CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Advanced C++ Course

Functional and generic programming utilities

CSCS

`std::tuple` is simple, why do we care?

- Generic programming in C++ is in many cases "remove as many constraints as possible": the fewer constraints, the more generic
  - *Don't overdo it, if there isn't a use case for it*
- Sometimes we introduce constraints without realizing it, sometimes we introduce bugs without realizing it
  - Important to understand the semantics and subtleties of the basic utilities when applying them to real problems
- This session will cover a basic set of C++ tools that are useful in generic programming, including many functional programming utilities

Warmup

- What are the requirements on `T`?

```
template <typename T>
void f(T&&) noexcept {
    // I am the most generic function, but I can't do anything
}
```

Warmup

- What are the requirements on `T`?

```
template <typename T>
void f(T&& t) noexcept {
   std::cout << t << '\n';
}
```

Warmup

- What are the requirements on `T`?

```
template <typename T>
void f(T&& t) noexcept {
    t.foo();
}
```

Warmup

- What are the requirements on `T`?

```
template <typename T>
void f(T&& t, bool flag) noexcept {
    if (flag) {
        t = T{};
    }
}
```

Warmup

- What are the requirements on `T` and `U`?

```
template <typename T, typename U>
void f(T&& t, U&& u) noexcept {
    t = u;
}
```

Warmup

- What are the requirements on `T` and `U`?

```
template <typename T, typename U>
void f(T&& t, U&& u) noexcept {
    t = std::forward<U>(u);
}
```

Warmup

- Constraints come in different forms
- Not only "has member function `foo`"
- Also:
  - Default constructibility
  - Copyability
  - Movability
  - Comparability
  - ...

Session overview

- C++ standard library basics:
  - `std::tuple` for storing a compile-time known number of potentially homogeneous types
    - by far the most commonly used utility in generic programming
  - `std::optional` for storing up to one type
  - `std::variant` for storing one of a compile-time known number of potentially homogeneous types
- Functional utilities
  - Lambdas and other function objects
  - Partial application, function invocation, etc.
- Finally: all of the above together

`std::tuple` : what is it not good for?

- If all elements in the tuple are known to be of the same type, prefer `std::array`

```cpp
std::array<int, 3> a{42, 43, 44};

// not
std::tuple<int, int, int> t{42, 43, 44};
```

`std::tuple` : what is it not good for?

- If you can give names to the members, prefer a struct

```
struct interval {
    double begin;
    double end;
};

interval i{1.0, 13.5};
// use i.begin and i.end

// not
using interval = std::tuple<double, double>
interval i{1.0, 13.5};
// use std::get<0>(i) and std::get<1>(i)
```

`std::tuple` : what is it good for? Generic programming!

- `std::tuple<Ts...>` 👍
- `std::tuple<T1, T2>` 🤦
- `std::tuple<int, double>` 👎

`std::tuple` : what is it good for?

- `std::tuple<Ts...>` : great for storing arguments for later use
- Common pattern to separate:
  - Description of work
  - Execution of work
- For example:
  - Store a task for later execution on a thread pool
  - Store a CUDA kernel for later execution with a given stream

```
template <typename... Ts>
struct mytype {
    // Unfortunately we can't do this
    // Ts... ts;
    // But we can do this
    std::tuple<Ts...> ts;
};
```

`std::tuple` : Kernel launcher

- Full CUDA example: https://godbolt.org/z/cqnE6WzM8

```
__global__ void fill(int* array, int n, int x);

int main() {
    int* array; int n; int x;

    auto k = make_kernel_launcher(
        256, (n + 256 - 1) / 256,
        fill,
        array, n, x);

    {
        // Initialize stream etc.
        cudaStream_t stream{};
        k(stream);
    }
}
```

```
template <typename F, typename... Ts>
struct kernel_launcher {
    int block_dim;
    int grid_dim;

    std::decay_t<F> f;
    std::tuple<std::decay_t<Ts>...> t;

    void operator()(cudaStream_t stream) { /* TODO */ }
};

template <typename F, typename... Ts>
auto make_kernel_launcher(
    int block_dim, int grid_dim, F&& f, Ts&&... ts) {
    return kernel_launcher<F, Ts...>(
        block_dim, grid_dim,
        std::forward<F>(f),
        std::tuple<std::decay_t<Ts>...>(std::forward<Ts>(ts)...)
    );
}
```

