

Vilniaus Universitetas
Matematikos ir Informatikos fakultetas
Kompiuterijos katedra

Saulius Narkevičius

Objektiškai Orientuotas Programavimas su C++

paskaitų konspektai

pavasaris 2005

Turinys

Pratarmė.....	5
Literatūra.....	7
I. Apžvalga.....	9
1.1. Pirmoji C++ programa.....	9
1.2. Antraštinių failų (h-failų) įtraukimas.....	11
1.3. Keletos modulių programos.....	12
1.4. Selektyvus kompiliavimas (make).....	15
1.5. Bendros taisykės (pattern rules) make-failuose.....	17
1.6. Keletas būtiniausių Unix komandų.....	19
1.7. Nuo C prie C++ per maisto prekių parduotuvę.....	23
1.8. Klasė = duomenys + metodai.....	24
1.9. Konteineriai ir iteratoriai.....	26
1.10. Palyginimo operatorius, konteinerių rūšiavimas ir failai.....	29
1.11. Dinaminis objektų sukūrimas ir naikinimas.....	33
1.12. Objektų tarpusavio sąryšiai.....	36
1.13. Paveldėjimas ir polimorfizmas.....	42
2. Inkapsuliacija.....	51
2.1. Objektai paremtas programavimas (object based programming).....	51
2.2. Klasė, objektas, klasės nariai.....	52
2.3. Klasės narių matomumas.....	55
2.4. Konstruktoriai ir destruktoriai.....	56
2.5. Konstruktorius pagal nutylėjimą.....	58
2.6. Kopijavimo konstruktorius.....	59
2.7. Konstruktoriai ir tipų konversija.....	61
2.8. Objektų masyvai.....	63
2.9. Objektas, kaip kito objekto laukas (agregacija).....	65
2.10. Objektų gyvavimo trukmė.....	67
2.11. Metodai, apibrėžti klasės aprašo viduje.....	70
2.12. Statiniai nariai.....	71
2.13. Klasės draugai.....	73

2.14.	Tipų aprašai klasės viduje (įdėtiniai tipai).....	75
2.15.	Vardų erdvės išsprendimo operatorius ::.....	76
2.16.	Konstantiniai laukai, laukai-nuorodos.....	77
2.17.	Konstantiniai metodai ir mutable-laukai.....	78
3.	Paveldėjimas ir polimorfizmas.....	81
3.1.	Trys OOP banginiai.....	81
3.2.	Paveldėjimas.....	82
3.3.	Konstruktoriai ir destruktoriai.....	85
3.4.	Bazinės klasės narių matomumas.....	86
3.5.	Metodų perkrovimas (overloading) ir pseudo polimorfizmas.....	87
3.6.	Virtualūs metodai ir polimorfizmas.....	88
3.7.	Virtualių metodų lentelės (VMT).....	90
3.8.	Statiniai, paprasti ir virtualūs metodai.....	91
3.9.	Polimorfizmas konstruktoriuose ir destruktoriuose.....	92
3.10.	Švariai virtualūs metodai ir abstrakčios klasės.....	93
3.11.	Švarus interfeisas.....	94
4.	Klaidų mėtymas ir gaudymas (exception handling).....	99
4.1.	Raktiniai žodžiai throw, try ir catch.....	99
4.2.	Skirtingų klaidų gaudymas.....	101
4.3.	Įdėtiniai try-blokai.....	102
4.4.	Automatinių objektų naikinimas steko vyniojimo metu.....	103
4.5.	Klaidų mėtymas konstruktoriuose ir destruktoriuose.....	104
4.6.	Nepagautos klaidos ir funkcija terminate().....	105
4.7.	Klaidų specifikacija ir netikėtos klaidos.....	106
4.8.	Standartinės klaidų klasės.....	107
5.	Vardų erdvės (namespace).....	109
5.1.	Motyvacija.....	109
5.2.	Raktinis žodis "using".....	111
5.3.	Vardų erdvių apjungimas.....	112
5.4.	Vardų erdvių sinonimai.....	112
5.5.	Vardų erdvės be pavadinimo.....	113
6.	Operatorių perkrovimas.....	115
6.1.	Motyvacija.....	115
6.2.	Perkraunami operatoriai.....	117
6.3.	Unariniai ir binariniai operatoriai.....	118
6.4.	Tipų konversijos operatoriai.....	120

7.	Apibendrintas programavimas (generic programming).....	121
7.1.	Programavimo stiliai.....	121
7.2.	Funkcijų šablonai.....	122
7.3.	Klasių šablonai.....	125
7.4.	Trumpai apie sudėtingesnes šablonų savybes.....	127
8.	Daugialypis paveldėjimas.....	131
8.1.	Daugialypio paveldėjimo pavyzdys.....	131
8.2.	Pasikartojančios bazinės klasės ir virtualus paveldėjimas.....	133
8.3.	Virtualios bazinės klasės konstravimo problema.....	134
8.4.	Objektų tipų identifikacija programos vykdymo metu.....	136
9.	OOP receptai.....	139
9.1.	Objektiškai orientuotos visuomenės folkloras.....	139
9.2.	Projektavimo šablonai (design patterns).....	141
9.3.	Programų pertvarkymas (refactoring).....	143
9.4.	Ekstremalus programavimas (eXtreme programming).....	145

Pratarmė

Objektinio Programavimo C++ kurse mokinsimės viso labo dviejų dalykų:

- objektiškai orientuoto programavimo (OOP) kaip tokio
- OOP naudojimo rašant programas su C++ programavimo kalba

Iš skaitytojo tikimasi praktinės darbo patirties su programavimo kalba C. Pastarosios išplėtimas C++ bus pristatytas nesistengiant pateikti visų programavimo kalbos detalių. Pasitikslinti detales galima žemiau pateiktame literatūros sąrašė. Nebūtina visko suprasti iškart: pradžia pasibandykite ir pripraskite. Kurso tikslas - supažindinti klausytojus su OOP ir C++ pagrindais, pakankamais programų kūrimui ir tolimesniam savarankiškam tobulinimuisi.

Šiuose konspektuose mes dažnai vartosime terminą objektinis programavimas turėdami omenyje objektiškai orientuotą programavimą.

Niekam ne paslaptis, jog programavimo kalba C++ yra kildinama iš programavimo kalbos C. Skaitytojas dar turėtų prisiminti, jog C buvo kuriama lygiagrečiai su operacine sistema *Unix* maždaug 1969-1973 metais *PDP-11* kompiuteriams kompanijoje *Bell Labs*. Jos tėvu laikomas *Dennis Ritchie*. 90% *UNIX* kodo buvo parašyta C kalboje. C kalbos pirmtakais laikomos dvi programavimo kalbos: *Algol68* ir *B*.

Toje pačioje kompanijoje *Bell Labs*, tik gerais dešimčia metų vėliau, *Bjarne Stroustrup* sukūrė patobulintą programavimo kalbą: "C su klasėmis". Neilgai trukus buvo nutarta, jog C kalbos išplėtimas Objektiškai Orientuoto Programavimo (OOP) priemonėmis nusipelno atskiros pavadinimo. Taip 1983 metais pirmą kartą paminėtas C++ vardas. Jos atsiradimą stipriai įtakėjo pirmoji objektiškai orientuota programavimo kalba *Simula67* (1962-1967).

Prireikė penkiolikos metų kol 1998-ųjų rugpjūtį buvo vienbalsiai patvirtintas C++ standartas ISO/IEC 14882 (Standard for the C++ Programming Language). Standarto rengimo eigoje buvo įnešta šiokių tokių pakeitimų į pačią programavimo kalbą. Gerokai išsiplėtė standartinė biblioteka: peržvelgti įvedimo/išvedimo srautai, atsirado klasė *string*, konteinerių šablonai, lokalizacija ir t.t.. Kompiliatorių kūrėjams prireikė dar dviejų-keturių metų kol jų produktai pradėjo daugiau ar mažiau realizuoti C++

standartą. Todėl nereikalaukite per daug iš C++ kompiliatorių, pagamintų prieš 2002-uosius metus.

Svarbiausias dalykas, kurį reikia žinoti apie C++, yra tai, jog C++ yra objektiškai orientuota kalba: ji pateikia klasės sąvoką (C kalbos struktūrų išplėtimas), kurios pagalba realizuojami trys Objektiškai Orientuoto Programavimo banginiai:

- inkapsuliacija
- paveldėjimas
- polimorfizmas

Aplink mus krūvos objektų: mašinos, kiemsargiai, medžiai. Kiekvienas jų turi savo būseną ir elgsenos ypatumus. OOP nutiesia labai patogų tiltą tarp objektų gyvenime ir objektų programoje. Objektiškai orientuotas yra ne tik programavimas: mes analizuojame aplinką, projektuojame jai talkinančias kompiuterines sistemas ir pagaliau programuojame tomis pačiomis sąvokomis – objektais, jų būsenomis, elgsena bei tarpusavio ryšiais. Syki perpratus OOP, paprastai jis tampa natūraliu programų kūrimo stiliumi. Sunku būna įsivaizduoti, kad kažkada gyventa be jo. Tai bene stipriausias instrumentas kovoje su programų sudėtingumu.

Literatūra

Paskaitas lydinčią medžiagą galima rasti internete: www.mif.vu.lt/~saulukas/oop2

1. **Bjarne Stroustrup:** *The C++ Programming Language; (Third Edition and Special Edition)*, Addison-Wesley, ISBN 0-201-88954-4 ir 0-201-70073-5, 1997.
<http://www.research.att.com/~bs>
2. **Bruce Eckel:** *Thinking in C++*; I ir II tomai; <http://www.bruceeckel.com>
3. C++ Library reference: <http://www.cplusplus.com/ref>
4. C/C++ Reference: <http://www.cppreference.com>
5. C++ Annotations: <http://www.icce.rug.nl/documents/cplusplus>
6. C++ FAQ LITE — *Frequently Asked Questions*:
<http://www.parashift.com/c++-faq-lite>

I. Apžvalga

I.I. Pirmoji C++ programa

Pati trumpiausia, nieko nedaranti, C++ programa:

```
// smallest.cpp  
  
int main() {}
```

Visos C++ programos prasideda nuo funkcijos *main()*, grąžinančios *int*-tipo reikšmę. Pagal nutylėjimą, grąžinama reikšmė nulis, kuri reiškia sėkmingą programos baigtį. Antroji mūsų programa ekrane atspausdins pasveikinimą:

```
// hello.cpp  
  
#include <iostream.h>  
  
int main ()  
{  
    cout << "Sveikas, pasauli!" << endl;  
}
```

Sukompiliuokime:

```
g++ hello.cpp
```

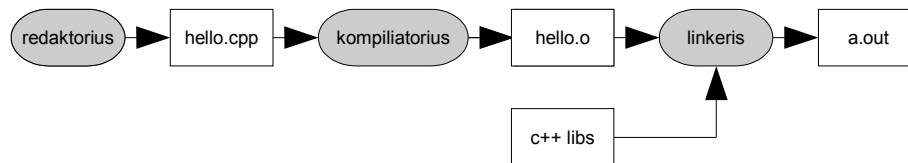
Bus pagaminta programėlė standartiniu pavadinimu *a.out* (arba *a.exe*), kurią paleidžiame tokiu būdu:

```
./a.out
```

Programa atspausdins ekrane:

```
Sveikas, pasauli!
```

Vieno failo kompiliavimą schematiškai pavaizduokime taip:



Programų kūrimas prasideda nuo to, jog jūsų pasirinkto redaktoriaus pagalba (pvz. integruoti failų komandierių *mc* ar *far* redaktoriai) sukuriamas tekstinis failas (*hello.cpp*) su programos išeities teksta. Komanda `g++ hello.cpp` iš karto atlieka du veiksmus: sukompiluoja kodą į atskirą modulį *hello.o* ir prijungia ("prilinkuoja") standartines bibliotekas, kad gauti savistovę paleidžiamą programą (*a.out* – *Unix*, *a.exe* – *Windows* terpėse). Šiuo paprasčiausiu atveju, tarpinis failas *hello.o* nėra išsaugomas diske.

1.2. Antraštinių failų (*h*-failų) įtraukimas

Griežtai žiūrint, ankstesniame skyrelyje pateikta programa *hello.cpp* yra parašyta "senuoju stiliumi". Maždaug 2000-ųjų metų ir vėlesni kompiliatoriai kompiliuoja šį kodą labai nenoriai, su krūva perspėjimų. Reikalas tas, jog po 1998 rugpjūčio mėnesį patvirtinto C++ programavimo kalbos standarto įsigaliojo naujas antraštinių failų įtraukimo stilius. Anksčiau C++ programuotojas traukdavo kvadratinę šaknį iš dviejų maždaug taip:

```
// headers_old.cpp

#include <math.h>
#include <iostream.h>

int main ()
{
    cout << "sqrt(2) = " << sqrt(2.0) << endl;
}
```

Pagal naująją C++ standartą nebereikia rašyti failo išplėtimo ".h" prie standartinių antraštinių failų. Be to, visi C-kalbos *h*-failai (pvz. *stdlib.h*, *stdio.h*, *math.h* ir t.t.) gauna priekyje raidę "c". Prisiminkime jog tokių standartinių antraštinių failų kompiliatorius ieškos parametrais ar aplinkos kintamaisiais nurodytuose kataloguose:

```
// headers_new.cpp

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

int main ()
{
    cout << "sqrt(2) = " << sqrt(2.0) << endl;
}
```

Apie raktinius žodžius *using namespace* mes plačiau pakalbėsime kitame skyriuje. Dabar tik išiminkite, jog magišką frazę *using namespace std* reikia rašyti po visų antraštinių failų įtraukimo. Tai, be kita ko, reiškia, jog visos standartinės C++ bibliotekos funkcijos ir klasės yra vardų erdvėje *std*.

Naujasis stilius galioja tik standartiniams C++ *h*-failams. Jūsų pačių rašyti *h*-failai, kaip ir anksčiau, įtraukiami rašant failo vardą tarp kabučių:

```
#include "mano.h"
```

Prisiminkime, jog tokių *h*-failų kompiliatorius pradžioje ieškos einamajame kataloge, o vėliau parametrais ar aplinkos kintamaisiais nurodytuose kataloguose.

1.3. Keletos modulių programos

Mūsų menkučiai pavyzdėliai apsiribojo vienu moduliu (*cpp*-failu). Kiek rimtesnė programa neišvengs keletos modulių – taip patogiau kodą prižiūrėti ir kompiliuoti. Pasitreniruokime rašydami programą apie protingą vaikiną *CleverBoy*. Ši programa susideda iš trijų failų: *CleverBoy.h* su *CleverBoy.cpp* ir *main.cpp*.

Faile *CleverBoy.h* turime klasės *CleverBoy* aprašą, t.y. modulio interfeisą. Aprašo užtenka, kad žinotume, ką duotoji klasė gali daryti, nesigilinant į tai, kaip ji tai daro:

```
// CleverBoy.h

#ifndef __CleverBoy_h
#define __CleverBoy_h

#include <string>

class CleverBoy
{
    std::string cleverWords;
public:
    CleverBoy(std::string cleverWords);
    std::string getCleverWords();
};
#endif // __CleverBoy_h
```

Faile *CleverBoy.cpp* turime klasės *CleverBoy* realizaciją. Tokia *h*- ir *cpp*-failų pora yra natūralus būdas nusakyti modulį: interfeisą ir realizaciją. Atkreipkime dėmesį į tai, jog po raktinių žodžių *using namespace std* galime naudoti klasės *string* trumpąjį vardą. Šie raktiniai žodžiai skirti naudoti tik *cpp*-failuose. Tuo tarpu *h*-faile mums teko nurodyti pilną vardą *std::string*.

```
// CleverBoy.cpp

#include "CleverBoy.h"
using namespace std;

CleverBoy::CleverBoy(string cw)
{
    cleverWords = cw;
}

string CleverBoy::getCleverWords()
{
    return cleverWords;
}
```

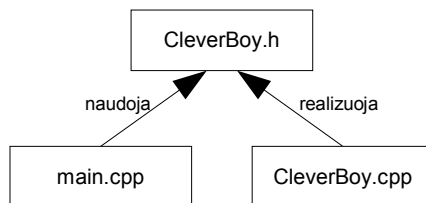
Trečiasis failas *main.cpp* naudoja modulį *CleverBoy*:

```
// main.cpp

#include <iostream>
#include "CleverBoy.h"
using namespace std;

int main()
{
    CleverBoy cleverBoy("Do not worry, C ++ is easy:");
    cout << "O dabar paklauskime protingu žodžiu:" << endl;
    cout << cleverBoy.getCleverWords() << endl;
}
```

Priklausomybę tarp šių trijų failų galime pavaizduoti taip:



Pastaba: nepamirškite *h*-failus apskliausti `#ifndef/#define/#endif` konstrukcijomis. Taip išvengsite daugkartinio to paties failo įtraukimo sudėtingesniuose projektuose. Tiesa sakant, nuo šių žodžių privalo prasidėti visų *h*-failų rašymas. Tai gero tono ženklas.

Turime du modulius: *CleverBoy.cpp* ir *main.cpp*. Vienas jų realizuoja *CleverBoy.h* faile aprašytąjį interfeisą, antrasis jį naudoja. Patys *h*-failai atskirai nėra kompiliuojami. Abu modulius sukompiliuoti ir sujungti ("sulinkuoti") į vieną programą galime keliais būdais, kaip antai:

```
g++ CleverBoy.cpp main.cpp
```

Arba, jei einamajame kataloge nėra pašalinių *cpp*-failų, tai tiesiog:

```
g++ *.cpp
```

Arba, jei norime vietoje standartinio vardo *a.out* (*a.exe*) parinkti prasmingesnę gautos programos vardą, pvz. *CleverBoy.exe*:

```
g++ -o CleverBoy.exe CleverBoy.cpp main.cpp
```

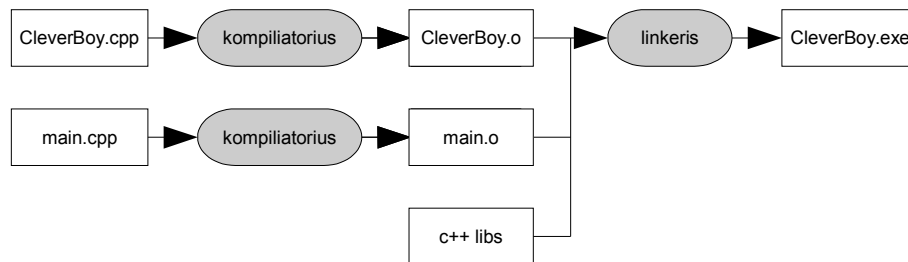
Kiekvienu atveju buvo atlikti trys veiksmai. Pavyzdžiui, vietoje paskutinės komandos galėjome parašyti tris atskiras komandas, darančias tiksliai tą patį:

```
g++ -c CleverBoy.cpp           - kompiliavimas
g++ -c main.cpp                - kompiliavimas
g++ -o CleverBoy.exe CleverBoy.o main.o - linkavimas
```

Kompiliatoriaus raktas `-c` nurodo, jog reikia tik kompiliuoti ir nenaudoti linkavimo, t.y. iš tekstinio *cpp*-failo pagaminti dvejetainį *o*-failą. Patys savaime *o*-failai negali būti vykdomi. Kad jie taptų savistove vykdoma programa, juos dar reikia apjungti (sulinkuoti) tarpusavyje, kartu prijungiant (savaime, be papildomų nurodymų) standartines C++ bibliotekas.

Toks ir yra klasikinis C++ programų kūrimo ciklas:

- naudojame keletą modulių (*cpp*-failų), kurių kiekvienas kompiliuojamas atskirai. Visi *cpp*-failai, išskyrus failą su funkcija `main()`, turi atitinkamus *h*-failus
- gauti *o*-failai susiejami į vieną *exe*-failą, kartu prijungiant C++ bibliotekas



Dvejojančius skaitytojus galime užtikrinti, jog programa *CleverBoy.exe* tikrai atspausdins:

```
O dabar paklauskime protingu zodziu:
Do not worry, C++ is easy:)
```

1.4. Selektivus kompiliavimas (*make*)

Mūsų projektelyje tėra du moduliai. Praktikoje jų būna dešimtys ir šimtai. Programų skaidymas į modulis – nuostabus dalykas, siekiant suvaldyti didelių sudėtingų programų kodą. Daug lengviau skyriumi tvarkyti nedidelius failus negu kapstyti po vieną monstrą. Pirmą kartą kompiliuojant, kompiliuojami visi moduliai – tai neišvengiama. Tačiau vėliau pakanka perkompiliuoti tik naujai pakeistus *cpp*-failus ir sulinkuoti su likusiais anksčiau sukompiliuotais *o*-failais. Tenka rašyti nemažai komandų. Be to, kiekvieną sykį reikia nepamiršti perkompiliuoti visus pakeistus modulius, antraip neišvengsime bėdų. Toks pastovus sekimas, kas buvo pakeista, o kas ne, yra labai nuobodus užsiėmimas. Jau nekalbant apie tai, jog užsižiopsoti ir pamiršti yra labai žmogiška.

Gyvenimą iš esmės palengvina plačiai paplitusi programėlė *make*. Ji seka, kuriuos *cpp*-failus reikia perkompiliuoti, o kurių ne. Tuo tikslu atskirame faile (*make*-faile) mes surašome, ką norime pagaminti, ir kokie žingsniai turi būti atlikti tikslui pasiekti. Programėlė *make* dirba selektyviai: atlieka tik tuos žingsnius, kuriuos reikia, o ne visus drauge. Klasikinis *make*-failo pavadinimas yra *makefile*:

```
# makefile

CleverBoy.exe : CleverBoy.o main.o
    g++ -o CleverBoy.exe CleverBoy.o main.o

CleverBoy.o : CleverBoy.cpp CleverBoy.h
    g++ -c CleverBoy.cpp

main.o : main.cpp CleverBoy.h
    g++ -c main.cpp
```

Tuomet, kokius failus beredaguotume, komandinėje eilutėje užteks parašyti:

```
make
```

Ekrane pamatysime, kaip po vieną žingsnį gimsta reikiami *o*-failai, ir vėliau pagaminamas *exe*-failas.

make-failas susideda iš taisyklių:

```
tikslas : prielaidos-atskirtos-tarpais
komanda-tikslui-pasiekti
```

Pagal nutylėjimą, programa *make* stengiasi gauti pirmąjį faile paminėtą tikslą (mūsų atveju *CleverBoy.exe*). Jei bent viena prielaida tikslui pasiekti (failai, nuo kurių priklauso tikslas) yra naujesnė, nei tikslo failas (patikrinama pagal failų datas), tai vykdoma nurodyta komanda-tikslui-pasiekti. Priklausomybė tikrinama rekurentiškai:

prielaidos failai taip pat yra atnaujinami, jei jie priklauso nuo kitų pasikeitusių failų.

Pavyzdžiui, jei paredaguosite ir išsaugosite failą *main.cpp*, tai komanda *make* rekurentiškai atseks, jog tikslui *CleverBoy.exe* pasiekti, reikia pirma atnaujinti *main.o* (naudojant komandą *g++ -c main.cpp*), o vėliau *CleverBoy.exe* (*g++ -o CleverBoy.exe CleverBoy.o main.o*). Analogiškai, jei paredaguosime ir išsaugosime failą *CleverBoy.h*, tai bus atnaujinti visi trys nuo jo priklausantys failai: *main.o*, *CleverBoy.o* ir *CleverBoy.exe*.

Pastaba: dauguma *make*-programų griežtai reikalauja, kad prieš komandą tikslui pasiekti būtų padėtas lygiai vienas tarpas arba tabuliacijos ženklas.

make-failai nėra griežtai skirti C++ programų kūrimui. Juos galima naudoti ir kituose projektuose, kur galutinis rezultatas priklauso nuo keletos tarpinių modulių.

