SAUGUS TIPŲ SUVEDIMAS.
DINAMINĖ INFORMACIJA APIE TIPĄ (RTTI)

Įvadiniame programavimo kurse jau buvo aptarti duomenų tipų transformavimo klausimai, tokie kaip neišreikštinio tipų transformavimo taisyklės ir tiesioginio (išreikštinio) tipų transformavimo operacijos.

Kai naudojamas neišreikštinis tipų transformavimas skirtingų tipų operandai suvedami į aukštesnį (ilgesnį) tipą, o priskyrimo operacijos rezultatas suvedamas į tipą kintamojo, esančio iš kairės nuo operacijos ženklo. C kalboje ir pradinėse C++ versijose tam tikros transformacijos buvo traktuojamos kaip saugios ir kompiliatorius jas vykdė automatiškai, net be perspėjimų. Paminėtinos tokios „saugios“ transformacijos kaip char->(unsigned) int, enum ir bitų laukai -> (unsigned long) int, bool->int. Tipų reikšmių intervalai C kalboje:

```
char:           -128 .. 127
int :           -32768 .. 32767
unsigned int :  0 .. 65535
```

Kadangi rezultato tipas gali tapti tiek aukštesnis, tiek žemesnis, galimas tikslumo arba ženklo praradimas, tad pagrindinė rekomendacija C++ kalboje yra vengti neišreikštinio tipų transformavimo. Tiesioginis (išreikštinis) tipų transformavimas leidžia programuotojui pačiam nustatyti reikalingas transformacijas ir tarsi padėti kompiliatoriui:

//C-stiliaus tipų suvedimas – *(tipas) išraiška*

```
int a=8, b=5;
float kint1, kint2;
double d = (double)10;

kint1 = (float)(a/b);
kint2 = (float)a/b;
kint2 = (float)a/(float)b;
```

//Funkcijos stilius – *tipas (išraiška)*

```
kint1 = float(b);
```

Kaip jau minėta, toks tipų transformavimas (buvo) tradiciškai naudojamas C kalboje, tad jį pradėjo taikyti ir C++ programuotojai. Tačiau, C++ yra griežto tipizavimo kalba, o čia nėra jokios tipų kontrolės!?

Todėl C++ kalboje atsirado saugesni tipų suvedimo (transformavimo)

mechanizmai, leidžiantys atlikti statinę ir dinaminę tipų kontrolę. Vykdant statinę tipų kontrolę, vietoje aukščiau pateiktų konstrukcijų tipo *(tipas)*, programuotojams rekomenduojama taikyti naujus operatorius *static_cast<>()*, *const_cast<>()* ir (mažiau) *reinterpret_cast<>()*.

Operatorius *static_cast* atlieka tipų transformaciją, remiantis statine informacija apie tipus, kuri gaunama kompiliavimo metu vykdant sintaksinę išraiškų analizę. Bendroji šio operatoriaus išraiška:

static_cast <naujo_tipo_vardas> (išraiška);

Taikant ***static_cast*** galima atlikti tiek standartinį, tiek nestandartinį tipų suvedimą: galima transformuoti vieną skaitmeninį tipą į kitą, *enum* tipą į *int*, rodyklė į rodyklę, paveldėjimo ryšiais susietus tipus. Pvz.,

Pavyzdys 14.1. Statinis standartinių tipų suvedimas.

```
#include <iostream>
using namespace std;

int main() {
   int n;
   double d=19.75;
   n = static_cast<int>(d);
   cout << "Reiksme n = " << n << endl;

   int* ptri;
   ptri = static_cast<int*>(NULL);
   cout << "Reiksme ptri = " << ptri << endl;

   int day;
   enum Weekend {Saturday, Sunday};
   Weekend dw;
   day = static_cast<int>(Sunday);
   day +=4;      //ši operacija būtų negalima be static_cast<>()
   cout << "Reiksme day = " << day << endl;
  return 0;
}
```

Programos vykdymo rezultatas:

Reiksme n = 19
Reiksme ptri = 0
Reiksme day = 5

Svarbu, kad naudojant ***static_cast*** galima ne tik suvesti išvestinės klasės rodyklę į bazinį tipą (kylantis suvedimas), bet ir transformuoti rodyklę į bazinį tipą į rodyklę į išvestinį tipą (krintantis suvedimas), kas bendru atveju nėra saugu. Tad pastarasis naudojamas tik tada, kai operandų tipai

nėra polimorfiniai ir kai naujas tipas yra vienareikšmė duotos klasės (išraiškos, kintamojo) išvestinė klasė.

