From Functions to Coroutines

40 Years Evolution

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.net

Evolution of Callable

```
template<typename Callable, typename T>
T operation(Callable callable, T a, T b) { return callable(a, b); }

int add1(int a, int b) { return a + b; }

struct Add2 {
    int operator()(int a, int b) const { return a + b; }
};

operation(add1, 1900, 98);                                    // 1998

Add2 add2;
operation(add2, 1900, 98);                                    // 1998

operation([](int a, int b){ return a + b; }, 2000, 11);     // 2011
operation([](auto a, auto b){ return a + b; }, 2000, 14);  // 2014

                                                              // 2020
operation([](std::integral auto a, std::integral auto b){ return a + b; }, 2000, 20);
```

Callable

- Function
- Function Overloading
- Function Object
- Lambda Expression
- Coroutine

Callable

- **Function**
- Function Overloading
- Function Object
- Lambda Expression
- Coroutine

Function

A function is a sequence of instructions that performs a specific task, packaged as a unit.

- Implementation
  - Each function call creates a stack frame on a stack data structure
  - The stack frame contains the private data of the function call (parameters, locals and the return address)
  - At the end of the function, the stack frame is deleted

Function in mathematic **!=** Function in programming

Pure Functions

Pure Functions (Mathematical functions)

- Produce the same result when given the same arguments (referential transparency)
- Have no side-effects
- Don’t change the state of the program

- Advantages
  - Easy to test and to refactor
  - The call sequence of functions can be changed
  - Automatically parallelizable
  - Results can be cached

Pure Functions

Working with pure functions is based on discipline

- Use common functions, meta-functions, `constexpr`, or `consteval` functions

  - Function

```
int powFunc(int m, int n){
    if (n == 0) return 1;
    return m * powFunc(m, n-1);
}
```

  - Meta-Function

```
template<int m, int n>
struct PowMeta {
    static int const value = m * PowMeta<m, n-1>::value;
};
template<int m>
struct PowMeta<m, 0> {
    static int const value = 1;
};
```

Pure Function

- `constexpr` Function (C++14)

```
constexpr int powConstexpr(int m, int n) {
    int r = 1;
    for(int k=1; k<=n; ++k) r*= m;
    return r;
}
```

- `consteval` Function (C++20)

```
consteval int powConsteval(int m, int n) {
    int r = 1;
    for(int k=1; k<=n; ++k) r*= m;
    return r;
}
```

➡ **1024 == powFunc(2, 10) == PowMeta<2, 10>()::value**
== powConstexpr(2, 10) == powConsteval(2, 10)

Callable

- Function
- **Function Overloading**
- Function Object
- Lambda Expression
- Coroutine

Function Overloading

Function overloading allows it to create multiple functions with the same name but different parameters.

- The compiler tries to find single best fit function based on overload resolution.

- Implementation:
  - The Compiler decorates the function names with the function parameters ➡ Name Mangling

Function	GCC 8.2	MSVC 19.16
`print(int)`	`_Z5printi`	`?print@@YAXH@Z`
`print(double)`	`_Z5printd`	`?print@@YAXN@Z`
`print(const char*)`	`_Z5printPKc`	`?print@@YAXPEBD@Z`
`print(int, double, const char*)`	`_Z5printidPKc`	`?print@@YAXHNPEBD@Z`

Function Overloading

- The single best fitting function
  - Functions: Fewer and less costlier conversions are better
  - Functions and function template specialisations: functions are better
  - Function template specialization: More specialized function template are better
  - Concepts: More constrained function templates are better

The full story: [Overload resolution on cppreference.com]()

Function Overloading

- Templates

```
void onlyDouble(double) {}

template <typename T>
void onlyDouble(T) = delete;

int main() {
    onlyDouble(3.14);      // OK
    onlyDouble(3.14f);     // ERROR
}
```

Function Overloading

- Concepts

```
template<std::forward_iterator I> void advance(I& iter, int n) {}
template<std::bidirectional_iterator I> void advance(I& iter, int n){}
template<std::random_access_iterator I> void advance(I& iter, int n){}

std::forward_list myFL {1, 2, 3};
std::list  myL{1, 2, 3};
std::vector myV{1, 2, 3};

std::list<int>::iterator lIt = myL.begin();
advance(lIt, 1);        // std::bidirectional_iterator

std::vector<int>::iterator vIt = myV.begin();
advance(vIt, 1);        // std::random_access_iterator

std::forward_list<int>::iterator fwIt = myFL.begin();
advance(fwIt, 1);       // std::forward_iterator
```

Callable

- Function
- Function Overloading
- **Function Object**
- Lambda Expression
- Coroutine

Function Object

A function object (aka functor) is an object that can be invoked such as a function.

- A function object can have state.

- Implementation:
  - The compiler maps a function call on an object `obj` `obj(arguments)` onto the function call operator `obj.operator()(arguments)`.
  - Function objects analyzed with C++ Insights

Function Object

- Operators
  - Arithmetic
    - `std::plus, std::minus, std::multiplies, std::divides, std::modulus, std::negate`
  - Comparisons
    - `std::equal_to, std::not_equal_to, std::greater; std::less, std::greater_equal, std::less_equal`
  - Logical
    - `std::logical_and, std::logical_or, std::logical_not`
  - Bitwise
    - `std::bit_and, std::bit_or, std::bit_xor, std::bit_not`
- A few examples on C++ Insights.