Labai patogiu papildyti *make*-failą dar viena taisykle *clean*. Ji išvalo viską, ką sugeneravo kompiliatorius, ir palieka mūsų pačių rašytus išeities failus. Taisyklę geriausia pridėti į failo pabaigą, kad ji nebūtų pagrindinė:

```
clean:
    rm -f *.o
    rm -f *.exe
```

Šiuo atveju mums nereikalingus darbinius failus išvalys komanda *make* su nurodytu tikslu:

```
make clean
```

Unix operacinės sistemos komanda *rm* skirta failams trinti. Parameras **.o* (atitinkamai **.exe*) nurodo trinti visus failus, kurių vardai baigiasi simboliais *“.o”* (atitinkamai **.exe*). Tiek *Unix*, tiek ir *Windows* sistemų komandose žvaigždutė *“*”* reiškia bet kokią simbolių seką (taip pat ir tuščią). Taip kad nerašykite *rm **, nes ištrinsite visus einamojo katalogo failus! Raktas *-f* nurodo, kad nespausdinti klaidos pranešimo, jei nėra ką trinti.

Skyrelio pabaigai palyginkime sąvokas:

- *compile* - kompiliuoti vieną modulį: iš *cpp*-failo gauname *o*-failą
- *link* - susieti modulius: iš *o*- ir *lib*-failų gauname *exe*-failą
- *make* - perkompiliuoti tik pasikeitusius modulius ir susieti į vieną *exe*-failą
- *build* - perkompiliuoti viską, nekreipiant dėmesio į keitimo datas

1.5. Bendros taisykės (*pattern rules*) *make*-failuose

Yra dar daug patogių dalykų, kurie palengvina programuotojo gyvenimą rašant *make*-failus. Mes aptarsime du iš jų: bendras taisyklės ir automatinę *o*-failo prielaidų radimą.

Kai turime dešimtis *cpp*-failų, tai visai neapsimoka prie kiekvieno iš jų rašyti `g++ -c` ir t.t.. Tuo tikslu galime aprašyti bendrą taisyklę, kaip iš *cpp*-failų gauti *o*-failus (analogiškai: iš *o*-failų gauti *exe*-failą). Pažvelkime į pagerintą *make*-failo variantą:

```
# nicemake

compile      : CleverBoy.exe

CleverBoy.exe : CleverBoy.o main.o
CleverBoy.o   : CleverBoy.cpp CleverBoy.h
main.o        : main.cpp CleverBoy.h

#-----

clean:
    rm -f *.o
    rm -f *.exe

build: clean compile

%.exe: %.o
    g++ -o $@ $^

%.o: %.cpp
    g++ -c $<
```

Kadangi failo vardas nėra *makefile*, tai turime naudoti parametrą *-f*:

```
make -f nicemake
```

Eilutės, esančios virš brūkšnio, nusako priklausomybes tarp failų. Eilutės, esančios žemiau brūkšnio, nusako bendras taisyklės. Bendros taisyklės taikomos failams, pagal failų vardų pabaigas (*.cpp .o .exe*), vietoje specialiųjų simbolių įstatant atitinkamų failų vardus:

`$@` - tikslo failo vardas

`$<` - pirmojo prielaidos failo vardas

`$^` - visų prielaidos failų vardai, atskirti tarpais

Atkreipkime dėmesį, jog pirmasis *make*-failo tikslas yra *compile*. Toks tikslo užvardinimas leidžia žemiau brūkšnio aprašyti patogų tikslą *build: clean compile*,

kuris viską pradžioje išvalo, o paskui sukompiluoja. Tokiu būdą daugumos mūsų programų *make*-failai žemiau brūkšnio bus identiški. Skirsis tik aukščiau brūkšnio išvardinti moduliai su savo prielaidomis. Visų mūsų *make*-failų vardai toliau bus *makefile*, kad nereikėtų rašinėti rakto *-f*.

Pastaba: jei *make*-failo viduje rašote labai ilgas eilutes, suskaldykite jas į kelias, kiekvienos pabaigoje rašydami simbolį `"\"`, po kurio iš karto seka *enter*-klavišas.

Labai pravartu žinoti dar vieną kompiliatoriaus *g++* savybę – galimybę automatiškai sugeneruoti *o*-failų prielaidų tekstinę eilutę:

```
g++ -MM main.cpp
```

Ekrane bus atspausdinta viena eilutė:

```
main.o: main.cpp CleverBoy.h
```

Unix ir *Windows* terpėse visų komandinės eilutės komandų išvedimas į ekraną gali būti nukreiptas į failą simbolio `>` pagalba:

```
g++ -MM main.cpp > tarpinis.txt
```

Vėlgi galima pasinaudoti simbolio `"**"` privalumais:

```
g++ -MM *.cpp
```

Ekrane pamatysime:

```
CleverBoy.o: CleverBoy.cpp CleverBoy.h  
main.o: main.cpp CleverBoy.h
```

1.6. Keletas būtiniausių *Unix* komandų

Žemiau pateikiama keletas būtiniausių *Unix* komandų, skirtų darbui su failine sistema. Komandas reikia įvedinėti specialiame komandiniame lange, mūsų atveju vadinamame *shell*.

Naujai paleistas komandinis langas paprastai vartotoją nukelia į jo asmeninį "namų katalogą" (*home directory*). Pasidomėkime, kokie failai yra saugomi namų kataloge:

```
ls
```

Komanda atspausdins einamajame kataloge esančių failų ir kitų katalogų vardus. Jei norime matyti failų sukūrimo datas, dydžius bei požymį, ar failas yra failas, ar katalogas, rašome:

```
ls -l
```

Kai kurie failai laikomi tarnybiniais-paslėptais. Jų vardai prasideda taško simboliu. Juos dažniausiai kuria ne pats vartotojas, o jo naudojamos aplikacijos. Komanda *ls* pagal nutylėjimą jų nerodo. Pamatyti visus einamojo katalogo failus galima rakto *-a* pagalba:

```
ls -al
```

Mažytė gudrybė, kaip susikurti naują tekstinį failą, kad po to jį galėtume redaguoti:

```
ls > naujas_failas.txt
```

Jei nenorime kurti naujo failo, o tik prirasyti prie jau egzistuojančio failo pabaigos, vietoje simbolio ">" naudokime simbolius ">>".

Jei norime sužinoti einamojo katalogo pilną kelią, renkame:

```
pwd
```

Jei mūsų kataloge yra *oop2* kurso pakatalogis, tai į jį patekti galime taip:

```
cd oop2
```

Atgal grįžtame su komanda (dar sakoma, "grįžti į tėvinį katalogą" arba "grįžti katalogu auksčiau" turint omenyje katalogų medžio metaforą):

```
cd ..
```

Jei norime atsidurti pačioje katalogų šaknyje renkame:

```
cd /
```

Komandos `cd` pagalba mes galime nukeliauti į bet kurį katalogą. Namo, į *home*-katalogą, iš bet kurios vietos grįžtame taip:

```
cd ~
```

Keliaudami po katalogus galime pastoviai naudoti komandą *pwd*, kuri mums pasufleruos buvimo vietą. Pagrindiniai `cd` komandos variantai trumpai:

<code>cd Pakatalogis</code>	- nukelia gylyn į Pakatalogį
<code>cd Pakat/Mano</code>	- nukelia gylyn per du pakatalogius
<code>cd ..</code>	- grąžina atgal (aukštyn) per vieną katalogą
<code>cd ../..</code>	- grąžina aukštyn per du katalogus
<code>cd /</code>	- nukelia į katalogų medžio šaknį
<code>cd /usr/bin</code>	- nukelia į absoliutų kelią (skaičiuojant nuo šaknies)
<code>cd ~</code>	- nukelia į namų katalogą
<code>cd .</code>	- nieko nekeičia, nes "." reiškia einamąjį katalogą

Dabar turėtų būti aišku, kodėl failas *a.out* iš einamojo katalogo yra paleidžiamas taip:

```
./a.out
```

Pakeisti failo vardą arba perkelti jį į kitą katalogą galima su komanda:

```
mv dabartinis_vardas.txt naujas_vardas.txt
mv dabartinis_vardas.txt naujas_kelias/
mv dabartinis_vardas.txt naujas_kelias/naujas_vardas.txt
```

Visiškai analogiškai veikia ir failų kopijavimo komanda, tik ji neištrina originalo:

```
cp dabartinis_vardas.txt naujas_vardas.txt
cp dabartinis_vardas.txt naujas_kelias/
cp dabartinis_vardas.txt naujas_kelias/naujas_vardas.txt
```

Failas trinamas taip:

```
rm failas.txt
```

Jei norime ištrinti visus failus su galūne *.o*:

```
rm *.o
```

Jei norime ištrinti rekursyviai iš visų pakatalogių:

```
rm -r *.o
```

Įspėjimas !!!

Du, dar geriau, septynis kartus pagalvokite, prieš rinkdami komandas:

```
rm *  
rm -r *
```

Pirmoji ištrins visus einamojo katalogo failus, išskyrus pakatalogius. Antroji rekursyviai ištrins ne tik visus failus, bet ir visus pakatalogius.

Katalogai kuriami ir naikinami su komandomis:

```
mkdir NaujasKatalogas  
rmdir NereikalingasKatalogas
```

Komanda *less* leidžia peržiūrėti tekstinį failą. Iš jos išiname paspaudę "q". Ji rodo failą po vieną puslapį ekrane. Jei norime visą failo turinį išvesti į ekraną, naudojame komandą *cat*:

```
less mano_failas.txt  
cat  mano_failas.txt
```

Jei kataloge yra daug failų, ir visas jų sąrašas netelpa ekrane, nukreipkime komandos *ls* išėjimą į komandos *less* įėjimą:

```
ls -al | less
```

Jei mes surenkame *shell*'ui nežinomą komandą, tai jis bandys surasti sistemoje nurodytuose kataloguose atitinkamą programą, pvz.:

```
g++ hello1.cpp
```

Kartais pravartu žinoti, iš kokio katalogo *shell*'as iškasė šią programą. Tuo tikslu renkame:

```
which g++
```

Verta žinoti:

Labai patogiu naudotis tabuliacijos klavišu renkant komandas. Renkant failo ar pakatalogio vardą užtenka surinkti tik jo pradžią ir spausti tabuliacijos klavišą - failo pabaiga bus parinkta ir atspausdinta vienareikšmiškai.

Jei pamiršote, kaip naudotis viena ar kita komanda, pvz. *cp* ar *g++* surinkite:

```
man cp  
man g++
```

Turėsite gerokai mažiau vargo, jei pirmąją savo komanda pasirinksite *mc* - *Midnight*

Comander. Tuomet failų sąrašas visuomet bus ekrane, kopijavimas - tikras malonumas, o įsiūtas tekstų redaktorius dar ir paryškins C++ sintaksę.

Beje, redaguojant su vidiniu *mc* programos redaktoriumi tekstinius failus, sukurtus *DOS/Windows* terpėje, galite aptikti simbolius ^M kiekvienos eilutės pabaigoje. Taip yra todėl, jog *Unix* sistemoje eilutės pabaiga koduojama vienu specialiu simboliu, o *DOS/Windows* – dviem, kurių pirmasis ir yra vaizduojamas kaip ^M. Konvertuoti tarp dviejų tekstinių failų formatų pirmyn-atgal galima komandomis:

```
dos2unix dosfailas.txt
unix2dos unixfailas.txt
```

Antra komanda vartojama rečiau, nes dauguma *Windows* tekstų redaktorių (nebent išskyrus *NotePad*) supranta ir *Unix* eilučių pabaigas.

1.7. Nuo C prie C++ per maisto prekių parduotuvę

Pailiustruokime C++ klasių sąvoka maisto prekių parduotuvės pavyzdžiu. Čia turime maisto produktus su savo pavadinimais ir kainomis. Taip pat turime pardavimų sąrašą: kokį produktą, kada pardavėme, už kiek ir kokį kiekį. Be abejo, tai bus tik labai supaprastintas realios maisto prekių parduotuvės modelis. O pradėsime mes dar iš toliau: apsirasykime C stiliaus maisto produkto struktūrą (ne klasę) *Food* su dviem laukais (*name* ir *price*):

```
// food1.cpp

#include <cstring>
#include <iostream>

using namespace std;

struct Food
{
    char    name [80];
    double price;
};

void printPrice (Food& food, double amount)
{
    cout << "    " << amount << " units of "
         << food.name << " costs "
         << (food.price * amount) << endl;
}

int main ()
{
    Food bread = {"bread", 2.50};
    Food milk  = {"milk",  1.82};

    printPrice(bread, 0.5);
    printPrice(milk,  1.5);
    strcpy(milk.name, "beer");
    printPrice(milk,  1.5);
}
```

Funkcijos *main()* viduje sukuriame du kintamieji tipo *Food*:

bread	milk
name: "bread"	name: "milk"
price: 2.50	price: 1.82

Ši programa ekrane atspausdins:

```
0.5 units of bread costs 1.25
1.5 units of milk costs 2.73
1.5 units of beer costs 2.73
```

1.8. Klasė = duomenys + metodai

Žiūrint iš gero programavimo tono pusės, programoje *food1* yra mažiausiai du prasto skonio dalykai:

1. funkcija *printPrice()* dirba su struktūros *Food* tipo kintamaisiais, tačiau pati funkcija nėra struktūros dalis,
2. funkcijoje *main()* mes patys kišame nagus prie pieno vardo. Idealiu atveju pats pienas turi spręsti, ar mes galime keisti jo vardą ir ne.

Išspręsimė abi problemas aprašydami klasę *Food*. C++ **klasė** yra C struktūros sąvokos išplėtimas. Klasėje saugomi ne tik duomenys, bet ir funkcijos manipuliuojančios jais. Tokios funkcijos vadinamos **metodais**. Tiek klasės duomenys (**atributai**), tiek ir klasės metodai yra vadinami **klasės nariais**. Klasės tipo kintamieji vadinami **objektais**.

Klasėse mes galime vienus narius padaryti privačiais, kad iš išorės nebūtų galima jų naudoti, o tik pačios klasės metodų viduje, kaip pailiustruota programoje *food2*:

```
// food2.cpp

class Food
{
private:
    string name;
    double price;

public:
    Food (string name, double price);

    string getName    ()      {return name;}
    double getPrice   ()      {return price;}
    void   setPrice    (double p) {price = p;}

    void   printPrice (double amount);
};
```

Matome, jog klasė *Food* turi tuos pačius duomenų laukus, kaip ir ankstesnės programos struktūra *Food*, tik dar prisidėjo keletas viešų (*public*) metodų, pasiekiamų klasės išorėje. Atkreipkime dėmesį, jog mes leidome keisti maisto kainą metodo *setPrice()* pagalba, bet neleidome keisti maisto vardo. Tiesiog pasirinkome nerašyti analogiško metodo *setName()*. Vienos eilutės metodai aprašyti ir realizuoti pačiame klasės apraše, o didesnieji apibrėžiami atskirai:

```

Food::Food (string n, double p)
{
    name = n;
    price = p;
}

void Food::printPrice (double amount)
{
    cout << " " << amount << " units of " << name
        << " costs " << (price * amount) << endl;
}

```

Dabar atkreipkime dėmesį į pasikeitusią funkciją *printPrice()*, kuri virto metodu. Visų pirma, metodų, realizuotų klasės išorėje, vardai gauna priešdėlį „KlasėsVardas:“. Be to, nebėra pirmojo argumento *food*. Mat metodą galima iškviesti tik konkrečiam objektui (klasės tipo kintamajam), kuris ir atstos šį argumentą. Ir būtent jo laukais *name* ir *price* yra manipuluojama metodo viduje be papildomų priedais.

Klasės naudojimas primena struktūrų naudojimą: metodai pasiekiami taip pat, kaip ir duomenys:

```

int main ()
{
    Food bread ("bread", 2.50);
    Food milk  ("milk", 1.82);

    bread.printPrice (0.5);
    milk.printPrice  (1.5);
    milk.setPrice    (2.10);
    milk.printPrice  (1.5);
}

```

Programa *food2* atspausdins:

```

0.5 units of bread costs 1.25
1.5 units of milk costs 2.73
1.5 units of milk costs 3.15

```

Klasės kintamieji (objektai) aprašomi taip pat, kaip ir visi kiti C/C++ kintamieji. Vienintelis skirtumas yra tas, jog visi objektai privalo būti inicializuoti. Tai daroma specialaus metodo, **konstruktoriaus**, pagalba. Konstruktoriaus vardas sutampa su klasės vardu, be to, jis negrąžina jokios reikšmės, netgi *void*. Jis kviečiamas lygiai vieną kartą, objekto sukūrimo metu, kartu perduodant reikalingus argumentus.

Skirtingai nuo programos *food1*, mes nebeturime teisės rašyti *strcpy(milk.name, "beer")* programoje *food2*, nes laukas *name* (kaip ir *price*) yra privatus. Šioje vietoje buvo nuspręsta, kad po maisto produkto sukūrimo nebegalima keisti jo pavadinimo, tačiau leidžiama keisti jo kainą viešojo metodo *setPrice()* pagalba.

1.9. Konteineriai ir iteratoriai

Programose retai pavyksta apsiriboti pavienių objektų naudojimu. Dažnai tenka dirbti su objektų masyvais ir sąrašais. Standartinė C++ biblioteka pateikia ištisą rinkinį dinamiškai augančių konteinerių bei algoritmų darbui su jais. Mes pasinaudosime dviem iš jų: vektoriumi (masyvu) ir sąrašu. Konteineriuose saugosime tuos pačius programos *food2* objektus *Food*:

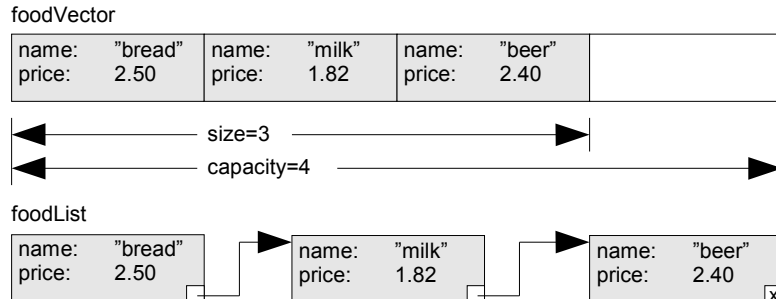
```
// food3.cpp

...
#include <vector>
#include <list>
...
int main ()
{
    vector<Food> foodVector;
    list<Food>    foodList;
    for (;;)
    {
        string name;
        double price;
        cout << "Enter food name ('q' - quit): ";
        cin >> name;
        if (name == "q") break;
        cout << "Price of one unit of " << name << ": ";
        cin >> price;

        Food food(name, price);
        foodVector.push_back(food);
        foodList .push_back(food);
    }
    ...
}
```

Standartinių konteinerių objektai yra apibrėžiami nurodant juose saugomų elementų tipą tarp < ir >. Ši konstrukcija, vadinama šablonais (*template*), detaliau bus aptarta daug vėliau.

Kaip matome, abiejų tipų konteineriai prigrūdami duomenimis metodo *push_back()* pagalba. Vektorius yra labai panašus į paprastą C kalbos masyvą. Jo elementus galime indeksuoti. Vektorių papildymas elementais nėra greitas veiksmas, nes kuomet elementų kiekis (*size*) viršija vektoriaus vidinio buferio talpą (*capacity*), tenka išskirti naują buferį ir į jį perkopijuoti visus elementus bei ištrinant senąjį buferį. Tuo tarpu sąrašai auga greitai, bet jų elementus galime perbėgti tik iš eilės.



Žemiau esantis kodo fragmentas demonstruoja, jog vektoriai yra indeksuojami taip pat, kaip ir C kalbos masyvai.

```

cout << "Vector size=" << foodVector.size()
      << " capacity=" << foodVector.capacity() << endl;
for (int i = 0; i < foodVector.size(); i++)
    foodVector[i].printPrice(2.00);
  
```

Įvedę tris produktus, gausime:

```

Vector size=3 capacity=4
2 units of bread costs 5
2 units of milk costs 3.64
2 units of beer costs 4.8
  
```

Tuo tarpu sąrašo elementai perbėgami **iteratorių** pagalba. Tai specialios paskirties objektai, kurie elgiasi labai panašiai, kaip rodyklė į vieną konteinerio elementą: operatorius ++ perveda iteratorių prie sekančio elemento, o operatorius * - grąžina objektą, į kurį iteratorius rodo šiuo metu. Konteineriai turi tokius metodus *begin()* ir *end()*, kurių pirmasis grąžina iteratorių, rodantį į pirmąjį elementą, o antrasis - į elementą, esantį už paskutiniojo. Iteratorių mechanizmas veikia ne tik sąrašams, ir ne tik vektoriams, bet ir visiems kitiems standartiniams konteineriams:

```

cout << "List size=" << foodList.size() << endl;
list<Food>::iterator li = foodList.begin();
for (; li != foodList.end(); li++)
    (*li).printPrice(3.00);    // not so nice style

cout << "Iterating through vector:" << endl;
vector<Food>::iterator vi = foodVector.begin();
for (; vi != foodVector.end(); vi++)
    cout << vi->getName() << " "    // better this way
         << vi->getPrice() << endl;
  
```

Šis kodo fragmentas atspausdins:

```
List size = 3
  3 units of bread costs 7.5
  3 units of milk costs 5.46
  3 units of beer costs 7.2
Iterating through vector:
bread 2.5
milk 1.82
beer 2.4
```

Kaip ir su C kalbos rodyklėmis į struktūras, taip ir su iteratoriais, struktūrų ir klasių laukus galime pasiekti dviem būdais:

```
(*iterator).printPrice(3.00);    // not so nice style
iterator->printPrice(3.00);        // better this way
```

Antrasis būdas yra stiliškai priimtinesnis.

1.10. Palyginimo operatorius, konteinerių rūšiavimas ir failai

Praeitame skyrelyje duomenis įvedinėjome iš klaviatūros. Šiame skyrelyje mes juos pakrausime iš failo, surūšiuosime ir išsaugosime naujame faile.