Pavyzdys 14.2. Statinis klasių tipų suvedimas.

```
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {  };
class Derived : public Base  { };
int main() {
   Derived derOb, *pderOb;
   Base* pBase;
   pBase = &derOb;
   if ((pderOb = static_cast< Derived*>(pBase)) !=0)
     cout << "Rezultato rodyklės tipas yra: " << typeid(pderOb).name()<<endl;
  return 0;
}
```

Programos vykdymo rezultatas:

*Rezultato rodyklės tipas yra: Derived**

Tačiau, pvz., norint transformuoti *int* tipą į rodyklę, bus gauta kompiliavimo klaida. Tokiu atveju, jei yra poreikis transformuoti visiškai nesusietus tarpusavyje duomenų tipus, vietoje ***static_cast*** galima taikyti ***reinterpret_cast*** operatorių.

```
void addr;
int* intPtr = static_cast<int*>(addr);
Point* poPtr = reinterpret_cast< Point*> (20);
```

Statinis šio operatoriaus pobūdis reiškia, kad jis netikrina išraiškų tipų programos vykdymo metu. Tai nustato didesnę atsakomybę programuotojui, bet, antra vertus, pagreitina programos vykdymą.

Kai reikia atlikti tik ribotą transformavimą pakeičiant nekintamą tipą į kintamą (t.y. šalinant modifikatorių ***const*** iš objekto tipo) arba atvirkščiai (t.y. šalinant modifikatorių ***volatile***), galima taikyti *const_cast* operatorių:

```
const  char* addr1;
char* addr2=const_cast< char*>addr1;
```

Statinės tipų transformacijos operatorių naudojimas suteikia programuotojams tam tikrų privalumų. Patogiau modifikuoti programas, nes analizuojant kodą, visas transformacijas lengva atpažinti. Taip pat

galima taikyti vadinamąsias „minimalias“ transformacijas, kurios mažiausiai keičia tipą. Tačiau, tipų kontrolė čia minimali, o *reinterpret_cast* atveju – tipų transformacija ne mažiau pavojinga, nei tradicinė C stiliaus transformacija.

DINAMINIS TIPŲ NUSTATYMAS IR SAUGUS SUVEDIMAS

C++ kalboje dinaminio tipų nustatymo priemonių paskirtis yra dvejopa: nustatyti tam tikros reikšmės (pvz. rodyklės) tipą programos vykdymo metu ir atlikti saugų tipų transformavimą.

Kaip jau minėta anksčiau, rodyklė į tam tikros klasės objektą galima transformuoti į rodyklę į bazinę klasę. Tačiau, atvirkštinis transformavimas nėra saugus, t. y. kompiliatorius negali garantuoti, kad rodyklės į bazinę klasę transformavimas į rodyklę į išvestinę klasę visada duos teisingą rezultatą. Programuotojas, dirbdamas su savo sukurtomis klasėmis, gali tai padaryti, bet jam reikia numatyti priemones, kurios saugotų informaciją apie rodyklės tipus.

Informaciją apie tipus programos vykdymo metu suteikia dinaminio tipų nustatymo priemonės (angl. *Run-Time Type Information, RTTI)*. Šios RTTI priemonės taikomos rodyklėms bei nuorodoms ir apima tokius tris pagrindinius elementus:

- operatorius ***dynamic_cast***;
- klasė ***type_info***;
- operatorius ***typeid***.

Būtent operatorius ***typeid*** pateikia informaciją apie dinaminius tipus kaip nuorodą į klasės ***type_info*** objektą.