`std::tuple` : constructing

`Ts...`	`auto t =`	`decltype(t)`
`int, double&, mytype`	`std::tuple(ts...)`	`std::tuple<int, double, mytype>`
`int, double&, mytype`	`std::make_tuple(ts...)`	`std::tuple<int, double, mytype>`
`int, double&, mytype`	`std::tuple<std::decay_t<Ts>...>(ts...)`	`std::tuple<int, double, mytype>`
`int, double&, mytype`	`std::tuple<Ts...>(ts...)`	`std::tuple<int, &double, mytype>`
`int, double&, mytype`	`std::forward_as_tuple(ts...)`	`std::tuple<int&, double&, mytype&>`

`std::tuple` : constructing but with forwarding

`Ts...`	`auto t =`	`decltype(t)`
`int, double&, mytype`	`std::tuple(std::forward<Ts>(ts)...)`	`std::tuple<int, double, mytype>`
`int, double&, mytype`	`std::make_tuple(std::forward<Ts>(ts)...)`	`std::tuple<int, double, mytype>`
`int, double&, mytype`	`std::tuple<std::decay_t<Ts>...>(std::forward<Ts>(ts)...)`	`std::tuple<int, double, mytype>`
`int, double&, mytype`	`std::tuple<Ts...>(std::forward<Ts>(ts)...)`	`std::tuple<int, &double, mytype>`
`int, double&, mytype`	`std::forward_as_tuple(std::forward<Ts>(ts)...)`	`std::tuple<int&&, double&, mytype&&>`

`std::optional`

- Traditional use case: failure

```
template <typename T>
T div(T x, T y) noexcept {
    return x / y; // May be undefined behaviour
}

template <typename T>
std::optional<T> safeish_div(T x, T y) noexcept {
    if (y == T{0}) { return std::nullopt; }
    else { return x / y; }
}
```

- However, `std::expected` (C++23) or exceptions are generally still a better choice because they can tell you *why* something failed

`std::optional`

- Traditional use case: missing value

```
std::vector<int> v1;
int x1 = v1.back(); // Undefined behaviour

myvector<int> v2;
// Use of x2 requires explicit checking that it's valid
std::optional<int> x2 = v2.back();
```

`std::optional`

- Generic programming use case: storing non-default-constructible types
- For example: value filled in asynchronously by another thread

```
struct mytype {
    mytype() = delete;
    mytype(int x) : x(x) {}
    // copy and move constructors/assignment
};
template <typename T>
struct mycontainer {
    // Requires that T is default-constructible
    T x{};
};
// mycontainer<mytype> c{}; // not ok
mycontainer<mytype> c{mytype(42)}; // ok
```

```
struct mytype {
    mytype() = delete;
    mytype(int x) : x(x) {}
    // copy and move constructors/assignment
};
template <typename T>
struct mycontainer2 {
    // x can be filled in later without constraints on T
    std::optional<T> x;
};
mycontainer<mytype> c{}; // ok
mycontainer<mytype> c{mytype(42)}; // ok
```

`std::variant`

- Closed set of homogeneous types, one is active
- Like union, but type safe

Setting

```
std::variant<int, std::string> v{"hello"};
// The second alternative is now active
std::print("Alternative {} is active\n", v.index());
// Is the first alternative active?
std::print("Is the first alternative active: {}\n",
    std::holds_alternative<T>(v));
```

Getting

```
// Access with std::get
std::string x1 = std::get<1>(v); // ok
std::string x2 = std::get<std::string>(v); // ok
// throws std::bad_variant_access
// std::string y1 = std::get<0>(v);
// throws std::bad_variant_access
// std::string y2 = std::get<int>(v);
```

Visiting

```
// Or with a generic function
std::visit(visitor, v);
```

std::variant

- Use case: abstract syntax tree
- Full example: https://godbolt.org/z/dG1jb7x84

```
template <typename... Ts>
using up = std::unique_ptr<Ts...>;

struct lit; struct add; struct mul;
using ast = std::variant<lit, up<add>, up<mul>>;

struct lit { int x; };
struct add { ast x, y; };
struct mul { ast x, y; };

int eval(ast const&);
struct visitor {
    auto operator()(lit const& l) const { return l.x; }
    auto operator()(up<add> const& a) const { return eval(a->x) + eval(a->y); }
    auto operator()(up<mul> const& m) const { return eval(m->x) * eval(m->y); }
};
int eval(ast const& a) { return std::visit(visitor{}, a); }
```