Tam, kad standartiniai konteineriai galėtų palyginti du maisto objektus *Food*, mes turime aprašyti specialų metodą: perkrautą operatorių “mažiau”. Apie operatorių perkrovimą plačiau kalbėsime vėlesniuose skyriuose. Dabar tiesiog susitaikykime su kiek keistoka *operator<* sintakse. Pateikta *operator<* versija lygina tik maisto pavadinimus:

```
// food4.cpp

class Food
{
private:
    string name;
    double price;

public:
    Food (const string& name, double price);

    string getName    () const    {return name;}
    double getPrice   () const    {return price;}
    void  setPrice     (double p) {price = p;}
    void  printPrice   (double amount) const;

    bool operator < (const Food& f) const;
};

Food::Food (const string& n, double p)
{
    name  = n;
    price = p;
}

void Food::printPrice (double amount) const
{
    cout << " " << amount << " units of " << name
          << " costs " << (price * amount) << endl;
}

bool Food::operator < (const Food& f) const
{
    return name < f.getName();
}
```

Kiek modernizuotame klasės *Food* apraše yra naudojami trys nauji tamptai susiję dalykai:

1. Objektų perdavimas metodams per nuorodą (ampersendo ženklas &). Be šio ženklo metodams ir funkcijoms perduodama objekto kopija (kaip ankstesniuose pavyzdžiuose). Kuomet objektai yra didesni, perduodant kopiją bereikalingai užimama vieta steke ir lėtėja programos veikimas. Perduodant metodui nuorodą į objektą, pats objektas nėra kopijuojamas - metodas dirba su originalu. Darbas su nuoroda niekuom nesiskiria nuo darbo su pačiu originalu.
2. Kadangi perduodant objektus pagal nuorodą metodas dirba su originalais, jis potencialiai gali pakeisti originalų objektą, t.y. priskirti naujas reikšmes objekto laukams arba iškviesti objekto metodus, kurie pakeis jo vidinę būseną. Tokių ketinimų neturintis metodas pasižada savo viduje nekeisti originalaus objekto prirašydamas žodelį *const* prie parametru (pvz., *const string& name*), o kompiliatorius pasirūpina, kad pažado būtų laikomasi.
3. Taigi, metodas gavo konstantinę nuorodą į objektą ir nebegali jo keisti. Taip pat jis negali kviesti objekto metodų, kurie keistų jo būseną. Gali kviesti tik konstantinius metodus, kurių antraštės gale prirašytis žodelis *const*. (pvz. *getName*, *printPrice*). Konstantinių metodų viduje negalima keisti objekto laukų. Mūsų pavyzdyje metodas keistu pavadinimu "*operator <*" gauna konstantinę nuorodą *f* į klasės *Food* objektą. Objektui *f* kviečiamas metodas *getName*, kuris savo ruožtu aprašytas kaip konstantinis: jis nekeičia objekto *f*, tik grąžina jo pavadinimo kopiją. Tuo tarpu metodas *setPrice* nėra konstantinis, nes keičia objekto lauką *price*.

Kuomet turime perkrautą palyginimo operatorių, mes galime jį naudoti tuo pačiu būdu, kaip ir baziniams C++ tipams:

```
int main ()
{
    Food bread ("bread", 2.50);
    Food milk  ("milk", 1.82);
    cout << "Compare food names:" << endl
         << bread.getName() << " < " << milk.getName() << " = "
         << (bread < milk) << endl;
    ...
}
```

Šis kodo fragmentas atspausdins:

```
Compare food names:
bread < milk = 1
```

Toliau mes vietoje įvedimo iš klaviatūros naudosime skaitymą iš failo *input.txt*. Šio failo viduje yra penkios tekstinės eilutės:

```

razinos  5.99
pienas   1.80
alus     1.78
duona    1.20
grybai   16.99

```

Skaitymas iš failo labai primena skaitymą iš klaviatūros. Abiem atvejais C++ terminologija sakoma, kad duomenys skaitomi iš **įvedimo srauto**. Tik pirmu atveju srautas sujungtas su failu (klasės *ifstream* objektu), antruoju – su klaviatūra (globalus objektas *cin*). Mes naudosime failo objekto metodą *good()*, kuris pasako, ar pavyko prieš tai buvusi skaitymo operacija:

```

vector<Food> foodVector;
list <Food> foodList;

ifstream inFile ("input.txt");
string name;
inFile >> name;
while (inFile.good())
{
    double price;
    inFile >> price;
    foodVector.push_back(Food(name, price));
    foodList .push_back(Food(name, price));
    inFile >> name;
}
inFile.close();

```

Vektoriaus (masyvo) elementai rūšiuojami globalios funkcijos *sort()* pagalba. Pastaroji reikalauja dviejų parametrų: iteratoriaus, rodančio į pirmąjį elementą, ir iteratoriaus, rodančio į elementą, **esantį už paskutiniojo**:

```

sort(foodVector.begin(), foodVector.end());
cout << "Vector sorted by name:" << endl;
for (int i = 0; i < foodVector.size(); i++)
    cout << foodVector[i].getName() << " "
        << foodVector[i].getPrice() << endl;

```

Šis kodo fragmentas atspausdins:

```

Vector sorted by name:
alus 1.78
duona 1.2
grybai 16.99
pienas 1.8
razinos 5.99

```

Kaip minėjome praeitame skyrelyje, standartinių konteinerių iteratoriai elgiasi labai panašiai, kaip ir įprastos C/C++ kalbos rodyklės į struktūras ar klases. Todėl algoritmai, dirbantys su dinaminiais vektoriais, yra pritaikyti darbui ir su įprastiniais C/C++ masyvais. Surūšiuokime sveikųjų skaičių masyvą:

```
int intArray[] = {1, 2, 15, 7, -2, 14, -20, 6};
sort(&intArray[0], &intArray[8]);
cout << "Sorted integers:" << endl;
for (int i = 0; i < 8; i++)
    cout << intArray[i] << " ";
cout << endl;
```

Atkreipkite dėmesį, jog masyve yra aštuoni elementai, ir paskutiniojo indeksas yra 7. Tačiau standartiniu pabaigos požymiu laikoma rodyklė į neegzistuojantį elementą, esantį iškart už paskutiniojo. Ekrane pamatysime:

```
Sorted integers:
-20 -2 1 2 6 7 14 15
```

Tuo tarpu sąrašas surūšiuojamas su jo paties metodu *sort()*. Surūšiuotus duomenis išsaugosime faile *output.txt*. Rašymas į failą labai primena spausdinimą ekrane: abiem atvejais rašome į **išvedimo srautą**. Tik pirmuoju atveju srautas sujungiamas su failą atstovaujančiu objektu (tipo *ofstream* – *output file stream*), o antruoju – su ekraną atitinkančiu globaliu objektu *cout*:

```
foodList.sort();

ofstream outFile("output.txt");
list<Food>::iterator li = foodList.begin();
for (; li != foodList.end(); li++)
    outFile << li->getName() << " "
    << li->getPrice() << endl;
outFile.close();
```

Atkreipkite dėmesį, jog baigus darbą su failu (tiek skaitymą, tiek ir rašymą), kviečiamas metodas *close()*, kuris atlieka reikiamus baigiamuosius darbus ir atlaisvina darbui su failu naudotus resursus.

1.11. Dinaminis objektų sukūrimas ir naikinimas

Iki šiol mes naudojome tik vadinamuosius **lokalius**, dar vadinamus, **automatinius** objektus. T.y. kintamojo vardas nusako patį objektą, kuris yra automatiškai sukuriamas jo aprašymo vietoje ir automatiškai sunaikinamas, kuomet programos vykdymas išeina iš jo aprašymo bloko (metodo, ciklo ir pan.), apriboto figūriniais sklaustais {}. Taip pat mes naudojome nuorodas (konstantines) į objektus, kaip parametrus metodui. Abiem atvejais, kintamasis, viso savo gyvavimo metu yra susietas su vienu ir tuo pačiu objektu.

Tuo tarpu objektiškai orientuotas programavimas yra sunkiai įsivaizduojamas, be **rodyklių** į **dinaminis objektus**. Šiuo atveju kintamasis (rodyklė) nusako ne patį objektą, o rodo į dinaminėje atmintyje išreikštai (operatoriaus *new* pagalba) sukurtą objektą. Programos vykdymo metu ta pati rodyklė gali būti nukreipta ir į kitus objektus. Taip pat, jai galima priskirti ypatingą reikšmę nulį (sąvokos “*null*” analogą), kuri reiškia, kad rodyklė šiuo metu į nieką nerodo. Dinaminiai objektai nėra automatiškai sunaikinami – tą padaro pats programuotojas operatoriaus *delete* pagalba. Šio operatoriaus iškvietimo užmiršimas vadinamas **atminties nutekėjimu** – tai viena dažniausių programuotojų klaidų.

Dinaminių objektų laukai (duomenys ir metodai) pasiekiami operatoriaus *->* pagalba:

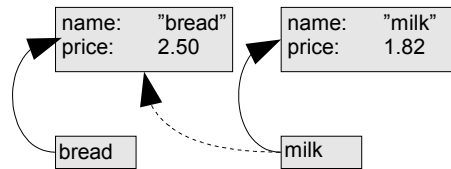
```
// food5.cpp

int main ()
{
    Food* bread = new Food ("bread", 2.50);
    Food* milk = new Food ("milk", 1.82);
    cout << "Two dynamically created objects:" << endl
         << bread->getName() << " and "
         << milk ->getName() << endl;
    delete milk;    milk = 0;
    milk = bread;
    cout << "Name of the milk: " << milk->getName() << endl;
    delete bread;  bread = 0;    milk = 0;
}
```

Šis kodo fragmentas atspausdins:

```
Two dynamically created objects:
bread and milk
Name of the milk: bread
```

Atmintyje patys objektai saugomi atsietai, nuo į juos rodančių rodyklių. Turime keturis objektus atmintyje: dvi rodykles ir du klasės *Food* objektus.



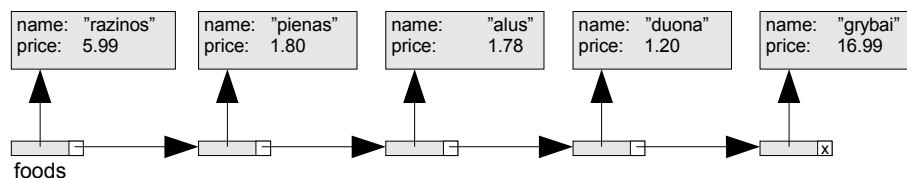
Punktyru pažymėta, jog programos vykdymo eigoje rodyklė *milk* buvo nukreipta į kitą objektą. Prisiminkime, jog pačios rodyklės užima vienodai vietos atmintyje, nepriklausomai nuo to, į kokio dydžio objektą jos rodo.

Ankstesniame skyrelyje mes turėjome sąrašą, kuriame saugojome klasės *Food* objektus. Dabar apibrėžkime sąrašą, kuriame saugosime rodykles į klasės *Food* objektus. Pačius duomenis imsime iš jau žinomo failo *input.txt*:

```
list<Food*> foods;

ifstream inFile ("input.txt");
string name;
inFile >> name;
while (inFile.good())
{
    double price;
    inFile >> price;
    foods.push_back(new Food(name, price));
    inFile >> name;
}
inFile.close();
```

Tokiu būdu sąrašo *foods* elementai yra rodyklės į dinamiškai sukurtus objektus:



Norint surūšiuoti tokį sąrašą nebeišeis pasinaudoti ankstesniame skyrelyje aprašytu palyginimo operatoriumi (*operator <*), nes jis buvo skirtas lyginti *Food* objektams, o ne *Food**. Tuo tikslu apsirašysime dar vieną standartinės bibliotekos stiliaus keistenybę: klasę-funkciją, kuri nusako tą patį palyginimo operatorių kitu būdu. Tai klasė, turinti tik vieną metodą, labai keistu pavadinimu "*operator[]*", kuris palygina maisto pavadinimus:

```

class FoodPtrNameCompare
{
public:
    bool operator() (const Food* a, const Food* b) const
    {
        return (a->getName() < b->getName());
    }
};

```

Sąrašo metodui *sort()* perduodamas papildomas parametras: laikinas objektas-funkcija, atsakingas už sąrašo elementų (rodyklių) palyginimą:

```

foods.sort(FoodPtrNameCompare());
list<Food*>::iterator li = foods.begin();
cout << "Food sorted by name:" << endl;
for (; li != foods.end(); li++)
    cout << (*li)->getName() << " "
        << (*li)->getPrice() << endl;

```

Prisiminkime, jog išraiška *(*li)* grąžina konteinerio elementą, į kurį rodo iteratorius. Mūsų atveju elementas yra rodyklė į objektą, todėl jo metodai kviečiami taip:

```

(*li)->getName()

```

Šis kodo fragmentas atspausdins:

```

Food sorted by name:
alus 1.78
duona 1.2
grybai 16.99
pienas 1.8
razinos 5.99

```

Be savo keistos sintaksės, klasė-funkcija mums leidžia tą patį sąrašą surūšiuoti ir kitais būdais, pvz. pagal kainą. Tuo tikslu apsirašome klasę-funkciją *FoodPtrPriceCompare*:

```

class FoodPtrPriceCompare
{
public:
    bool operator() (const Food* a, const Food* b) const
    {
        return (a->getPrice() < b->getPrice());
    }
};

```

Ji naudojama lygiai taip pat, kaip ir klasė-funkcija *FoodPtrNameCompare*. Analogiškas kodo fragmentas atspausdins:

```

Food sorted by price:
1.2 duona
1.78 alus
1.8 pienas
5.99 razinos
16.99 grybai

```

1.12. Objektų tarpusavio sąryšiai

Ankstesniuose pavyzdėliuose mes turėjome vienintelę klasę *Food*. Tam, kad maisto parduotuvė galėtų pradėti pardavinėti produktus, aprašykime pardavimo klasę *Selling*. Čia mes saugome informaciją apie vieną pardavimą: kurį maisto produktą kada pardavėme, kiek vienetų pardavėme ir kokia buvo vieno produkto vieneto kaina.

```
// selling.h

class Food; // forward declaration

class Selling
{
private:
    const Food* food;
    std::string date;
    int         itemCount;
    double      itemPrice;
public:
    Selling (const Food*      food,
            const std::string& date,
            int               itemCount,
            double            itemPrice);
    const Food* getFood      () const {return food;}
    std::string getDate      () const {return date;}
    int         getItemCount () const {return itemCount;}
    double      getItemPrice () const {return itemPrice;}
    double      getProfit    () const;
    void        print        () const;
};
```

Klasėje *Food* uždraudėme keisti kainas ir palikome tik naujų pirkimų registravimą:

```
// food.h

class Food
{
private:
    std::string      name;
    double           price;
    std::vector<Selling*> sellings;
public:
    Food (const std::string& name, double price);
    std::string getName      () const {return name;}
    double      getPrice     () const {return price;}
    double      getProfit    () const;
    void        print        () const;
    void        addSelling   (Selling* selling);
    int         getSellingCount () const {return sellings.size();}
    Selling*    getSelling   (int i) {return sellings[i];}
};
```

Kadangi klasės *Food* ir *Selling* abi turi rodyklių viena į kitą, tai mums tenka viename iš failų (pasirinkome *selling.h*) naudoti išankstinį klasės apibrėžimą be jos kūno (*class Food;*).

Vieno pardavimo metu uždirbtas pelnas nėra atskirai saugomas. Jis apskaičiuojamas pagal kitus duomenis:

```
// selling.cpp

#include <iostream>
#include "selling.h"
#include "food.h"
using namespace std;

Selling::Selling (const Food*      food,
                  const std::string& date,
                  int               itemCount,
                  double            itemPrice)
{
    this->food      = food;
    this->date      = date;
    this->itemCount = itemCount;
    this->itemPrice = itemPrice;
}

double Selling::getProfit () const
{
    return itemCount * (itemPrice - food->getPrice());
}

void Selling::print () const
{
    cout << food->getName()
          << " " << date
          << " " << itemCount
          << " " << itemPrice
          << " pelnas=" << getProfit()
          << endl;
}
```

Visi metodai ir konstruktoriai gauna vieną papildomą “nematomą” parametą *this*, rodantį į objektą, kuriam yra iškviestas metodas. Mes parametą *this* naudojame išreikštai tik ten, kur klasės narių vardai sutampa su metodo parametrų vardais.

Analogiškai, vieno produkto visų pardavimų pelnas irgi yra skaičiuojamas metodo pagalba. Atkreipkime dėmesį, jog metodas *getProfit* yra konstantinis, todėl jo viduje iteruoti per lauko *sellings* elementus galime tik konstantinio iteratoriaus pagalba:

```
// food.cpp

#include <iostream>
#include "food.h"
using namespace std;

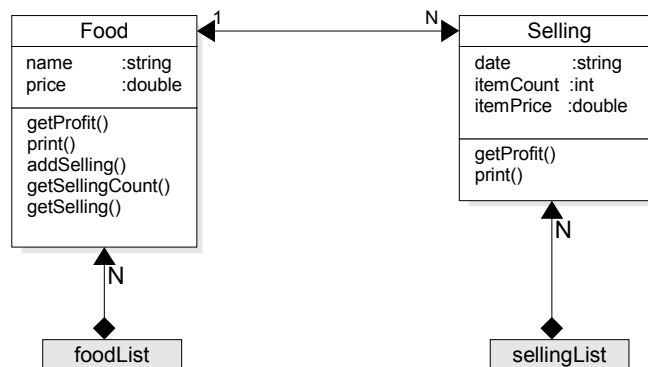
Food::Food (const string& name, double price)
{
    this->name = name;
    this->price = price;
}

double Food::getProfit () const
{
    double profit = 0;
    vector<Selling*>::const_iterator iter = sellings.begin();
    for (; iter != sellings.end(); iter++)
        profit += (*iter)->getProfit();
    return profit;
}

void Food::print () const
{
    cout << name << " " << price
         << " pelnas=" << getProfit() << endl;
}

void Food::addSelling(Selling* selling)
{
    sellings.push_back(selling);
}
```

Abi klases ir jų tarpusavio sąryšį galime pavaizduoti grafiškai:



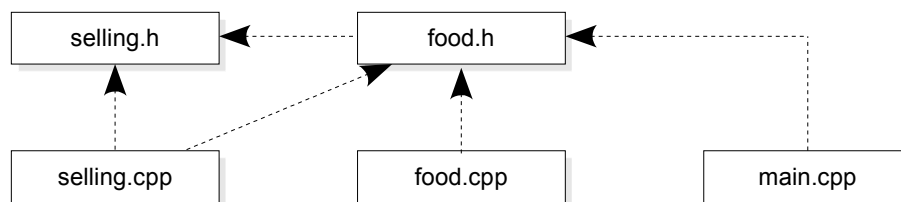
Klasės vaizduojamos kaip trijų dalių stačiakampis: viršutinėje dalyje – klasės vardas, vidurinėje – duomenų laukai (jų tipai po dvitaškio), apatinėje dalyje – metodai. Jei klasė turi nuorodą į kitą klasę, tai šį ryšį vaizduojame atitinkamos krypties rodykle su skaičiais: klasė *Selling* turi rodyklę į vieną (1) *Food* objektą, o klasė *Food* turi ištisą rodyklių masę į daug (N) klasės *Selling* objektų.

Dežutės *foodList* ir *sellingList* nusako kintamuosius: sąrašus iš rodyklių atitinkamai į *Food* ir *Selling* objektus. Rombas viename rodyklės gale sako, jog *foodList* susideda iš *N* klasės *Food* objektų. Toks ryšys vadinamas **agregacija**. Sąrašai *foodList* ir *sellingList* yra lokalūs funkcijos *main* kintamieji. Geras programavimo tonas įpareigoja vengti globalių kintamųjų.

Programa susideda iš pagrindinio “amžinojo” ciklo: rodyti meniu, laukti vartotojo įvedimo ir įvykdyti pasirinktą komandą:

```
// main.cpp
...
typedef std::list<Food*>    FoodList;
typedef std::list<Selling*> SellingList;
...
int main ()
{
    FoodList    foodList;
    SellingList sellingList;
    for (;;)
    {
        cout << endl
              << "Maisto produktai:"          << endl
              << "  1 - sarasas"                << endl
              << "  2 - sukurti"                << endl
              << "  3 - parduoti"               << endl
              << "  4 - produkto detales"       << endl
              << "  5 - laikotarpio pelnas"     << endl
              << endl
              << "  0 - baigti darba"          << endl;
        string key;
        cin >> key;
        if (key == "0") return 0;
        else if (key == "1") printFoodList(foodList);
        else if (key == "2") addNewFood(foodList);
        else if (key == "3") sellFood(foodList, sellingList);
        else if (key == "4") showFoodDetails(foodList);
        else if (key == "5") showProfit(sellingList);
        else cout << endl << "Neteisinga komanda..." << endl;
    }
}
```

Maisto parduotuvės išeities tekstai susideda iš penketos failų, kurių *include*-ryšiai pavaizduoti punktyrinėmis rodyklėmis.



Žemiau pateikiamos likusios modulio *main.cpp* funkcijos:

```
// main.cpp

Food* selectFood (FoodList& foodList)
{
    for (;;)
    {
        string name;
        printFoodList(foodList);
        cout << "Pasirinkite produkta ('#'-baigti): ";
        cin >> name;
        if (name == "#")
            return 0;
        FoodList::iterator iter = foodList.begin();
        for (; iter != foodList.end(); iter++)
            if (name == (*iter)->getName())
                return *iter;
    }
    return 0; // unreachable

void printFoodList (FoodList& foodList)
{
    cout << "-- produktų sąrašas " << foodList.size() << endl;
    FoodList::iterator iter = foodList.begin();
    for (; iter != foodList.end(); iter++)
        (*iter)->print();
}

void addNewFood (FoodList& foodList)
{
    string name;
    double price;
    cout << "-- naujas maisto produktas --" << endl;
    cout << "pavadinimas: "; cin >> name;
    cout << "kaina: "; cin >> price;
    foodList.push_back(new Food(name, price));
}

void sellFood (FoodList& foodList, SellingList& sellingList)
{
    cout << "-- produkto pardavimas --" << endl;
    Food* food = selectFood(foodList);
    if (food == 0) return;
    string date;
    int itemCount;
    double itemPrice;
    food->print();
    cout << "pardavimo data (yyyy.mm.dd): "; cin >> date;
    cout << "vienetų skaičius: "; cin >> itemCount;
    cout << "vieneto kaina: "; cin >> itemPrice;
    Selling* selling =
        new Selling(food, date, itemCount, itemPrice);
    food->addSelling(selling);
    sellingList.push_back(selling);
}
```

```

void showFoodDetails(FoodList& foodList)
{
    cout << "-- produkto pardavimo detales --" << endl;
    Food* food = selectFood(foodList);
    if (food == 0) return;
    cout << "Maisto pardavimai: ";
    food->print();
    for (int i = 0; i < food->getSellingCount(); i++)
        food->getSelling(i)->print();
}

void showProfit(SellingList& sellingList)
{
    string dateFrom;
    string dateTo;
    cout << "-- laikotarpio pardavimu pelnas --" << endl;
    cout << "    nuo kada (yyyy.mm.dd): "; cin >> dateFrom;
    cout << "    iki kada (yyyy.mm.dd): "; cin >> dateTo;
    double totalProfit = 0;
    SellingList::iterator iter = sellingList.begin();
    for (; iter != sellingList.end(); iter++)
        if (dateFrom <= (*iter)->getDate() &&
            (*iter)->getDate() <= dateTo)
        {
            (*iter)->print();
            totalProfit += (*iter)->getProfit();
        }
    cout << "laikotarpio pelnas: " << totalProfit << endl;
}

```

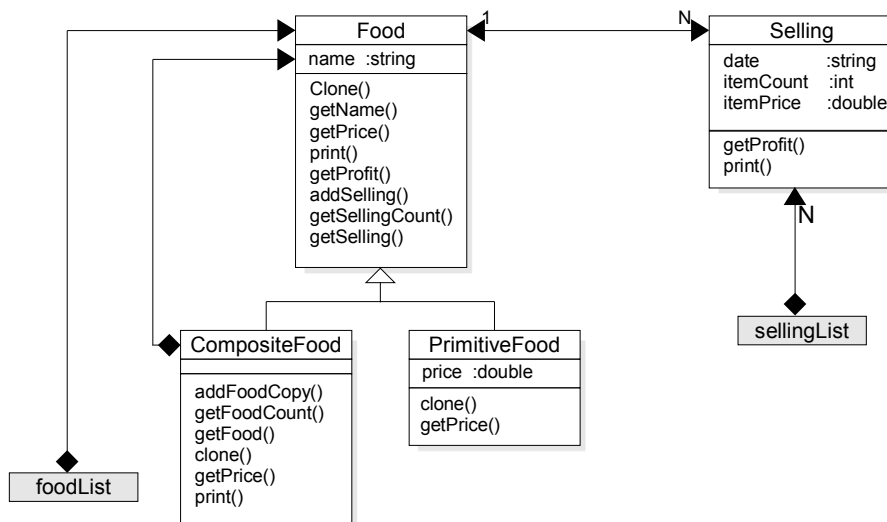
1.13. Paveldėjimas ir polimorfizmas

Paskutinį įvadinės dalies skyrelį paskirkime likusiems dviems (iš trijų) objektiškai orientuoto programavimo (OOP) banginiams: paveldėjimui ir polimorfizmui, kurie visuomet eina drauge.

