C++ Lambda Idioms

Timur Doumler

@timur_audio

C++North
18 July 2022

C++11

```
struct Person {
    String name;
    Date dateOfBirth;
};

std::vector<Person> people = {
    {"John Doe", "1953-02-25" },
    {"Vasiliy Pupkin", "1986-04-26"},
    {"Erika Mustermann", "1964-08-12" }
};
```

```cpp
struct Person {
    String name;
    Date dateOfBirth;
};

std::vector<Person> people = {
    {"John Doe", "1953-02-25" },
    {"Vasiliy Pupkin", "1986-04-26"},
    {"Erika Mustermann", "1964-08-12" }
};

void sortPeople() {
    std::sort(
        people.begin(), people.end(),
        [](const Person& lhs, const Person& rhs) {
            return lhs.name < rhs.name;
        });
}
```

[expr.prim.lambda]/2

The closure type is not an aggregate type. An implementation may define the closure type differently from what is described below provided this does not alter the observable behavior of the program other than by changing:

(2.1) — the size and/or alignment of the closure type,

(2.2) — whether the closure type is trivially copyable ([class.prop]), or

(2.3) — whether the closure type is a standard-layout class ([class.prop]).

An implementation shall not add members of rvalue reference type to the closure type.

3 The closure type for a *lambda-expression* has a public inline function call operator (for a non-generic lambda) or function call operator template (for a generic lambda) ([over.call]) whose parameters and return type are described by the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* respectively, and whose *template-parameter-list* consists of the specified *template-parameter-list*, if any. The *requires-clause* of the function call operator template is the *requires-clause* immediately following < *template-parameter-list* >, if any. The trailing *requires-clause* of the function call operator or operator template is the *requires-clause* of the *lambda-declarator*, if any.

Your code

```
[](const Person& lhs, const Person& rhs) {
    return lhs.name < rhs.name;
};
```

Compiler

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

Your code

```
[](const Person& lhs, const Person& rhs) {
    return lhs.name < rhs.name;
};
```

Compiler

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

Your code

```
[](const Person& lhs, const Person& rhs) mutable {
    return lhs.name < rhs.name;
}
```

Compiler

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

```
[](const Person& lhs, const Person& rhs) {
    return lhs.name < rhs.name;
};
```

Your code

Compiler

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

Your code

```
[](const Person& lhs, const Person& rhs) {
    return lhs.name < rhs.name;
};
```

Compiler

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

```cpp
[](const Person& lhs, const Person& rhs) -> bool {
    return lhs.name < rhs.name;
};
```

Your code

Compiler

```cpp
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

Your code

```
[](const Person& lhs, const Person& rhs) noexcept {
    return lhs.name < rhs.name;
};
```

Compiler

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const noexcept {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

Your code

```
[](const Person& lhs, const Person& rhs) [[nodiscard]] {
    return lhs.name < rhs.name;
};
```

Compiler

```
[[nodiscard]] struct  __lambda_1 {  // Error: ‘nodiscard’ cannot be applied to types
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };
};

__lambda_1();
```

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };

    __lambda_1() = delete; // not default-constructible!
    __lambda_1& operator=(const __lambda_1&) = delete; // not assignable!
};

__lambda_1();  // this instance is auto-generated by the compiler, so no error
```

```cpp
void legacy_call(int(*f)(int)) {
    std::cout << f(7) << '\n';
}

int main() {
    legacy_call([](int i){  // OK, implicit conversion to function pointer
        return i * i;
    }); // prints 49
}
```

```cpp
void legacy_call(int(*f)(int)) {
    std::cout << f(7) << '\n';
}

int main() {
    legacy_call([](int i){  // OK, implicit conversion to function pointer
        return i * i;
    }); // prints 49
}
```

[expr.prim.lambda]/8

8 The closure type for a non-generic *lambda-expression* with no *lambda-capture* whose constraints (if any) are satisfied has a conversion function to pointer to function with C++ language linkage having the same parameter and return types as the closure type's function call operator. The conversion is to “pointer to `noexcept` function” if the function call operator has a non-throwing exception specification. The value returned by this conversion function is the address of a function `F` that, when invoked, has the same effect as invoking the closure type's function call operator on a default-constructed instance of the closure type. `F` is a constexpr function if the function call operator is a constexpr function and is an immediate function if the function call operator is an immediate function.

```
struct __lambda_1 {
    inline bool operator()(const Person& lhs, const Person& rhs) const {
        return lhs.name < rhs.name;
    };

    __lambda_1() = delete; // not default-constructible!
    __lambda_1& operator=(const __lambda_1&) = delete; // not copyable or assignable!

    using __func_type = bool(*)(const Person&, const Person&);
    inline operator __func_type() const noexcept {
        return &__invoke;
    }

private:
    static inline bool __invoke(const Person& lhs, const Person& rhs) {
        return lhs.name < rhs.name;
    }
};

__lambda_1();
```