Function Object

- Reference Wrappers: `std::reference_wrapper<type>`
  - Stores a reference in a copyable function object
  - Two helper functions:
    - `std::ref` creates a reference wrapper
    - `std::cref` creates a constant reference wrapper

```
#include <functional>
#include <vector>

int main() {
    int a{2011};
    std::vector<std::reference_wrapper<int>> myIntVec{ std::ref(a) };
    a = 2014;
    myIntVec[0] << std::endl;    // 2014
}
```

Function Object

- Function wrapper and partial function application
  - `std::function`: wraps callable objects with specified function call signature
  - `std::bind`: binds arguments to a function object
  - `std::bind_front`: binds arguments, in order, to a function object

Function Object

```
using namespace std::placeholders;

std::function<int(int)> minus1 = std::bind(std::minus<int>(), 2020, _1);
std::cout << minus1(9);         // 2011

std::function<int()> minus2 = std::bind(minus1, 9);
std::cout << minus2();          // 2011

std::function<int(int, int)> minus3 = std::bind(std::minus<int>(), _2, _1);
std::cout << minus3(9, 2020); // 2011

std::function<int(int)> plus1 = std::bind_front(std::plus<int>(), 2000);
std::cout << plus1(20);         // 2020

std::function<int()> plus2 = std::bind_front(std::plus<int>(), 2000, 20);
std::cout << plus2();           // 2020
```

Callable

- Function
- Function Overloading
- Function Object
- **Lambda Expression**
- Coroutine

Lambda Expression

A lambda expression (anonymous function) is a function definition that is not bound to an identifier.

- Steps in the evolution of lambdas
  - C++11: Lambda expressions
  - C++14: Generic lambda expressions
  - C++20: Template parameters for lambda expressions

- Implementation:
  - The compiler generates a class with an overloaded function call operator.
  - Lambda Expressions with C++ Insights

Lambda Expression

- Template parameters for lambda expressions

```
auto lambdaGeneric = [](const auto& container) { return container.size(); };

auto lambdaVector = []<typename T>(const std::vector<T>& vec) {
    return vec.size();
};

auto lambdaVectorIntegral = []<std::integral T>(const std::vector<T>& vec) {
    return vec.size();
};

std::deque<int> deq;               // OK for lambdaGeneric
std::vector<double> vecDouble;     // OK for lambdaGeneric, lambdaVector
std::vector<int> vecInt;           // OK for lambdaGeneric,
                                   //        lambdaVector, lambdaVectorIntegral
```

Callable

- Function
- Function Overloading
- Function Object
- Lambda Expression
- **Coroutine**

Coroutine

Coroutine are generalized subroutines that can be suspended and resumed.

- A coroutine consists of
  - A promise object to manipulate it from the inside
  - A handle to resume or destroy it from the outside
  - State containing (dynamic allocated)
    - Parameters
    - Representation of the suspension point
    - Locals and temporaries

Coroutine

- Implementation:
  - The compiler transforms the coroutine (having `co_return`, `co_yield`, or `co_await`) to something such-as the following

```
{
    Promise promise;
    co_await promise.initial_suspend();
    try {
        <function body>
    }
    catch (...) {
       promise.unhandled_exception();
    }
FinalSuspend:
    co_await promise.final_suspend();
}
```

Coroutine

- Implementation (`infiniteDataStream.cpp`):
  - Coroutine execution triggers the following steps
    - Coroutine activation frame is allocated
    - Parameters are copied to the coroutine activation frame
    - Promise `prom` created
    - Generator `gen` created
    - Handle to the promise created and locally stored `prom.get_return_object()`
    - Promise initially suspended: `prom.initial_suspend()`
    - Coroutine reaches suspension point
      - Generator `gen` is returned to the caller
    - Caller resumes the coroutine using the generator and ask for the next value: `gen.nextValue()`
    - Generator `gen` destroyed
    - Promise `prom` destroyed

Callable

- Function
- Function Overloading
- Function Object
- Lambda Expression
- Coroutine

Future Directions

- Uniform function call syntax
  - `x.f(y)` and `f(x, y)` should be equivalent

  - Syntax based on the proposal (Bjarne Stroustrup and Herb Sutter, 2015) [N4474.pdf]

    - General strategy
      - `x.f(y):` First look for `x.f(y)`, then for `f(x, y)`
      - `f(x, y):` First look for `f(x, y)`, then for `x.f(y)`

    - Pointers
      - `p->f(y)` and `f(p, y)` should be equivalent

Future Directions

- Benefit of the universal function call syntax
  - I don‘t know if `f` is a function or a member function.
    ⇨ Calling `x.f()` or `f(x)` is just equivalent.
  - Instead of `f(x)`, I want to call `x.f()`.
    ⇨ Convenient syntax for function composition (fluent interface)

```
std::string startString = "Only for testing purpose.";

std::string upperString = upper(startString);
std::vector<std::string> upperStrings = split(upperString);

std::vector<std::string> upperStrings = startString.upper().split();

// upperStrings = ['ONLY', 'FOR', 'TESTING', 'PURPOSE.']
```