Operatorius *dynamic_cast;*

Šis operatorius skirtas rodyklės arba nuorodos tipo *saugiam transformavimui* (*suvedimui*) į kitą tipą. Saugus čia reiškia, kad suvedimas į užduotą tipą įvyksta tik tada, kai šie tipai susieti paveldėjimo ryšiais. Priešingu atveju, tipo suvedimas neįvyksta ir generuojama išimtinė situacija (arba gražinama rodyklė NULL).

Bendroji šio operatoriaus išraiška:

dynamic_cast <tipo_vardas> (išraiška);

Ši operacija transformuoja (suveda) išraiškos reikšmę (rodyklę arba nuorodą) į nurodytą naują tipą. Nurodytas tipas gali būti arba rodyklė, arba nuoroda į duotame kontekste apibrėžtą tipą arba betipinė rodyklė

(void*). Kai vyksta transformavimas, pirma, patikrinamas išraiškos tipas. Jei šio tipo transformavimas į naują tipą neįmanomas, gražinama rodyklė NULL, o jei naudojama nuoroda, generuojama išimtis **bad_cast**. Pvz.:

```
class A { ... };
class B : public A  { ... };
class C { ...};

int main ()
{
        B* pb = new B;
        funkcija (pb);
}
void funkcija (A* pa)
{
        B* pb = dynamic_cast<B*> (pa);      //leidžiama transformacija
        C* pc = dynamic_cast<C*> (pa);      //rezultatas NULL
}
```

Klasė *type_info*.

Ši klasė yra apibrėžta antraštiniame faile <*typeinfo*> ir skirta suteikti informaciją apie duomenų tipą. Jos struktūra:

```
class type_info
{
   public:
        virtual ~type_info();
        int  operator== (const type_info& t) const;
        int  operator!= (const type_info& t) const;
        int  before(const type_info& t) const;
        const  char* name() const;
        const  char* raw_name() const;        //funkcija nebenaudojama
        size_t hash_code() const;             //tik nuo C++11
};
```

Kadangi šioje klasėje apibrėžtas tik privatus konstruktorius, programuotojas negali kurti šios klasės objektų. Antra vertus, klasėje yra perkrauti operatoriai lygu ir nelygu, tad jos objektus galima lyginti tarpusavyje, o metodas *name()* pateikia tipo vardą (t.y. rodyklę į jį).

Kiti metodai yra tarnybiniai ir taikomosiose programose beveik naudojami. Metodas *hash_code()* pateikia tam tikrą reikšmę, kuri sutampa tiems patiems tipams; Metodas *before()* gražina *1* (*true)*, jei iškviečiamas objektas yra aukštesnis objektų hierarchijoje nei objektas, naudojamas

kaip parametras, tačiau tai nesusiję su programos klasių hierarchija ar paveldėjimu.

Gauti informaciją apie tipą (t.y. *type_info* klasės objektą) galima taikant ***typeid*** operaciją.

Operatorius *typeid*.

Šis operatorius (funkcija) leidžia gauti nuorodą į klasės *type_info* objektą, kuris aprašo duotos išraiškos tipą. Bendras operatoriaus formatas:

typeid (tipo_vardas)

arba

typeid (išraiška);

Kai *typeid* taikomas rodyklei (ar nuorodai), kuri rodo į polimorfinę bazinę klasę, gaunama informacija apie tikrąjį objekto tipą, į kurį ši rodyklė rodo jau programos vykdymo metu (t.y. dinaminis tipas). Taip, žemiau pateiktame pavyzdyje, *t1* saugos informaciją apie išvestinę klasę:

```
class B { ... };
class D : public B  { ... };
class C { ...};
void funkcija ()
{
        D* pd = new D;
        B* pb = pd;
        ...
        const type_info& t = typeid(pb);        // t turi informaciją apie rodyklės tipą
        const type_info& t1 = typeid(*pb);      //čia, t1 turi informaciją apie D tipą
        ...
}
```

Kai *typeid* operatoriui perduodama rodyklė (ar nuoroda) į ne polimorfinį tipą, jis gražina statinį išraiškos tipą. Pavyzdžiui, jei *typeid* perduodamas standartinio tipo kintamasis ar išraiška, jis gražina šį standartinį tipą.

Pavyzdys 14.3 Žr. *cast_formatai.cpp*.