`std::variant`

- `std::monostate` : an empty tag type that can be used to make `std::variant` default constructible

```
// First type is active after default construction
// Fails to compile if mytype is not default constructible
std::variant<mytype, int> v;
```

```
// Always compiles, no matter what mytype is
std::variant<std::monostate, mytype, int> v;
```

`std::variant`

- Use case: Implementing `std::optional`!

- ```
  template <typename T>
  struct optional {
      std::variant<std::monostate, T> v;
  };
  ```
  ```

Questions about `tuple`, `optional`, or `variant`?

Functional utilities

- Not just functions: C++ has many other things that behave like a function
  - lambdas, `operator()`, `std::function`, `std::bind_front`
- And many utilities that operate on functions or are useful in conjunction with those utilities:
  - `std::invoke`, `std::apply`, `std::reference_wrapper`

Lambda use case 1: algorithms as higher-order functions

- Sometimes functions take another function as a parameter: "higher order function"
- Treating functions as data
- Example: `transform` takes a range as input and modifies the elements with a function
- What can we use in place of `???`

```
std::vector<int> x{1, 4, 7, 1, 4};
std::ranges::transform(x, ???);
```

Lambda use case 1: algorithms as higher-order functions

- Sometimes functions take another function as a parameter: "higher order function"
- Treating functions as data
- Example: `transform` takes a range as input and modifies the elements with a function
- We can pass regular functions into `transform`

```
int triple(int x) { return 3 * x; }

std::vector<int> x{1, 4, 7, 1, 4};
std::ranges::transform(x, triple);
```

Lambda use case 1: algorithms as higher-order functions

- Sometimes functions take another function as a parameter: "higher order function"
- Treating functions as data
- Example: `transform` takes a range as input and modifies the elements with a function
- We can also use *lambdas*: "function literals"

```
std::vector<int> x{1, 4, 7, 1, 4};
std::ranges::transform(x, [](int x) { return 3 * x; });
```

Lambda use case 1: algorithms as higher-order functions

- Sometimes functions take another function as a parameter: "higher order function"
- Treating functions as data
- Example: `transform` takes a range as input and modifies the elements with a function
- Lambdas can also be assigned to variables, but we don't know their type

```
std::vector<int> x{1, 4, 7, 1, 4};
auto triple = [](int x) { return 3 * x; };
std::ranges::transform(x, triple);
```

Lambda use case 2: adding state

- Sometimes we want to use variables from outside the lambda/function
- Example: `partition` partitions an input range into two parts based on a predicate
- We can again use functions or lambdas in place of `???`

```
std::vector<int> x{1, 4, 7, 1, 4};
// partition vector into even and odd
std::ranges::partition(x, ???);
```

Lambda use case 2: adding state

- Sometimes we want to use variables from outside the lambda/function
- Example: `partition` partitions an input range into two parts based on a predicate
- We can again use functions or lambdas in place of `???`

```
bool even(int x) { return x % 2 == 0; }

std::vector<int> x{1, 4, 7, 1, 4};
// partition vector into even and odd
std::ranges::partition(x, even);
```

Lambda use case 2: adding state

- Sometimes we want to use variables from outside the lambda/function
- Example: `partition` partitions an input range into two parts based on a predicate
- We can again use functions or lambdas in place of `???`

```
std::vector<int> x{1, 4, 7, 1, 4};
// partition vector into even and odd
auto even = [](int x) { return x % 2 == 0; };
std::ranges::partition(x, even);
```

Lambda use case 2: adding state

- Sometimes we want to use variables from outside the lambda/function
- Example: `partition` partitions an input range into two parts based on a predicate
- How do we use `pivot` in `pred`?