Pirmasis OOP banginis kurį mes jau matėme – **inkapsuliacija**. Kartais sakoma, kad inkapsuliacija, tai kai klasėje kartu su duomenimis yra metodai, operuojantys tais duomenimis. Tačiau mes mąstykime, jog inkapsuliacija, tai klasės realizacijos detalių paslėpimas po gerai apgalvotu interfeisu. Išities tekstuose inkapsuliacija pasireiškė dviem būdais:

- Klasės interfeisas (aprašas) buvo iškeltas į atskirą antraštinį *h*-failą, kurį gali matyti ir naudoti (įtraukti) kiti moduliai (*cpp*-failai). Tuo tarpu klasės realizacija (apibrėžimas) nuosavame modulyje (*cpp*-faile) paprastai nėra matomas kitiems moduliams. Tokia inkapsuliacija yra atkeliavusi iš modulinio programavimo, kuris dera ne tik su OOP, bet ir su kitais programavimo stiliais (procedūriniu, struktūriniu ir t.t.).
- Skirtingai nuo C/C++ struktūrų duomenų-laukų, kurie yra vieši (*public*), klasės duomenys-laukai dažniausiai būna paslėpti (*private*) nuo tiesioginio skaitymo ir keitimo. Jais manipuliuoja vieši (*public*) klasės metodai. Tai OOP inkapsuliacija.

Paveldėjimą ir polimorfizmą pailiustruosime toliau vystydami maisto prekių parduotuvės pavyzdį. Modeliuosime nedalomus ir sudėtinius maisto produktus. Pavyzdžiui, laikysime, kad druska, miltai, sviestas, dešra ir pan. yra nedalomi maisto produktai (*PrimitiveFood*), o sumuštinis, pyragas ir t.t. - dalomi (*CompositeFood*).



Mūsų klasių hierarchijoje bendriausia maisto produkto sąvoką nusako klasė *Food*. Joje yra duomenys ir metodai būdingi visiems maisto produktams:

```

// food.h

class Food
{
private:
    std::string      name;
    std::vector<Selling*> sellings;

public:
    Food (const std::string& name);
    Food (const Food& food);

    virtual Food*   clone      () const = 0;

    std::string     getName    () const {return name;}
    virtual double  getPrice   () const = 0;

    virtual void    print     (const std::string& margin = "") const;

    double          getProfit   () const;

    void            addSelling   (Selling* selling);
    int             getSellingCount () const {return sellings.size();}
    Selling*        getSelling   (int i) {return sellings[i];}
};
  
```

Rodyklė su dideliu baltu trikampi viename gale žymi paveldėjimo sąryšį. Klasės

PrimitiveFood ir *CompositeFood* paveldi (turi savyje) visus bazinės klasės *Food* laukus (duomenis ir metodus). Pavyzdžiui, tiek paprasti, tiek ir sudėtiniai maisto produktai turi pavadinimą (*name*). Todėl pavadinimas saugomas bazinėje klasėje *Food*, o išvestinės klasės *PrimitiveFood* ir *CompositeFood* jį paveldi iš klasės *Food*.

Žodelis *virtual* žymi **polimorfinius** metodus. Virtualūs metodai gali būti perrašyti paveldėtose klasėse. Pavyzdžiui, paprasti produktai savyje turi kainą, o sudėtiniai produktai jos nesaugo savyje – jie apskaičiuoja kainą virtualaus metodo *getPrice* viduje sumuodami sudėtinių dalių kainas. Kadangi bazinėje klasėje *Food* mes iš viso negalime pateikti prasmingos metodo *getPrice* realizacijos, tai jis pažymėtas kaip “švariai virtualus” pabaigoje prirašant “= 0”. Švariai virtualūs metodai neturi jokio kūno (realizacijos), netgi tuščio. Klasė, kuri turi bent vieną švariai virtualų metodą vadinama **abstrakčia** klase. Mes negalime kurti abstrakčios klasės objektų. Abstrakčios klasės visuomet turi išvestines klases, kurios pateikia trūkstamas švariai virtualių metodų realizacijas.

Metodas *print* turi vienintelį parametą *margin* (paraštė) su reikšme pagal nutylėjimą. Tokį metodą galime kviešti su vienu parametru arba be parametru, tuomet bus perduota reikšmė pagal nutylėjimą.

Atsirado papildomas konstruktorius, priimančias vienintelį argumentą - konstantinę nuorodą į tos pačios klasės objektą: *Food (const Food& food)*. Toks konstruktorius vadinamas **kopijavimo konstruktoriumi**. Mūsų pavyzdyje jis naudojamas metoduose *clone*, kad pagaminti tikslų objekto kopiją.

Klasė *PrimitiveFood* paveldi visus laukus iš bazinės klasės *Food*, ir pateikia savas (perrašydama) švariai virtualių metodų *clone* ir *getPrice* realizacijas. Čia taip pat rasime nuosavą kopijavimo konstruktorių. Pats paveldėjimas yra nusakomas užrašu “: *public Food*”:

```
// food.h

class PrimitiveFood : public Food
{
private:
    double price;

public:
    PrimitiveFood (const std::string& name, double price);
    PrimitiveFood (const PrimitiveFood& food);

    virtual Food*   clone      () const;
    virtual double  getPrice   () const {return price;}
};
```

Sudėtinio maisto produkto klasė *CompositeFood* turi tik pavadinimą, kurį paveldi iš klasės *Food*. Visą kitą savyje saugo sudėtinės produkto dalys – kiti klasės *Food*

objektai:

```
// food.h

class CompositeFood : public Food
{
private:
    std::vector<Food*> foods;

public:
    CompositeFood (const std::string& name);
    CompositeFood (const CompositeFood& food);

    void    addFoodCopy (const Food* food);
    int     getFoodCount () const          {return foods.size();}
    Food*   getFood (int i)                {return foods[i];}
    const Food* getFood (int i) const {return foods[i];}

    virtual Food*   clone      () const;
    virtual double  getPrice  () const;
    virtual void    print (const std::string& margin = "") const;
};
```

Atkreipkime dėmesį, jog sudėtinio maisto klasė net tik perrašo švariai virtualius metodus *clone* ir *getPrice*, bet ir pateikia savą virtualaus metodo *print* realizaciją: jame atspausdina ir visas sudėtines savo dalis.

Gal kas nors jau pastebėjo, jog *CompositeFood* susideda iš rodyklių į *Food*. O mes juk sakėme, kad klasė *Food* turi švariai virtualių metodų, todėl ji yra abstrakti klasė, ir būdama abstrakčia klase negali turėti jokių sukurtų objektų. Reikalas tame, jog paveldėjimas susieja klases *yra-ryšiu (is-a)*. T.y. *PrimitiveFood yra Food*, ir *CompositeFood* irgi *yra Food*. Arba galime įsivaizduoti, kad *PrimitiveFood* ir *CompositeFood* turi savyje klasę *Food*. Programose tai reiškia: ten, kur reikia rodyklės (nuorodos) į bazinės klasės *Food* objektus galima perduoti rodyklės (nuorodos) į paveldėtų klasių *PrimitiveFood* ir *CompositeFood* objektus.

Šiame pavyzdyje mes turime failus su tais pačiais vardais, kaip ir praeitame: *food.h/cpp*, *selling.h/cpp* ir *main.cpp*. Failai *selling.h* ir *selling.cpp* yra identiškai praeito pavyzdžio bendravardžiams. Faile *food.h* mes jau matėme klasių *Food*, *PrimitiveFood* ir *CompositeFood* aprašus, o atitinkamame realizacijos faile *food.cpp* turime metodų apibrėžimus (kūnus).

Klasės *Food* metoduose mes nepamatysime didesnių naujovių:

```
// food.cpp

Food::Food (const string& name)
: name (name)
{
}

Food::Food (const Food& food)
: name(food.name)
{
}

void Food::print (const string& margin) const
{
    cout << margin << name << " " << getPrice() << endl;
}

double Food::getProfit () const
{
    double profit = 0;
    vector<Selling*>::const_iterator iter = sellings.begin();
    for (; iter != sellings.end(); iter++)
        profit += (*iter)->getProfit();
    return profit;
}

void Food::addSelling(Selling* selling)
{
    sellings.push_back(selling);
}
```

Konstruktorių viduje panaudota speciali duomenų laukų inicializavimo sintaksė. Ją galime naudoti tik konstruktoriuose. Taip siekiama suvienodinti bazinės klasės konstruktoriaus kvietimą ir laukų inicializavimą kaip kad klasėje *PrimitiveFood*:

```
// food.cpp

PrimitiveFood::PrimitiveFood (const string& name, double price)
: Food (name),
  price (price)
{
}

PrimitiveFood::PrimitiveFood (const PrimitiveFood& food)
: Food (food),
  price (food.price)
{
}

Food* PrimitiveFood::clone() const
{
    return new PrimitiveFood(*this);
}
```

Klasė *PrimitiveFood* yra *Food*, arba dar kartais laisvu žargonu sakoma, jog klasė *PrimitiveFood* turi savyje klasę *Food*. Todėl ji privalo inicializuoti ne tik savo

duomenų laukus (*price*), bet ir paveldėtus (*name*). Geras programavimo tonas įpareigoja klasės *Food* laukų inicializaciją palikti pačiai klasei *Food*, iškviečiant atitinkamą jos konstruktorių.

Metode *clone* mes vėl naudojame paslėptą parametrą *this*, kuris rodo į klasės *PrimitiveFood* objektą, kuriam iškviestas šis metodas. Kartu į pagalbą pasitelkę kopijavimo konstruktorių, mes grąžiname rodyklę į dinamiškai sukurtą kopiją.

Panašiai realizuota ir sudėtinio maisto klasė *CompositeFood*. Ji savyje saugo maisto objektų kopijas, kurias pasigamina metodo *clone* pagalba.

```
// food.cpp

CompositeFood::CompositeFood (const string& name)
: Food (name)
{
}

CompositeFood::CompositeFood (const CompositeFood& food)
: Food (food)
{
    for (int i = 0; i < food.getFoodCount(); i++)
        addFoodCopy(food.getFood(i));
}

Food* CompositeFood::clone() const
{
    return new CompositeFood(*this);
}

void CompositeFood::addFoodCopy (const Food* food)
{
    foods.push_back(food->clone());
}

double CompositeFood::getPrice () const
{
    double price = 0;
    for (int i = 0; i < foods.size(); i++)
        price += foods[i]->getPrice();
    return price;
}

void CompositeFood::print (const string& margin) const
{
    Food::print(margin);
    for (int i = 0; i < foods.size(); i++)
        foods[i]->print(margin + "    ");
}
```

Metode *print* matome bazinės klasės metodo kvietimo pavyzdį *Food::print(margin)*. Pastarasis atspausdina (kaip jau matėme klasėje *Food*), *CompositeFood* klasės vardą ir suminę kainą. O toliau yra spausdinamos sudėtinės dalys, patraukiant jas keliais tarpais į dešinę. Vieno programos veikimo metu buvo atspausdintas toks tekstas:

```

pienas 1.85
duona 0.85
sviestas 1.99
desra 2.15
sumustinis 4.99
    duona 0.85
    sviestas 1.99
    desra 2.15
alus 1.75
uzkanda 6.74
    sumustinis 4.99
        duona 0.85
        sviestas 1.99
        desra 2.15
    alus 1.75

```

Čia matome, jog paprastų maisto produktų buvo penki: pienas, duona, sviestas, dešra ir alus. Sudėtinis produktas “sumuštinis” susidėjo iš duonos, sviesto ir desros kopijų. O sudėtinis produktas “uzkanda” savyje turėjo lygiai du produktus: sudėtinį sumuštinį ir paprastą alų. Šis programos rezultatas demonstruoja, jog paveldėjimas drauge su polimorfizmu leidžia daugelyje programos vietų tiek sudėtinius, tiek ir paprastus objektus traktuoti vienodai.

Mums beliko panagrinėti failą *main.cpp*. Kad ir kaip bebūtų keista, jis praktiškai nepasikeitė, tik vietoje meniu punkto “sukurti” atsirado du punktai: “sukurti paprastą produktą” ir “sukurti sudėtinį produktą”. Nauja funkcija *addPrimitiveFood* yra tokia pati, kaip ir ankstesnio pavyzdžio *addNewFood*, todėl iš esmės failas *main.cpp* tepasipildė nauja funkcija *addCompositeFood*:

```

// main.cpp

void addCompositeFood (FoodList& foodList)
{
    string name;
    cout << "-- naujas sudetinis produktas --" << endl;
    cout << "pavadinimas: "; cin >> name;
    CompositeFood* composite = 0;
    for (;;)
    {
        cout << "-- pasirinkite produkto " << name
              << " dali --" << endl;
        Food* food = selectFood(foodList);
        if (food == 0)
            break;
        if (composite == 0)
            composite = new CompositeFood(name);
        composite->addFoodCopy(food);
    }
    if (composite != 0)
        foodList.push_back(composite);
}

```

Iš funkcijos *addCompositeFood* matome, jog naujus sudėtinius produktus galime konstruoti tik iš sąrašo *foodList* jau esančių produktų kopijų.

Viena esminė detalė, kurios mes nepalietėme įvade (neskaitant tūkstančio nepaliestų mažiau esminių detalių) yra objektų naikinimas. Mes visą laiką tikrai kūrėme objektus ir nė sykiu nenaikinome nebereikalingų objektų. Šio neišvengiamo meno pasimokinsime vėlesniuose skyriuose.

Įvado pabaigai pasižiūrėkime naujausią *make*-failo redakciją. Šis failas mums leidžia ne tik selektyviai kompiliuoti (*compile*) ir trinti sugeneruotus failus (*clean*), bet ir viską besąlygiškai perkompiliuoti (*build = clean + compile*), ir net paleisti sukompiliuotą programą (*run*). Kad nereiktų daugelyje vietų rašinėti ta patį failo vardą *main.exe*, įsivedėme kintamąjį *TARGET*:

```
# makefile

TARGET=main.exe

compile: $(TARGET)

run: compile
    $(TARGET)

main.exe      : main.o food.o selling.o
main.o        : main.cpp food.h selling.h
food.o        : food.cpp food.h selling.h
selling.o     : selling.cpp selling.h

#-----

clean:
    rm -f *.o
    rm -f *.exe

build: clean compile

%.exe: %.o
    g++ -o $@ $^

%.o: %.cpp
    g++ -c $<
```


2. Inkapsuliacija

2.1. Objektais paremtas programavimas (*object based programming*)

Objektais paremtu programavimo (dar ne OOP) esmė yra inkapsuliacija: duomenys ir funkcijos (metodai) dirbančios su jais laikomi kartu. Pavyzdžiui, grafinė figūra *Apskritimas* ne tik saugo savyje centro koordinates ir spindulį, bet dar ir moka pati save nupiešti ekrane ar printeryje, išsaugoti save faile ir t.t.. Niekas neturi teisės tiesiogiai keisti centro koordinatės ar spindulio dydžio, jei *Apskritimas* nepateikia atitinkamų metodų tam atlikti.

Apibrėžimas: inkapsuliacija – tai realizacijos paslėpimas po gerai apgalvotu interfeisu.

Objektais paremtas programavimas neturi paveldėjimo ir polimorfizmo sąvokų - tai objektiškai orientuoto programavimo privilegija.

2.2. Klasė, objektas, klasės nariai

C++ kalboje klasė yra struktūros sąvokos išplėtimas, t.y. struktūros papildymas funkcijomis.

Neformalus apibrėžimas: klasė = duomenys + metodai

Žemiau aprašyta fiksuoto maksimalaus dydžio sveikųjų skaičių steko klasė:

```
// stack.h

#ifndef __stack_h
#define __stack_h

class Stack
{
private:
    static const int MAX_SIZE = 10;

    int elements [MAX_SIZE];
    int size;

public:
    Stack ();

    void push (int element);
    int pop ();
    int peek ();
    bool isEmpty ();
};

#endif // __stack_h
```

Nepamirškite kabliataškio klasės aprašo pabaigoje. Antraip daugelis kompiliatorių pateikia labai miglotą klaidos pranešimą, ir ne h-faile, o kažkur cpp-failo viduryje. Klasė piešiama kaip dėžutė iš trijų sekcijų:

Stack	- klasės vardas
size: int elements: int[]	- duomenys
push (element: int) pop (): int peek (): int isEmpty (): bool	- metodai

Klausimas ne į temą: kas skaniau – spageti ar ravioli (James Rumbaugh, OMT, tekstas dėžutėse).

Klasė – duomenų tipas. Klasės tipo kintamieji (egzemplioriai) vadinami objektais. Objektams galioja tos pačios taisyklės, kaip ir visiems C/C++ kintamiesiems: jie gali būti statiniai, automatiniai arba dinaminiai.

Klasės nariai būna dviejų rūšių: nariai-duomenys (laukai) ir nariai-funkcijos (metodai). Metodai visuomet kviečiami kažkokiam objektui.

Žemiau pateiktas steko klasės panaudojimo pavyzdyje į tris stekus atitinkamai dedami sveikieji skaičiai, jų kvadratai ir kubai:

```
// demostack.cpp

Stack values;                                // global variable

int main ()
{
    Stack squares;                            // automatic variable
    Stack* cubes = new Stack;                 // dynamic variable

    for (int i = 1; i <= 10; i++)
    {
        values.push(i);
        squares.push(i*i);
        cubes->push(i*i*i);
    }

    while (!values.isEmpty())
        cout << values.pop() << " "
              << squares.pop() << " "
              << cubes->pop() << endl;

    delete cubes;
}
```

Programa *demostack.exe* atspausdins ekrane:

```
10 100 1000
9 81 729
8 64 512
7 49 343
6 36 216
5 25 125
4 16 64
3 9 27
2 4 8
1 1 1
```

Klausimas ne į temą: kokias funkcijas turi atlikti programa? Žiūrint kam ją naudosime (Ivar Jacobson, *Use Cases*, žmogeliukai prie ovalų).

Klasės metodai aprašomi nurodant jų pilną vardą. Tam naudojamas vardų erdvės išsprendimo operatorius (::):

```

// stack.cpp

#include "stack.h"

//-----
Stack :: Stack ()
{
    size = 0;
}
//-----
void Stack :: push (int element)
{
    if (size < MAX_SIZE)
    {
        elements [size] = element;
        size += 1;
    }
}
//-----
int Stack :: pop ()
{
    if (size > 0)
    {
        size -= 1;
        return elements [size];
    }
    return 0;
}
//-----
int Stack :: peek ()
{
    if (size > 0)
        return elements [size - 1];
    return 0;
}
//-----
bool Stack :: isEmpty ()
{
    return (size <= 0);
}

```

Kiekvienas objektas turi nuosavą duomenų egzempliorių. Tuo tarpu egzistuoja tik vienas kiekvieno metodo egzempliorius, bendras visiems objektams. Metodo kūno viduje visuomet yra apibrėžta rodyklė *this*, rodanti į objektą, kuriam šis metodas yra iškviestas. Steko metodą *peek()* mes galėjome užrašyti ir kitaip:

```

int Stack :: peek ()
{
    if (this->size > 0)
        return this->elements [this->size - 1];
    return 0;
}

```

Klausimas ne į temą: kuri profesija senesnė – fiziko, inžinieriaus ar programuotojo (Grady Booch, Booch method, debesėliai).

2.3. Klasės narių matomumas

Klasės metodų viduje yra matomi visi tos pačios klasės nariai: tiek duomenys, tiek ir metodai. Metodai laisvai manipuliuoja savo objekto būseną ir yra atsakingi už jos konsistentiškumą (teisingumą). Dažniausiai klasės metodai vieninteliai tiksliai žino, kaip teisingai elgtis su objekto būseną. Tuo tarpu kodui, naudojančiam klasę, neleidžiama kišti nagų prie objekto būsenos, o tik naudoti viešuosius metodus.

Tiek duomenys, tiek ir metodai gali būti privatūs (*private*) arba vieši (*public*). Tolesniuose skyriuose, kalbėdami apie paveldėjimą, susipažinsime ir su apsaugotais nariais (*protected*). Šie raktiniai žodžiai gali eiti bet kokia tvarka ir kartotis kiek norima kartų. Privatūs nariai yra pasiekiami tik klasės metodų viduje, o viešieji nariai pasiekiami visiems. Žemiau pateiktas kodas nekompetetingai kiša nagus prie steko vidinės realizacijos. Kompiliatorius išves klaidos pranešimą:

```
int main ()
{
    Stack stack;
    stack.size = -1; // error: Stack::size is not accessible
                   //           in function main()
}
```

C++ kalboje struktūra (*struct*) yra ta pati klasė, kurios narių matomumas pagal nutylėjimą yra *public*. Klasės narių matomumas pagal nutylėjimą yra *private*. Geras programavimo stilius reikalauja visuomet išreikštininiu būdu nurodyti klasės narių matomumą.

2.4. Konstruktoriai ir destruktoriai

Apsirašykime steką, kuris dinamiškai didiną elementams skirtą masivą. Išsiskiria dvi sąvokos: steko dydis – kiek steke yra elementų, ir steko talpa – kokio dydžio yra šiuo metu dinamiškai sukurtas elementų masyvas (nors nebūtinai užpildytas).

```
class Stack
{
private:
    int* elements;
    int size;
    int capacity;
public:
    Stack (int initialCapacity = 4);
    ~Stack ();
    void push (int element);
    int pop ();
    int peek ();
    bool isEmpty ();
};
```

Sukūrus steko objektą, reikia pasirūpinti, kad būtų teisingai inicializuoti steko laukai *elements*, *capacity* ir *size*. Tuo tikslu C++ turi specialų metodą - konstruktorių. Jo tikslas - sukonstruoti objektą, t.y. inicializuoti jo duomenis. Sukūrus klasės egzempliorių (objektą), visuomet yra iškviečiamas konstruktorius - to neįmanoma išvengti. Konstruktoriaus vardas yra toks pats kaip ir klasės. Konstruktorių gali būti keli, tuomet jie skirsis argumentų sąrašais (signatūromis). Jie negrąžina jokios reikšmės, netgi *void*:

```
Stack::Stack (int initialCapacity)
{
    if (initialCapacity < 1)
        initialCapacity = 1;
    capacity = initialCapacity;
    size = 0;
    elements = new int [capacity];
}
```

Naikinant objektą, jis privalo atlaisvinti visus naudojamus bendrus resursus: dinaminę atmintį, atidarytus failus ir t.t.. Tuo tikslu C++ turi specialų metodą - destruktorių. Sunaikinus klasės egzempliorių (objektą), visuomet yra iškviečiamas destruktorius - to neįmanoma išvengti. Destruktoriaus vardas yra toks pats kaip ir klasės, tik su bangele priekyje (~). Klasė gali turėti tik vieną destruktorių, kuris neturi jokių argumentų ir negrąžina jokios reikšmės, netgi *void*:

```
Stack::~~Stack ()
{
    delete[] elements;
}
```

Yra plačiai paplitę naudoti konstruktorių ir destruktorių poroje: konstruktorius pasiima resursus (pvz., sukuria dinaminis kintamuosius, atidaro failus), o destruktorius atlaisvina juos (pvz., atlaisvina dinaminę atmintį, uždaro failus).

