TEMPLATES - ADVANCED TECHNIQUES

CODERS SCHOOL

ŁUKASZ ZIOBROŃ

AGENDA

1. Intro and pre-test
2. Deduction guides
3. Overload resolution rules
4. SFINAE - `std::enable_if`
5. Template template parameters
6. Variadic templates
7. Fold expressions
8. CRTP
9. Recap

REPO GH `coders-school/templates`

https://github.com/coders-school/templates/tree/cr/module2

```
git clone -b cr https://github.com/coders-school/templates.git
```

SOMETHING ABOUT YOU

- Have you ever seen/used the SFINAE technique?
- Have you heard about a static polymorphism?
- What do you expect from today's session?

ŁUKASZ ZIOBROŃ

NOT ONLY A PROGRAMMING XP

- Front-end dev, DevOps & Owner @ Coders School
- C++ and Python developer @ Nokia & Credit Suisse
- Team leader & Trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Web developer (HTML, PHP, CSS) @ StarCraft Area

EXPERIENCE AS A TRAINER

- C++ online course @ Coders School
- Company trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr & UWr
- Nokia Academy @ Nokia

PUBLIC SPEAKING EXPERIENCE

- code::dive conference
- code::dive community
- Academic Championships in Team Programming
- Coders School YouTube channel

HOBBIES

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy

CONTRACT

- 🎰 Vegas rule
- 🗣️ Discussion, not a lecture
- ☕ Additional breaks on demand
- ⌚ Be on time after breaks

PRE-TEST 1/3 🤯

IS THIS CODE CORRECT?

```
template<typename... Args> bool f(Args... args) {
    return (args && ... && args);
}
```

1. Yes
2. No, the parameter pack is used twice in fold-expression
3. No, the ellipsis operator is used incorrectly in fold-expression
4. No, `Args&&` should be used as a function argument

PRE-TEST 2/3 🤯

WHAT IS THE OUTPUT OF THIS PROGRAM?

```
#include <iostream>

template <typename T>
void fun(T)
{
  static int i;
  std::cout << ++i;
}

int main()
{
  fun(1);
  fun(1.0);
  fun(2);
}
```

PRE-TEST 3/3 🤯

MARK TRUE STATEMENTS ABOUT THIS CODE

```
template<typename T> class MyDeleter {};
template<typename T, typename Policy> class Handle { /* ... */ };
Handle<FILE, MyDeleter> h;
```

1. This code compiles
2. `MyDeleter` cannot be used as a `Policy` template argument
3. `template<typename T, template typename Policy>` - this should make this code work
4. `template<typename T, template <typename ...> typename Policy>` - this should make this code work
5. `template<typename T, template <typename> typename Policy>` - this should make this code work

DEDUCTION GUIDES

CODERS SCHOOL

TEMPLATE TYPE DEDUCTION FOR CLASSES (C++17)

Template types can be automatically deduced by the compiler thanks to... template functions.

The compiler uses class constructors to achieve that.

From C++17 you can write a code like this:

```
std::vector v = {1, 2, 3};  // std::vector<int> is deduced
std::list l{'c', 'd', 'b'}; // std::list<char> is deduced
```

That's not gonna work:

```
// std::vector v1;          // compilation error, vector of what?
// std::vector v2(10)       // compilation error, 10 elements of what?
```

USER-DEFINED DEDUCTION GUIDES

- The syntax is similar to function declaration with a trailing return type, except that it uses the name of a class template as the function name.
- It's not a function and does not have a body.

```
template<class T>
struct container {
    container(T t);
    template<class Iter> container(Iter beg, Iter end);
};

// additional deduction guide
template<class Iter>
container(Iter b, Iter e) -> container<typename std::iterator_traits<Iter>::value_type>;
```

```
container c1{2.71};                       // deduces T = double using an automatic guide
std::vector v = {1, 2, 3, 4};             // deduces std::vector<int>
container c2{v.begin(), v.end()};         // deduces T = int
auto c3 = container{v.begin(), v.end()};  // deduces T = int
//container c4{5, 6};                     // Error: no std::iterator_traits<int>::value_type
```