```
std::vector<int> x{1, 4, 7, 1, 4};
int pivot = get_pivot(v);
// partition vector into bigger and smaller than pivot
auto pred = [](int x) { return x > pivot; }; // Does not compile!
std::ranges::partition(x, pred);
```

Lambda use case 2: adding state

- Sometimes we want to use variables from outside the lambda/function
- Example: `partition` partitions an input range into two parts based on a predicate
- `[]` allows "capturing" variables from outer scope
  - Lambdas also sometimes called "closures", because they "close over" the environment where they are defined

```
std::vector<int> x{1, 4, 7, 1, 4};
int pivot = get_pivot(v);
// partition vector into bigger and smaller than pivot
auto pred = [pivot](int x) { return x > pivot; };
std::ranges::partition(x, pred);
```

Lambda use case 3: C++ overloads and function templates

- Sometimes we want to pass not only one function to another function, but a whole "overload set", i.e. multiple functions with the same name
- Example: print in another thread with `std::async`

```cpp
void print(double);
void print(struct important_data);
template <typename T>
void print(T);

std::vector<int> v{1, 2, 3, 4};
auto all_prints = &print; // Does not work!
auto one_print = &print<std::vector<int>>; // Ok, but only one function
auto f = std::async(all_prints, v);
```

Lambda use case 3: C++ overloads and function templates

- Sometimes we want to pass not only one function to another function, but a whole "overload set", i.e. multiple functions with the same name
- Example: print in another thread with `std::async`
- Can only take the address of one specific overload
- Lambdas can help us delay the choice of overload
  - Advanced: sometimes abstracted away as a macro (https://github.com/rollbear/lift/blob/3927d06415f930956341afd5bc223f912042d7e4/include/lift.hpp#L20-L29)

```
void print(double);
void print(struct important_data);
template <typename T>
void print(T);

std::vector<int> v{1, 2, 3, 4};
auto all_prints = [](auto x) { print(x); };
auto f = std::async(all_prints, v);
```

Lambdas, formally

https://en.cppreference.com/w/cpp/language/lambda

Syntax

[*captures*] (*params*) *specs* *requires*(optional) { *body* }	(1)	
[*captures*] *attr* (*params*) *specs* *requires*(optional) { *body* }	(1)	(since C++23)
[*captures*] { *body* }	(2)	
[*captures*] *attr* *specs* { *body* }	(2)	(since C++23)
[*captures*] < *tparams* > *requires*(optional) (*params*) *specs* *requires*(optional) { *body* }	(3)	(since C++20)
[*captures*] < *tparams* > *requires*(optional) *attr* (*params*) *specs* *requires*(optional) { *body* }	(3)	(since C++23)
[*captures*] < *tparams* > *requires*(optional) { *body* }	(4)	(since C++20)
[*captures*] < *tparams* > *requires*(optional) *attr* *specs* { *body* }	(4)	(since C++23)

1) Full form.
2) Omitted parameter list: function takes no arguments, as if the parameter list were ().
3) Same as (1), but specifies a generic lambda and explicitly provides a list of template parameters.
4) Same as (2), but specifies a generic lambda and explicitly provides a list of template parameters.

Lambdas, formally

- Lambdas are syntax sugar over structs with `operator()`, i.e. the call operator
- The following are equivalent, except we don't know the type of `real_lambda`

```
struct my_lambda {
   int x;

   auto operator()(int y) const { return x * y; }
}

int x = 42;
auto emulated_lambda = my_lambda{x};
auto real_lambda = [x](int y) { return x * y; };
```

Lambdas, formally

- Captures can be implicit or explicit, by value or by reference
- Capture `x` explicitly by reference

```
struct my_lambda {
   int& x;

   auto operator()(int y) const { return x * y; }
}

int x = 42;
auto emulated_lambda = my_lambda{x};
auto real_lambda = [&x](int y) { return x * y; };
```

Lambdas, formally

- Captures can be implicit or explicit, by value or by reference
- Capture `x` implicitly by reference, `z` explicitly by value

```
struct my_lambda {
   int& x;
   int z;

   auto operator()(int y) const { return x * y * z; }
}

int x = 42;
int z = 3;
auto emulated_lambda = my_lambda{x, z};
auto real_lambda = [&, z](int y) { return x * y * z; };
```

Lambdas, formally

- Captures can be implicit or explicit, by value or by reference
- Capture `x` explicitly by reference, `z` implicitly by value

