

NAUJOS C++ KALBOS KONSTRUKCIJOS.

C++11, C++14, C++17 STANDARTAI.

Naujų C++ standartų rengimas – tai, iš esmės valdoma, programavimo kalbos evoliucija. Darbo grupė WG21 nuolat renka aukščiausios kvalifikacijos specialistų siūlymus, juos analizuoja, išdiskutuoja, fiksuoja ir tinkamai parengtus, adaptuotus pasiūlymus įtraukia į C++ standartą [ISO/IEC 14882](http://www.iso.org/iso/14882). Su peržiūros procesu galima susipažinti čia: http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html

2018 m. buvo atlikta jau 100-oji kalbos revizija (žr. *C++ Standard Core Language Active Issues, Revision 100, 2018-04-11*). Galutinį ISO kalbos standartą (pvz., 2017 m.) patvirtina atitinkamas standartų komitetas (*JTC1/SC22/WG21 – The C++ Standards Committee - ISO C++*), galutinis ISO standartas yra mokamas. Tačiau, C++ standartų komitetas ir darbo grupė pateikia visus darbinis dokumentus atviroje prieigoje *GitHub* (<https://github.com/cplusplus/draft>) taip pat prieinamas ir darbinio standarto varianto *pdf* formato dokumentas – Draft: <http://www.open-std.org/jtc1/sc22/wg21/>

Esminiai kalbos pokyčiai prasidėjo nuo C++11 ir nors C++ išlieka procedūrinio (imperatyvaus) ir objektinio programavimo kalba, joje dera vis daugiau apibendrinto (meta-programavimo), funkcinio ir loginio programavimo savybių. Šios naujos savybės taip pat užtikrina saugesnę ir patogesnę profesionaliam naudojimui kalbą. Naujų C++ kalbos konstrukcijų evoliuciją patartina stebėti būtent nuo esminių pokyčių, atliktų 2011 m. (žr. pvz., <https://en.wikipedia.org/wiki/C%2B%2B11>).

Pagrindiniai pokyčiai apima tokias sritis:

- Naujos kalbos branduolio (angl. *core language*) konstrukcijos ir jų semantika (*rvalues*, *move semantika*, *POD – plain old data*);
- Kalbos branduolio naudojamumo didinimas (tipų išvedimas, *lambda* išraiškos, unifikuotas inicializavimas (*uniform initialization*), intervalinis *for* ciklas (*range-for-loops*), *nullptr* konstanta);
- Kalbos funkcionalumo praplėtimas (tipas *long long int*, protingos rodyklės, tipizuotas vardinis tipas *enum*, nauji vartotojo apibrėžiami literalai, paralelinio programavimo atminties modelis, gijų tvarkymas, ir kt.);
- Standartinės bibliotekos papildymas (pvz., *Threading Library*) ir gerinimas.

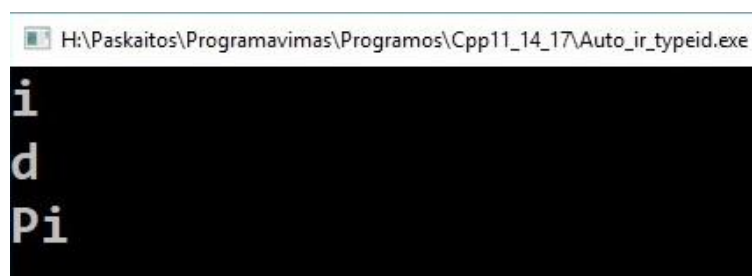
TIPŲ IŠVEDIMAS C++.