EXAMPLE WITH `std::variant`

```
#include <functional>
#include <iostream>
#include <string>
#include <variant>

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

// Deduction guide
template <typename... Ts>
make_visitor(Ts...) -> make_visitor<Ts...>;

using variant = std::variant<int, std::string, double>;

int main()
{
  {
    const auto v0 = variant{42.2};

    std::visit(make_visitor{
      [](int){std::cout << "int\n";},
```

Source

MORE ON CPPREFERENCE.COM

OVERLOAD RESOLUTION RULES

CODERS SCHOOL

OVERLOAD RESOLUTION

```
1 void foo(unsigned i) {
2     std::cout << "unsigned " << i << "\n";
3 }
4
5 template <typename T>
6 void foo(const T& t) {
7     std::cout << "template " << t << "\n";
8 }
```

What is the result of calling `foo(42)`?

`template 42`

Why `const int &` is a better match than `unsigned`?

Const reference can bind to r-values. No conversion is needed, there is an exact match, so this option is chosen.

OVERLOAD RESOLUTION

```
1 void foo(unsigned i) {
2     std::cout << "unsigned " << i << "\n";
3 }
4
5 template <typename T>
6 void foo(T& t) {
7     std::cout << "template " << t << "\n";
8 }
```

What is the result of calling `foo(42)` now?

`unsigned 42`

Why `unsigned` is a better match than `int &`?

Reference (non-const) cannot bind to r-values. There is only one function matching. Implicit conversion from int to unsigned is applied.

OVERLOAD RESOLUTION

```
1 void foo(unsigned i) {
2     std::cout << "unsigned " << i << "\n";
3 }
4
5 void foo(double i) {
6     std::cout << "double " << i << "\n";
7 }
```

What is the result of calling `foo(42)`?

`error: call of overloaded ‘foo(int)’ is ambiguous`

Why?

Promotion to double and conversion to unsigned are equally viable.

FUNCTION OVERLOADS VS FUNCTION SPECIALIZATIONS

```
template<class T> void f(T);     // #1: overload for all types
template<>        void f(int*); // #2: specialization of #1 for pointers to int
template<class T> void f(T*);    // #3: overload for all pointer types

f(new int{1});
```

WHICH FUNCTION IS CHOSEN?

It calls #3, even though specialization of #1 would be a perfect match.

RULES

- Only non-template and primary template overloads participate in overload resolution
- The specializations are not overloads and are not considered
- Only after the overload resolution selects the best-matching primary function template, its specializations are examined to see if one is a better match

SFINAE

CODERS SCHOOL

SIMPLE SFINAE EXAMPLE

```
template <typename T>
void foo(T arg) {}

template <typename T>
void foo(T* arg) {}
```

Calling `foo(42)` makes a compiler to generate two functions.

```
void foo(int arg) {}
void foo(int* arg) {}
```

The compiler cannot substitute `42` as an argument to the second function. It would cause a compilation error. Therefore this overload is discarded.

There is no compilation error - this is how SFINAE works.

If the first function is missing, there is a compilation error.

SFINAE

Substitution Failure Is Not An Error is a meta-programming technique.

> *This rule applies during overload resolution of function templates: When substituting the explicitly specified or deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error.*
>
> -- *cppreference.com*

Rationale: have a universal interface without letting the caller to decide which implementation should be called. The selection of an optimal implementation is done by a compiler and is coded by library creators.