```
struct my_lambda {
   int& x;
   int z;

   auto operator()(int y) const { return x * y * z; }
}

int x = 42;
int z = 3;
auto emulated_lambda = my_lambda{x, z};
auto real_lambda = [=, &x](int y) { return x * y * z; };
```

Lambdas, formally

- Captures can be implicit or explicit, by value or by reference
- Capture `this` by value or reference

```
class my_class {
   int x;

   auto f() {
      // *this copied into lambda capture
      auto t = std::thread([*this](){ /* ... */ });
      t.detach();
   }

   auto g() {
      // this is a pointer, beware dangling pointer access!
      auto t = std::thread([this]() { /* ... */ });
      t.detach();
   }
};
```

Lambdas, formally

- Captures can be given new names inside the lambda body with initializer syntax
- Can capture by value and by reference also with initializers

```
void g(std::tuple<int, double> const& t);

void f(std::tuple<int, double>&& t1) {
    std::jthread([t2 = std::move(t1)]() {
        g(t2);
    });
}
```

Lambdas, formally

- Which overload is called below?

```
void g(std::tuple<int, double>&& t);
void g(std::tuple<int, double> const& t);

void f(std::tuple<int, double>&& t1) {
    std::jthread([t2 = std::move(t1)]() {
        g(std::move(t2));
    });
}
```

Lambdas, formally

- Which overload is called below?
- Call operator is const by default
  - `g(std::tuple<int, double> const&)` is called!
  - https://godbolt.org/z/xs4cdYh8f

```
void g(std::tuple<int, double>&& t);
void g(std::tuple<int, double> const& t);

void f(std::tuple<int, double>&& t1) {
    std::jthread([t2 = std::move(t1)]() {
        g(std::move(t2));
    });
}
```

Lambdas, formally

- Call operator is const by default
- But can be made `mutable`

```
struct my_lambda {
   type x;

   auto operator()() /* const */ { return g(std::move(x)); }
}

type x{};
auto emulated_lambda = my_lambda{std::move(x)};
auto real_lambda = [x = std::move(x)](int y) mutable { return g(std::move(x)); };
```

Lambdas, formally

- Lambda parameters can be `auto` (or `auto&` or `auto&&`) since C++14
- Generates a templated `operator()` for you

```
struct my_lambda {
   template <typename T1, typename T2>
   auto operator(T1 x1, T2 x2)() {}
}

auto emulated_lambda = my_lambda{};
auto real_lambda = [](auto x1, auto x2) {};
```

Lambdas, formally

- Lambda parameters can be `auto` (or `auto&` or `auto&&`) since C++14
- Generates a templated `operator()` for you
- `auto&&` acts as a forwarding reference

```
struct my_lambda {
   template <typename T1, typename T2>
   auto operator(T1&& x1, T2 x2)() {}
}

auto emulated_lambda = my_lambda{};
auto real_lambda = [](auto&& x1, auto x2) {};
```

Lambdas, formally

- Lambda parameters can be `auto` (or `auto&` or `auto&&`) since C++14
- Generates a templated `operator()` for you
- Since C++20 a lambda can be explicitly templated

```
struct my_lambda {
   template <typename T1, typename T2>
   auto operator(T1&& x1, T2 x2)() {}
}

auto emulated_lambda = my_lambda{};
auto real_lambda = []<typename T2>(auto&& x1, T2 x2) {};
```

Lambdas, formally

- Capturing packs can be done with tuples or explicit packs since C++20

Since C++20:

```
template <typename... Ts>
void f(Ts&&... ts) {
    auto ff1 = [ts...]() {};
    auto ff2 = [...ts = std::forward<Ts>(ts)]() {};
}
```

In C++14:

```
template <typename... Ts>
void f(Ts&&... ts) {
    auto ff1 = [t = std::tuple<std::decay_t<Ts>...>(ts...)]() {};
    auto ff2 = [t = std::tuple<std::decay_t<Ts>...>(std::forward<Ts>(ts)...)]() {};
}
```

Lambdas, formally

- `auto` is the default return type and often sufficient
- Return type can be specified explicitly with the trailing `->` syntax
- Can be used for SFINAE simply not accidentally returning the wrong type