Lambda Idiom 1:

Unary plus trick

```cpp
void legacy_call(int(*f)(int)) {
    std::cout << f(7) << '\n';
}

int main() {
    legacy_call([](int i){  // OK, implicit conversion to function pointer
        return i * i;
    }); // prints 49
}
```

…but what if we need *explicit* conversion to function pointer?

```
int main() {
    auto f = static_cast<int(*)(int)>([](int i){ return i * i; });
    static_assert(std::is_same_v<decltype(f), int(*)(int)>);
}
```

```
int main() {
    auto f = +[](int i){ return i * i; };
    static_assert(std::is_same_v<decltype(f), int(*)(int)>);
}
```

Lambda captures

```cpp
int i = 0;
int j = 0;
auto f = [=] {
    return i == j;
};
```

```cpp
int i = 0;
int j = 0;
auto f = [=] {
    return i == j;
};
```

Your code

```cpp
struct __lambda_2 {
    __lambda_2(int i, int j)
      : __i(i), __j(j)
    {}

    inline bool operator()() const {
        return __i == __j;
    }

    private:
        int __i;
        int __j;
};
__lambda_2(i, j);
```

Compiler

[expr.prim.lambda]/10

For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member is the referenced type if the entity is a reference to an object, an lvalue reference to the referenced function type if the entity is a reference to a function, or the type of the corresponding captured entity otherwise. A member of an anonymous union shall not be captured by copy.

11 Every *id-expression* within the *compound-statement* of a *lambda-expression* that is an odr-use ([basic.def.odr]) of an entity captured by copy is transformed into an access to the corresponding unnamed data member of the closure type.

```cpp
int i = 0;
int j = 0;
auto f = [&] {
    return i == j;
};
```

Your code

```cpp
struct __lambda_2 {
    __lambda_2(int& i, int& j)
      : __i(i), __j(j)
    {}

    inline bool operator()() const {
        return __i == __j;
    }

    private:
        int& __i;
        int& __j;
};
__lambda_2(i, j);
```

Compiler

[expr.prim.lambda]/12

12 An entity is *captured by reference* if it is implicitly or explicitly captured but not captured by copy. It is unspecified whether additional unnamed non-static data members are declared in the closure type for entities captured by reference. If declared, such non-static data members shall be of literal type.

```
int i = 0;
int j = 0;
auto f = [&] {
    return i == j;
};
```

Your code

```
struct __lambda_2 {
    __lambda_2(int& i, int& j)
      : __i(i), __j(j)
    {}

    inline bool operator()() const {
        return __i == __j;
    }

    private:
        int& __i;
        int& __j;
};
__lambda_2(i, j);
```

Compiler

```
struct X {
    void printAsync() {
        callAsync([this] {
            std::cout << i << '\n';
        });
    }

private:
    int i = 42;
};
```

```cpp
struct X {
    void printAsync() {
        callAsync([this] {
            std::cout << i << '\n';
        });
    }

private:
    int i = 42;
};
```

Your code

```cpp
struct X {
    void printAsync() {
        struct __lambda_3 {
            __lambda_3(X* _this)
              : __this(_this)
            {}

            void operator()() const {
                std::cout << __this->i << '\n';
            }

        private:
            X* __this;
        };

        callAsync(__lambda_3(this));
    }
private:
    int i = 42;
};
```

Compiler

```cpp
struct X {
    void printAsync() {
        callAsync([this] {
            std::cout << i << '\n';
        });
    }

private:
    int i = 42;
};
```

Your code

```cpp
struct X {
    void printAsync() {
        struct __lambda_3 {
            __lambda_3(X* _this)
              : __this(_this)
            {}

            void operator()() const {
                std::cout << __this->i << '\n';
            }

        private:
            X* __this;
        };

        callAsync(__lambda_3(this));
    }
private:
    int i = 42;
};
```