`type_traits` LIBRARY

<type_traits> on cppreference.com

std::enable_if

C++11 has a metaprogramming helper struct - `std::enable_if`. It is a compile-time switch for enabling or disabling some templates.

```
template <bool Condition, class T = void>;
struct enable_if {};

template <class T>
struct enable_if<true, T> { using type = T; };
```

- If `Condition` is `true`, accessing internal type by `enable_if<Condition, T>::type` is valid.
- If `Condition` is `false`, accessing internal type by `enable_if<Condition, T>::type` is invalid and substitution is not correct - SFINAE works.

std::enable_if_t

C++14 defines a helper type:

```
1 template <bool B, class T = void>
2 using enable_if_t = typename enable_if<B, T>::type;
```

Using both is equivalent.

```
1 template <
2     typename T,
3     typename std::enable_if<std::is_integral<T>::value, T>::type* = nullptr
4 > void function(T& t) {}
```

```
1 template <
2     typename T,
3     typename std::enable_if_t<std::is_integral_v<T>, T>* = nullptr
4 > void function(T& t) {}
```

Why `* = nullptr`?

enable_if VARIATIONS

```
template<class T>     // #1 return type
auto construct(T* t) ->
typename std::enable_if_t<std::has_virtual_destructor_v<T>, T>*
{ return new T{}; }
```

```
template<class T>     // #2 parameter
T* construct(
    T* t,
    typename std::enable_if_t<std::has_virtual_destructor_v<T>>* = nullptr
) { return new T{}; }
```

```
template<
    class T,          // #3 template parameter - usual choice from C++11
    typename std::enable_if_t<std::has_virtual_destructor_v<T>>* = nullptr
> T* construct(T* t)
{ return new T{}; }
```

enable_if VARIATIONS

The most elegant way

```
template <typename T>
using HasVirtDtor = std::enable_if_t<std::has_virtual_destructor_v<T>>;
```

```
template<
    class T,            // the same as #3 - template parameter
    typename = HasVirtDtor<T>
> T* construct(T* t)
{ return new T{}; }
```

CONCEPTS (C++20)

```
template <class T>
concept HasVirtualDtor = std::has_virtual_destructor_v<T>;

template <HasVirtualDtor T>
T* construct(T* t)
{ return new T{}; }
```

From C++20 concepts should replace the usage of `std::enable_if`.

TASK

Take a look into `module2/shapes` directory.

Write an `insert()` function in `main.cpp`. It should allow inserting only subclasses of Shape to the collection. Parameters other than `Circle`, `Square` or `Rectangle` should not compile. Use SFINAE. Use proper type_traits.

The function should create a `shared_ptr` from the object passed as the first parameter and put it in the collection, which should be provided as the second parameter.

Hints:

- `std::is_base_of`
- `std::remove_reference`
- `std::remove_cv`

SOLUTION

```
template <typename T>
using removeCvRef = std::remove_cv_t<std::remove_reference_t<T>>;
template <typename T>
using isBaseOfShape = std::enable_if_t<std::is_base_of_v<Shape, removeCvRef<T>>>;

template <
    class T,
    typename = isBaseOfShape<T>
>
void insert(T&& item, Collection& collection) {
    collection.emplace_back(make_shared<removeCvRef<T>>(item));
}
```

if constexpr

```
template <typename T>
using removeCvRef = std::remove_cv_t<std::remove_reference_t<T>>;

template <class T>
void insertC(T&& item, Collection& collection) {
    if constexpr (std::is_base_of_v<Shape, removeCvRef<T>>) {
        collection.emplace_back(make_shared<removeCvRef<T>>(item));
    } else {
        std::cout << "Sorry\n";
    }
}
```

- can replace SFINAE mechanism
- available from C++17
- the result is not a substitution error, but usually empty code

CONCLUSIONS

- SFINAE is a very powerful and difficult technique
- Proper usage may not be too ugly 😉
- Where applicable, use below techniques instead of SFINAE:
  - tag dispatch
  - `static_assert` (C++11)
  - `if constexpr` (C++17)
  - concepts (C++20)

TEMPLATE TEMPLATE PARAMETERS

CODERS SCHOOL

EXAMPLE

We want to use a template class as a template argument.

```
template<typename T> class MyDeleter {};
template<typename T, typename Policy> class Handle { /* ... */ };
Handle<FILE, MyDeleter> h;
```