Žiūrint iš vartotojo pusės, dinaminis stekas elgiasi taip pat, kaip ir fiksuoto dydžio stekas. Tik šį sykį galime neriboti savęs ir prigrūsti ne 10, o pvz. 20 skaičių. Parametrai konstruktoriams perduodami objekto konstravimo metu. Žemiau esančiame kodo fragmente pradinė steko values talpa bus 40 elementų, squares - 4 elementai (konstruktorius su parametru pagal nutylėjimą), cubes - 20 elementų:

```
// dynastack.cpp

#include <iostream>
#include "stack.h"
using namespace std;

Stack values(40); // global variable

int main ()
{
    Stack squares; // automatic variable
    Stack* cubes = new Stack(20); // dynamic variable

    for (int i = 1; i <= 20; i++)
    {
        values.push (i);
        squares.push (i*i);
        cubes->push (i*i*i);
    }

    while (!values.isEmpty())
        cout << values.pop () << " "
             << squares.pop () << " "
             << cubes->pop () << endl;

    delete cubes;
}
```

2.5. Konstruktorius pagal nutylėjimą

Konstruktorius pagal nutylėjimą yra iškviečiamas be argumentų, t.y. jis neturi argumentų, arba visi jo argumentai turi reikšmes pagal nutylėjimą. Mūsų dinaminis stekas turi konstruktorių pagal nutylėjimą, kuris turi vieną argumentą, pradinę tuščio steko talpą, su jam priskirta reikšme pagal nutylėjimą:

```
class Stack {  
    ...  
public:  
    Stack (int initialCapacity = 4);  
    ~Stack ();  
    ...  
};
```

Pastaba: tik tuo atveju, kai klasė neturi jokio konstruktoriaus, kompiliatorius sugeneruos konstruktorių pagal nutylėjimą.

Išvadėlė: galima apsirašyti klasę, kuri neturės konstruktoriaus pagal nutylėjimą, o tik konstruktorius, reikalaujančius parametrų.

2.6. Kopijavimo konstruktorius

C++ vienas iš konstruktorių turi specialę paskirtį: konstruktorius, kurio vienintelis parametras yra nuoroda į tos pačios klasės objektą. Jis vadinamas kopijavimo konstruktoriumi. Apsirašykime standartine datos klasę su konstruktoriumi pagal nutylėjimą ir su kopijavimo konstruktoriumi:

```
// copyconstr.cpp

class Date
{
private:
    int year;
    int month;
    int day;
public:
    Date (int year = 2003, int month = 2, int day = 28);
    Date (const Date& date);
    ~Date ();

    int getYear () const {return year;}
    int getMonth () const {return month;}
    int getDay () const {return day;}
    void print (const string& prefix) const;
};
```

Kartu su interfeisu tame pačiame *cpp*-faile ir klasės realizacija:

```
Date::Date (int year, int month, int day)
{
    this->year = year;
    this->month = month;
    this->day = day;
    print("Date");
}
//-----
Date::Date (const Date& date)
{
    year = date.getYear();
    month = date.getMonth();
    day = date.getDay();
    print("Date(Date&)");
}
//-----
Date::~Date ()
{
    print("~Date");
}
//-----
void Date::print (const string& prefix) const
{
    cout << prefix << "("
        << year << " " << month << " " << day << ")" << endl;
}
```

Kopijavimo konstruktorių galima iškviešti dvejopai: kaip ir visus kitus konstruktorius objekto kūrimo metu, arba naudojant priskyrimo simbolį. Priskyrimo ženklas objekto sukūrimo metu kviečia kopijavimo konstruktorių, o ne priskyrimo operatorių:

```
int main ()
{
    Date yesturday (2003, 2, 27);
    Date date1 = yesturday; // copy constructor
    Date date2 (yesturday); // preferred syntax
}
```

Programa atspausdins:

```
Date(2003 2 27)
Date(Date&)(2003 2 27)
Date(Date&)(2003 2 27)
~Date(2003 2 27)
~Date(2003 2 27)
~Date(2003 2 27)
```

Jei klasė neturi kopijavimo konstruktoriaus, tuomet jį sugeneruos kompiliatorius. Šiuo atveju klasės duomenys bus inicializuoti panariui. Mūsų klasės *Date* atveju kompiliatoriaus sugeneruotas kopijavimo konstruktorius veiktų lygiai taip pat, kaip ir mūsų parašytasis. Jis netiktų tuomet, kai klasė savyje turi rodyklės į kitus objektus. Tuomet būtų kopijuojamos tik pačios rodyklės (adresai), o ne objektai, į kuriuos jos rodo.

2.7. Konstruktoriai ir tipų konversija

Panagrinėkime kompleksinius skaičius, kurie turi realiąją ir menamąją dalis:

```
// typeconstr.cpp

class Complex
{
private:
    double r;
    double i;

public:
    Complex (double re=0, double im=0) : r(re), i(im) {}
    Complex add (const Complex& c);

    void print();
};
```

Trumputė klasės realizacija:

```
Complex Complex::add (const Complex& c)
{
    return Complex(r+c.r, i+c.i);
}

void Complex::print ()
{
    cout << "(" << r << ", " << i << ")" << endl;
}
```

Atlikime keletą skaičiavimų su kompleksiniais skaičiais:

```
int main ()
{
    Complex a (10, 2);
    a = a.add( Complex(2.0));
    a.print();
    a = a.add( Complex(2, -1) );
    a.print();
    a = a.add( 0.5 );
    a.print();
}
```

Programa atspausdins:

```
(12, 2)
(14, 1)
(14.5, 1)
```

Kiek keistai atrodo funkcijos *main* kodo eilutė:

```
a = a.add( 0.5 );
```

Juk klasė *Complex* neturi metodo

```
Complex Complex::add (double d);
```

C++ už "programuotojo nugaros" atlieka automatinę tipų konversiją naudojant konstruktorių. Aukščiau užrašyti eilutė yra ekvivalentiška žemiau esančiai:

```
a = a.add( Complex(0.5) );
```

Toks išreikštinis ar neišreikštinis konstruktoriaus kvietimas sukuria laikiną automatinį objektą, egzistuojantį tik išraiškos skaičiavimo metu. O ką, jei mes nenorime, kad kompiliatorius už programuotojo nugaros kvieštų konstruktorių tipų konversijai atlikti ir laikinam objektui sukurti? T.y. mes patys norime valdyti, kas ir kada yra kviečiama. Tuo tikslu, aprašant konstruktorių, naudojamas raktinis žodis *explicit*:

```
class Complex
{
    ...
    explicit Complex (double re=0, double im=0);
};

void f ()
{
    Complex a (10, 2);
    ...
    a = a.add(0.5);           // syntax error
    a = a.add(Complex(0.5));  // OK
}
```

2.8. Objektų masyvai

Jei klasės turi konstruktorių pagal nutylėjimą, tuomet mes galime apsirašyti jos objektų masyvą. Kita alternatyva - inicializatorių sąrašas. Pasinaudokime klase *Date* iš ankstesnio skyrelio:

```
// array.cpp

int main ()
{
    Date defaultDates [2];
    Date customDates[] = {Date(1500, 1, 3), Date(1999, 4, 20)};
    cout << customDates[1].getYear() << endl;
}
```

Programa atspausdins:

```
Date(2003 2 28)
Date(2003 2 28)
Date(1500 1 3)
Date(1999 4 20)
1999
~Date(1999 4 20)
~Date(1500 1 3)
~Date(2003 2 28)
~Date(2003 2 28)
```

Nepamirškime, kad C++ kalboje nėra skirtumo tarp rodyklės į atskirą objektą ir rodyklės į pirmąjį masyvo elementą. Programuotojas pats atsakingas už tai, kad objektai, sukurti su *new* būtų sunaikinti su *delete*, o objektai sukurti su *new[]* būtų sunaikinti su *delete[]*:

```
// newarray.cpp

int main ()
{
    Date* date = new Date(1000, 1, 1);
    Date* two = new Date [2];
    cout << date->getYear() << endl;
    cout << two[0].getYear() << endl; // same as bellow
    cout << two->getYear() << endl;
    delete date;
    delete[] two;
}
```

Ši programa atspausdins:

```
Date(1000 1 1)
Date(2003 2 28)
Date(2003 2 28)
1000
2003
2003
~Date(1000 1 1)
~Date(2003 2 28)
~Date(2003 2 28)
```

Paprastai masyvuose saugomi nedideli objektai. Esant didesniems objektams, dalyvaujantiems klasių hierarchijose, masyvuose saugomos rodyklės į juos.

2.9. Objektas, kaip kito objekto laukas (agregacija)

Panagrinėkime atvejį, kai objekto duomenimis yra kiti objektai, pvz. asmuo turi vardą, gimimo datą ir ūgį:

```
// person.cpp

class Person
{
private:
    string name;
    Date birth;
    float height;
public:
    Person ();
    Person (const string& name, const Date& birth, float height);
    ~Person ();
};
```

Konstruojant objektą pradžioje bus sukonstruoti objekto laukai, o tik po to bus iškvieštas paties objekto konstruktorius:

```
Person::Person ()
{
    height = 0;
    cout << "Person()" << endl;
}

int main ()
{
    Person person;
}
```

Person konstruktorius pradžioje iškvies objekto *name* konstruktorių pagal nutylėjimą, tuomet objekto *date* konstruktorių, o tik po to įvykdys savo paties kūną ir inicializuos lauką *height*. Aukščiau pateiktas kodo fragmentas atspausdins:

```
Date(2003 2 28)
Person()
~Person() 0
~Date(2003 2 28)
```

Galima išreikštiniu būdu nurodyti, kurį duomenų nario konstruktorių kviešti. Tuomet dar prieš objekto konstruktoriaus kūną dedamas dvitaškis, po kurio vardijami kableliais atskirti objekto laukų konstruktoriai:

```
Person::Person (const string& n, const Date& b, float h)
: name(n), birth(b), height(h)
{
    cout << "Person() " << name << " " << height << endl;
}
```

```
int main ()
{
    Person zigmas("zigmas", Date(1984, 4, 17), 178.5);
}
```

Atkreipkite dėmesį, kaip funkcijoje *main* klasės *Person* konstruktoriui perduodamas laikinas klasės *Date* objektas, kuris egzistuoja tik konstruktoriaus kvietimo metu ir yra iš karto sunaikinamas. Žemiau pateiktame programos išvedime pirmasis *Date* konstruktorius ir pirmasis destruktorius priklauso šiam laikinajam objektui:

```
Date(1984 4 17)           - sukuriamas laikinas objektas
Date(Date&)(1984 4 17)
Person() zigmas 178.5
~Date(1984 4 17)          - sunaikinamas laikinas objektas
~Person() zigmas 178.5
~Date(1984 4 17)
```

C++ kalboje tokią objekto laukų inicializavimo užrašymo būdą galima naudoti ne tik laukams objektams, bet ir baziniams C++ tipams (*int*, *char*, *double*, ...). Laukų konstruktoriai iškviečiami ta tvarka, kuria laukai yra aprašyti klasės apraše, o ne ta tvarka, kaip jie išvardinti konstruktoriuje inicializatorių sąraše.

Konstruojant objektą, pradžioje sukonstruojami jo objektai-laukai ir tik po to kviečiamas objekto konstruktoriaus kūnas. Destruktorių kūnai kviečiami atvirkščia tvarka.

2.10. Objektų gyvavimo trukmė

Panagrinėkime, kaip ir kada yra kuriami ir naikinami objektai. Tuo tikslu apsirašykime klasę, su besipasakojančiais konstruktoriumi ir destruktoriumi:

```
// lifecycle.cpp

class A
{
private:
    string text;
public:
    A(const char* t) : text(t) {cout << " " << text << endl;}
    ~A() {cout << "~" << text << endl;}
};
```

Dinaminiai objektai konstruojami operatoriaus *new* pagalba, o naikinami su *delete*:

```
int main ()
{
    A* d = new A ("dynamic");
    delete d;
}
```

Šis kodo fragmentas atspausdins:

```
dynamic
~dynamic
```

Automatiniai objektai konstruojami tuomet, kai programos vykdymas pasiekia jų aprašymo vietą (funkcijos kūną, kodo bloką), o naikinami vykdymui paliekant kodo bloką. C++ kalboje kiekvienos ciklo iteracijos metu yra įeinama ir išeinama iš ciklo kūno, todėl ciklo kūną aprašyti automatiniai objektai bus konstruojami ir naikinami kiekvienos iteracijos metu. Ciklo inicializavimo dalyje aprašyti objektai konstruojami tik vieną kartą - prieš visas iteracijas, ir naikinami joms pasibaigus, t.y. tokie objektai gyvuoja tik vieno ciklo sakinio viduje ir neegzistuoja jo išorėje. Ankstyvieji C++ kompiliatoriai elgdavosi truputį kitaip, ne pagal standartą: ciklo inicializavimo dalyje aprašyti kintamieji ir toliau gyvuodavo už ciklo ribų.

Jei funkcija grąžina lokalų objektą kaip savo rezultatą, tuomet į žaidimą gali įsijungti ir kopijavimo konstruktorius (jei optimizuojantis kompiliatorius neišvengs jo kvietimo).

```

void hasCycle()
{
    A l("local");
    cout << "enter hasCycle()" << endl;
    int i = 0;
    for (A ic("init cycle"); i < 2; i++)
    {
        A cb("cycle body");
    }
    cout << "leave hasCycle()" << endl;
}

int main
{
    hasCycle();
}

```

Šis kodo fragmentas atspausdins:

```

local
enter hasCycle()
init cycle
cycle body
~cycle body
cycle body
~cycle body
~init cycle
leave hasCycle()
~local

```

Globalūs statiniai objektai konstruojami prieš vykdant funkciją *main*, o naikinami po to, kai išeinama iš *main*, arba kai programa baigiama standartinės funkcijos *exit* pagalba. Jie nebus naikinami, jei programa bus užbaigta funkcijos *abort* pagalba.

```

A gs("global static");

int main ()
{
    cout << "enter main()" << endl;
    cout << "leave main()" << endl;
}

```

Šis kodo fragmentas atspausdins:

```

global static
enter main()
leave main()
~global static

```

Lokalūs statiniai objektai yra vartojami labai retai. Jie konstruojami, kai programos vykdymas pirmą kartą pasiekia jų aprašymo vietą, o naikinami programos pabaigoje:

```

void hasLocalStatic()
{
    cout << "enter hasLocalStatic()" << endl;
    static A ls("local static");
    cout << "leave hasLocalStatic()" << endl;
}

int main ()
{
    cout << "enter main()" << endl;
    hasLocalStatic();
    hasLocalStatic();
    cout << "leave main()" << endl;
}

```

Šis kodo fragmentas atspausdins:

```

enter main()
enter hasLocalStatic()
  local static
leave hasLocalStatic()
enter hasLocalStatic()
leave hasLocalStatic()
leave main()
~local static

```

Jei programos vykdymas nepasiekia funkcijos/metodo su lokaliu statiniu objektu, tai jis niekada nebus sukonstruotas ir sunaikintas.

2.11. Metodai, apibrėžti klasės aprašo viduje

Metodai, apibrėžti klasės aprašo viduje (*h*-faile) yra *inline*-metodai, t.y. kompiliatorius jų iškviatimo vietoje stengsis tiesiogiai įterpti metodo kūną, o ne kviesti klasės realizacijos modulyje esantį kūną. Tuo būdu sutaupomas metodo kvietimo laikas:

```
class String
{
    ...
    const char* getChars() const {return buffer;}
};

int main()
{
    String text ("hello");
    const char* chars = text.getChars(); // chars = text.buffer
    ...
}
```

Praktikoje klasės viduje apibrėžiami tik kai kurie trumpučiai vienos eilutės metodai, tuo pačiu įnešant daugiau aiškumo į klasės aprašą. *inline*-metodai gali būti apibrėžti ir klasės išorėje, naudojant raktinį žodį *inline*:

```
inline const char* String::getChars() const
{
    return buffer;
}
```

Bet kokių būdu aprašytas *inline*-metodas – tai tik rekomendacija kompiliatoriui, kad jis vietoje metodo kvietimo, įterptų metodo kūną. Kompiliatoriai gali to ir neatlikti. Kai kurie griežtesni kompiliatoriai spausdina įspėjimus apie *inline*-metodus, kurių kūnai niekada nebus įterpti į kodą, o visuomet bus formuojamas metodo kvietimas.

2.12. Statiniai nariai

Statiniai klasės metodai yra panašūs į paprastas C kalbos funkcijas. Jie nesusieti su jokių konkrečių objektu, todėl jų viduje nėra rodyklės *this*. Statiniai klasės duomenys yra panašūs į globalius kintamuosius. Skirtumas yra tas, jog statiniai nariai (duomenys ir metodai) yra aprašomi klasės viduje ir, analogiškai kaip ir įprasti metodai, gali prieiti prie klasės narių. Be to, statiniams nariams galioja matomumo taisyklės (*private*, *protected*, *public*). Statiniai duomenys yra bendri visiems klasės egzemplioriams, skirtingai nuo nestatinių duomenų, kuriuos kiekvienas objektas turi nuosavus. Žemiau esantis pavyzdys skaičiuoja, kiek iš viso buvo sukurta duotos klasės objektų ir kiek iš jų dar gyvi:

```
// counted.h

class Counted
{
private:
    static int createdCount;
    static int livingCount;

public:
    Counted ();
    ~Counted ();

    static int getCreatedCount () {return createdCount;}
    static int getLivingCount  () {return livingCount;}
};
```

Statinius narius duomenis reikia ne tik aprašyti klasės viduje h-faile, bet ir apibrėžti kartu su klasės realizacija cpp-faile.

```
// counted.cpp

#include "counted.h"

int Counted::createdCount = 0;
int Counted::livingCount  = 0;

Counted::Counted ()
{
    createdCount += 1;
    livingCount  += 1;
}

Counted::~~Counted ()
{
    livingCount -= 1;
}
```

Pasinaudokime tuom, jog ciklo viduje aprašyti objektai sukuriami kiekvienos iteracijos pradžioje, o jos pabaigoje vėl sunaikinami. Kadangi statiniai nariai nesusieti

su jokių konkrečių tos klasės objektu, jie kviečiami nurodant jų pilną vardą: *klasės-vardas::metodo-vardas*:

```
// main.cpp

#include <iostream>
#include "counted.h"
using namespace std;

int main ()
{
    Counted c1;
    Counted c2;
    for (int i = 0; i < 10; i++)
    {
        Counted c3;
    }
    cout << "Created: " << Counted::getCreatedCount() << endl;
    cout << "Living: " << Counted::getLivingCount() << endl;
}
```

Šis kodo fragmentas atspausdins:

```
Created: 12
Living: 2
```

Ir dar vienas praktiškas pavyzdys: klasė, galinti turėti ne daugiau kaip vieną objektą. Projektuotojai šiai šabloninei situacijai sugalvojo pavadinimą – *Singleton*. Triukas: paslėpkime konstruktorių (padarykime jį *private*) ir pateikime statinį viešą metodą, grąžinantį rodyklę į vienintelę klasės egzempliorių. Žemiau randame ištrauką:

```
class Singleton
{
private:
    static Singleton* instance;
    Singleton ();
public:
    static Singleton* getInstance ();
    ...
};
//-----
Singleton* Singleton::instance = 0;
Singleton* Singleton::getInstance()
{
    if (instance == 0)
        instance = new Singleton();
    return instance;
}
//-----
int main ()
{
    for (int i = 0; i < 10; i++)
        Singleton::getInstance()->doSomething(i*i);
    Singleton::getInstance()->print();
}
```

2.13. Klasės draugai

Paprastam klasės metodui garantuojami trys dalykai:

1. metodas turi priėjimo teises prie apsaugotos klasės aprašo dalies
2. metodo matomumas nusakomas raktiniais žodžiais *public*, *protected* ir *private*
3. turi būti kviečiamas klasės objektui (turi rodyklę *this*)

Statiniams klasės metodams (*static*) galioja tik pirmi du punktai.

Funkcijos-draugai (*friend*) pasižymi tik pirmąja savybe. Pavyzdys: sudėkime du stekus į vieną:

```
// friend.cpp

class Stack
{
private:
    int* elements;
    int size;
    int capacity;
    ...
    friend Stack* addStacks (const Stack* s1, const Stack* s2);
};

Stack* addStacks (const Stack* s1, const Stack* s2)
{
    Stack* result = new Stack();
    result->size = s1->size + s2->size;
    result->capacity = result->size + 1;
    result->elements = new int [result->capacity];
    for (int i = 0; i < s1->size; i++)
        result->elements[i] = s1->elements[i];
    for (int i = 0; i < s2->size; i++)
        result->elements[s1->size + i] = s2->elements[i];
    return result;
}
//-----
int main ()
{
    Stack values;
    Stack squares;
    for (int i = 1; i <= 7; i++)
    {
        values.push (i);
        squares.push (i*i);
    }
    Stack* sum = addStacks(&values, &squares);
    while (!sum->isEmpty())
        cout << sum->pop() << " ";
    delete sum;
}
```

Programa atspausdins:

```
49 36 25 16 9 4 1 7 6 5 4 3 2 1
```

Nesvarbu kurioje klasės dalyje aprašysime (*private*, *protected* ar *public*) klasės draugą. Jį vis vieną bus galima naudoti klasės išorėje (tarsi *public* narį), o jis pats turės priėjimą prie visų klasės duomenų.

Klasės draugais gali būti ne tik paprastos funkcijos, bet ir kitos klasės metodai:

```
class ListIterator
{
    ...
    int* next();
};

class List
{
    ...
    friend int* ListIterator::next();
};
```

Klasės draugu gali būti ir kita klasė, t.y. visi kitos klasės metodai:

```
class List
{
    ...
    friend class ListIterator;
};
```

2.14. Tipų aprašai klasės viduje (įdėtiniai tipai)

Klasės viduje mes galime aprašyti bet kokius kitus tipus, pavyzdžiui, išvardijamus tipus (*enum*), *typedef*-konstrukcijas, kitas klases:

```
// innertype.cpp

class List
{
public:
    typedef string Element;
    enum Position {FIRST, LAST};
private:
    class Node {
    private:
        Element element;
        Node* next;
    public:
        void removeTail ();
    };
};
```

Tai yra įprastiniai tipai, tik jų vardai turi būti pilnai kvalifikuoti:

```
void List::Node :: removeTail()
{
    if (next != 0)
    {
        next->removeTail();
        delete next;
        next = 0;
    }
}

int main () {
    List::Element element = "stabdis";
    List::Position position = List::LAST;
    cout << element << " " << position << endl;
}
```

Įdėtiniais tipams galioja klasės narių matomumo taisyklės:

```
int main ()
{
    List::Node node; // klaida - List::Node yra private
}
```

Įdėtinės klasės neturi jokių privilegijų. Pvz. klasės *List::Node* metodai negali prieiti prie apsaugotų klasės *List* narių.

2.15. Vardų erdvės išsprendimo operatorius ::

Mes jau vartojome operatorių (::):

- norėdami apibrėžti metodo realizaciją klasės išorėje
- naudodami viešuosius statinius klasės narius klasės išorėje
- naudodami klasės viduje apibrėžtus tipus.

