO
- `tc::string<char>` uses “correct” comparison
- `tc::string<char16_t>` is the same as `std::u16string`

Use algorithms; not special member functions.

think-cell

String views

```
template <typename T>
using span = tc::subrange<tc::iterator_base<T*>>;
```

- A subrange whose index type is given by a pointer.
- String arguments are not different from other subranges.
- We do not use `std::string_view`.

think-cell

String formatting

dhfTcWTnM

```
fmt::format("The answer is {}.\n", 42); // The answer is 42.
```

```
std::vector<unsigned char> mac = …;
fmt::format("{:02x}", fmt::join(mac, ":")); // aa:bb:cc:dd:ee:ff
```

- Format string has placeholders to control value.
- Embedded DSL and custom functions to control formatting.
- Adding formatting support to types requires format specifier parsing.
- Eagerly create `std::string` or push to output iterator.

think-cell

String formatting

7rjqr4vsY

```
tc::concat("The answer is "_tc, tc::as_dec(42), ".\n"_tc);
```

```
std::vector<unsigned char> mac = …;
tc::join_with_separator(":"_tc,
    tc::transform(mac, tc_fn(tc::as_padded_hex)));
```

- No format string, string is concatenated from pieces.
- No DSL, normal functions control formatting.
- Adding formatting support to types requires writing a function that returns a range.
- Lazily describe range.

think-cell

tc::placeholders

For internationalization:

```
tc::placeholders("Hello, {0}."_tc, "World"_tc);
```

```
tc::placeholders("{pi} is our favorite number."_tc,
                 tc::named<"pi">(tc::as_dec(3.14)));
```

Unlike `std::format()`, localized strings don't need to include formatting info.

Files and I/O

Not yet public.

Files are just ranges.

Not yet public.

Files are just ranges.

Reading File is a generator range of `unsigned char`.

```
tc::for_each(file, [](unsigned char byte) {
    // Do something.
});
```

think-cell

Not yet public.

Files are just ranges.

Reading File is a generator range of `unsigned char`.

```
tc::for_each(file, [](unsigned char byte) {
    // Do something.
});
```

Writing File provides an appender for `tc::append()`.

```
tc::append(file, tc::size_prefixed(rng), tc::as_blob(data));
```

think-cell

Example

think-cell

Example: Generate download page

```
std::vector<Build> builds = {
    {12, 0, …},
    {12, 1, …},
    {11, 0, …},
    {11, 1, …},
};
```

think-cell 12

- Build 0: ...
- Build 1: ...

think-cell 11

- Build 0: ...
- Build 1: ...

think-cell

Example: Generate download page

```
tc::adjacent_unique_range(builds,
        tc::projected(tc::fn_equal(), tc_member(.major_version))),
```

Example: Generate download page

```
                                  tc::transform(
  tc::adjacent_unique_range(builds,
          tc::projected(tc::fn_equal(), tc_member(.major_version))),
  [&](auto&& builds_same_major)  {










  })
```

Example: Generate download page

```
                                  tc::transform(
  tc::adjacent_unique_range(builds,
          tc::projected(tc::fn_equal(), tc_member(.major_version))),
  [&](auto&& builds_same_major)  {
    tc_auto_cref(version, tc::front(builds_same_major).major_version);
    return tc::concat("<h4>think-cell "_tc, tc::as_dec(version), "</h4><ul>"_tc,








                  "</ul>"_tc);
  })
```

Example: Generate download page

```
                               tc::transform(
tc::adjacent_unique_range(builds,
        tc::projected(tc::fn_equal(), tc_member(.major_version))),
[&](auto&& builds_same_major)  {
  tc_auto_cref(version, tc::front(builds_same_major).major_version);
  return tc::concat("<h4>think-cell "_tc, tc::as_dec(version), "</h4><ul>"_tc,
          tc::join(tc::transform(builds_same_major, [&](auto&& build) {
              return tc::concat("<li>"_tc,
                      tc::placeholders(_GT("Build {0}"),
                          tc::as_dec(build.build_nr)),
                      render_download_link(build),
                     "</li>"_tc);
          })),
          "</ul>"_tc);
})
```

Example: Generate download page

```
tc::append(http_stream, tc::join(tc::transform(
  tc::adjacent_unique_range(builds,
          tc::projected(tc::fn_equal(), tc_member(.major_version))),
  [&](auto&& builds_same_major)  {
    tc_auto_cref(version, tc::front(builds_same_major).major_version);
    return tc::concat("<h4>think-cell "_tc, tc::as_dec(version), "</h4><ul>"_tc,
            tc::join(tc::transform(builds_same_major, [&](auto&& build) {
                return tc::concat("<li>"_tc,
                        tc::placeholders(_GT("Build {0}"),
                            tc::as_dec(build.build_nr)),
                        render_download_link(build),
                       "</li>"_tc);
            })),
            "</ul>"_tc);
  })));
```

Many more features

- Enum reflection traits
  - `tc::is_enum_value<Enum>(v)`
  - `tc::all_values<Enum>` is generator of all enum values
  - `tc::enumset<Enum>` (bitset)
- Specialized data structures
  - `tc::optional<T&>`
  - `tc::static_vector<T, N>`
  - `tc::dense_map<Enum, T>`
- ...

think-cell

Conclusion

github.com/think-cell/think-cell-library

We're hiring: think-cell.com/cppnow

jonathanmueller.dev/talk/think-cell-library

@foonathan@fosstodon.org
youtube.com/@foonathan