Programavimo kalbose *tipų išvedimas* reiškia automatinį išraiškos duomenų tipo nustatymą. Klasikinėje C++ kalboje iki C++11 kiekvieną duomenų tipą reikėjo deklaruoti (kompiliavimo metu), tokiu būdu ribojant išraiškos reikšmės vykdymo metu. Pradedant nuo C++11 į kalbą įtraukti nauji baziniai žodžiai, kurie leidžia programuotojui nedeklaruoti tipą, o palikti jo išvedimą pačiam kompiliatoriui. Turėdami tokias galimybes, programuotojai gali taupyti laiko, nes nebereikia apibrėžti dalykus, kuriuos kompiliatorius ir taip jau žino. Kadangi visi tipai yra išvedami kompiliavimo metu, kompiliavimo laikas šiek tiek padidėja, bet tai neturi įtakos programos vykdymo trukmei.

Bazinis žodis **auto** nurodo, kad deklaruojamo kintamojo tipas bus nustatytas automatiškai iš jo inicializavimo. Funkcijų atveju, jei funkcijos grąžinamos reikšmės tipas yra automatinis, tai jis bus vertinamas pagal **return** operatoriuje nurodytos išraiškos tipą programos vykdymo metu.

Pavyzdys 12.1. Tipo išvedimas su *auto*.

```
// C++ program to demonstrate type inference with auto
#include <iostream>
#include <typeinfo>
using namespace std;
int main()
{
    auto x = 10;
    auto y = 3.55;
    auto ptr = &x;
    cout << typeid(x).name() << endl
         << typeid(y).name() << endl
         << typeid(ptr).name() << endl;
    return 0;
}
```



Bazinis žodis **auto** ypač naudingas, kai objekto tipo nurodymas

reikalauja daug žodžių arba kai šis tipas automatiškai sugeneruojamas, t. y. šablonuose. Pvz., tipinis iteratorių naudojimas vektoriuje:

```
void func(const vector<int> &vi)
{
    vector<int>::const_iterator ci=vi.begin();
}
```

Dabar, vietoj šios ilgos konstrukcijos, galima paskelbti tokį iteratorių:

```
auto ci=vi.begin();
```

Bazinis žodis **decltype**. Ši kalbos konstrukcija tikrina deklaruotą objekto tipą arba išraiškos tipą. Jei **auto** leidžia paskelbti tam tikro tipo kintamąjį, tai **decltype** yra tam tikras operatorius, kuris leidžia išgauti duomenų tipą iš išraiškos arba kintamojo.

Pavyzdys 12.2. Tipo išvedimas su *decltype*.

```
//C++ program to demonstrate use of decltype
```

```
#include <iostream>
```

```
#include <typeinfo>
```

```
using namespace std;
```

```
int fun1() { return 10; }
```

```
char fun2() { return 'g'; }
```

```
int main()
```

```
{
```

```
// x duomenų tipas yra toks pats kaip fun1() grąžinamos reikšmės tipas
```

```
decltype(fun1()) x;
```


```
decltype(fun2()) y; //y tipas toks pats kaip fun2()
```

```
cout << typeid(x).name() << endl;
```

```
cout << typeid(y).name() << endl;
```

```
return 0;
```

```
}
```

 H:\Paskaitos\Programavimas\Programos\Funkcija_max.exe



Pateiksime dar vieną pavyzdį – bendrą *auto* ir *decltype* naudojimą. Čia pateikta šabloninė funkcija *min_type()*, kuri grąžina mažiausią iš dviejų

skaičių. Patys skaičiai gali būti bet kurio skaitmeninio tipo. Gražinamas tipas nustatomas pagal gaunamo mažiausio skaičiaus tipą.

Pavyzdys 12.3. Bendras *auto* ir *decltype* naudojimas.

/ C++ program: type inference with auto and decltype

```
#include <iostream>
```

```
#include <typeinfo>
```

```
using namespace std;
```

```
template<class A, class B>
```

```
auto findMin(A a, B b) -> decltype(a < b ? a : b)
```

```
{
```

```
    return (a < b) ? a : b;
```

```
}
```

```
int main()
```

