



Uniwersytet
Wrocławski

Inne nowości w C++11/14

Zbigniew Koza

Wydział Fizyki i Astronomii

auto

- **auto** = „kompilatorze, sam się domyśl, jaki jest typ tej zmiennej”

```
#include <iostream>
#include <list>

int main()
{
    std::list<std::string> lista {"Ala", "Ela", "Olek"};
    for (auto it = lista.begin(); it != lista.end(); ++it)
        std::cout << *it << "\n";
}
```



```
Ala
Ela
Olek
```

auto jako wartość funkcji

```
#include <iostream>

// błąd w c++11, w c++14 jest OK
auto minimum(int a, int b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << minimum(2, 5) << "\n";
}
```

- C++11: error

```
2.auto.cpp:4:26: error: 'minimum' function uses 'auto' type specifier without trailing return type
auto minimum(int a, int b)
                        ^
2.auto.cpp:4:26: note: deduced return type only available with -std=c++14 or -std=gnu++14
```

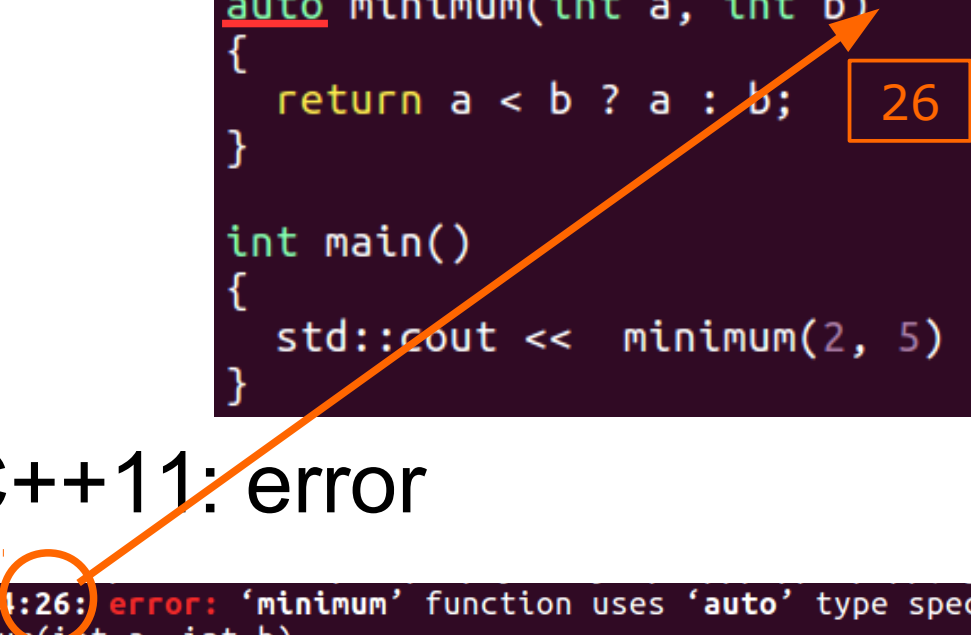
- C++14: OK

„Trailing return type”?

```
#include <iostream>

// błąd w c++11, w c++14 jest OK
auto minimum(int a, int b)
{
    return a < b ? a : b;
}

int main()
{
    std::cout << minimum(2, 5) << "\n";
}
```



- C++11: error

```
2.auto.cpp:4:26: error: 'minimum' function uses 'auto' type specifier without trailing return type
  auto minimum(int a, int b)
                        ^
2.auto.cpp:4:26: note: deduced return type only available with -std=c++14 or -std=gnu++14
```

Trailing return type

- C++11:

```
auto minimum(int a, int b) -> int
{
    return a < b ? a : b;
}
```

Jaki w tym sens?

- C++14

```
auto minimum(int a, int b)
{
    return a < b ? a : b;
}
```

Działa

Trailing return type + **decltype**

```
#include <iostream>

template <typename T1, typename T2>
auto suma(T1 a, T2 b) -> decltype(a + b)
{
    return a + b;
}

int main()
{
    std::cout << suma(2, 5) << "\n";
    std::cout << suma(2, 5.5) << "\n";
    std::cout << suma('a', 5) << "\n";
}
```

- Teraz to zaczyna mieć sens

Trailing return type + **decltype**

```
#include <iostream>

template <typename T1, typename T2>
auto suma(T1 a, T2 b) -> decltype(a + b)
{
    return a + b;
}

int main()
{
    std::cout << suma(2, 5) << "\n";
    std::cout << suma(2, 5.5) << "\n";
    std::cout << suma('a', 5) << "\n";
}
```

- Teraz to zaczyna mieć sens

a i b użyte
po swojej
deklaracji

decltype

```
#include <iostream>

int main()
{
    decltype('a' + 2) x = 7;
    std::cout << x << "\n";
}
```