Compiler

Lambda capture gotchas

```cpp
int main() {
    static int i = 42;

    auto f = [=]{ ++i; };
    f();

    return i;
}
```

```
int main() {
    static int i = 42;

    auto f = [=]{ ++i; };
    f();

    return i;
}
```

```
int main() {
    static int i = 42;

    auto f = [=]{ ++i; };
    f();

    return i;  // returns 43!
}
```

```
int main() {
    static int i = 42;

    auto f = []{ ++i; };
    f();

    return i;  // returns 43!
}
```

```cpp
int i = 42;

int main() {
    auto f = []{ ++i; };
    f();

    return i;  // returns 43!
}
```

```
int main() {
    constexpr int i = 42;

    auto f = []{ std::cout << i << '\n'; };  // OK: 'i' is not odr-used
    f();
}
```

```
int main() {
    constexpr int i = 42;

    auto f = []{ std::cout << i << '\n'; };  // OK: 'i' is not odr-used
    f();
}
```

```
int main() {
    constexpr int i = 42;

    auto f = []{ std::cout << &i << '\n'; };  // Error: 'i' odr-used but not captured
    f();
}
```

```
int main() {
    constexpr int i = 42;

    auto f = [&]{ std::cout << &i << '\n'; };  // OK, prints 42
    f();
}
```

```cpp
int main() {
    const int i = 42;

    auto f = []{ std::cout << i << '\n'; };  // OK: 'i' is implictly constexpr
    f();
}
```

```cpp
int main() {
    const float f = 42.0f;

    auto f = []{ std::cout << i << '\n'; };  // Error: 'f' is not captured
    f();
}
```

Lambda Idiom 2:

Immediately Invoked Function Expressions (IIFE)

```
int main() {
    []{ std::cout << “Hello, World!\n”; }();
}
```

```
int main() {
    []{ std::cout << “Hello, World!\n”; }();
}
```

```
int main() {
    // some code...
    Foo foo;

    if (hasDatabase) {
        foo = getFooFromDatabase();
    }
    else {
        foo = getFooFromElsewhere();
    }
}
```

```cpp
int main() {
    // some code...
    Foo foo;  // Error: Foo is not default-constructible

    if (hasDatabase) {
        foo = getFooFromDatabase();
    }
    else {
        foo = getFooFromElsewhere();
    }
}
```

```
int main() {
    // some code...
    const Foo foo;

    if (hasDatabase) {
        foo = getFooFromDatabase();    // Error: cannot assign to const object
    }
    else {
        foo = getFooFromElsewhere();  // Error: cannot assign to const object
    }
}
```

```cpp
int main() {
    // some code...
    const Foo foo = hasDatabase
        ? getFooFromDatabase()
        : getFooFromElsewhere();
}
```

```
Foo getFoo() {
   if (hasDatabase) {
       return getFooFromDatabase();
   }
   else {
       return getFooFromElsewhere();
   }
}
```

```
int main() {
    // some code...
    const Foo foo = getFoo();
}
```

```
Foo getFoo(bool hasDatabase) {
   if (hasDatabase) {
       return getFooFromDatabase();
   }
   else {
       return getFooFromElsewhere();
   }
}

int main() {
    // some code...
    const Foo foo = getFoo(hasDatabase);
}
```

```
int main() {
    // some code...
    const Foo foo = [&]{
        if (hasDatabase) {
            return getFooFromDatabase();
        }
        else {
            return getFooFromElsewhere();
        }
    }();
}
```

```cpp
int main() {
    // some code…
    std::vector<Foo> foos;

    foos.emplace_back([&]{
        if (hasDatabase) {
            return getFooFromDatabase();
        }
        else {
            return getFooFromElsewhere();
       }
    }());
}
```

```
int main() {
    // some code...
    std::vector<Foo> foos;

    foos.emplace_back([&]{
        if (hasDatabase) {
            return getFooFromDatabase();
        }
        else {
            return getFooFromElsewhere();
        }
    }());
}
```

```
int main() {
    // some code…
    std::vector<Foo> foos;

    foos.emplace_back(std::invoke([]{   // since C++17
        if (hasDatabase) {
            return getFooFromDatabase();
        }
        else {
            return getFooFromElsewhere();
        }
    }));
}
```

Lambda Idiom 3:

Call-once Lambda

(Daisy Hollman: *“What you can learn from being too cute”*)

```cpp
struct X {
    X() {
        static auto _ = []{ std::cout << "called once!"; return 0; }();
    }
};

int main() {
    X x;
    X x2, x3;
}
```