It causes a compilation error:

```
error: type/value mismatch at argument 2 in template parameter list for ‘template<class T, class Policy> class Handle’
```

We tried to pass a template class instead of a specific class (like instantiated template class). This should work:

```
Handle<FILE, MyDeleter<FILE>> h;
```

TEMPLATE OF TEMPLATE

To allow using a template class, we need to use so-called `template template` arguments.

```
template<typename T> class MyDeleter {};
template<typename T, template<typename> typename Policy> class Handle { /* ... */ };
// template<typename T, typename Policy> class Handle { /* ... */ };  // previously
Handle<FILE, MyDeleter> h;
```

TEMPLATE TEMPLATE PARAMETERS

```
template<template<parameter-list> typename|class name> declaration
template<template<parameter-list> typename|class> declaration

template<template<parameter-list> typename|class name = default> declaration
template<template<parameter-list> typename|class = default> declaration

template<template<parameter-list> typename|class ... name> declaration
template<template<parameter-list> typename|class ...> declaration
```

MORE EXAMPLES

```
template<typename T1, typename T2>
struct data {
    T1 val1;
    T2 val2;
};
```

```
template<size_t N, template<typename, typename> typename Data, typename T1, typename T2>
struct data_array {
  std::array<Data<T1, T2>, N> data;
};
```

```
data_array<16, data, int, double> a1;
a1.data[0].val1 = 4;
a1.data[0].val2 = 3.14;

data_array<8, std::pair, std::string, std::string> a2;
a2.data[0].first = "Mickey";
a2.data[0].second = "Mouse";
```

TASK

1. Write a template class `Holder`.
    1. The class should have a `vector<T>` as a member field.
    2. The class should have a constructor, that can take any of 4 sequential containers (`vector`, `list`, `forward_list`, `deque`). This constructor should copy all values to the internal vector.
    3. The template parameter of the class `T` should be automatically deduced. Write a deduction guide if needed.
2. Add a print function for printing the content of the internal `vector<T>` field
3. Add support of `std::array<T, N>`. The constructor should allow copying the content to the internal vector.

SOLUTION #1 (EASIER)

```
template <typename T>
class Holder {
    std::vector<T> v_;
public:
    template <typename Container>
    Holder(const Container & c)
        : v_{c.begin(), c.end()}
    {}

    void print() {
        std::copy(v_.begin(), v_.end(), std::ostream_iterator<T>(std::cout, " "));
    }
};

template <typename Container>
Holder(const Container & c) -> Holder<typename Container::value_type>;
```

- No template template arguments
- Simple deduction guide

SOLUTION #2

```
template <typename T, template <typename...> typename Container>
class Holder {
    std::vector<T> v_;
public:
    Holder(const Container<T> & c)
        : v_{c.begin(), c.end()}
    {}

    template <size_t N>
    Holder(const std::array<T, N> & a)
        : v_{a.begin(), a.end()}
    {}

    void print() {
        std::copy(v_.begin(), v_.end(), std::ostream_iterator<T>(std::cout, " "));
    }
};

template <typename T, size_t N>
Holder(const std::array<T, N> & a) -> Holder<T, std::vector>;
```

- Template template arguments
- Additional constructor for `std::array`
- Additional and counterintuitive deduction guide for array

VARIADIC TEMPLATES

CODERS SCHOOL

MOTIVATION

Variadic templates can be used to create template functions or classes that accept any numer of arguments of any type.