Typem x jest typ
wyrażenia 'a' + 7

- Użyteczne zwłaszcza w szablonach

Jak wyświetlić typ funkcji?

```
int print(int const& a)
{
    std::cout << a << "\n";
    std::cout << __FUNCTION__ << "\n";
    std::cout << __PRETTY_FUNCTION__ << "\n";
}

int main()
{
    print(0);
}
```



```
0
print
int print(const int&)
```

- To działa nawet w C++98

Jak wyświetlić typ zmiennej lub wyrażenia?

```
#include <iostream>
#include <list>
#include <typeinfo> // z c++ 98

int main()
{
    std::list<std::string> lista {"Ala", "Ela", "Olek"};
    auto it = lista.begin();
    std::cout << "1) " << typeid(lista).name() << "\n";
    std::cout << "2) " << typeid(it).name() << "\n";
    std::cout << "3) " << typeid(*it).name() << "\n";
    std::cout << "4) " << typeid('a' + 88).name() << "\n";
}
```



```
1) NSt7__cxx114listINS_12basic_stringIcSt11char_traitsIcESaIcEEEEIS5_EEE
2) St14_List_iteratorINSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEE
3) NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
4) i
```

- To też działa nawet w C++98

Llanfairpwllgwyngyllgogerychwyrndrobwlllantysiliogogoch?

```
1) NSt7__cxx114listINS_12basic_stringIcSt11char_traitsIcESaIcEEEEsaIS5_EEE
2) St14_List_iteratorINSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEE
3) NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
4) i
```



```
./a.out | c++filt -t
```



```
1) std::__cxx11::list<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, s
td::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >
2) std::_List_iterator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >
3) std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
4) int
```

- To też jest znane z C++98

Dekorowanie nazw / *name mangling*

Przykład:

- „Dekorowanie nazw (ang. name mangling, name decoration) – technika stosowana przez kompilatory współczesnych języków programowania w celu wygenerowania unikatowych nazw funkcji, struktur, klas oraz innych typów danych (wikipedia)”

Dekorowanie nazw / *name mangling*

```
1) NSt7__cxx114listINS_12basic_stringIcSt11char_traitsIcESaIcEEEEIS5_EEE  
2) St14_List_iteratorINSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEE  
3) NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE  
4) i
```

- Udekorowane nazwy to jest to, co widzi „linker”. Dzięki niej może on rozróżniać funkcje o tych samych nazwach i łączyć pliki obiektowe wygenerowane różnymi kompilatorami lub z programów napisanych w innych językach, np. C i C++.

extern "C" {...}

```
#ifdef __cplusplus
extern "C" {
#endif

void *memset (void *, int, size_t);
char *strcat (char *, const char *);
int  strcmp (const char *, const char *);
char *strcpy (char *, const char *);

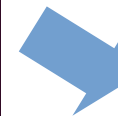
#ifdef __cplusplus
}
#endif
```

- Sposób na łączenie C++ z bibliotekami napisanymi w C (znany w C++98)

constexpr

```
constexpr int silnia(int n)
{
    return n <= 1 ? 1 : n*silnia(n-1);
}

int main()
{
    constexpr int N = 100;
    int a[N]; // OK
    int b[silnia(5)]; // OK
    std::cout << sizeof(b)/sizeof(b[0]) << "\n";
    int n;
    std::cin >> n;
    std::cout << silnia(n) << "\t" << &N << "\n";
}
```



```
120
10
3628800 0x7ffc77060cf8
```

- Wskazówka, że wyrażenie może być wartością stałą znaną w czasie kompilacji

constexpr a const

- constexpr ma nieco większą „moc” niż const
- W praktyce optymalizujący kompilator może traktować wiele użycí const jak constexpr
- Główna różnica: składnia (i **szablony!!!**)

nullptr

- Wskaźniki zerowy w C: **NULL** (== 0)
- Wskaźniki zerowy w C++98: **0**
- Wskaźnik zerowy w C++11: **nullptr**

```
void f(int);  
void f(char *);  
int main()  
{  
    f(0);           //Która f będzie wywołana?  
    f(NULL);        //Która f będzie wywołana?  
    f(nullptr);     //Która f będzie wywołana?  
}
```

nullptr

- Nowy literał oznaczający wskaźnik „zerowy” (niewskazujący żadnego ważnego o adresu)

Pętle for

```
#include <iostream>
#include <map>

int main()
{
    std::map<int, std::string> map;
    map[1] = "jeden";
    map[11] = "jedenaście";

    for(const auto& para : map)
    {
        std::cout << para.second << "\n";
        for (auto c : para.second)
            if (c == 'e')
                std::cout << "e\n";
    }

    int tab[] {1,2,3,4,5};
    for(int &j : tab)
    {
        j = j*j;
    }
}
```