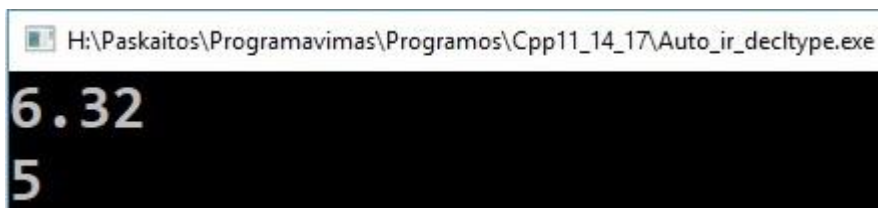
```
{
```

```
    cout << findMin(7, 6.32) << endl;
```

```
    cout << findMin(7.4, 5) << endl;
```

```
    return 0;
```

```
}
```



Pastaba. Iš tikrųjų, bazinis žodis **auto** nėra naujas C++ kalboje, jis buvo dar C kalboje. Tačiau, C++11 iš esmės pakeitė jo reikšmę, nes **auto** nebežymi programinio objekto su automatiniu atminties tipu, tad jos sena reikšmė nebegalioja.

UNIFIKUOTAS INICIALIZAVIMAS

Klasikinėje C++ kalboje yra mažiausiai keturi skirtingi inicializacijos nurodymo būdai (notacijos), kai kurie iš jų persidengė. Tokia painiava tikrai nepadeda programuotojams. Be to, anksčiau negalima buvo inicializuoti klasės narių, POD masyvo narių ir masyvų, talpinamų į dinaminę atmintį panaudojant operatorių **new** [].

C++11 standartas leidžia atsikratyti inicializavimo painiavos ir siūlo naudoti vieningą riestinių skliaustų notaciją (angl. *uniform brace*

notation, curly brackets). Naujos notacijos pavyzdžiai:

```
class C
{
    int a;
    int b;

    public:
        C(int i, int j);
};
C c {0,0};           //Tik C++11 ! Tai ekvivalentiška: C c(0,0);

int* a = new int[3] { 1, 2, 0 }; //Tik C++11 !

class X
{
    int a[4];

    public:
        X() : a{1,2,3,4} {}      //Tik C++11 !, masyvo nario inicializ.
};

class C
{
    int a=7;                    //Tik C++11 !, klasės nario inicializ.

    public:
        C();
};
```

Anksčiau taip pat nebuvo paprasto būdo **inicializuoti STL konteinerį**. Pavyzdžiui, norint inicializuoti vektorių, sudarytą iš simbolių eilučių, reikėjo sekos iš kelių **push_back()** kvietimų:

```
vector <string> vs;
vs.push_back("alpha");
vs.push_back("beta");
vs.push_back("gamma");
```

C++11 kiekvienam konteineriui įvedė vadinamąjį sekos konstruktorių angl. *sequence constructor*). Jo dėka konteinerių inicializaciją dabar daroma žymiai paprasčiau, tiesiogiai naudojant inicializavimo sąrašą (angl. *initializer list*), pavyzdžiui:

```
vector <string> vs{"alpha", "beta", "gamma"};
vector<int> vi {1,2,3,4,5,6};
vector<double> vd {0.5, 1.33, 2.66};
map<string, string> stars
```

```
{ {"Superman", "+1 (212) 545-7890"}, {"Batman", "+1 (212) 545-987"} };
```

Naujuose C++ standartuose nulinę rodyklės konstantą žymi naujas bazinis žodis **nullptr**. Tai leidžia išvengti dviprasmiškumo ir klaidingų situacijų naudojant NULL makrosą arba literalą 0, nes skirtingai nuo jų **nullptr** yra griežtai tipizuota konstanta. Pavyzdžiui:

```
void f(int);           // Pirma funkcija
void f(char *);        // Antra funkcija

//C++03
f(0);                  //Dviprasmiškumas: neaišku kuri f funkcija iškviečiama?

//C++11
f(nullptr)             //Vienareikšmis kreipinys į antrą funkciją
```

