










My ideas for a talk. . .


. . .


. . . let's visit `#random` on slack for an idea


Italian C++ Community on Slack


 Italian C++ Co...  **#random** ☆ | 👤 311 | 🗨️ 2 | Non-work banter and water cooler co...      


 **asd** 8:25 AM
Where is my socket?


 **asd** 8:55 AM
No UI, no party


 **asd** 9:29 AM
They are in love with compile-time stuff only


 **asd** 8:22 AM
std::web_view... really?

 **asd** 8:10 AM
It takes way toooooooo long to compile!!

 **asd** 8:15 AM
STL containers are meh, really really meh

 **asd** 2:22 AM
Allocators are part of the types of the containers... c'mon!

 **asd** 2:29 AM
Welcome to the UTF nightmare

 **asd** 2:59 AM
Exceptions system is clearly a joke

Modern C++: yay or nay?

So, to sum up:

- C++ could not be worse than this.
- *Modern C++* is even worse than the good, old C++98.
- The Standard Template Library is all wrong, no doubts about it.
- Containers have some (many?) design problems.
- The C++ Standards Committee is focusing only on useless things.
- ...and so on.

Join us on slack and leave your complaint! :)

Is it really like this? Let's find out together.

Idioms

You're doing it right

Michele Caini

skypjack

Novemeber 30, 2019



Released under CC BY-SA 4.0

C++ has its idioms

Programming idioms

An idiom is a phrase that doesn't make literal sense, but makes sense once you're acquainted with the culture in which it arose. Programming idioms are no different. They are the little things you do daily in a particular programming language or paradigm that only make sense to a person after getting it

Courtesy of WikiWikiWeb.

Good old one: erase-remove

Intent

To eliminate elements from a container.

Some idioms have never changed over time...

```
template<typename Type>
inline void cleanup(std::vector<Type> &vec, const Type &value) {
    auto it = std::remove(vec.begin(), vec.end(), value);
    vec.erase(it, vec.end());
}
```

...and they probably never will.

Heads up!

Don't forget about *copy&swap*, another glorious hero of all time.

Evergreen idioms

A better (?) definition

A programming idiom is the usual way to code a task in a specific language.

Many other idioms are with us from the beginning:

- Resource Acquisition Is Initialization (RAII).
- Curiously Recurring Template Pattern (CRTP).
- Pointer to Implementation (PImpl).
- Copy & Swap.
- ...

Some others have been *deprecated* instead and we won't miss them.

Null pointer

Intent

To distinguish between 0 and a null pointer.

In a perfect world it would be:

```
#define NULL ((void *)0)
```

If only this were possible: `char * str = NULL;`

`void *` doesn't convert to `T *` but *int-to-pointer* conversions exist:

```
#define NULL 0
```

Problem: 0 vs 0L and overload resolution.

Null pointer

Intent

To distinguish between 0 and a null pointer.

The `nullptr` idiom (based on the *return type resolver* idiom):

```
const struct nullptr_t {
    template<class T> operator T *() const { return 0; }
    template<class C, class T> operator T C:: *() const { return 0; }

private:
    void operator &() const;
} nullptr = {};
```

Sounds familiar? `char *ch = nullptr;`

Now a **keyword** of the language.

Move constructor

Intent

To transfer the ownership of a resource held by an object to another object.

It takes advantage of some less known features of the language:

```
template <class Type> struct proxy { Type *res; };

template <class Type> struct movable {
    movable(Type *r = 0) : res(r) {}
    ~movable() { delete res; }

    movable(movable &o): res(o.res) { o.res = 0; }
    movable(proxy<Type> o): res(o.res) {}
    movable & operator=(movable &o) { /* copy-and-swap. */ }
    movable & operator=(proxy<T> o) { /* copy-and-swap. */ }
    void swap(movable &o) { std::swap(res, o.res); }

    operator proxy<T>() { proxy<T> p; p.res = res; res = 0; return p; }

private:
    Type *res;
};
```

Move constructor

Intent

To transfer the ownership of a resource held by an object to another object.

To sum up:

- In case of non-const reference we steal the resource.
- In case of const reference we export the resource through a proxy.
- The source is set in a *valid but unspecified* state.
- Indirection, *unique ownership* model.

Now an **utility** (as in <utility>):

```
my_type other{std::move(instance)};
```

Final class

Intent

To prevent a class from further subclassing or inheritance.

`friend`-ness is the key, `private`-ness is the way:

```
class base {
    ~base() {}
    friend class derived;
};
```

```
struct derived: base { /* ... */ };
```

Let's inherit from derived:

```
struct my_class: derived { /* ... */ };
my_class instance;
```

Errors occur only in case of instantiations but *it works*.

Final class

Intent

To prevent a class from further subclassing or inheritance.

Nowadays `final` is a **keyword** resolved no matter what:

```
struct derived final { /* ... */ };  
struct my_class: derived { /* ... */ };
```

It works always, everywhere.

Heads up!