C++14

Generic lambdas

```
std::map<int, std::string> httpErrors = {
    {400, "Bad Request"},
    {401, "Unauthorised"},
    {403, "Forbidden"},
    {404, "Not Found"}
};

std::for_each(
    httpErrors.begin(), httpErrors.end(),
    [](const auto& item) {
        std::cout << item.first << ':' << item.second << '\n';
    });
```

```cpp
std::map<int, std::string> httpErrors = {
    {400, "Bad Request"},
    {401, "Unauthorised"},
    {403, "Forbidden"},
    {404, "Not Found"}
};

std::for_each(
    httpErrors.begin(), httpErrors.end(),
    [](const auto& item) {
        std::cout << item.first << ':' << item.second << '\n';
    });
```

```
[](auto i){
    std::cout << i << '\n';
};
```

```cpp
[](auto i){
    std::cout << i << '\n';
};
```

Your code

```cpp
struct __lambda_6 {
    template <typename T>
    void operator()(T i) const {
        std::cout << i << '\n';
    }

    template <typename T>
    using __func_type = void(*)(T i);

    template <typename T>
    inline operator __func_type<T>() const noexcept {
        return &__invoke<T>;
    }

private:
    template <typename T>
    static void __invoke(T i) {
        std::cout << i << '\n';
    }
}; __lambda_6();
```

Compiler

```cpp
[](auto i){
    std::cout << i << '\n';
};
```

Your code

```cpp
struct __lambda_6 {
    template <typename T>
    void operator()(T i) const {
        std::cout << i << '\n';
    }

    template <typename T>
    using __func_type = void(*)(T i);

    template <typename T>
    inline operator __func_type<T>() const noexcept {
        return &__invoke<T>;
    }

private:
    template <typename T>
    static void __invoke(T i) {
        std::cout << i << '\n';
    }
}; __lambda_6();
```

Compiler

```
[](auto i){
    std::cout << i << '\n';
};
```

Your code

```
struct __lambda_6 {
    template <typename T>
    void operator()(T i) const {
        std::cout << i << '\n';
    }

    template <typename T>
    using __func_type = void(*)(T i);

    template <typename T>
    inline operator __func_type<T>() const noexcept {
        return &__invoke<T>;
    }

private:
    template <typename T>
    static void __invoke(T i) {
        std::cout << i << '\n';
    }
}; __lambda_6();
```

Compiler

```
void legacy_call(int(*f)(int)) {
    std::cout << f(7) << '\n';
}

int main() {
    legacy_call([](auto i){  // OK, implicit conversion to int(*)(int)
        return i * i;
    }); // prints 49
}
```

```
int main() {
    auto fptr = +[](auto i){  // Error: can't deduce template argument
        return i * i;
    });
}
```

```cpp
std::vector<std::string> v;
auto f = [&v](auto&& item) {
    v.push_back(std::forward<decltype(item)>(item));
};
```

```cpp
std::vector<std::string> v;
auto f = [&v](auto&& item) {
    v.push_back(std::forward<decltype(item)>(item));
};
```

Your code

```
std::vector<std::string> v;
auto f = [&v](auto&& item) {
    v.push_back(std::forward<decltype(item)>(item));
};
```

Compiler

```
struct __lambda_7 {
    __lambda_7(std::vector<std::string>& _v)
      : __v(_v) {}

    template <typename T>
    void operator()(T&& item) const {
        __v.push_back(std::forward<decltype(item)>(item));
    }

private:
    std::vector<std::string>& __v;
};
```

```
auto f = [](auto&&... args) {
    (std::cout << ... << args);    // Fold expression (since C++17)
};

f(42, "Hello", 1.5);
```

```
auto twice = [](auto&& f) {
    return [=]{ f(); f(); };
}

auto print_hihi = twice([]{ std::cout << “hi”; });
print_hihi();
```

Init capture

```cpp
struct Widget {};
auto ptr = std::make_unique<Widget>();

auto f = [ptr=std::move(ptr)] {        // move happens here
    std::cout << ptr.get() << '\n';
};

assert(ptr == nullptr);       // assert passes
f();
```

```
struct Widget {};
auto ptr = std::make_unique<Widget>();

auto f = [ptr=std::move(ptr)] {        // move happens here
    std::cout << ptr.get() << '\n';
};

assert(ptr == nullptr);        // assert passes
f();
```

```
auto f = [ptr=std::move(ptr)] {
    std::cout << ptr.get() << '\n';
};
```