Standartinė C++11/14 biblioteka apibrėžia **naujus algoritmus**, atvaizduojančius aibių teorijos operacijas **all_of()**, **any_of()** ir **none_of()**. Pateiktame pavyzdyje predikatas **ispositive()** taikomas diapazonui $[first, first + n]$ ir naudoja minėtus algoritmus diapazono elementų savybėms nustatyti:

```
all_of(first, first+n, ispositive());    // ar visi elementai teigiami?
any_of(first, first+n, ispositive());    // ar yra bent vienas teigiamas?
none_of(first, first+n, ispositive());    // ar visi elementai neigiami?
```

Naujas **copy_n()** algoritmas leidžia, pvz., nukopijuoti n vieno masyvo elementų į kitą:

```
#include <algorithm>
int source[5]={0,12,34,50,80};
int target[5];

copy_n(source,5,target);
```

Naujas **iota()** algoritmas sukuria nuosekliai didėjančių reikšmių diapazoną, tarsi priskirdamas pradinę reikšmę pirmam diapazono elementui ir didindamas šią reikšmę prefiksiniu inkrementu ++:

```
#include <numeric>
int a[5]={0};
char c[3]={0};

iota(a, a+5, 10);           // a keičiamas į {10,11,12,13,14}
iota(c, c+3, 'a');          // c keičiamas į {'a','b','c'}
```

REIKŠMIŲ INTERVALO CIKLAS

Nuo C++11 standarto į C++ kalbą įtrauktas reikšmių intervalo (diapazono) ciklas *for* (angl. *range-for-loop*). Toks žinomo kartojimų skaičiaus ciklas perrenka visas reikšmes iš duoto intervalo. Lyginant su tradiciniu *for* ciklu, toks intervalinis ciklas ypač patogus ir aiškus, kai dirbama, pvz., su visais konteinerio elementais.

Bendroji intervalinio *for* ciklo sintaksė

```
for ( range_declaration : range_expression ) loop_statement
```

čia:

range_declaration – tai diapazono deklaracija, kurią sudaro vardinio kintamojo skelbimas; šio kintamojo tipas yra sekos elementų tipas, o pati seka apibrėžiama arba kaip *range_expression* arba per nuoroda į šį tipą. Šioje išraiškoje dažnai naudojamas *auto* automatiniam tipo išvedimui.

range_expression – tai bet kokia išraiška, kuri apibrėžia tinkamą seką arba inicializacijos sąrašas (angl. *braced-init-list*).

loop_statement – ciklo sakiniai (bet kokie).

Pavyzdys 12.4. reikšmių intervalo ciklo taikymas.

```
// Range-for loop iliustracija
#include <iostream>
#include <vector>
#include <map>
using namespace std;

int main() {
    // Iteracija per visą vektorių
    std::vector<int> v = {0, 1, 2, 3, 4, 5};
    for (auto i : v)
        cout << i << ' ';
    cout << endl;
    // tiesioginis inicializacijos sąrašo naudojimas
    for (int n : {0, 1, 2, 3, 4, 5})
        cout << n << ' ';
    cout << endl;
    // Iteracija per standartinį masyvą
    int a[] = {0, 1, 2, 3, 4, 5};
```

```

for (int n : a)
    cout << n << ' ';
cout << endl;
// Ciklo vykdymas visiems masyvo elementams
for (int n : a)
    cout << "Ciklas " <<n<< ' ';
cout << '\n';
// Simbolių sekos string atskirų simbolių spausdinimas
string str = "Labas";
for (char c : str)
    cout << c << ' ';
cout << endl;
// Atvaizdo reikšmių porų (raktas - reikšmė) spausdinimas
map <int, int> MENUO({{1, 31}, {2, 28}, {3, 31}});
for (auto i : MENUO)
    cout << '{'<<i.first<< ", "<<i.second<<"}\n";
}

```

```

0 1 2 3 4 5
0 1 2 3 4 5
0 1 2 3 4 5
Ciklas 0 Ciklas 1 Ciklas 2 Ciklas 3 Ciklas 4 Ciklas 5
L a b a s
{1, 31}
{2, 28}
{3, 31}

Process returned 0 (0x0)    execution time : 0.086 s
Press any key to continue.

```