```
struct my_lambda {
   int operator(int x1, int x2)() { return x + y; }
}

auto emulated_lambda = my_lambda{};
auto real_lambda = [](int x, int y) -> int { return x + y; };
```

`std::function` use case: API boundary with `std::function`

- Not everything needs to be templated etc. for optimal performance
- A usable API may be more important
- Example: register a function to be called at startup of a library
- We can use function pointers

```
void register_startup_handler(void(*)(const configuration&));
void print_config(const configuration&);

register_startup_handler(print_config);
```

`std::function` use case: API boundary with `std::function`

- Not everything needs to be templated etc. for optimal performance
- A usable API may be more important
- Example: register a function to be called at startup of a library
- We can use function pointers, but lambdas are not function pointers (most of the time: https://godbolt.org/z/Wb6xx6Kz1)

```
void register_startup_handler(void(*)(const configuration&));

auto print_config = [](const configuration&) { /* do stuff */ };
register_startup_handler(print_config);
```

`std::function` use case: API boundary with `std::function`

- Not everything needs to be templated etc. for optimal performance
- A usable API may be more important
- Example: register a function to be called at startup of a library
- We can use function pointers, but lambdas are not function pointers (most of the time: https://godbolt.org/z/Wb6xx6Kz1)

```
void register_startup_handler(void(*)(const configuration&));

int x = 42;
auto print_config = [x](const configuration&) { /* do stuff with x as well */ };
register_startup_handler(print_config); // Does not work!
```

`std::function` use case: API boundary with `std::function`

- Not everything needs to be templated etc. for optimal performance
- A usable API may be more important
- Example: register a function to be called at startup of a library
- We can template, but not necessary in this case

```
template <typename F>
void register_startup_handler(F&&);

int x = 42;
auto print_config = [x](const configuration&) { /* do stuff with x as well */ };
register_startup_handler(print_config);
```

`std::function` use case: API boundary with `std::function`

- Not everything needs to be templated etc. for optimal performance
- A usable API may be more important
- Example: register a function to be called at startup of a library
- `std::function` : type-erased callable wrapper
  - Takes function anything that looks like it could be callable with the right signature

```
void register_startup_handler(std::function<void(const configuration&)>);

int x = 42;
auto print_config = [x](const configuration&) { /* do stuff with x as well */ };
register_startup_handler(print_config);
```

`std::function`, formally

- Type-erased callable wrapper
- Type-erasure implies heap-allocation and virtual functions and in turn some overhead
- Will not be inlined
- Can hide implementation in source file

Callables, summarized

- Function pointers: use for C compatibility or if you want to guarantee stateless callables
- Take by generic template parameter if you want the most generic interface
  - Can constrain with `std::invocable` if necessary
- Take by `std::function` if you want a simple API where the implementation can be hidden in a source file

Function parameter	Plain function	Stateless lambda	Stateful lambda	`std::function`
Function pointer	ok	ok	no	maybe
Template	ok	ok	ok	ok
`std::function`	ok	ok	ok	ok

Other useful functional utilities

`std::bind_front` : "hard-code" the first arguments of a function

- `std::bind_front` exists since C++20
- Does a "partial application" of the function
- `std::bind_front` will not implicitly call the function when all arguments have been supplied (not even possible in the general case)
- `std::bind` also exists, but prefer lambdas or `std::bind_front` whenever possible; `std::bind` has hairy corner cases that make it error-prone: https://godbolt.org/z/xPE6fa1K9

```
int f(double, std::string);
int x = 42; std::string y = "hello";

// bind_front reduces the arity of the function by the number of (non-function) arguments passed to bind_front
std::bind_front(f)(x, y);
std::bind_front(f, x)(y);
std::bind_front(f, x, y)();
```

`std::apply` : unpack a tuple into separate arguments

- Similar to unpacking with `*` in Python
- Passes each element of tuple-like objects as separate arguments to the callable

```cpp
int f(int, std::string);

std::tuple<int, std::string> t{42, "hello"};

// Equivalent to f(42, "hello")
std::apply(f, t);
```

`std::apply` : unpack a tuple into separate arguments

- Useful together with `std::bind_front`

```cpp
int f(double, int, std::string);

std::tuple<int, std::string> t{42, "hello"};