Your code

Compiler

```
struct __lambda_8 {
    __lambda_8(std::unique_ptr<Widget> _ptr)
        : __ptr(std::move(_ptr))
    {}

    inline void operator()() const {
        std::cout << __ptr.get() << '\n';
    }

private:
    std::unique_ptr<Widget> __ptr; // type deduced as if by 'auto' decl
};
__lambda_8(std::move(ptr));
```

Lambda Idiom 4:

Init capture optimisation

(Bartłomiej Filipek: *“C++ Lambda Story”*)

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [str = prefix + "bar"](const std::string& s) {
        return s == str;
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [str = prefix + "bar"](const std::string& s) {
        return s == str;
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

C++17

```
auto f = []() constexpr {
    return sizeof(void*);
};

std::array<int, f()> arr = {};
```

```
auto f = []() constexpr {
    return sizeof(void*);
};

std::array<int, f()> arr = {};
```

Class template argument deduction (CTAD)

```
std::vector vec = {1, 1, 2, 3, 5, 8, 13};  // std::vector<int> deduced
```

Lambda Idiom 5:

Lambda overload set

```cpp
template <typename... Ts>
struct overload : Ts... {
    using Ts::operator()...;
};
```

```
template <typename... Ts>
struct overload : Ts... {
    using Ts::operator()...;
};

int main() {
    overload f = {
        [](int i){ std::cout << "int thingy"; },
        [](float f){ std::cout << "float thingy"; }
    };
}
```

```cpp
template <typename... Ts>
struct overload : Ts... {
    using Ts::operator()...;
};

int main() {
    overload f = {
        [](int i){ std::cout << "int thingy"; },
        [](float f){ std::cout << "float thingy"; }
    };

    f(2); // prints int thingy
    f(2.0f); // prints float thingy
}
```

```cpp
template <typename... Ts>
struct overload : Ts... {
    using Ts::operator()...;
};

int main() {
    overload f = {
        [](int i){ std::cout << "int thingy"; },
        [](float f){ std::cout << "float thingy"; }
    };

    std::variant<int, float> v = 2.0f;
    std::visit(f, v); // prints float thingy
}
```

C++20

```
struct Widget {
    float x, y;
};

auto [x, y] = Widget();
auto f = [=] {
    std::cout << x << ", " << y << "\n";
};
```

```cpp
auto foo(auto... args) {
    std::cout << sizeof...(args) << '\n';
}

template <typename... Args>
auto delay_invoke_foo(Args... args) {
    return [args...]() -> decltype(auto) {
        return foo(args...);
    };
}
```

```
auto f = [](int i) consteval {
    return i * i;
};
```

```cpp
auto f = [](int i) consteval {
    return i * i;
};

f(5);   // OK, constant expression

int x = 5;
f(x);   // Error, not a constant expression
```

- **lambdas allowed in unevaluated contexts**
- **lambdas without captures are now:**
  - **default-constructible**
  - **assignable**

```
class Widget {
    auto f = []{};  // Error: data member cannot be ‘auto’
};
```

```
class Widget {
    decltype([]{}) f;   // OK since C++20
};
```

```
template <typename T>
using MyPtr = std::unique_ptr<T, decltype([](T* t) { myDeleter(t); })>;

MyPtr<Widget> ptr;
```

```
using WidgetSet = std::set<
    Widget,
    decltype([](Widget& lhs, Widget& rhs) { return lhs.x < rhs.x; })
>;

WidgetSet widgets;
```

Templated lambdas

```cpp
std::vector<int> data = {1, 2, 3, 4, 5};
std::erase_if(
    data,
    [](auto i) {
        return i % 2;
    }
);
```

```
std::vector<int> data = {1, 2, 3, 4, 5};
std::erase_if(
    data,
    []<typename T>(T i) {
        return i % 2;
    }
);
```

```
std::vector<float> data = {1, 2, 3, 4, 5};
std::erase_if(
    data,
    []<typename T>(T i) {
        return i % 2;
    }
);

// Error: invalid operands for binary expression
```

```
std::vector<float> data = {1, 2, 3, 4, 5};
std::erase_if(
    data,
    []<typename T>(T i) requires std::integral<T> {
        return i % 2;
    }
);

// Error: ‘float’ does not satisfy constraint ‘integral’
```