Thanks to override we've also less headaches.

```
struct derived: base { void func() override { /* ... */ } };
```

Enable if

Intent

To allow function overloading based on arbitrary properties of types.

SFINAE lovers thank:

```
template<bool, typename = void>
struct enable_if {};
```

```
template<typename Type>
struct enable_if<true, Type> {
    typedef Type type;
};
```

Heads up!

is_same, is_base_of, is_invocable, void_t, decay_t, ...

Enable if

Intent

To allow function overloading based on arbitrary properties of types.

std::, you're doing it right:

```
template<typename Type, typename Func>
typename std::enable_if<std::is_integral_v<Type>>::type
void do_something(Func func) {
    if constexpr(std::is_invocable<Func, Type>::value) {
        // ...
    } else {
        // ...
    }
}
```

Helper types (`_t`) and variable templates (`_v`) did the rest.

C++ evolves, idioms evolve

What really is an idiom?

A language-specific way of solving a problem that can arise from a lack.

Many other idioms have been *deprecated* over time:

- Safe bool (welcome `explicit`).
- Type generator (welcome alias templates).
- Shrink-to-fit (welcome `.shrink_to_fit()`).
- Type safe enum (welcome `enum` class).
- Traits and utilities of any type.
- ...

Some others have been revised in a more *modern* key (as in *modern C++*).

Variadic template and pack expansion

Intent

To define template parameter lists of any size and consume them.

Old-fashioned variadic templates:

```
template<typename T1>
void func(T1 t1) { /* ... */ }
```

```
template<typename T1, typename T2>
void func(T1 t1, T2 t2) { /* ... */ }
```

```
// ...
```

```
template<typename T1, typename T2, typename T3, typename T4, typename T5>
void func(T1 t1, T2 t2, T3 t3, T4 t4, T5 t5) { /* ... */ }
```

```
// ...
```

I've seen things that you people wouldn't believe.

Variadic template and pack expansion

Intent

To define template parameter lists of any size and consume them.

True variadic templates introduced new short-lived idiom(s):

```
template<typename... Func>
void invoke_all(Func... func) {
    int _[] { 0, (func(), 0)... };
    (void)_;
}
```

Another option was recursive functions with fallback.

There was no (easy) way to *unfold* a parameter pack.

Variadic template and pack expansion

Intent

To define template parameter lists of any size and consume them.

Fold expressions solved the problem:

```
template<typename... Func>
void invoke_all(Func... func) { (func(), ...); }
```

In fact, they gave us much more:

```
template<auto... Test, typename... Func>
int invoke_all(Func... func) {
    (Test() || ...) ? (0 + ... + func()) : 0;
}
```

Include guard macro

Intent

To allow inclusion of a header file multiple times.

```
#ifndef MY_GUARD
#define MY_GUARD
```

Dear, old inclusion guard macros:

- Compile-time scalability: for each TU, parse and include headers.
- Fragility: active macros matter, such as `#define` `std` "Standard"
- Agree on disagree: many years, no convergence. To `__` or not to `__`?
- All or nothing: things are private in name only, not in fact.

Not to mention the use of `pragma once`.

```
#endif // MY_GUARD
```

Include guard macro

Intent

To allow inclusion of a header file multiple times.

A module is a *producer* that knows what to export:

```
export module hello;
export const char * message() { return "Hello, \u00a0world!"; }
```

Consumers see only public stuff:

```
import hello;
int main() { std::cout << message() << std::endl; }
```

It took only 30 years to get them but here they are finally!

Comparable object

Intent

To provide consistent relational operators for a type.

Making a type comparable has never been easier:

- Define `operator==` and `operator!=`.
- Define `operator<` and `operator<=`.
- Define `operator>` and `operator>=`.

Verbosity to the rescue. Hands up who has never done this:

```
operator!=(T lhs, T rhs) -> !(lhs == rhs)
```

Those are the things that make you hate the language.

Comparable object

Intent

To provide consistent relational operators for a type.

With the *spaceship operator* a compiler will generate everything for me:

```
auto operator<=>(const T &) const = default;
```

Allow out of order comparisons and non-default semantic:

```
std::strong_ordering std::weak_ordering std::partial_ordering  
std::strong_equality std::weak_equality
```

Support indistinguishable/distinguishable, incomparable/comparable values.

Function object (aka *functor*)

Intent

To create objects that *look like* functions.

Function like objects were rather common some time ago:

```
struct functor {  
    void operator(int value) { /* ... */ }  
};
```

```
// ...
```

```
std::for_each(first, last, functor());
```

Many objects, more code to maintain, no way to design one-off solutions.
They also had pros though: they were reusable.

Function object (aka *functor*)

Intent

To create objects that *look like* functions.

Nowadays we have (possibly generic) lambda functions:

```
std::for_each(first, last, [](int value) { /* ... */ });  
std::for_each(first, last, [](auto value) { /* ... */ });
```

We are also going to have everything we want:

```
std::for_each(first, last, [<typename... Args>(Args &&... args) {  
    /* ... */  
});
```

No more `std::forward<decltype(Args)>(args)...`!