- Uproszczenie zapisu

```
jeden
e
e
jedenaście
e
e
e
```

override i final

```
class B
{
    public:
        virtual void f() {std::cout << "B::f" << "\n";}
};

class C : public B
{
    public:
        virtual void f() override final {std::cout << "C::f" << "\n";}
};

class D : public C
{
    public:
        virtual void f() override {std::cout << "D::f" << "\n";}
};
```

```
11_override.cpp:18:18: error: virtual function 'virtual void D::f()'
    virtual void f() override {std::cout << "D::f" << "\n";}
                   ^
```

```
11_override.cpp:12:18: error: overriding final function 'virtual void C::f()'
    virtual void f() override final {std::cout << "C::f" << "\n";}
                   ^
```

final

```
class A final
{
    public:
        int a;
};

class B : A
{
    public:
        int f() { return a; }
};
```



```
12_override.cpp:9:7: error: cannot derive from 'final' base 'A' in derived type 'B'
    class B : A
    ^
```

- Może też zapobiegać dalszemu dziedziczeniu klasy

Final jako technika optymalizacji

```
class A
{
    public:
        virtual int f() = 0;
};

class B : A
{
    public:
        int f() override final {return 2;}
        int g() { return 2 + f(); }
};
```

Kompilator może
wywołać f() „inline”

- „dewirtualizacja”

override i final

- **override**

pomaga uniknąć
wielu bardzo
nieprzyjemnych
błędów

- **final**

blokuje
dziedziczenie
i służy głównie
optymalizacji kodu



Funkcje wirtualne

shared_ptr, unique_ptr, weak_ptr

- **shared_ptr<T>**: wskaźnik do zasobu, który ma mieć jednego właściciela (z możliwością transferu własności)
- **unique_ptr<T>**: wskaźnik do zasobu, który może mieć wielu właścicieli (ze zliczaniem referencji)
- **weak_ptr<T>**: jak shared_ptr, ale bez zliczania referencji

unique_ptr

```
#include <iostream>
#include <memory>

void f(int* p)
{
    std::cout << *p << std::endl;
}

int main()
{
    std::unique_ptr<int> p1 { new int(10) };
    std::unique_ptr<int> p2 = std::move(p1);

    if(p1)
        f(p1.get());

    (*p2)++;

    if(p2)
        f(p2.get());
}
```



shared_ptr

```
#include <iostream>
#include <memory>

void f(int* p)
{
    std::cout << *p << std::endl;
}

int main()
{
    std::shared_ptr<int> p1 { new int(10) };
    std::shared_ptr<int> p2 { p1 };

    if(p1)
        f(p1.get());

    (*p2)++;

    if(p2)
        f(p2.get());
}
```



10
11

std::begin(T), std::end(T)

```
#include <algorithm>

int main()
{
    int arr[] = {1, 2, 3, 4};
    std::for_each(std::begin(arr), std::end(arr), [](int n) {
        std::cout << n << " ";
    });
    std::vector<int> v {-1, -2, -3, -4};
    std::for_each(std::begin(v), std::end(v), [](int n) {
        std::cout << n << " ";
    });
    std::cout << "\n";
}
```



1 2 3 4 -1 -2 -3 -4

- Pozwala w jednolity sposób traktować wszystkie kontenery i tablice z języka C

Inicjalizacja „klamrami”

- C++98:
co najmniej 3
sposoby
inicjalizacji

```
int main()
{
    int a = 0;
    int b (1);
    std::pair<int,int> {2, 3};
}
```

- C++11 =
C++98 + „klamry”

```
int main()
{
    int a = 0;
    int b (1);
    std::pair<int,int> {2, 3};

    int c = {0};
    int d {0};
    std::vector<int> e {7}; // ??
    std::vector<int> f {0,1,2,3,4};
    std::map<int, std::string> g{
        {1, "one"}, {2, "two"}, {3, "three"}
    };

    std::vector<int> h {e};
    std::vector<int> k {std::move(e)};
}
```


#include <initializer_list>

```
#include <iostream>
#include <initializer_list>

struct X {
    X() = default;
    X(const X&) = default;
};

struct Q {
    Q() = default;
    Q(Q const&) = default;
    Q(std::initializer_list<int> arg)
    {
        for (auto a: arg)
            std::cout << a << "\n";
    }
};

int main() {
    X x;
    X x2 { x };
    Q q1 { 1 };
    Q q2 { 2 };
    Q q3 { 3, 4, 5 };
}
```



- Nowy, „zachłanny” konstruktor
- Ma większy priorytet niż konstruktor kopiujący
- Umożliwia inicjalizację listą wartości