C++23

deducing ‘this’

```
template <typename T>
struct optional
{
    constexpr T& value() & {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T const& value() const& {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T&& value() && {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }

    constexpr T const&& value() const&& {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }
};
```

```
template <typename T>
struct optional
{
    constexpr T& value() & {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T const& value() const& {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T&& value() && {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }

    constexpr T const&& value() const&& {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }
};
```

```cpp
template <typename T>
struct optional
{
    constexpr T& value() & {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T const& value() const& {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T&& value() && {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }

    constexpr T const&& value() const&& {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }
};
```

```
template <typename T>
struct optional
{
    constexpr T& value() & {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T const& value() const& {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T&& value() && {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }

    constexpr T const&& value() const&& {
        if (has_value())
            return std::move(this->m_value);

        throw bad_optional_access();
    }
};
```

```cpp
template <typename T>
struct optional
{
    constexpr T& value() & {
        return const_cast<T&>(
            static_cast<optional const&>(
                    *this).value());
    }

    constexpr T const& value() const& {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T&& value() && {
        return const_cast<T&&>(
            static_cast<optional const&>(
                *this).value());
    }

    constexpr T const&& value() const&& {
        return static_cast<T const&&>(
            value());
    }

};
```

```cpp
template <typename T>
struct optional
{
    constexpr T& value() & {
        return const_cast<T&>(
            static_cast<optional const&>(
                *this).value());
    }

    constexpr T const& value() const& {
        if (has_value())
            return this->m_value;

        throw bad_optional_access();
    }

    constexpr T&& value() && {
        return const_cast<T&&>(
            static_cast<optional const&>(
                *this).value());
    }

    constexpr T const&& value() const&& {
        return static_cast<T const&&>(
            value());
    }

};
```

```
template <typename T>
struct optional
{
    constexpr T& value() & {
        return value_impl(*this);
    }

    constexpr T const& value() const& {
        return value_impl(*this);
    }

    constexpr T&& value() && {
        return value_impl(move(*this));
    }

    constexpr T const&& value() const&& {
        return value_impl(move(*this));
    }

private:
    template <typename Opt>
    static decltype(auto)
    value_impl(Opt&& opt) {
        if (!opt.has_value())
            throw bad_optional_access();

        return std::forward<Opt>(opt).m_value;
    }

};
```

```
template <typename T>
struct optional
{
    constexpr T& value() & {
        return value_impl(*this);
    }

    constexpr T const& value() const& {
        return value_impl(*this);
    }

    constexpr T&& value() && {
        return value_impl(move(*this));
    }

    constexpr T const&& value() const&& {
        return value_impl(move(*this));
    }

private:
    template <typename Opt>
    static decltype(auto)
    value_impl(Opt&& opt) {
        if (!opt.has_value())
            throw bad_optional_access();

        return std::forward<Opt>(opt).m_value;
    }

};
```

```cpp
template <typename T>
struct optional
{
    constexpr T& value() & {
        return value_impl(*this);
    }

    constexpr T const& value() const& {
        return value_impl(*this);
    }

    constexpr T&& value() && {
        return value_impl(move(*this));
    }

    constexpr T const&& value() const&& {
        return value_impl(move(*this));
    }

private:
    template <typename Opt>
    static decltype(auto)
    value_impl(Opt&& opt) {
        if (!opt.has_value())
            throw bad_optional_access();

        return std::forward<Opt>(opt).m_value;
    }

};
```

```
template <typename T>
struct optional {
    template <typename Self>
    constexpr auto&& value(this Self&& self) {
        if (!self.has_value())
            throw bad_optional_access();

        return std::forward<Self>(self).m_value;
    }
};
```

```
template <typename T>
struct optional {
    template <typename Self>
    constexpr auto&& value(this Self&& self) {
        if (!self.has_value())
            throw bad_optional_access();

        return std::forward<Self>(self).m_value;
    }
};
```

```
template <typename T>
struct optional {

    constexpr auto&& value(this auto&& self) {
        if (!self.has_value())
            throw bad_optional_access();

        return std::forward<Self>(self).m_value;
    }
};
```

Deducing this

Document #:	P0847R7
Date:	2021-07-12
Project:	Programming Language C++
Audience:	EWG
Reply-to:	Gašper Ažman <gasper.azman@gmail.com> Sy Brand <sibrand@microsoft.com> Ben Deane, ben at elbeno dot com <ben@elbeno.com> Barry Revzin <barry.revzin@gmail.com>