Do you know `printf()` function (from C)?

```
#include <cstdio>

int main() {
   printf("Hello %s, you are %d years old\n", "John", 25);
   printf("Just a text\n");
   return 0;
}
```

PARAMETER PACK

A template parameter pack is a template parameter that accepts zero or more template arguments

```
template<typename ... Types> struct Tuple {};
Tuple<> t0;             // Types contains no arguments
Tuple<int> t1;          // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float
```

```
template<typename ... Types> void f(Types ... args);
f();                  // OK: args contains no arguments
f(1);                 // OK: args contains one argument: int
f(2, 1.0);            // OK: args contains two arguments: int and double
```

SYNTAX

Templates with a variable number of arguments use the new syntax of parameter pack, which represents many (or zero) parameters of a template.

```
template<class ... Types>
void variadic_foo(Types&& ... args)
{ /*...*/ }

int main() {
    variadic_foo(1, "", 2u);
    return 0;
}
```

```
template<class ... Types>
class variadic_class
{ /*...*/ };

int main() {
    variadic_class<float, int, std::string> v1{};  // default c-tor
    variadic_class v2{2.0, 5, "Hello"}; // automatic deduction from C++17
}
```

UNPACKING FUNCTION PARAMETERS

Unpacking group parameters uses new syntax of elipsis operator - `...`

In the case of function arguments, it unpacks them in the order given in the template function call.

TWO WAYS OF PASSING ARGUMENTS

```
template <typename... Args>
void print_all(Args&&... args) {
    printMany(args...);
    // expands to printMany(arg1, arg2, ..., argN);
}

template <typename... Args>
void print_each(Args&&... args) {
    (printOne(args), ...);  // fold expression from C++17
    // expands to printOne(arg1), printOne(arg2), ..., printOne(argN);
}

int main() {
    print_all("Hello!", "Lukin");
    print_each("Hello!", "Lukin");
    return 0;
}
```

UNPACKING PARAMETER TYPES

```
Foo...                  => Foo1, Foo2, Foo3, etc
vector<Foo>...          => vector<Foo1>, vector<Foo2>, vector<Foo3>, etc
tuple<Foo...>(bar...) => tuple<Foo1, Foo2, Foo3, etc>(bar1, bar2, bar3, etc)
&bar...                 => &bar1, &bar2, &bar3, etc
&&bar...                => &&bar1, &&bar2, &&bar3, etc
```

EXAMPLE WITH `std::variant`

```
#include <functional>
#include <iostream>
#include <string>
#include <variant>

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

// Deduction guide
template <typename... Ts>
make_visitor(Ts...) -> make_visitor<Ts...>;

using variant = std::variant<int, std::string, double>;

int main()
{
  {
    const auto v0 = variant{42.2};

    std::visit(make_visitor{
      [](int){std::cout << "int\n";},
```

Source

HEAD TAIL RECURSION

It is also possible to use recursion to unpack every single argument. It requires the variadic template Head/Tail and non-template function to be defined.

```
void variadic_foo() {}

template<class Head, class... Tail>
void variadic_foo(Head const& head, Tail const&... tail) {
    takesOneParam(head);    //action on head
    variadic_foo(tail...);
}
```

HEAD TAIL RECURSION IN CLASSES

It is possible to unpack all types at once (e.g. in the case of a base class that is a variadic template class) or using partial and full specializations.

```
template<int... Number>
struct Sum;

template<int Head, int... Tail>
struct Sum<Head, Tail...> {
    const static int RESULT = Head + Sum<Tail...>::RESULT;
};

template<>
struct Sum<> {
    const static int RESULT = 0;
}

constexpr auto value = Sum<1, 2, 3, 4, 5>::RESULT; // = 15
```

HANDLING INHERITANCE FROM VARIADIC CLASSES

```
template<class... Types>
struct Base
{};

template<class... Types>
struct Derived : Base<Types...>
{};
```

`sizeof...` OPERATOR

`sizeof...` returns the number of parameters in a parameter pack.

```
template<class... Types>
struct NumOfArguments {
    const static unsigned value = sizeof...(Types);
};

constexpr auto num = NumOfArguments<A, B, C>::value;  // 3
```

TASK

Write a `make_array()` function, that can create `std::array<T, N>` from all values passed to it.

```
std::array<int, 3> a = make_array(1, 2, 3);
std::array<double, 4> b = make_array(1.1, 2.2, 3.3, 4.4);
```

