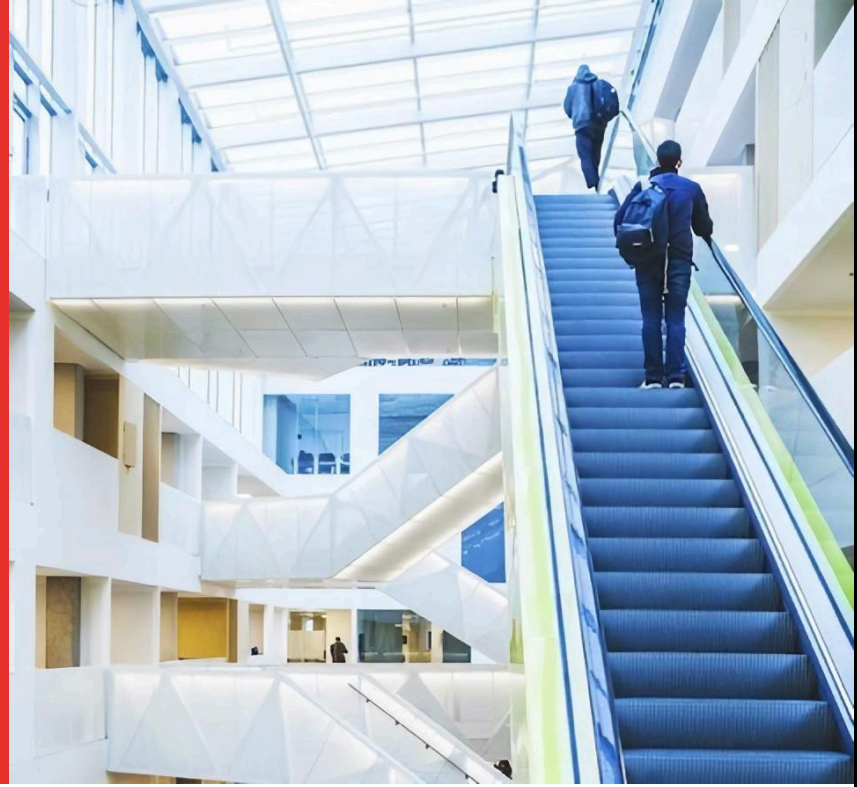
OVERZICHT LES

- Types
- Templates
- Variadic Templates
- SFINAE & if constexpr
- Constexpr / Consteval
- Type Traits & Concepts (C++20)



TYPES

C++ (Statisch)

- Types zijn compile-time vastgelegd
- Fouten worden vaak al tijdens compilatie ontdekt
- Templates genereren voor elk type een aparte instantie

```cpp
#include <iostream>

int main() {
    int a = 5;
    a = "Hello"; // Error
    std::cout << "a: " << a << "\n";
    return 0;
}
```

Python (Dynamisch)

- Types worden pas tijdens runtime bepaald
- Flexibeler, maar typefouten kunnen later optreden
- Gebruik van `type()` en duck typing

```python
def print_value(x):
    print("Value:", x)
    print("Type:", type(x))

print_value(5)
print_value("Hello")
```

TEMPLATES

Bepaal max van twee integers

```cpp
float max2(float a, float b) {
    return a > b ? a : b;
}

std::string max2(std::string a,
                 std::string b) {
    return a > b ? a : b;
}

gz::math::Color max2(gz::math::Color a,
                     gz::math::Color b) {
    return a > b ? a : b;
}
```

```python
def max(a, b):
    return a if a > b else b
```

FUNCTIE TEMPLATES

Bepaal max van twee integers

```cpp
auto max(auto a, auto b) {
    return a > b ? a : b;
}

std::cout << max(11, 12) << "\n";
std::cout << max(2.71828, 3.14159) << "\n";
std::cout << max(gz::math::Color::Blue,
                 gz::math::Color(0.3, 0.4, 0.5)
                )
          << "\n";
```

```python
def max(a, b):
    return a if a > b else b
```

FUNCTIE TEMPLATES GEBRUIK

```cpp
template <typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

std::cout << max<float>(42, 6.5) << "\n";
std::cout << max<int>(2.71828, 3.14159) << "\n";
std::cout << max(gz::math::Color::Blue,
                 gz::math::Color(0.3, 0.4, 0.5)
             )
          << "\n";
```

Je *kan* zelf template parameters opgeven.