Iterator range

Intent

To specify a range without worrying about the data structure.

Initially, it must have seemed like a good idea:

```
std::for_each(container.begin(), container.end(), std::move(func));
```

Among the other benefits:

- Generic algorithms.
- Write it once, use it everywhere.
- No matter what our container is.

Iterator range

Intent

To specify a range without worrying about the data structure.

Of course, filtering and then transforming values isn't that good:

```
std::vector<int> num = { 0, 1, 2, 3, 4, 5 };

std::vector<int> even;
std::copy_if(num.begin(), num.end(), std::back_inserter(even),
    [](int n) { return n % 2 == 0; });

std::vector<int> square;
std::transform(begin(even), end(even), std::back_inserter(square),
    [](int n) { return n * n; });
```

Many problems, algorithms aren't even *composable*.

Range-based `for` loop didn't add much here.

Iterator range

Intent

To specify a range without worrying about the data structure.

Ranges make the whole thing much more *intuitive*:

```
auto res = num
  | ranges::view::filter([](int n) { return n % 2 == 0; })
  | ranges::view::transform([](int n) { return n * n; });
```

Using iterators without having to deal with iterators.

Idioms *disappear* when raised to a higher level.

The idiom that made history

From C++98 to C++20

A journey along N revisions and through many changes.

There is an idiom that more than others:

- Meets a need that we all have sooner or later.
- Has existed since the dawn and will always exist.
- Has evolved a revision of the standard at a time.
- Is a proof of how the language has improved in many respects.

Its name is...

Member detector

Intent

To detect a specific member attribute, function or type in a class.

Not so modern C++:

```
template<typename T>
struct detector {
    struct fallback { int x; };
    struct derived: T, fallback {};

    template<typename U, U> struct check;
    typedef char one[1]; typedef char two[2];

    template<typename U> static one & func(check<int fallback:: *, &U::x> *);
    template<typename> static two & func(...);

public:
    enum { value = sizeof(func<derived>(0)) == 2 };
};
```

To use as `detector<my_type>::value` with `enable_if` or similar.

Member detector

Intent

To detect a specific member attribute, function or type in a class.

One-off solution in Modern C++:

```
template<typename T> auto f(int)
-> decltype(&T::x, void()) { /* ... */ }

template<typename T> void f(char) { /* ... */ }
template<typename T> void f() { return f<T>(0); }
```

Tag dispatching and `decltype` work like a charm together.

Heads up!

Do you remember the *choice trick*?

Member detector

Intent

To detect a specific member attribute, function or type in a class.

Structured solution in Modern C++:

```
template<typename T, std::void_t<>>
struct has_x: std::false_type {};

template<typename T>
struct has_x<T, std::void_t<decltype(&T::x)>>: std::true_type {};

template<typename T>
void f() {
    static_assert(std::has_x<T>::value);
    // ...
}
```

Use `std::enable_if_t` for a compile-time `if-else`-like statement.

Member detector

Intent

To detect a specific member attribute, function or type in a class.

`if constexpr` reduced the necessity for SFINAE/`static_assert`:

```
template<typename T>
void f() { if constexpr(std::has_x<T>::value) { /* ... */ } }
```

Concepts make it possible to no longer pollute function declarations:

```
template <class Type>
concept HasX = requires(Type type) { type.x; };
```

```
template<HasX Type>
void f() { /* ... */ }
```

Idioms: you're doing it right

Many other idioms have evolved in the meantime but let's sum it up:

- C++ could ~~not~~ be better than this but it's pretty good already.
- Modern C++ has given us a lot and much more awaits us.
- The standard template library is improving a little at a time.
- Containers have fewer problems than they once had.
- The C++ Standards Committee is doing a good job overall.

Join us on slack and leave your ~~complaint~~ feedback! :)

Because we like to complain but all in all we love C++.

The curious case of `std::shared_ptr`

The class template you don't expect

Someone even wanted to show us what it means to exploit idioms.

`std::shared_ptr` is as elegant as it's practical:

- Smart pointer: I said smart, not `auto_ptr`.
- Intrusive reference counting: more or less, due to shared ownership.
- Type erasure: `std::shared_ptr<void>`.
- CRTP: `std::enable_shared_from_this<T>`.
- Who knows what others...

It put idioms into practice without many of us even realizing it.

Standard Template Library meets Idioms

Examples at hand

The Standard Template Library is a great resource to learn from.

`std::shared_ptr` isn't the only case of idioms though:

- `std::unique_ptr`: another smart pointer.
- `std::variant`: a *safe* tagged union.
- `std::optional`: an alternative to returning a pointer.
- `std::any`: type erasure in the guise of a class.
- `std::cout` and the others: nifty counter.
- `<algorithm>` and containers: a jumble of idioms.
- ...




Questions?

Italian C++ Conference 2019



Novemeber 30, Parma

Links

- [More C++ Idioms](#) 
- [What is a programming idiom?](#) 
- [Programming idiom](#) 
- [C++ patterns](#) 