Šis operatorius taipogi leidžia pasiekti globaliuosius vardus, kuriuos konkrečioje situacijoje paslepia lokalūs kintamieji ar parametrai:

```
class MyClass
{
    ...
    int value;
};

int value; // global variable

void MyClass::myMethod (int value)
{
    this->value = 1;    // class member
    value      = 2;    // method parameter
    ::value    = 3;    // global variable
}
```

2.16. Konstantiniai laukai, laukai-nuorodos

Konstantas ir nuorodas būtina inicializuoti jų apibrėžimo vietoje. Vėliau mes nebegalime keisti konstantos reikšmės, o nuoroda taip ir liks susieta su tuo pačiu objektu:

```
const int MAX_SIZE = 15;
Stack& reference = anotherStack;
```

Trumpai prisiminke rodykles ir konstantas:

```
const char *      pointerToConstString      = "s1";
char const *      pointerToConstString      = "s2";
char * const      constPointerToString      = "s3";
const char * const constPointerToConstString = "s4";
const char const * constPointerToConstString = "s5";
```

Jei klasė turi konstantinių laukų arba laukų nuorodų, jie privalo būti inicializuoti konstruktoriaus inicializatorių sąrašė:

```
class List
{
public:
    const int MAX_SIZE;
    Node&      defaultNode;
    List (int maxSize, Node& defNode);
};

List::List (int maxSize, Node& defNode)
: MAX_SIZE      (maxSize),
  defaultNode   (defNode)
{
}
```

Statiniai konstantiniai laukai yra inicializuojami aprašymo vietoje (senesni kompiliatoriai reikalaudavo apibrėžti ir inicializuoti tokią konstantą *cpp*-faile, kartu su kitais statiniais laukais):

```
class List
{
public:
    static const int DEFAULT_MAX_SIZE = 18;
};

int main ()
{
    cout << List::DEFAULT_MAX_SIZE << endl;
}
```

2.17. Konstantiniai metodai ir *mutable*-laukai

O kas tuomet, kai konstanta yra pats objektas? Tuomet, po jo apibrėžimo, mes nebeturime teisės keisti nei vieno jo lauko. Taip pat tokiam objektui negalime kviesti metodų, jei jie nėra konstantiniai. Konstantiniai metodai turi raktinį žodelį *const* savo signatūros pabaigoje ir savo viduje negali keisti objekto laukų:

```
// constmet.cpp

class Date
{
public:
    int getYear    () const {return year;}
    int getMonth   () const {return month;}
    int getDay     () const {return day;}
};

const Date defaultDate (2003, 3, 14);

int main ()
{
    cout << defaultDate.getYear    () << " "
          << defaultDate.getMonth   () << " "
          << defaultDate.getDay     () << endl;
}
```

Žodelis *const* priklauso metodo signatūrai, todėl klasė gali turėti du metodus su tais pačiais argumentų sąrašais, bet besiskiriančiais *const* žodelio buvimu. Žemiau pateikti metodai grąžina nuorodą į objekto laukus, t.y. jie palieka galimybę keisti laukus iš išorės, todėl mes negalime jų aprašyti konstantiniais:

```
// constmet.cpp

class Date
{
public:
    int& getYear    () {return year;}
    int& getMonth   () {return month;}
    int& getDay     () {return day;}
};

int main ()
{
    Date date2;
    date2.getYear() = 2004; // non-const method called
    cout << date2.getYear    () << endl;
}
```

Kartais objekto loginė būseną nepasikeis, nors ir bus pakeisti kai kurie laukai. Dažniausiai tai būna apskaičiuojami laukai, atliekantys *cache*-stiliaus funkcijas. Jie aprašomi su žodeliu *mutable*. Yra leidžiama keisti konstantinio objekto *mutable*-

laukus. Žemiau mes iliustruojame situacija, kuomet metodas *getString* grąžina tekstinę eilutę. Laikome, jog pastaroji gali būti apskaičiuojama iš kitų objekto laukų, tik jos skaičiavimas labai ilgai užtrunka. Todėl mes saugome požymį *cacheValid*, kuris parodo, ar nuo paskutinio eilutės skaičiavimo pasikeitė objekto laukai, ar ne. Jei laukai pasikeitė, tai esame priversti iš naujo suskaičiuoti *cacheString* pagal naujas laukų reikšmes. Pati *cacheString* nelaikome objekto būseną, nes yra apskaičiuojama iš kitų objekto laukų:

```
class Date
{
private:
    mutable bool    cacheValid;
    mutable string  cacheString;

    void computeCacheString() const;
    ...
public:
    ...
    string getString() const;
};

string Date::getString() const
{
    if (!cacheValid)
    {
        computeCacheString();
        cacheValid = true;
    }
    return cacheString;
}
```


3. Paveldėjimas ir polimorfizmas

3.1. Trys OOP banginiai

Mes jau artimai susipažinome su vienu iš Objektiškai Orientuoto Programavimo banginių - inkapsuliacija. Šiame skyrelyje panagrinėsime likusius du - paveldėjimą ir polimorfizmą.

Paveldėjimas ir polimorfizmas gerokai palengvina daugkartinį kodo panaudojimą:

- galime naudotis jau anksčiau parašytais klasėmis kaip bazinėmis tam, kad sukurti savas, specializuotas, kartu paveldint visus bazinės klasės duomenis ir kodą. Belieka tik prirašyti paveldėtai klasei specifinius laukus ir metodus;
- kodas, manipuliuojantis objektais, dažnai gali su visa klasių hierarchija dirbti vienodai, t.y. jis neturi būti dubliuojamas kiekvienai klasei atskirai, o rašomas tik bazinei klasei.

Šie du teoriniai pamastymai plačiau atsiskleis, kuomet panagrinėsime keletą pavyzdžių. Vienas natūraliausių objektiškai orientuoto programavimo pritaikymų - kompiuterinė grafika. Tarkime, turime abstrakčią figūros sąvoką *Shape*, ir konkrečias figūras *Circle*, *Square* ir *Rectangle*. Tuomet:

- paveldėjimas leidžia užrašyti natūralų faktą: *Circle*, *Square* ir *Rectangle* yra figūros. OOP sąvokomis sakoma, kad jos paveldi iš klasės *Shape*. Tokiu būdu, jei bazinė figūra turės centro koordinates, tai ir paveldėjusios figūros jas turės.
- polimorfizmas leidžia realizuoti apskritimų, kvadratų ir stačiakampių elgsenos skirtumus, pvz. paišymą, ploto skaičiavimą ir pan..

Grafiniuose pavyzdžiuose naudosime biblioteką *Qt*, kurią galima rasti internete adresu www.trolltech.com

Naudojimosi šia biblioteka dokumentacija su pamokomis yra adresu doc.trolltech.com

3.2. Paveldėjimas

Turime geometrinę figūrą su centro koordinatėmis x ir y , kuri piešia save kaip vieną tašką:

```
// shapes.h

class Shape
{
protected:
    int x;
    int y;

public:
    Shape (int cx, int cy);

    int  getCenterX () const {return x;}
    int  getCenterY () const {return y;}
    void setCenter  (int cx, int cy) {x=cx; y=cy;}

    void draw      (QPainter* p);
};
```

Apskritimas, kvadratas ir stačiakampis taip pat yra figūros turinčios centrą. Objektiškai orientuoto programavimo terminologijoje žodelis "yra" keičiamas žodeliu "paveldi". Taigi, apskritimas, kvadratas ir stačiakampis paveldi iš figūros visas jos savybes bei prideda papildomų duomenų ir metodų. Šis faktas C++ kalboje užrašomas tokiu būdu.

```
class Circle : public Shape
{
protected:
    int radius;

public:
    Circle (int cx, int cy, int radius);
    int  getRadius () const {return radius;}
    void setRadius (int r) {radius = r;}
    void draw      (QPainter* p);
};

class Square : public Shape
{
protected:
    int width;

public:
    Square(int cx, int cy, int width);
    int  getWidth () const {return width;}
    void setWidth  (int w) {width=w;}
    void draw      (QPainter* p);
};
```

```

class Rectangle : public Shape
{
protected:
    int width;
    int height;

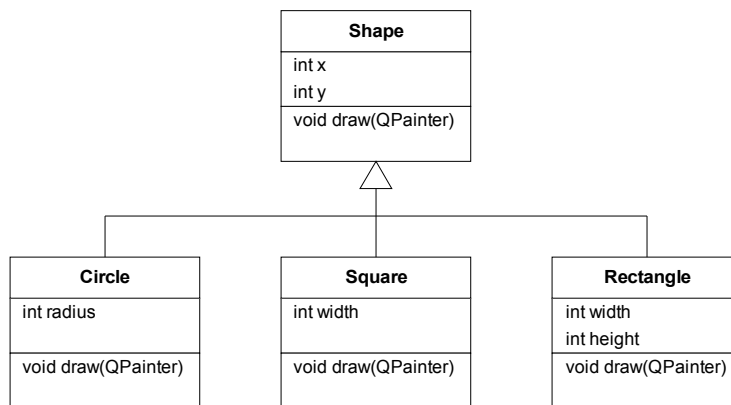
public:
    Rectangle(int cx, int cy, int width, int height);

    int  getWidth  () const {return width;}
    int  getHeight () const {return height;}
    void setSize   (int w, int h) {width=w; height=h;}

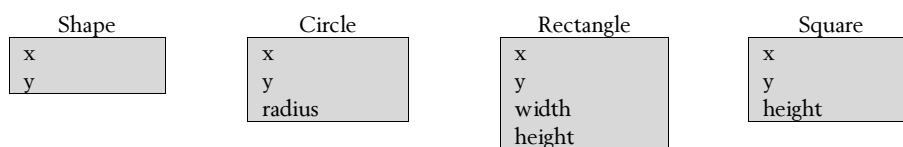
    void draw      (QPainter* p);
};

```

Gauname klasių hierarchiją, kurią grafiškai vaizduojame taip:



Klasė *Circle* paveldi visus duomenis laukus ir metodus iš klasės *Shape*. Tarytumei laukai *x* ir *y*, bei metodai *draw* ir t.t. būtų tiesiogiai aprašyti klasėje *Circle*. Kompiuterio atmintyje objektai atrodo maždaug taip:



Sakoma, kad klasė *Shape* yra klasės *Circle* bazinė klasė (*super-class*). Savo ruožtu klasė *Circle* paveldi iš klasės *Shape*, arba klasė *Circle* yra klasės *Shape* poklasė (*sub-class*).

Kiekviena figūra save piešia vis kitaip:

```

void Shape::draw(QPainter* p)
{
    p->drawPoint(x, y);
}
void Circle::draw(QPainter* p)
{
    p->drawEllipse(x-radius, y-radius, radius*2, radius*2);
}
void Square::draw(QPainter* p)
{
    p->drawRect(x-width/2, y-width/2, width, width);
}
void Rectangle::draw(QPainter* p)
{
    p->drawRect(x-width/2, y-height/2, width, height);
}

```

Pagalbinė klasė *Screen* parūpina vietą ekrane, kurioje figūros gali save nusipiešti:

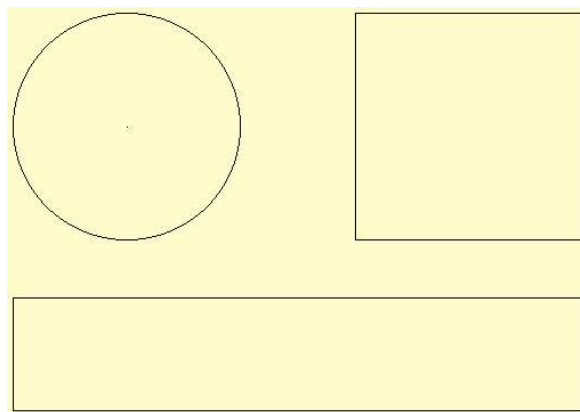
```

Screen::Screen(QWidget* parent, const char* name)
: QWidget(parent, name)
{
    setPalette(QPalette(QColor(250, 250, 200)));
    shape = new Shape(200, 200);
    circle = new Circle(200, 200, 100);
    square = new Square(500, 200, 200);
    rectangle = new Rectangle(350, 400, 500, 100);
}
void Screen::paintEvent(QPaintEvent*)
{
    QPainter p(this);

    shape->draw(&p);
    circle->draw(&p);
    square->draw(&p);
    rectangle->draw(&p);
}

```

Ekrane pamatysime:



3.3. Konstruktoriai ir destruktoriai

Paveldėta klasė yra konstruojama griežtai nustatytu būdu, nepriklausomai nuo to, kokia tvarka konstruktoriai surašyti inicializatorių sąrašė:

1. konstruojama bazinei klasei priklausanti objekto dalis
2. konstruojami paveldėtos klasės laukai-duomenys
3. įvykdomas paveldėtos klasės konstruktoriaus kūnas

```
Circle::Circle (int cx, int cy, int r)
: Shape(cx, cy),      // 1.
  radius(r)           // 2.
{
    // 3.
}
```

Jei inicializatorių sąrašė nėra nurodytas konkretus bazinės klasės konstruktorius, tai kviečiamas konstruktorius pagal nutylėjimą. Jei tokio nėra, kompiliatorius praneša apie klaidą.

Kaip visuomet: destruktoriai kviečiami tiksliai atvirkščia tvarka.

3.4. Bazinės klasės narių matomumas

Naudojant paveldėjimą, šalia *private* ir *public* klasės narių matomumo labai plačiai vartojamas tarpinis modifikatorius *protected* - apsaugotieji nariai. Jie pasiekiami paveldėjusios klasės metoduose, bet nepasiekiami klasės išorėje.

Taigi, paveldėjusios klasės metodai gali prieiti prie bazinės klasės *public* ir *protected* narių. Bazinės klasės *private* nariai nėra pasiekiami paveldėjusioje klasėje.

Klasės išorėje yra matomi tik *public* nariai.

	bazinės klasės narių matomumas		
nuosavi klasės metodai pasiekia	public	protected	private
paveldėjusios klasės metodai mato	public	protected	
išorinės funkcijos pasiekia	public		

3.5. Metodų perkrovimas (*overloading*) ir pseudo polimorfizmas

Klasę *Circle* su klase *Shape* sieja vadinamasis *is-a* ryšys: klasės *Circle* objektai gali būti naudojami visur, kur reikalaujama *Shape* klasės objektų. Pvz. jei funkcija reikalauja *Shape** tipo parametro, tai galime perduoti *Circle** objektą.

Paveldėjusios klasės ne tik pridėjo savų duomenų ir metodų, bet dar ir perkrovė metodą *draw*. Kuris iš metodų bus iškviestas žinoma jau kompiliavimo metu ir priklauso tik nuo kintamojo ar rodyklės tipo. Tai nėra tikrasis polimorfizmas. Pailiustruokime pavyzdžiu:

```
// screen.cpp

void Screen::paintEvent(QPaintEvent*)
{
    QPainter p(this);

    Shape* shapeTmp = shape;
    shapeTmp->draw(&p);

    shapeTmp = circle;
    shapeTmp->draw(&p);

    shapeTmp = square;
    shapeTmp->draw(&p);

    shapeTmp = rectangle;
    shapeTmp->draw(&p);
}
```

Programa nupieš keturis taškus, atitinkančius *shape*, *circle*, *square* ir *rectangle* centrus, o ne pačias figūras. Kompiliatorius sakinyje *shapeTmp->draw(&p)* mato, jog rodyklės *shapeTmp* tipas yra *Shape**, todėl kviečia metodą *Shape::draw*. Taigi, šioje vietoje turime paveldėjimą, bet neturime polimorfizmo. Polimorfizmas - sekančiame skyrelyje.

3.6. Virtualūs metodai ir polimorfizmas

Tam, kad išspręsti praeito skyrelio problemą, pasinaudosime virtualiais metodais. Prirašykime raktinį žodį *virtual* prie visų metodo *draw* aprašymų (prie realizacijų rašyti nereikia):

```
// shapes.h

class Shape
{
    ...
    virtual void draw(QPainter* p);
};

...

class Rectangle : public Shape
{
    ...
    virtual void draw(QPainter* p);
};
```

Dabar praeito skyrelio metodas *Screen::paintEvent()* ekrane nupieš tašką, apskritimą, kvadratą ir stačiakampį. Virtualūs (polimorfiniai) metodai kviečiami ne pagal deklaruotą rodyklės tipą (mūsų atveju *Shape**), o pagal realų objekto tipą, kuris nustatomas ne kompiliuojant, o programos vykdymo metu. Sakoma, kad virtualūs metodai yra ne perkraunami (overload), bet perrašomi (override).

Vieną kartą virtualus - visada virtualus! Bazinėje klasėje paskelbus metodą virtualiu, paveldėtoje klasėje jis bus virtualus nepriklausomai nuo to, ar parašysime žodį *virtual*, ar ne. Geras tonas reikalauja prie visų virtualių metodų paveldėjusiose klasėse pakartotinai rašyti *virtual*.

Praktinis pastebėjimas: bazinė klasė privalo turėti virtualų destruktorių. Tuomet būsime tikri, jog tokios komandos, kaip *delete shape*, iškvies destruktorių pagal realų objekto tipą, antraip neišvengsime bėdos.

Dažnai būna, kad nemenka dalis kodo dirba su rodykle į bazinę klasę ir nesirūpina, kokios klasės (bazinės ar paveldėtos) objektas yra iš tikrųjų. Tai bene pats nuostabiausias OOP mechanizmas. Pvz. parašykime globalią funkciją *draw10Steps()*, kuri nupieš po 10 duotos figūros vaizdų po žingsnelį juos perstumdama:

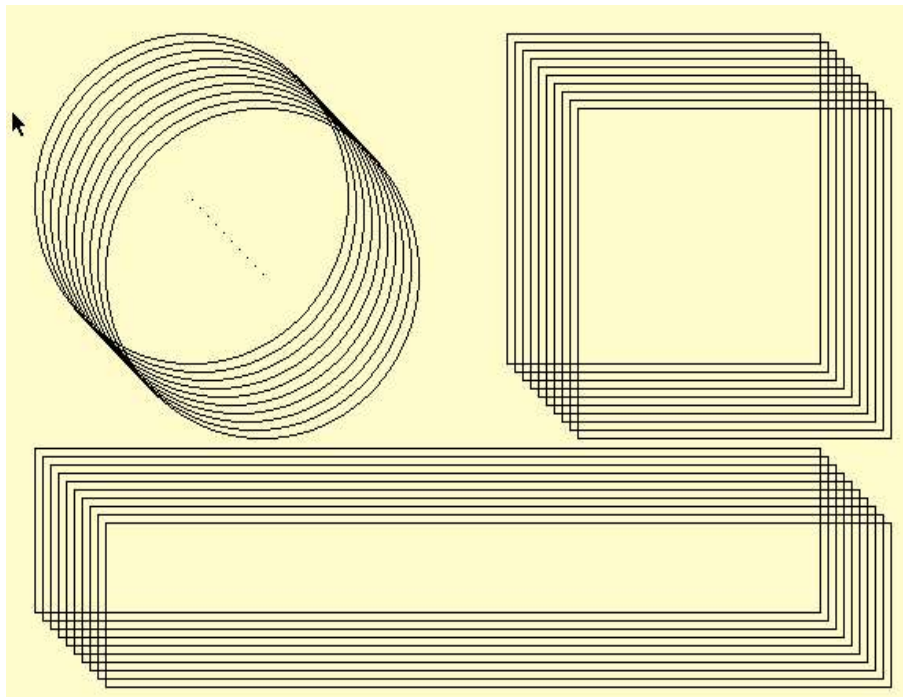
```

void draw10steps (QPainter* p, Shape* shape)
{
    int x = shape->getCenterX();
    int y = shape->getCenterY();
    for (int i=0; i < 10; i++)
    {
        shape->setCenter(x + i*5, y + i*5);
        shape->draw(p);
    }
    shape->setCenter(x, y);
}

void Screen::paintEvent(QPaintEvent*)
{
    QPainter p(this);
    draw10steps(&p, shape);
    draw10steps(&p, circle);
    draw10steps(&p, square);
    draw10steps(&p, rectangle);
}

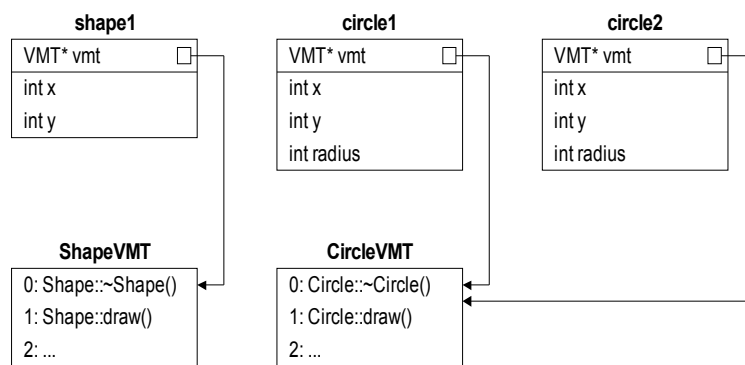
```

Ekrane pamatysime:



3.7. Virtualių metodų lentelės (VMT)

Kaip įgyvendinama ši virtualių metodų magija? Tuo tikslu kiekvienas objektas, turintis bent vieną virtualų metodą, kartu gauna ir nematomą rodyklę į virtualių metodų lentelę. Tokios klasės vadinamos polimorfinėmis. Kiekvienai polimorfinei klasei (o ne atskiram objektui) yra lygiai po vieną VMT (*virtual method table*), kur surašyti virtualių metodų adresai. Pasipaišykime tris objektus atmintyje:



Kviečiant virtualų metodą, pradžioje objekte surandama rodyklė į virtualių metodų lentelę, po to iš lentelės žinomu indeksu imamas metodo adresas. Pvz., visų klasių, paveldėjusių nuo *Shape*, virtualių metodų lentelėse metodo *draw* adresas bus patalpintas tuo pačiu indeksu (mūsų pavyzdyje jis lygus 1).

Programuotojas kode rašo metodo vardą, o kompiliatorius pakeičia jį indeksu VMT'e. Kompiliavimo metu yra žinomas tik metodo indeksas, o konkretus adresas randamas programos vykdymo metu. Taigi, virtualiu metodu kvietimas yra kiek lėtesnis, nei nevirtualių. Paprastai beveik visi klasės metodai turėtų būti virtualūs, tačiau, programos veikimo greičio sumetimais, virtualiais daromi tik tie, kurie klasės kūrėjo nuomone bus perrašyti paveldėtoje klasėje.

O kas, jei klasės *Circle* metodas *draw* nori iškviesti klasės *Shape* metodą *draw*, tam, kad padėti taškelį savo centre? Tuo tikslu vartojamas pilnai kvalifikuotas metodo vardas:

```
void Circle::draw(QPainter* p)
{
    Shape::draw(p);
    p->drawEllipse(x-radius, y-radius, radius*2, radius*2);
}
```

3.8. Statiniai, paprasti ir virtualūs metodai

Dabar mes jau žinome visas tris metodų rūšis:

1. statiniai - kviečiami nesusietai su jokia konkrečiu objektu. Metodo adresą žinomas kompiliavimo metu.
2. paprasti (ne virtualūs) - kviečiami konkrečiam klasės objektui. Metodo viduje apibrėžtas žodelis *this*. Metodo adresą žinomas jau kompiliavimo metu.
3. virtualūs - kviečiami konkrečiam klasės objektui. Metodo viduje apibrėžtas žodelis *this*. Metodo adresą randamas tik vykdymo metu.

3.9. Polimorfizmas konstruktoriuose ir destruktoriuose

Panagrinėkime pavyzdį su trimis paprastutėmis klasėmis:

```
// abc.cpp

class A
{
public:
    A () {print();}
    virtual ~A () {print();}
    virtual void print () {cout << "A";}
};

class B : public A
{
public:
    B () {print();}
    virtual ~B () {print();}
    virtual void print () {cout << "B";}
};

class C : public B
{
public:
    C () {print();}
    virtual ~C () {print();}
    virtual void print () {cout << "C";}
};

int main ()
{
    C c;
}
```

Ši programa atspausdins:

ABCCBA

C++ konstruktoriuose ir destruktoriuose (skirtingai nuo Java) polimorfizmas neveikia: nors klasė C ir perrašė virtualų metodą *print()*, tačiau konstruojant klasei B priklausančią dalį kviečiamas B::*print()*, o konstruojant klasei A priklausančią dalį kviečiamas A::*print()*. Analogiškas Java pavyzdys atspausdintų "CCCCC".

