

# C++17

by

Dr. Günter Kolousek

# Neuerungen – eine Auswahl...

- ▶ structured bindings
- ▶ template deduction
- ▶ variable definitions inside `if`, `switch`
- ▶ `string_view`
- ▶ `optional`, `any`, `variant`
- ▶ guaranteed copy elision
- ▶ `inline` variables
- ▶ parallel algorithms!
- ▶ filesystem!
- ▶ `byte`

# structured bindings

```
#include <iostream> // structured_bindings.cpp
using namespace std;
int main() {
    int arr[2]{1, 1};

    // creates an array int tmp[2]
    // copies arr into tmp
    // x == &tmp[0], y == &tmp[1]
    auto [x, y]{arr};
    arr[0] = 2;
    cout << x << endl; // -> 1

    // xr == &arr[0], yr == &arr[1]
    auto& [xr, yr] = arr;
    arr[0] = 3;
    cout << xr << endl; // -> 3
}
```

# structured bindings – 2

```
#include <iostream> // structured_bindings2.cpp
#include <tuple>
using namespace std;
int main() {
    int x{1};
    float y{2.5};
    string z{"abc"};

    const auto& [a, b, c]{
        tuple<int, float&, string>{x, y, move(z)};

        cout<< a <<' ' << b <<' ' << c <<endl; // 1 2.5 abc
        // a = 0; // -> const!
        x = 42;
        cout << "a=" << a << endl; // -> 1
        b = 3.5;
        cout << "y=" << y << endl; // -> 3.5
        // c[0] = "x"; -> const!
        cout << "z=' ' << z << "' ' << endl; // ' '
    }
}
```

# structured bindings – 3

```
#include <iostream> // structured_bindings3.cpp
using namespace std;
struct User {
    string name;  int age;
};

User get_user() { return User{"maxi", 42}; }

int main() {
    string name;
    int age;
    [name, age]{get_user()};
    cout << name << ' ' << age << endl;
    auto [name2, age2]{get_user()};
}
```

# template deduction

```
#include <iostream> // class_template_deduction.cpp
#include <vector>
#include <utility> // pair
using namespace std;
int main() {
    vector v{1, 2, 3, 4}; // vector<int>
    pair p{1, 2.5}; // pair<int, double>

    cout << p.first << ' ' << p.second << endl;
}
```

# variables inside if, switch

```
#include <iostream> // variables_if.cpp
using namespace std;
```

```
int foo() {
    return 1;
}
```

```
int main() {
    if (int i{foo()}; i != 0) {
        cout << i << endl;
    }
}
```

# string\_view

*immutable view, copyable, cheap!!!*

```
#include <iostream> // string_view.cpp
```

```
#include <string_view>
```

```
using namespace std;
```

```
void f(string_view v) { cout << v << endl; }
```

```
int main() {  
    char cstr[]{"!hello!"}; string str{"world"};  
    string_view cstr_view{cstr + 1, 5};  
    string_view str_view{str};  
  
    cout << cstr_view << ' ';  
    f(str_view); // -> hello world  
    cout << cstr_view[0] << endl; // -> h  
    // cstr_view[0] = 'x'; immutable  
}
```



# string\_view-2

```
#include <iostream> // string_view2.cpp
#include <string_view>
using namespace std;

int main() {
    string str{"hello world"};
    string_view str_view{str}; // no copying!
    string_view str_view2{str_view.substr(0, 5)};
    cout << str_view2 << endl; // -> hello
    cout << str_view2.find("l") << endl; // -> 2
}
```

# optional

```
#include <iostream> // optional.cpp
#include <optional>
using namespace std;
optional<int> result(bool answer) {
    if (answer) return optional<int>{42};
    else return nullopt; // not set
}
int main() { // operator*: unchecked!
    cout << *result(false) << endl; //-> 2 =>has to be checked!
    try { // value() checked
        cout << result(false).value();
    } catch (bad_optional_access& e) {
        cout << e.what() << endl; // -> bad optional access
    }
    if (!result(false))
        cout << "not set!!!" << endl;
    cout << result(false).value_or(-1) << endl; // -> -1
    auto answer{result(true)};
    if (answer) // here we check it!!!
        cout << "answer:" << *answer << endl; // 42
}
```

# any

```
#include <vector>
#include <any>
#include <iostream> // any.cpp
using namespace std;

int main() {
    cout << boolalpha;
    vector<any> v{true, 2017, string{"abc"}, 3.14};
    cout << "any_cast<bool>v[0]: "
        << any_cast<bool>(v[0]) << endl; // true
    cout << "any_cast<int>v[0]: "
        << any_cast<int>(v[1]) << endl; // 2017
    cout << "v[0].type().name(): "
        << v[0].type().name() << endl; // b
    cout << "v[1].type().name(): "
        << v[1].type().name() << endl; // i
}
```

# variant

```
#include <iostream> // variant.cpp
using namespace std;
#include <variant>
int main() {
    variant<int, double, string> v;
    cout << get<int>(v) << endl; // -> 0
    try {
        cout << get<double>(v) << endl;
    } catch (bad_variant_access& e) {
        cerr << e.what() << endl; // -> Unexpected index
    }
    v = 42;
    cout << get<int>(v) << endl; // 42
    v = 3.1415926;
    cout << get<double>(v) << endl; // 3.14159
    cout << get<1>(v) << endl; // 3.14159
    v = "abc";
    cout << get<string>(v) << endl; // abc
}
```

# guaranteed copy elision

```
#include <iostream> // copy_elision.cpp
using namespace std;

struct Data {
    Data() = default;
    Data(const Data&) {
        cout << "copy cons" << endl; }
    int a;
    double b;
};

Data f() {
    return Data{};
}

int main() {
    Data d{f()}; // *no* output at all
}
```

# inline variables

```
#ifndef PERSON_H
#define PERSON_H

struct Person { // inline_variables.h
    public:
        int id{};
        static inline int next_id{};
        Person() : id{next_id++} {}
};

inline Person root;
#endif
```

# inline variables – 2

```
#include <iostream> // inline_variables.cpp
using namespace std;
```

```
#include "inline_variables.h"
```

```
int main() {
    cout << Person::next_id << endl;
    Person p1;
    cout << p1.id << endl;
    cout << Person::next_id << endl;
}
```

# byte

- ▶ nur als Schnittstelle zum Speicher
  - ▶ kein arithmetischer Typ
  - ▶ Größe wie char
- ▶ aus ganzer Zahl: `byte b{123};`
- ▶ in ganze Zahl: `int i{b.to_integer()};`
  - ▶ auch `static_cast<int>` möglich
- ▶ keine Arithmetik, aber bitweise Operationen
  - ▶ `&, |, ^, !, &=, |=, ^=`
  - ▶ `>>=, >>, <<, <<=`