Lambda Idiom 6:

Recursive lambdas

```
// Naive approach:

int main() {
    auto f = [](int i) {
        if (i == 0) return 1;
        return i * f(i - 1);  // Error: 'f' cannot appear in its own
    };                        // initialiser!

    std::cout << f(5);
}
```

```
// std::function: works, but too much overhead: type erasure, indirection,
// extra object, memory allocation

int main() {
    std::function<int(int)> f = [&](int i) {
        if (i == 0) return 1;
        return i * f(i - 1);
    };

    std::cout << f(5); // prints 120
}
```

```
// Y combinator: works, but too complicated / impractical

int main() {
    auto f = [&](auto&& self, int i) {
        if (i == 0) return 1;
        return i * self(self, i - 1);
    };

    auto recursive = [](auto&& f, auto&&... args) {
        return f(f, std::forward<decltype(args)>(args)...);
    };

    std::cout << recursive(f, 5); // prints 120
}
```

```cpp
// C++ deducing this: it just works :)

int main() {
    auto f = [&](this auto&& self, int i){
        if (i == 0) return 1;
        return i * self(i - 1);
    };

    std::cout << f(5); // prints 120
}
```

Lambda Idioms 5 + 6:

Recursive lambda overload set

(Ben Deane: *“Deducing this patterns”*)

```
struct Leaf {};
struct Node;
using Tree = std::variant<Leaf, Node*>;
struct Node {
    Tree left, right;
};

template <typename... Ts>
struct overload : Ts... { using Ts::operator()...; }

int countLeaves(const Tree& tree) {
    return std::visit(overload{
        [] (const Leaf&) { return 1; },
        [] (this const auto& self, const Node* node) -> int {
            return visit(self, node->left) + visit(self, node->right);
        }
    }, tree);
}
```

```
struct Leaf {};
struct Node;
using Tree = std::variant<Leaf, Node*>;
struct Node {
    Tree left, right;
};

template <typename... Ts>
struct overload : Ts... { using Ts::operator()...; }

int countLeaves(const Tree& tree) {
    return std::visit(overload{
        [] (const Leaf&) { return 1; },
        [] (this const auto& self, const Node* node) -> int {
            return visit(self, node->left) + visit(self, node->right);
        }
    }, tree);
}
```

```cpp
struct Leaf {};
struct Node;
using Tree = std::variant<Leaf, Node*>;
struct Node {
    Tree left, right;
};

template <typename... Ts>
struct overload : Ts... { using Ts::operator()...; }

int countLeaves(const Tree& tree) {
    return std::visit(overload{
        [] (const Leaf&) { return 1; },
        [] (this const auto& self, const Node* node) -> int {
            return visit(self, node->left) + visit(self, node->right);
        }
    }, tree);
}
```

```cpp
struct Leaf {};
struct Node;
using Tree = std::variant<Leaf, Node*>;
struct Node {
    Tree left, right;
};

template <typename... Ts>
struct overload : Ts... { using Ts::operator()...; }

int countLeaves(const Tree& tree) {
    return std::visit(overload{
        [] (const Leaf&) { return 1; },
        [] (this const auto& self, const Node* node) -> int {
            return visit(self, node->left) + visit(self, node->right);
        }
    }, tree);
}
```

C++ Lambda Idioms

Timur Doumler

@timur_audio

C++North
18 July 2022

Bonus Material

Lambda Idiom 7:

Variable template lambda

(Björn Fahller)

```cpp
std::vector<std::string> v;
auto f = [&v](auto&& item) {
    v.push_back(std::forward<decltype(item)>(item));
};
```

Your code

Compiler

```cpp
struct __lambda_7 {
    __lambda_7(std::vector<std::string>& _v)
      : __v(_v) {}

    template <typename T>
    void operator()(T&& item) const {
        __v.push_back(std::forward<decltype(item)>(item));
    }

private:
    std::vector<std::string>& __v;
};
```

Timur Doumler @timur_audio · Jun 10

This was my most recent "WTF? Wait... Ohh!" C++ moment.

What's your favourite, most surprising, weirdest such C++ moment?