Tokiu būdu išvengiama nepageidaujamų šalutinių efektų. Jei konstruktoriuose veiktų polimorfizmas, tai būtų įmanoma iškviesti paveldėjusios klasės perrašytą metodą anksčiau, nei paveldėtoji klasė bus sukonstruota.

3.10. Švariai virtualūs metodai ir abstrakčios klasės

Kartais bazinėje klasėje negali būti prasmingos virtualaus metodo realizacijos. Jeigu mes sutarsime, kad klasė `Shape` nusako abstrakčią figūrą, neturinčią jokio vaizdo ekrane, netgi taško, tuomet galime metodą `draw()` padaryti švariai virtualų (*pure virtual*), prirašydami "= 0" jo antraštėje:

```
class Shape
{
    ...
    virtual void draw(QPainter* p) = 0;
};
```

Klasė, turinti bent vieną švariai virtualų metodą, vadinama abstrakčia. Kompiliatorius neleidžia sukurti abstrakčios klasės objektų. Švariai virtualus metodas neturi jokio kūno, netgi tuščio. Tai daugiau pažadas, kad paveldėtos klasės pateiks savo realizacijas. Jei paveldėta klasė nerealizuoja visų bazinės klasės švariai virtualių metodų, tai ir ji pati tampa abstrakčia.

3.II. Švarus interfeisas

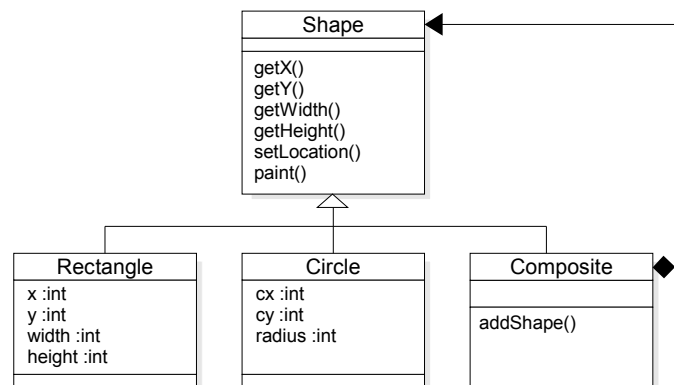
Neformalus apibrėžimas: jei klasė neturi savo duomenų, turi tuščią virtualų destruktorių ir visi jos metodai yra vieši (*public*) ir švariai virtualūs, tai tokią klasę vadinsime švariu interfeisu.

Švarus interfeisas - puiki priemonė labai aiškiai atskirti sąvokas "interfeisas" ir "realizacija". Jei klasė paveldi iš švaraus interfeiso ir realizuoja visus jo metodus, tai tokiu atveju dažnai sakoma, jog klasė "realizuoja interfeisą". Pavyzdžiui, aprašykime grafinę figūrą *Shape*, kurią nusako jos kairiojo viršutinio kampo koordinatės, bei plotis ir aukštis:

```
// project: composite, shape.h

class Shape
{
public:
    virtual ~Shape      () {}
    virtual int  getX    () const    = 0;
    virtual int  getY    () const    = 0;
    virtual int  getWidth() const    = 0;
    virtual int  getHeight() const   = 0;
    virtual void setLocation(int x, int y) = 0;
    virtual void paint   (QPainter* p) = 0;
};
```

Klasė *Shape* - švarus interfeisas. Klasėje *Shape* nesaugome duomenų apie koordinates ekrane, nes paveldėjusios klasės naudoja skirtingas strategijas. Pvz. stačiakampis saugo koordinates išreikštiniu būdu, o apskritimas skaičiuoja pagal savo centrą.



```

class Rectangle : public Shape
{
protected:
    int x;
    int y;
    int width;
    int height;
public:
    Rectangle (int cx, int cy, int width, int height);
    ...
};

class Circle : public Shape
{
protected:
    int cx;
    int cy;
    int radius;
public:
    Circle (int cx, int cy, int radius);
    virtual int    getX      () const {return cx - radius;}
    virtual int    getWidth   () const {return radius*2 + 1;}
    ...
};

```

Trečioji paveldėjusi klasė, *Composite*, yra sudėtinė figūra:

```

class Composite : public Shape
{
protected:
    vector<Shape*> shapes;
public:
    ...
    void addShape (Shape* shape);
};

```

Sudėtinė figūra nesaugo savo matmenų, bet kiekvieną kartą skaičiuoja iš naujo:

```

int Composite::getX() const
{
    if (shapes.size() <= 0)
        return 0;
    int minX = shapes[0]->getX();
    for (unsigned i = 1; i < shapes.size(); i++)
        if (minX > shapes[i]->getX())
            minX = shapes[i]->getX();
    return minX;
}

```

Atkreipkime dėmesį, jog sudėtinė figūra taip pat yra figūra, t.y. sudėtinė figūra gali susidėti ir iš kitų sudėtinių figūrų. Sukurkime apskritimą, stačiakampį, medį, susidedantį iš apskritimo ir stačiakampio, bei šunį. Šuns galva ir uodega kiek dirbtinai sudėsime į atskirą sudėtinę figūrą tam, kad pailustruoti, jog sudėtinė figūra gali susidėti iš bet kokių figūrų, tame tarpe ir kitų sudėtinių:

```

Screen::Screen(QWidget* parent, const char* name)
: QWidget(parent, name)
{
    circle = new Circle(150, 100, 50);
    rectangle = new Rectangle(250, 50, 400, 100);

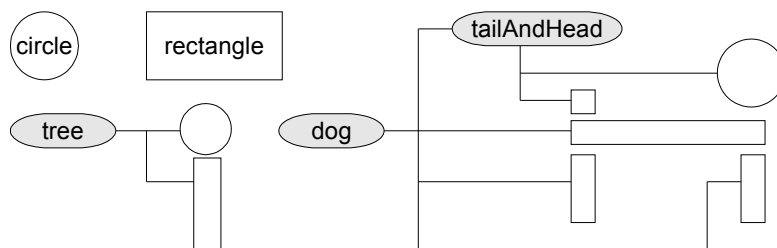
    tree = new Composite();
    tree->addShape(new Rectangle(600, 300, 20, 200));
    tree->addShape(new Circle(610, 250, 50));

    Composite* tailAndHead = new Composite();
    tailAndHead->addShape(new Rectangle(100, 380, 20, 20));
    tailAndHead->addShape(new Circle(280, 350, 50));

    dog = new Composite();
    dog->addShape(new Rectangle(100, 400, 200, 20));
    dog->addShape(new Rectangle(100, 420, 20, 40));
    dog->addShape(new Rectangle(280, 420, 20, 40));
    dog->addShape(tailAndHead);
}

```

Schematiškai objektus galėtume pavaizduoti taip:



Dar aprašykime dvi globalias funkcijas, kurių pirmoje piešia perstumtą figūrą, o antroji piešia punktyrinį rėmelį apie figūrą:

```

void paintMoved (QPainter* p, Shape* shape, int dx, int dy)
{
    shape->setLocation(shape->getX()+dx, shape->getY()+dy);
    shape->paint(p);
    shape->setLocation(shape->getX()-dx, shape->getY()-dy);
}

void paintBorder (QPainter* p, Shape* shape)
{
    int x = shape->getX();
    int y = shape->getY();
    int w = shape->getWidth();
    int h = shape->getHeight();
    p->setPen(Qt::DotLine);
    p->drawRect(x-2, y-2, w+4, h+4);
    p->setPen(Qt::SolidLine);
}

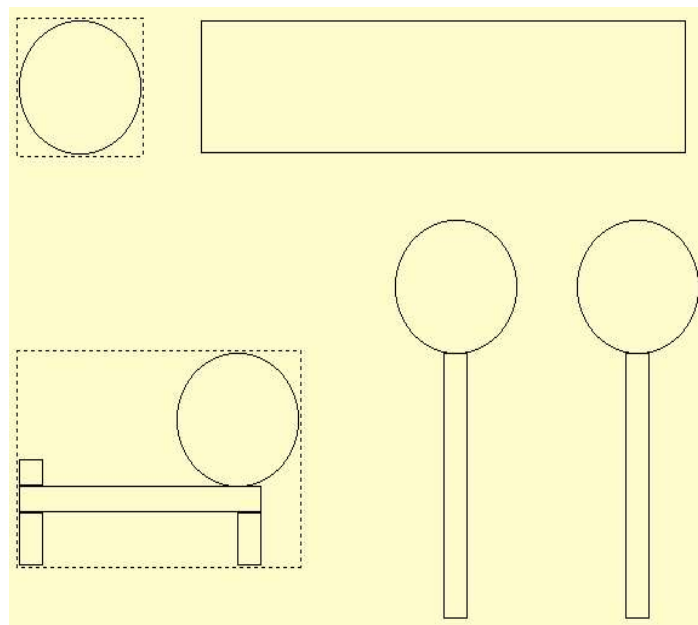
```

Nupieškime apskritimą, stačiakampį, medį ir šunį. Dar nupieškime kairėn pastumtą medį, bei rėmelius apie apskritimą ir šunį:

```
void Screen::paintEvent(QPaintEvent*)
{
    QPainter p(this);
    circle->paint(&p);
    rectangle->paint(&p);
    tree->paint(&p);
    dog->paint(&p);

    paintMoved(&p, tree, -150, 0);
    paintBorder(&p, circle);
    paintBorder(&p, dog);
}
```

Ekrane pamatysime:



Sudėtinė figūra *Composite* iliustruoja labai paplitusią objektiškai orientuoto programavimo architektūrą: klasė susideda iš savo paties bazinės klasės objektų, tiksliau, rodyklių į juos. Pvz. katalogas yra failas, susidedantis iš kitų failų, tame tarpe ir katalogų. Variklis yra automobilio dalis, susidedanti iš kitų dalių, kurios savo ruožtu irgi gali būti sudėtinės. Tokia pasikartojanti šabloninė situacija OOP projektuotojų yra vadinama *Composite*. Mes jau ne kartą susidūrėme su ja.

4. Klaidų mėtymas ir gaudymas (*exception handling*)

4.1. Raktiniai žodžiai *throw*, *try* ir *catch*

Prisiminkime dinaminį sveikųjų skaičių steką:

```
// dynastack.cpp

class Stack
{
    ...
public:
    ...
    void push    (int element);
    int  pop     ();
    int  peek    ();
    bool isEmpty ();
};
```

Metodas *pop* išima iš steko viršaus elementą ir jį grąžina. Jei stekas tuščias, grąžina nulį:

```
int Stack::pop ()
{
    if (size > 0)
        return elements[--size];
    return 0;
}
```

Tokiu būdu, klaidingai parašyta steko naudojimo programa, kuri netikrina požymio *isEmpty*, gali sėkmingai traukti iš steko nulių kiek panorėjusi. Daugeliu atveju, tokia situacija yra nepageidaujama. Šioje vietoje C++ siūlo patogų klaidų aptarnavimo mechanizmą: klaidų mėtymą ir gaudymą (*exception handling*). Metodas *pop* gali "mesti" klaidą, naudodamasis raktiniu žodeliu *throw*, o kodas, kviečiantis metodą *pop*, gali gaudyti klaidas raktinių žodžių *try* ir *catch* pagalba:

```
int Stack::pop ()
{
    if (size > 0)
        return elements[--size];
    throw string("pop() called for empty stack");
}
```

Tokiu būdu, kai stekas tuščias, vietoje to, kad grąžinti beprasmę reikšmę, mes pranešame apie klaidą mesdami ją. Atininkamai funkcijoje *main* mes tikimės metamų klaidų *try*-bloko pagalba ir pagaunama jas *catch*-bloke:

```
int main ()
{
    try
    {
        Stack squares;
        for (int i = 1; i <= 5; i++)
            squares.push (i*i);
        for (;;)
            cout << squares.pop () << endl;
    }
    catch (string& ex)
    {
        cout << "Exception caught: " << ex << endl;
    }
}
```

Programa atspausdins:

```
25
16
9
4
1
Exception caught: pop() called for empty stack
```

Žodelis *throw* turi lygiai vieną parametą - klaidos objektą, kuris yra metamas. Mūsų atveju yra metamas *string* tipo objektas, nešantis savyje pranešimą apie klaidą.

Raktinis žodis *try* su figūriniais skliaustais žymi bloką, kuriame tikimasi, jog gali įvykti klaida, t.y. bus "mesta klaida". Po jo iš karto privalo sekti *catch*-blokas su lygiai vienu argumentu, nusakančiu, kokia klaida yra gaudoma. Jei klaida nėra metama *try*-bloko viduje, tai programos vykdymas ignoruoja *catch*-bloką.

Klaidą gali mesti ne tik tiesiogiai *try*-bloke kviečiamos funkcijos ir metodai, bet ir pastarųjų viduje iškviestos funkcijos ir metodai.

4.2. Skirtingų klaidų gaudymas

Metamomis klaidomis gali būti bet kokio tipo objektai, pvz. jūsų aprašytos klasės ar bazinių C++ tipų *int*, *char*, *float* ir t.t.. Klasėms yra reikalavimas, kad jos turėtų kopijavimo konstruktorių, nes išmesta klaida, skriedama per funkcijų ir metodų kvietimo steką, pakeliui gali būti kopijuojama. Plačiai paplitusi praktika: kiekvienam klaidos tipui apsirasyti po klasę, nešančia visą reikiamą informaciją apie klaidą.

try-bloko viduje skirtingose vietose gali būti išmestos skirtingų tipų klaidos. Jas galima pagauti naudojant vieną paskui kitą einančius *catch*-blokus su atitinkamais parametrais. Be to, galima naudoti *catch*-bloką su daugtaškiu vietoje parametro, kuris pagauna visų tipų klaidas. Pavyzdžiui, turime aritmetinių išraiškų parserį, kuris tekstinę eilutę paverčia atitinkamu išraiškos objektu. Pastarasis turi metodą, grąžinantį išraiškos rezultatą, kaip *double* tipo reikšmę. Parseris gali išmesti klaidą, apie neteisingai užrašytą aritmetinę išraišką, o išraiškos objektas gali išmesti dalybos iš nulio klaidą:

```
Expression* exp = NULL;
try
{
    Parser parser;
    exp = parser.parseString("12 + 3*(9-5)");
    cout << exp->getValue() << endl;
}
catch (SyntaxError& ex)
{
    cout << "Syntax error: " << ex.getMessage() << endl;
}
catch (DivisionByZeroError& ex)
{
    cout << "Division by zero error" << endl;
}
catch (...)
{
    cout << "Unknown error" << endl;
}
delete exp;
```

Įvykus klaidai, ji bus perduota pirmajam *catch*-blokui, gaudančiam to tipo klaidą. Kiti *catch*-bloškai bus ignoruojami. Čia galioja paveldėjimas: jei *catch*-blokas gauda bazinės klaidų klasės objektus, tai pagaus ir paveldėtų klasių objektus. Tam, kad šis mechanizmas veiktų, reikia gaudyti nuorodas į objektą, o ne patį objektą.

Jei nėra atitinkamo *catch*-bloko ir nėra *catch*-bloko su daugtaškiu, tai klaida išlekia toliau iš einamosios funkcijos. Jei jos nepagauna joks kitas steke esančios funkcijos *catch*-blokas, tai iškviečiama globali funkcija *terminate*, kuri, pagal nutylėjimą, užbaigia programos darbą standartinės funkcijos *abort* pagalba.

4.3. Įdėtiniai *try*-blokai

Klaida laikoma pagauta ar aptarnauta nuo to momento, kai programos vykdymas įeina į atitinkamą *catch*-bloką. Tokiu būdu, *catch*-blokas pats savo ruožtu gali mesti naują klaidą:

```
try
{
    tableData = readTableData(file);
}
catch (IOException& ex)
{
    throw TableInputError("Can not get table data");
}
```

Kaip ir bet koks kitas blokas, *catch*-blokas gali turėti savyje kitus *try*- ir *catch*-blokus:

```
List* list = 0;
try
{
    list = createMyObjects();
    // do some file input/output
}
catch (IOException& ex)
{
    try { deleteMyObjects(list); } // do some clean up
    catch(...) {}                // catch all exceptions
}
```

Kartais *catch*-blokas pats negali pilnai apdoroti pagautos klaidos ir nori mesti ją toliau. Būtent *catch*-bloko viduje galima naudoti raktinį žodį *throw* be jokio klaidos objekto, kad pagautoji klaida būtų metama toliau:

```
try
{
    ...
}
catch (SomeError& ex)
{
    if (canHandle(ex))
        handleException(ex);
    else
        throw; // throw current SomeError object
}
```

4.4. Automatinių objektų naikinimas steko vyniojimo metu

Steko vyniojimas, tai procesas, kuomet išmesta klaida keliauja funkcijų iškvietimo steku, ieškodama atitinkamo *catch*-bloko. Pakeliui yra naikinami visi steke aprašyti ir pilnai sukonstruoti objektai. Objektas yra ne pilnai sukonstruotas, jei klaida išlėkė iš jo konstruktoriaus.

Panagrinėkime pavyzdį, kuriame atidarytas failas gali likti neuždarytas:

```
void useFile (const char* fileName)
{
    FILE* file = fopen(fileName, "wt");
    fprintf(file, "writting from useFile() \n");
    mayThrowAnException(file);
    fclose(file);
}
```

Funkcija *throwsAnException* išmeta klaidą, ir failas lieka neuždarytas - kitose programos vietose priėjimas prie jo yra uždraustas. Dabar pasinaudokime standartinės C++ bibliotekos klase *ofstream*, kuri savo destruktoriuje uždaro failą:

```
void useFile (const string& fileName)
{
    ofstream file (fileName);
    file << "writting from useFile() << endl;
    mayThrowAnException(file);
}
```

Tokiu būdu, failas *file* bus uždarytas nepriklausomai nuo to, ar iš funkcijos *useFile* išeinama normaliai, ir sunaikinami visi lokalūs objektai, ar išeinama dėl to, jog buvo išmesta klaida, ir sunaikinami visi steke esantys pilnai sukonstruoti objektai.

4.5. Klaidų mėtymas konstruktoriuose ir destruktoriuose

Klasių destruktoriai neturėtų mesti klaidos. Tarkime, buvo išmesta klaida ir steko vyniojimo metu kviečiami automatinių objektų destruktoriai. Jei kuris nors automatinis objektas tuo metu pats išmes klaidą iš savo destruktoriaus, tai bus iškviesta globali funkcija *terminate* kuri, pagal nutylėjimą, nutraukia programos vykdymą.

Jei klaida metama iš objekto konstruktoriaus, tai kviečiami destruktoriai toms objekto dalims, įskaitant bazines klases, kurios buvo pilnai sukonstruotos, o pačio objekto destruktorius nėra kviečiamas:

```
// constrex.cpp

class A
{
public:
    A() {cout << " A()" << endl;}
    virtual ~A() {cout << "~A()" << endl;}
};

class B : public A
{
public:
    B () {cout << " B()" << endl;
        throw string("construction of B failed");}
    ~B () {cout << "~B()" << endl;}
};

int main ()
{
    try
    {
        B b;
    }
    catch (string& ex)
    {
        cout << "exception caught: " << ex << endl;
    }
}
```

Programa atspausdins:

```
A()
B()
~A()
exception caught: construction of B failed
```

4.6. Nepagautos klaidos ir funkcija *terminate()*

Kaip jau buvo minėta, jei klaidos nepagauna nei vienas *catch*-blokas, arba jei vyniojant steką kuris nors destruktorius išmetė klaidą, tuomet kviečiama globali funkcija *terminate()*, kuri pagal nutylėjimą iškviečia funkciją *abort()*. Prisiminkime, jog standartinė funkcija *abort()*, skirtingai nuo funkcijos *exit()*, nesunaikina globalių objektų. Mes galime pateikti savo *terminate*-funkciją naudodamiesi standartine funkcija *set_terminate()*:

```
// terminate.cpp

void myTerminate()
{
    cout << "my terminate called" << endl;
    exit(1);
}

A globalA;

int main ()
{
    set_terminate(myTerminate);
    throw string("some error");
}
```

Programa atspausdins:

```
A()
my terminate called
~A()
```

4.7. Klaidų specifikacija ir netikėtos klaidos

Pagal nutylėjimą, funkcija ir metodas gali išmesti bet kokią klaidą. Nepakentų metodo ar funkcijos signatūroje nurodyti, kokias klaidas jis gali išmesti:

```
void a() { ... }
void b() throw(string, int) { ... }
void c() throw() { ... }
```

Funkcija *a()* gali išmesti bet kokią klaidą. Funkcija *b()* išmes tik *int* arba *string*, arba klaidą, paveldėtą nuo klasės *string*. Funkcija *c()* pasižada nemesti jokios klaidos.

Jei funkcija *b()* išmes kitos rūšies klaidą, arba funkcija *c()* išmes bet kokią klaidą, bus iškviesta globali funkcija *unexpected()*, kuri, pagal nutylėjimą, iškviės funkciją *terminate*.

Analogiškai, kaip ir *terminate()* funkcijos atveju, galime pateikti savo *unexpected()* funkciją pasinaudodami *set_unexpected()*:

```
// unexpected.cpp

void myUnexpected()
{
    cout << "my unexpected called" << endl;
    exit(1);
}

void throwsAnException ()
{
    throw string ("Some exception");
}

void promisedNotToThrow () throw()
{
    throwsAnException();
}

int main ()
{
    set_unexpected(myUnexpected);
    promisedNotToThrow();
}
```

4.8. Standartinės klaidų klasės

Standartinė C++ biblioteka pateikia klaidų klasių hierarchiją:

```
exception
|
+----bad_alloc . . . . . new
|
+----bad_cast. . . . . dynamic_cast
|
+----bad_exception
|
+----bad_typeid. . . . . typeid
|
+----ios_base::failure . . . . . ios_base::clear()
|
+----logic_error
|
|       +----domain_error
|       |
|       +----invalid_argument
|       |
|       +----length_error
|       |
|       +----out_of_range . . . at()
|
+----runtime_error
|
|       +----overflow_error
|       |
|       +----range_error
|       |
|       +----underflow_error
```

Visos standartinės klaidų klasės iš bazinės klasės *exception* paveldi metodą *what()*, kuris grąžina klaidos pranešimą. Standartinės bibliotekos klaidų klasių hierarchija - puikus klaidų grupavimo naudojant paveldėjimą pavyzdys: užtenka gaudyti bazinės klasės *exception* klaidas ir kartu pagausime visas paveldėtas klaidų klases. Prisiminkime: kad šis mechanizmas veiktų, reikia gaudyti nuorodas į objektą, o ne patį objektą.