HINTS

- variadic function
- `sizeof...` to deduce size
- `std::common_type` to deduce the type
- trailing return type with `->`
- perfect forwarding
- `constexpr`

SOLUTION

```
#include <array>
#include <iostream>
#include <type_traits>

template<typename... Ts>
constexpr auto make_array(Ts&&... ts) -> std::array<std::common_type_t<Ts...>, sizeof...(Ts)> {
    return { std::forward<Ts>(ts)... };
}

int main() {
    auto b = make_array(1, 2, 3);
    std::cout << b.size() << '\n';
    for(auto i : b)
        std::cout << i << ' ';
}
```

PERFECT FORWARDING

It is recommended to use `&&` and `std::forward` with variadic arguments to handle r-values and l-values correctly. It is so-called **perfect forwarding**.

```
#include <utility>

template<typename Type>
void single_perfect_forwarding(Type&& t) {
    callable(std::forward<Type>(t));
}

template<typename... Types>
void variadic_perfect_forwarding(Types&&... args) {
    callable(std::forward<Types>(args)...);
}
```

FOLD EXPRESSIONS

CODERS SCHOOL

FOLD EXPRESSIONS (C++17)

Fold-expressions allow writing compact code with variadic templates without using explicit recursion.

```
template<typename... Args> auto sum(Args... args) {
    return (args + ...);
    // return (... + args);     // the same
    // return (0 + ... + args); // the same
    // return (args + ... + 0); // the same
    // return args + ...;       // error - missing parentheses
}

template<typename... Args> bool f(Args... args) {
    return (args && ... && args); // error: both operands
}                                 // contain unexpanded
                                  // parameter packs
```

FOLD-EXPRESSIONS

Reduces (folds) a parameter pack over a binary operator op.

OPERATORS

```
+ - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= ==
!= <= >= && || , .* ->*
```

- unary right fold
  - `(pack op ...)`
- binary right fold
  - `(pack op ... op init)`
- unary left fold
  - `(... op pack)`
- binary left fold
  - `(init op ... op pack)`

EXAMPLES

```
template<typename... Args>
bool all(Args... args) {
    return (... && args);
}

bool b = all(true, true, true, false);
```

```
template<typename T, typename... Args>
void push_back_vec(std::vector<T>& v, Args&&... args) {
    (v.push_back(args), ...);
}

std::vector<int> v;
push_back_vec(v, 6, 2, 45, 12);
```

TASK

1. Write a `print()` function that can print anything on a screen. You should be able to pass any number of parameters to it. Use fold expressions.

```
print(1, 40, "string", 2.0);
```

   Difficult version: try to add a space between elements.

2. Write an `areEven()` function, that

   - prints all parameters on screen using above `print()` function
   - check if all numbers in parameter pack are even and return a proper boolean value

```
bool allEven = areEven(2, 4, 6, 9);
```

SOLUTION

```
template<typename ...Args>
void print(Args&&... args) {
    (std::cout << ... << args) << '\n';
}

template<typename ...Args>
void printSpace(Args&&... args) {
    ((std::cout << " " << args), ...) << '\n';
}

template<typename... Numbers>
auto areEven(Numbers... nums) {
    print(nums...);
    return ((nums % 2 == 0) && ...);
}
```

FOLD EXPRESSIONS

Default values when parameter pack is empty

Operator	Value
`*`	`1`
`+`	`int()`
`&`	`-1`
`|`	`int()`
`&&`	`true`
`||`	`false`
`,`	`void()`

CRTP

CODERS SCHOOL

DYNAMIC POLYMORPHISM

```
1  class Base {
2      virtual void setup() = 0;
3      virtual void run() = 0;
4      virtual void cleanup() = 0;
5  public:
6      virtual ~Base() = default;
7      void process() {
8          setup();
9          run();
10         cleanup();
11     }
12 };
13
14 class Derived : public Base {
15     void setup() override   { /* ... */ }
16     void run() override     { /* ... */ }
17     void cleanup() override { /* ... */ }
18 };
```