(Er zitten overigens prima min en max in `<algorithm>`, de STL les volgt later)

Elke template instantiatie is zijn eigen ding, daarom moeten ze in een header staan:
https://godbolt.org/z/jY8sWMajn

Voordelen/Nadelen

- Specifieke optimalisaties per type
- Kan leiden tot code bloat als er veel instanties worden aangemaakt

CLASS TEMPLATES & TEMPLATE VALUE PARAMETERS

- C++ templates kunnen niet alleen types, maar ook waarden als parameters nemen
- Voorbeeld: Het definiëren van een array met vaste grootte

```cpp
template<typename T, std::size_t N>
struct Array {
    T data[N];
};
```

https://godbolt.org/z/7hxjf5odq

Gebruik

`Array<int, 10>` definieert een array van 10 integers.

Voordeel

- Compile-time vastgelegde constante waarden kunnen maximaal geoptimaliseerd worden

Toepassing

- Gebruikt in containers zoals `std::array`.

VECTOR<BOOL>

- `std::vector<bool>` is een bekende specialisatie in de STL
- In plaats van een array van booleans (1 byte per bool) wordt een bit-packed representatie gebruikt

```cpp
template<typename T, std::size_t N>
class Array {
    T data[N];
};

template <std::size_t N>
class Array<bool, N> {
    unsigned char data[(N + 7) / 8];

    bool get(std::size_t i) const {
        return ((data[i / 8] >> (i % 8)) & 1) != 0;
    }

    void set(std::size_t i, bool b){
        if (b) {
            data[i / 8] |=  (1 << (i % 8));
        } else {
            data[i / 8] &= ~(1 << (i % 8));
        }
    }
};
```

FUNCTIE TEMPLATES & SPECIALISATIES

```cpp
#include <iostream>

// Algemene template voor logging
template<typename T>
void log(const T& value) {
    std::cout << "Log: " << value << "\n";
}

// Specialisatie voor C-strings (const char*)
template<>
void log<const char *>(const char * const value) {
    std::cout << "Log (C-string): " << value << "\n";
}
```

VARIADIC TEMPLATES: LOGGER VOOR MEERDERE ARGUMENTEN

```cpp
#include <iostream>

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

https://godbolt.org/z/hhfEen3sE

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

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

SFINAE VOORBEELD

```cpp
template<typename T>
struct is_integral_manual {
    static constexpr bool value = false;
};

template<> struct is_integral_manual<int> { static constexpr bool value = true; };
template<> struct is_integral_manual<unsigned long> { static constexpr bool value = true; };

template<bool isIntegral, typename T>
struct log_sfinae_helper;

template<typename T>
struct log_sfinae_helper<true, T> {
    static void log(const T& value) {
        std::cout << "Integral value: " << value << "\n";
    }
};

template<typename T>
struct log_sfinae_helper<false, T> {
    static void log(const T&) {
        std::cout << "Non-integral value\n";
    }
};

template<typename T>
void log_sfinae(const T& value) {
    log_sfinae_helper<is_integral_manual<T>::value, T>::log(value);
}
```

SFINAE & IF CONSTEXPR: CONDITIEEL COMPILEREN

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

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

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

SFINAE & IF CONSTEXPR: CONDITIEEL COMPILEREN

https://godbolt.org/z/MfP6ajM6n

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
void log_if(const T& value) {
    if constexpr (std::is_arithmetic_v<T>) {
        std::cout << "Arithmetic: " << value << "\n";
    } else {
        std::cout << "Non-arithmetic type\n";
    }
}

template<typename T>
auto log_sfinae(const T& value) -> std::enable_if_t<std::is_integral_v<T>> {
    std::cout << "Integral value: " << value << "\n";
}

template<typename T>
auto log_sfinae(const T& value) -> std::enable_if_t<!std::is_integral_v<T>> {
    std::cout << "Non-integral value\n";
```