> **Timur Doumler** @timur_audio · Jun 5
>
> TIL that local constexpr variables can be used inside a lambda without capturing:
>
> ```
> int main() {
>   constexpr int i = 42;
>   auto f = []{ std::cout << i << '\n'; };
>   f();  // compiles, prints 42
> }
> ```
>
> I've been staring at the standard for a while but still don't understand why...
>
> Show this thread

6 17

BjörnFahller @

@bjorn_fahller

Replying to @timur_audio

That you can make a lambda a variable template and access the template parameter in it.

```
template <typename T>
constexpr auto c_cast = [](auto x) { return (T)x; };
```

9:04 AM · Jun 10, 2022 · TweetDeck

```
template <typename T>
constexpr auto c_cast = [](auto x) {
    return (T)x;
};

int main() {
    return c_cast<int>(3.14159); // returns 3
}
```

Your code

```
template <typename T>
constexpr auto c_cast = [](auto x) {
    return (T)x;
};
```

Compiler

```
template <typename T>
struct __lambda_9 {
    template <typename U>
    inline auto operator()(U x) const {
        return (T)x;
    };
};

template <typename T>
auto c_cast = __lambda_9<T>();
```

Your code

```
template <typename T>
constexpr auto c_cast = [](auto x) {
    return (T)x;
};
```

Compiler

```
template <typename T>
struct __lambda_9 {
    template <typename U>
    inline auto operator()(U x) const {
        return (T)x;
    };
};

template <typename T>
auto c_cast = __lambda_9<T>();
```

```cpp
template <typename T>
constexpr auto c_cast = [](auto x) {
    return (T)x;
};
```

Your code

Compiler

```cpp
template <typename T>
struct __lambda_9 {
    template <typename U>
    inline auto operator()(U x) const {
        return (T)x;
    };
};

template <typename T>
auto c_cast = __lambda_9<T>();
```

```cpp
using ms = std::chrono::milliseconds;
using us = std::chrono::microseconds;
using ns = std::chrono::nanoseconds;

struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(std::chrono::duration_cast<us>).count();  // Error
}
```

```cpp
using ms = std::chrono::milliseconds;
using us = std::chrono::microseconds;
using ns = std::chrono::nanoseconds;

struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(std::chrono::duration_cast<us>).count();  // Error
}
```

```cpp
using ms = std::chrono::milliseconds;
using us = std::chrono::microseconds;
using ns = std::chrono::nanoseconds;

struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(std::chrono::duration_cast<us>).count();  // Error
}
```

```cpp
using ms = std::chrono::milliseconds;
using us = std::chrono::microseconds;
using ns = std::chrono::nanoseconds;

struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(std::chrono::duration_cast<us>).count(); // Error
}
```

std::chrono::duration_cast

Defined in header <chrono>

```
template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const std::chrono::duration<Rep,Period>& d);
```
(since C++11)

Converts a `std::chrono::duration` to a duration of different type `ToDuration`.

The function does not participate in overload resolution unless `ToDuration` is a specialization of `std::chrono::duration`.

No implicit conversions are used. Multiplications and divisions are avoided where possible, if it is known at compile time that one or more parameters are `1`. Computations are done in the widest type available and converted, as if by `static_cast`, to the result type only when finished.

Parameters

d - duration to convert

Return value

d converted to a duration of type `ToDuration`.

```
struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};






int main() {
    Time t(ns(3000));
    std::cout << t.convert(std::chrono::duration_cast<us>).count();  // Error
}
```

```cpp
struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

template <typename T>
constexpr auto duration_cast = [](auto d) {
    return std::chrono::duration_cast<T>(d);
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(duration_cast<us>).count();  // Works :)
}
```

```
struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

template <typename T>
constexpr auto duration_cast = [](auto d) {
    return std::chrono::duration_cast<T>(d);
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(duration_cast<us>).count();  // Works :)
}
```

```
struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

template <typename T>
constexpr auto duration_cast = [](auto d) {
    return std::chrono::duration_cast<T>(d);
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(duration_cast<us>).count();  // Works :)
}
```

```cpp
struct Time {
    std::variant<ms,ns> time;
    auto convert(const auto& converter) {
        return std::visit(converter, time);
    }
};

template <typename T>
constexpr auto duration_cast = [](auto d) {
    return std::chrono::duration_cast<T>(d);
};

int main() {
    Time t(ns(3000));
    std::cout << t.convert(duration_cast<us>).count();  // Works :)
}
```

C++ Lambda Idioms

Timur Doumler

@timur_audio

C++North
18 July 2022