STATIC POLYMORPHISM = CRTP

```
1  template<class Child>
2  class Base {
3  public:
4    void process()
5    {
6      static_cast<Child*>(this)->setup();
7      static_cast<Child*>(this)->run();
8      static_cast<Child*>(this)->cleanup();
9    }
10 };
11
12 class Derived : public Base<Derived> {  // CRTP
13   friend class Base<Derived>;
14   void setup()   { /* ... */ }
15   void run()     { /* ... */ }
16   void cleanup() { /* ... */ }
17 };
```

CRTP

- Curiously Recurring Template Pattern
- An idiom in C++ in which a class X derives from a class template instantiation using X itself as a template argument
- Sometimes called "Upside-Down Inheritance" due to the way it allows class hierarchies to be extended by substituting different base classes
- Compile time polymorphism at no extra cost (no vtable, no vptr)

POST-WORK

CODERS SCHOOL

make_unique

Implement `make_unique` function. It should be able to take any number of parameters and it should return a unique_ptr to the given type. The object of the given type should be created with all parameters passed to it. The main challenge is to properly unit tests for l-values and r-values.

```
struct MyType {
    MyType(int&, double&&, bool) { std::cout << "lvalue, rvalue, copy\n"; }
    MyType(int&&, double&, bool) { std::cout << "rvalue, lvalue, copy\n"; }
};

int main(){
    int lvalue{2020};
    std::unique_ptr<int> uniqZero = make_unique<int>();
    auto uniqEleven = make_unique<int>(2011);
    auto uniqTwenty = make_unique<int>(lvalue);
    auto uniqTypeLRC = make_unique<MyType>(lvalue, 3.14, true);
    auto uniqTypeRRC = make_unique<MyType>(2020, 3.14, true);
}
```

CODE REVIEW

Send me a link to the repository with your implementation for my code review.

LUKASZ@CODERS.SCHOOL

POST-TEST

The link will be sent after the training with all materials.

RECAP

CODERS SCHOOL

WHAT WAS THE MOST DIFFICULT THING?

WHAT HAVE YOU LEARNED TODAY?

POINTS TO REMEMBER

- Deduction guides help the compiler to deduce proper template classes
- SFINAE is used to disable code generation for a certain types
- To pass a template class to a template you need to use so-called template template arguments
- If you want a function that can take any number of arguments use variadic templates
- Use fold expressions to simplify operations on variadic arguments
- If possible, you can replace template functions with generic lambdas (from C++14). Just keep in mind that lambdas are objects and have a limited lifespan.
- If possible, use `if constexpr`, `static_assert` or concepts to emulate SFINAE.

PRE-TEST ANSWERS

PRE-TEST 1/3 🤯

IS THIS CODE CORRECT?

```
template<typename... Args> bool f(Args... args) {
    return (args && ... && args);
}
```

1. Yes
2. No, the parameter pack is used twice in fold-expression
3. No, the ellipsis operator is used incorrectly in fold-expression
4. No, `Args&&` should be used as a function argument

PRE-TEST 2/3 🤯

WHAT IS THE OUTPUT OF THIS PROGRAM?

```
#include <iostream>

template <typename T>
void fun(T)
{
  static int i;
  std::cout << ++i;
}

int main()
{
  fun(1);
  fun(1.0);
  fun(2);
}
```

112

PRE-TEST 3/3 🤯

MARK TRUE STATEMENTS ABOUT THIS CODE

```
template<typename T> class MyDeleter {};
template<typename T, typename Policy> class Handle { /* ... */ };
Handle<FILE, MyDeleter> h;
```

1. This code compiles
2. `MyDeleter` cannot be used as a `Policy` template argument
3. `template<typename T, template typename Policy>` - this should make this code work
4. `template<typename T, template <typename ...> typename Policy>` - this should make this code work
5. `template<typename T, template <typename> typename Policy>` - this should make this code work

RATE THIS TRAINING SESSION

CODERS SCHOOL