// Equivalent to f(3.14, 42, "hello")
std::apply(std::bind_front(f, 3.14), t));
```

`std::apply` exercise: finish `kernel_launcher`

- See exercise `tuple_storage_apply`

```
template <typename F, typename... Ts>
struct kernel_launcher {
    int block_dim;
    int grid_dim;

    std::decay_t<F> f;
    std::tuple<std::decay_t<Ts>...> t;

    void operator()(cudaStream_t stream) { /* TODO */ }
};
```

`std::apply` exercise: implement it

- See exercise `apply`
- Example implementation: https://godbolt.org/z/6nzd1cjn6

```
// Equivalent to f(42, 3.14)
std::apply(f, std::tuple(42, 3.14));
```

`std::apply` exercise: constructing tuples correctly

- See exercise `apply_tuple_bug`
- Is the following correct (equivalent to `g(ts...)`) for all `Ts`?

```
template <typename... Ts>
auto f(Ts... ts) {
    return std::apply(g, std::tuple(ts...));
}
```

`std::tuple` : constructing, with a twist

`Ts...`	`auto t =`	`decltype(t)`
`std::tuple<int, double&, mytype>`	`std::tuple(ts...)`	`std::tuple<int, double&, mytype>`
`std::tuple<int, double&, mytype>`	`std::make_tuple(ts...)`	`std::tuple<std::tuple<int, double&, mytype>>`
`std::tuple<int, double&, mytype>`	`std::tuple<std::decay_t<Ts>...>(ts...)`	`std::tuple<std::tuple<int, double&, mytype>>`
`std::tuple<int, double&, mytype>`	`std::tuple<Ts...>(ts...)`	`std::tuple<std::tuple<int, &double, mytype>>`
`std::tuple<int, double&, mytype>`	`std::forward_as_tuple(ts...)`	`std::tuple<std::tuple<int, double&, mytype>&&>`

`std::(c)ref` : obtain a copyable (const) reference to an object

- Value semantics like pointers, i.e. can be rebound
- Non-nullable like references
- `std::(c)ref` are functions which return `std::reference_wrapper<T>`
- Useful to opt-in to references in places that normally don't allow references

```
int f(int&, std::string);

int x = 42;
std::string y = "hello";

std::bind_front(f, std::ref(x))(y);
```

Lambda hack: Immediately invoked function/lambda expression

- Can (ab)use lambdas for complex initialization
- Ternary operator has no equivalent with `if-else`

```
auto x = pred() ? y : z;
```

Lambda hack: Immediately invoked function/lambda expression

- Can (ab)use lambdas for complex initialization
- Can't return only from a scope, only from the whole function

```
auto x = if (pred()) {
    ??? y; // nothing we can do here
} else {
    ??? z; // nothing we can do here
}
```

Lambda hack: Immediately invoked function/lambda expression

- Can (ab)use lambdas for complex initialization
- Can default initialize and then assign

```
// We have to know the type of x
T x;
if (pred()) {
    x = y;
} else {
    x = z;
}
```

Lambda hack: Immediately invoked function/lambda expression

- Can (ab)use lambdas for complex initialization
- We can return from a lambda without returning from the outer function!

```
// We can also make x const
const auto x = [&]() {
    if (pred()) {
        return y; // return from lambda, not from outer scope!
    } else {
        return z; // return from lambda, not from outer scope!
    }
}();
```

`overloaded` exercise: How does the following work?

- See exercise `ast`

```
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;

int main() {
    std::variant<int, std::string> v{"hi!"};
    std::visit(overloaded([](int x) { std::cout << "variant holds " << x << '\n'; },
                          [](std::string x) { std::cout << "variant holds " << x << '\n'; }),
               v)
}
```

Bonus: What does this do?

- https://quuxplusone.github.io/blog/2018/05/17/super-elider-round-2/
- https://akrzemi1.wordpress.com/2018/05/16/rvalues-redefined/
- https://godbolt.org/z/qh7sEePT5

```
template <typename F>
class foo
{
    F&& f;

public:
    explicit foo(F&& f) : f(std::forward<F>(f)) {}

    using type = std::invoke_result_t<F&&>;
    operator type() { return std::forward<F>(f)(); }
};
```