Kartu pamatysime, jog *vector<>* klasės perkrautas operatorius *[]* netikrina masyvo rėžių, o analogiškas metodas *at()* meta klaidą, jei elemento indeksas neatitinka masyvo elementų kiekio:

```

// standard_ex.cpp

try
{
    vector<int> v;
    v.push_back(2);
    v.push_back(4);
    v.push_back(8);
    cout << v[0] << endl;
    cout << v[7] << endl;
    cout << v.at(0) << endl;
    cout << v.at(7) << endl;
}
catch (const exception& ex)
{
    cout << "exception: " << ex.what() << endl;
}

```

Programa atspausdins:

```

2
1247039744
2
exception: vector [] access out of range

```

5. Vardų erdvės (*namespace*)

5.1. Motyvacija

Vardų erdvės skirtos spręsti globalių besikartojančių vardų konfliktus. Tarpusavyje susiję tipų, funkcijų ar globalių kintamųjų vardai gali būti apjungti į atskirą vardų erdvę. Mes jau susidūrėme su vardų erdve *std*, kurioje apibrėžtos visos standartinės bibliotekos klasės, funkcijos, globalūs kintamieji ir t.t.. Panagrinėkime pavyzdį: dvimačiai ir trimačiai taškai bei atstumai tarp jų:

```
// namespace1.cpp

namespace Graphics2D
{
    class Point
    {
    public:
        double x, y;
        Point (double x, double y);
    };
    double distance (const Point& p1, const Point& p2);
}

namespace Graphics3D
{
    class Point
    {
    public:
        double x, y, z;
        Point (double x, double y, double z);
    };
    double distance (const Point& p1, const Point& p2);
}
```

Funkcijų ir konstruktorių realizacija:

```
double Graphics2D::distance (const Point& a, const Point& b)
{
    return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}
Graphics2D::Point::Point (double _x, double _y)
: x (_x), y (_y) {}

// Graphics3D - analogiškai
```

Tokie vardų erdvėse apskliausti tipai naudojami įprastiniu būdu, tik jų vardai prasideda talpinančiosios vardų erdvės vardu:

```
int main ()
{
    Graphics2D::Point  a(0, 0);
    Graphics2D::Point  b(2, 2);
    Graphics3D::Point  c(0, 0, 0);
    Graphics3D::Point  d(2, 2, 2);
    cout << "a..b = " << Graphics2D::distance(a, b) << endl;
    cout << "c..d = " << Graphics3D::distance(c, d) << endl;
}
```

Programa atspausdins:

```
a..b = 2.82843
c..d = 3.4641
```

5.2. Raktinis žodis *"using"*

Raktinio žodžio *using* pagalba galima naudoti ir trumpesnius vardus:

```
// namespace2.cpp

int main ()
{
    using namespace Graphics2D;
    using Graphics2D::distance; // prefer to std::distance
    Point  a(0, 0);
    Point  b(2, 2);
    cout << "a..b = " << distance(a, b) << endl;
}
```

Užrašas *"using namespace Graphics2D"* reiškia, kad visi vardų erdvės *Graphics2D* identifikatoriai gali būti naudojami tiesiogiai, be vardų erdvės priešdėlio.

Galima išvardinti ir atskirus vardų erdvės identifikatorius, kuriuos norime vartoti tiesiogiai. Pavyzdžiui, užrašas *"using Graphics2D::distance"* atlieka dvi funkcijas:

1. leidžia tiesiogiai naudoti identifikatorių *distance* be *Graphics2D::* prefikso;
2. jei užrašo vietoje yra matomas kitas identifikatorius *distance*, tai pirmenybę teikti *Graphics2D* vardų erdvės identifikatoriui.

Mūsų atveju naudotas antrasis variantas, nes antraštinio failo *iostream* viduje jau yra identifikatorius *distance*.

Naudojant *using* konkrečiam funkcijos identifikatoriui, tampa matomos visos funkcijos su nurodytu vardu, nekreipiant dėmesio į argumentų sąrašą. Be to, raktinis žodis *using* įtakoja tik tą kodo bloką, kuriame yra naudojamas.

5.3. Vardų erdvių apjungimas

Vardų erdvės yra atviros, t.y. jas galima papildyti: kompiliatorius visas rastas vardų erdves su vienodu pavadinimu apjungia į vieną:

```
// namespace3.cpp

namespace Graphics2D
{
    class Point {...};
}

namespace Graphics2D
{
    double distance (const Point& p1, const Point& p2);
}
```

5.4. Vardų erdvių sinonimai

Vardų erdvėms galima sukurti sinonimus. Dažniausiai tai būna sutrumpinimai:

```
// namespace4.cpp

namespace G2 = Graphics2D;

int main ()
{
    G2::Point a (0, 0);
    G2::Point b (2, 2);
    cout << "a..b = " << G2::distance(a, b) << endl;
}
```

5.5. Vardų erdvės be pavadinimo

Vardų erdvės be pavadinimo naudojamos tik *cpp*-failuose (o ne pvz. *h*-failuose). Tarkime, turime du modulius *m1.cpp* ir *m2.cpp* priklausančius tam pačiam projektui, t.y. kartu susiejamus į vieną programą:

```
// m1.cpp

class Date {...};
int f(int i) {...}

// m2.cpp

class Date {...};
int f(int i) {...}
```

Klasė *Date* ir funkcija *f* yra pagalbiniai, todėl jie ir aprašyti ir apibrėžti ir naudojami tik savo modulių viduje, t.y. jie nepaminėti *h*-faile. Kompiliatorius tvarkingai kompiliuos. Tačiau linkeris išmes klaidą apie pasikartojančius identifikatorius skirtinguose moduluose. Naudojant vardų erdves be pavadinimo, galime priversti kompiliatorių sugeneruoti unikalius vardų erdvės pavadinimus kiekvienam moduliui atskirai:

```
// m1.cpp

namespace
{
    class Date {...};
    int f(int i) {...}
}
```

Toks užrašas yra ekvivalentiškas užrašui:

```
namespace $$$
{
    class Date {...};
    int f(int i) {...}
}

using namespace $$$;
```

Kur *\$\$\$* yra unikalus pavadinimas, galiojantis tik viename modulyje.

Neįvardintos erdvės pakeičia žodėlį *static* kuomet jis naudojamas su prasme "lokalus linkavimas". Raktinis žodis *static* turėtų būti naudojamas tik statiniams klasės nariams ir funkcijų statiniams vidiniams kintamiesiems aprašyti.

6. Operatorių perkrovimas

6.1. Motyvacija

Realizuokime kompleksinius skaičius su realiąja ir menamąja dalimis. Apibrėžkime kompleksinių skaičių sudėtį, daugybą, išvedimą į srautą:

```
// operator1.cpp

class Complex {
private:
    double r;
    double i;
public:
    Complex (double re=0, double im=0) : r(re), i(im) {}
    Complex add (const Complex& c);
    Complex multiply (const Complex& c);
    void print (ostream& os);
};
```

Apskaičiuokime reiškinį $d=a + b*c$:

```
int main ()
{
    Complex a (2.5, 1.5);
    Complex b (10, 2);
    Complex c (4);
    Complex d;
    d = b.multiply(c);
    d = a.add(d);
    d.print(cout);
}
```

Rezultatas:

(42.5+i9.5)

O ar negalėtume mes to paties reiškinio užrašyti įprastiniu matematiniu pavidalu?

```
int main ()
{
    ...
    d = a + b*c;
    cout << d << endl;
}
```

Galime! Tuo tikslu perkraukime tris C++ operatorius +, - ir <<:

```
// operator2.cpp

class Complex
{
private:
    double r;
    double i;
public:
    Complex (double re=0, double im=0) : r(re), i(im) {}
    Complex operator+ (const Complex& c);
    Complex operator* (const Complex& c);
    friend ostream& operator<< (ostream& os, const Complex& c);
};

Complex Complex::operator+ (const Complex& c)
{
    return Complex(r + c.r, i + c.i);
}

Complex Complex::operator* (const Complex& c)
{
    return Complex(r*c.r - i*c.i, r*c.i + i*c.r);
}

ostream& operator<< (ostream& os, const Complex& c)
{
    os << "(" << c.r << "+i" << c.i << ")";
    return os;
}

int main ()
{
    Complex a (2.5, 1.5);
    Complex b (10, 2);
    Complex c (4);
    Complex d;
    d = a + b*c;
    cout << d << endl;
}
```

Rezultatas bus tas pats:

(42.5+i9.5)

6.2. Perkraunami operatoriai

Perkrauti galima šiuos C++ operatorius:

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

Daug lengviau atsiminti vienintelius keturis operatorius, kurių perkrauti negalima:

:: . .* ?:

Galime perkrauti operatorių, t.y. veiksmus, kuriuos atlieka operatorius, bet jų vykdymo prioritetai bei nariškumai (unirinis ar binarinis) išlieka tie patys.

Perkrautam operatoriui galime suteikti bet kokią prasmę, tačiau geriau tuom nepiktnaudžiauti: tegu suma ir reiškia sumą, o sandauga – sandaugą ir pan..

Beje, operatorius galima kviesti ir išreikštiniu būdu, kaip įprastas funkcijas ar metodus. Pvz., vietoje:

```
d = a + b*c;
cout << d << endl;
```

Galime parašyti taip, kaip kompiliatorius "padaro mintyse":

```
// operator2.cpp

d = a.operator+(b.operator*(c));
operator<<(cout, d).operator<<(endl);
```

Matyt, nereikia ir sakyti, jog išreikštiniu būdu operatoriai nėra naudojami - juk jie todėl ir perkrauti, kad įneštu aiškumo, o ne painiavos.

6.3. Unariniai ir binariniai operatoriai

Binarinis (dviejų argumentų) operatorius gali būti aprašytas kaip klasės metodas su vienu parametru, arba kaip globali funkcija su dviem parametrais:

```
class Vector
{
    Vector operator* (double k)
};

Vector operator* (double k, Vector& v);

void f (Vector& v)
{
    Vector a = v * 2.0;      // v.operator*(2.0)
    Vector b = 2.0 * v;      // ::operator*(2.0, v)
    ...
}
```

Kartais privaloma aprašyti binarinį operatorių kaip funkciją ne narį, jei pirmasis jo argumentas yra ne tos klasės, kuriai skiriamas šis operatorius, objektas. Pvz:

```
ostream& operator<< (ostream& os, const Complex& c)
```

Tokie operatoriai dažnai būna klasės draugais (*friend*).

Unarinis (vieno argumento) operatorius gali būti aprašytas kaip klasės metodas be parametru, arba kaip globali funkcija su vienu parametru:

```
class ListIterator
{
    ListElement& operator++ (); // returns next element
    bool hasElements();
};

List operator~ (List& list); // reverse list order

void printAll (List& list)
{
    ListIterator iterator = list.getFirst();
    while (iterator.hasElements())
        cout << ++iterator << endl;
    List reversed = ~list;
    ...
}
```

Prisiminkime, jog operatoriai ++ ir -- gali būti prefiksiniai, pvz. ++*a*, kuomet jie pradžioje padidina kintamojo *a* reikšmę vienetu ir vėliau ją grąžina, arba postfiksiniai, pvz. *a*++, kuomet pradžioje gaunama kintamojo *a* reikšmė, o vėliau ji padidinama vienetu. Aukščiau yra aprašytas prefiksinis operatorius ++. Postfiksiniai operatoriai ++ ir -- yra aprašomi dirbtinai pridėdant nenaudojamą *int*-tipo argumentą:

```

class ListIterator
{
    ListElement& operator++ (int); // postfix
};

void printAll (List& list)
{
    ListIterator iterator = list.getFirst();
    while (iterator.hasMoreElements())
        cout << iterator++ << endl;
}

```

Dar kartą nepamirškite, kad perkraunant operatorius, programuotojas nustato, ką jie iš tiesų darys. Tik jis gali (ir privalo) užtikrinti, kad prefiksinis operatorius veiks kaip prefiksinis, o postfiksinis - kaip postfiksinis.

6.4. Tipų konversijos operatoriai

Prisiminkime, kad tipų konversija užrašoma taip:

```
double d = 8.15;
int i = (int) d;
```

Mes jau matėme, jog galima aprašyti kokią nors klasę X su konstruktoriumi, leidžiančiu išreikštiniu arba automatinio būdu kito tipo T objektą konvertuoti į klasės X tipo objektą ($T \rightarrow X$). Tipas T gali būti kita klasė ar bazinis C++ kalbos tipas.

Galima ir atvirkštinė konversija: klasėje X aprašyti operatorių, kuris klasės X objektą konvertuotų į tipo T objektą ($X \rightarrow T$). Pvz., standartinėje bibliotekoje *ifstream* tipo objektą galima konvertuoti į *int* tipo reikšmę:

```
class ifstream          // input file stream
{
    ...
    operator int () const {return errorOccured == false;}
};
```

Tokiu būdu, klasės *ifstream* objektai gali būti naudojami visur, kur ir sveikieji skaičiai:

```
// operator3.cpp

int main ()
{
    ifstream file ("integers.txt");
    int i;
    file >> i;
    while (file) // calles: file.operator int ()
    {
        cout << " " << i;
        file >> i;
    }
    cout << endl;
    file.clear();
    file.close();
}
```

Aprašant klasės viduje tipų konvertavimo operatorių, nereikia nurodyti grąžinamos reikšmės tipo - jis visuomet yra toks, į kurį konvertuojama.

Matyt, kad tokių tipo konvertavimo operatorių naudojimas turėtų būti labai saikingas, o gal net ir vengtinas. Juk tą patį programos ciklą galėjome užrašyti ir suprantamiau:

```
while (file.good()) {...}
```

7. Apibendrintas programavimas (*generic programming*)

7.1. Programavimo stiliai

Skirtingi programavimo stiliai akcentuoja skirtingas sąvokas:

- struktūrinis (*structured*): struktūros ir funkcijos, manipuliuojančios jomis
- objektais paremtas (*object based*): duomenys ir funkcijos drauge
- objektiškai orientuotas (*object oriented*): paveldėjimas ir polimorfizmas
- apibendrintas (*generic*): tipas, o ne kintamasis, gali būti kito tipo parametras.

Skaitytojas, matyt, galėtų išvardinti ir daugiau programavimo stilių, pvz., loginis programavimas (tipinis programavimo kalbos atstovas - *PROLOG*) besiriamiantis logine dedukcija, funkcinis programavimas, paremtas sąrašais ir rekursija, kur jau klasika tapusi programavimo kalba *LISP* (*list programming*) dar kartais vadinama dirbtinio intelekto assembleriu, ir t.t..

Mūsų nagrinėjamam apibendrintam programavimui palaikyti C++ kalba pateikia šablono (*template*) sąvoką. Susipažįstant su šablonais pagrindinis tikslas yra gauti pakankamai žinių, kad naudotis standartine C++ šablonų biblioteka (*STL* - *Standard Template Library*).

7.2. Funkcijų šablonai

Pačioje šablonų atsiradimo pradžioje kai kurie programuotojai juos laikė tik griozdišku programavimo kalbos C preprocesoriaus makrosų pakaitalu. Standartinis pavyzdys - funkcijos *min* ir *max*:

```
// minmax.cpp

#define MIN(x, y) ((x) < (y) ? (x) : (y))
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

Pakeiskime šiuos C makrosus C++ funkcijų šablonais:

```
template<typename T>
T min (T x, T y)
{
    return x < y ? x : y;
}

template<class T>
T max (T x, T y)
{
    return x > y ? x : y;
}
```

Prieš kiekvieną funkcijos šabloną eina užrašas *template<typename AAA>* arba *template <class AAA>*, kur *AAA* yra tipo parametro vardas. Funkcijos *min* šablono parametras yra tipas *T*, kuris nebūtinai turi būti klasės vardas. Reikalaujama, kad tipas turėtų konkrečias savybes, bet nereikalaujame, kad priklausytų konkrečiai klasių hierarchijai. Mūsų pavyzdyje tipui *T* turi būti apibrėžtos palyginimo operacijos *>* ir *<*. Tai gali būti bazinis C++ kalbos tipas, kuriam automatiškai apibrėžtos palyginimo operacijos arba pačių apsirašyta klasė, su perkrautais palyginimo operatoriais. Žemiau pateikiamas makrosų ir šablonų panaudojimo pavyzdys:

```
int a = 10;
int b = 20;
cout << "MIN(a, b)      = " << MIN(a, b) << endl;
cout << "MAX(a, b)      = " << MAX(a, b) << endl;
cout << "min(a, b)      = " << min(a, b) << endl;
cout << "max(a, b)      = " << max(a, b) << endl;
```

Funkcijos šablono aprašymas nesugeneruoja jokio kodo į objektinį failą. Kad funkcijos šablonas įgautų pavidalą, reikia funkciją iškviesti. Funkcijos šablonas kviečiamas taip pat, kaip ir įprastinė funkcija. Kompiliavimo metu kompiliatorius pats sukonstruos reikiamą funkcijos kūną pagal funkcijos argumentų tipus. Jei skirtingose programos vietose kviesime su skirtingų tipų argumentais, tai kiekvienam tipų rinkiniui kompiliatorius sugeneruos po vieną funkcijos šablono kūną. Galima kompiliatoriui ir išreikštiniu būdu pasakyti, kokios funkcijos mes pageidaujame. Žemiau esantys

sakiniai yra ekvivalentūs, nes kintamieji *a* ir *b* yra tipo *int*:

```
cout << "min(a, b)      = " << min(a, b) << endl;
cout << "min<int>(a, b) = " << min<int>(a, b) << endl;
```

Nepamirškime, kad preprocesoriaus makrosai - tai "bukas" vieno simbolių pakeitimas kitais dar prieš kompiliavimą. Nevykdomas tipų atitikimo patikrinimas, be to galimi šalutiniai efektai, pvz.:

```
a = 10;
b = 20;
cout << "MIN(++a, ++b) = " << MIN(++a, ++b) << endl;
a = 10;
b = 20;
cout << "min(++a, ++b) = " << min(++a, ++b) << endl;
```

Šis kodo fragmentas atspausdins:

```
MIN(++a, ++b) = 12
min(++a, ++b) = 11
```

Pirmoje rezultato eilutėje iliustruotas nepageidaujamas efektas: kintamasis *a* buvo inkrementuotas du kartus, nes užrašas *MIN(++a, ++b)* po makroso išplėtimo prieš kompiliavimą yra ekvivalentus tokiame užrašui:

```
((++a) < (++b) ? (++a) : (++b))
```

Kai kas argumentuoja, jog makrosų naudojimas veikia greičiau nei funkcijos šablonas, nes sutaupomas funkcijos kvietimas. Tačiau panaudojus raktinį žodėlį *inline* galime leisti kompiliatoriui šią nelygybę ištaisyti:

```
template<typename T>
inline T min (T x, T y)
{
    return x < y ? x : y;
}
```

Taip pat atkreipkime dėmesį į užrašą *template<typename T>*, kuris yra ekvivalentus (su ypatingai retomis išimtimis) užrašui *template<class T>*, tik istoriškai atsirado kiek vėliau. Mat C++ kūrėjai eilinį kartą be reikalo taupė naujo raktinio žodžio įvedimą. Toliau mes vartosime žodelį *typename*, kadangi jis geriau atspindi prasmę: *T* - yra bet koks tipas, nebūtinai klasė.

Funkcijos *min* ir *max* yra labai paprastas pavyzdys. Standartinėje bibliotekoje yra gerokai sudėtingesnių funkcijų šablonų, pvz. *sort* ir *stable_sort* šablonai, kurie surūšiuoja bet kurį masyvo stiliaus konteinerį turintį savyje bet kokio tipo elementus, kuriems apibrėžtas operatorius *<*.

Prisiminkime, jog paprastų funkcijų antraštės talpinamos į *h*-failus, o kūnai į *cpp*-

failus, taip atskiriant interfeisą nuo realizacijos. Su funkcijų šablonais popieriai yra gerokai prastesni: visas funkcijos kūnas turi būti matomas kompiliatoriui kompiliavimo metu, todėl jis talpinamas į *h*-failą. Tas pats galioja ir klasių šablonams. Tokiu būdu ne tik atskleidžiamos realizacijos detalės, bet ir labai žymiai padaugėja kompiliuojamų eilučių kiekis. Bene pagrindinis kompiliatoriaus rūpestis kompiliuojant keliasdešimties eilučių programą, naudojančią STL konteinerius, yra sukompiliuoti keliasdešimt tūkstančių eilučių iš standartinių *h*-failų, turinčių savyje šablonus.

7.3. Klasių šablonai

Šio kurso metu mes dažnai naudojome sveikųjų skaičių steką. Nesunku suvokti, kad lygiai taip pat galime apsirašyti ir kitų bazinių tipų stekus: *char*, *float*, *double*, *long* ir t.t.. Būtų labai varginantis darbas apsirašyti naują steko klasę kiekvienam baziniam ar pačių sukonstruotam tipui. Milžiniškas kodo pasikartojimas. Čia mes galime pasinaudoti klasių šablonais. Visur, kur naudojome žodelį *int* nusakyti steko elemento tipui, dabar naudokime šablono parametrą, tipą *T*:

```
// stack_template.h

template<typename T>
class Stack
{
private:
    T*   elements;
    int  size;
    int  capacity;

public:
    Stack (int initialCapacity = 4);
    ~Stack ();

    void push    (T element);
    T  pop      ();
    T  peek     ();
    bool isEmpty () const;
};
```

Dabar pilnas klasės vardas yra *Stack<T>*. Jos metodų realizacijos reikalauja priedašo *template<typename T>*:

```
template<typename T>
Stack<T>::Stack (int initialCapacity)
{
    if (initialCapacity < 1)
        initialCapacity = 1;
    capacity = initialCapacity;
    size     = 0;
    elements = new T [capacity];
}

...

template<typename T>
T Stack<T>::pop ()
{
    if (size > 0)
        return elements[--size];
    throw std::string("pop() called on empty stack");
}
```

Dabar mes galime turėti stekus su norimo tipo elementais:

```
int main ()
{
    Stack<double> squares;
    for (int i = 1; i <= 10; i++)
        squares.push (i*i / 100.0);
    while (!squares.isEmpty())
        cout << squares.pop() << endl;

    Stack<char> word;
    word.push('s');
    word.push('u');
    word.push('l');
    word.push('a');
    while (!word.isEmpty())
        cout << word.pop();
    cout << endl;

    Stack<string> names;
    names.push("Aldona");
    names.push("Steponas");
    names.push("Zigmas");
    names.push(string("Birute"));
    while (!names.isEmpty())
        cout << names.pop() << endl;
}
```

Atkreipkime dėmesį į tai, kad klasė *Stack* išskirinėja elementų masyvus ir grąžina elementus pagal reikšmę. Tai reiškia, jog elementai privalo turėti konstruktorių pagal nutylėjimą, korektišką priskyrimo operatorių bei kopijavimo konstruktorių. Standartinė klasė *string* pasižymi visomis šiomis savybėmis. Aukščiau esantis kodo fragmentas atspausdins:

```
1
0.81
0.64
0.49
0.36
0.25
0.16
0.09
0.04
0.01
alus
Birute
Zigmas
Steponas
Aldona
```

7.4. Trumpai apie sudėtingesnes šablonų savybes

Ankstesnių skyrelių turėtų pakakti, kad galėtumėti naudotis standartinės C++ bibliotekos šablonais. Tuo tarpu šiame skyrelyje trumpai paminėsime kiek subtilesnes šablonų galimybes.

Šablonas gali turėti keleta parametrų:

Be to, parametrais gali būti ne tik tipų vardai, bet ir konstantos. Pvz., fiksuoto dydžio buferis:

```
template<typename T, int size> class Buffer {...};
Buffer<char, 12> charBuffer;
Buffer<Record, 8> recordBuffer;
```

Šablono parametrai gali nusakyti algoritmą:

Jie gali nurodyti, kokią operaciją atlikti su argumentais. Pvz., lyginant simbolines eilutes tenka atsižvelgti į kokrečiai kalbai būdingą abėcėlę: raidė "Ą" (A nosinė) eina po raidės "A" ir prieš raidę "B", kas nėra tiesa, jei lygintume tik simbolių kodus. Tuo tikslu galime apsisątyti palyginimo klasę su dviem statiniais metodais:

```
class ComparatorLT
{
public:
    static bool equal (char a, char b) {...}
    static bool less  (char a, char b) {...}
};
```

Kartu aprašyti eilučių palyginimo funkcijos šabloną, kuris ima kaip parametą palyginimo klasę. Ši funkcija grąžina reikšmę 0, kai eilutės lygios, neigiamą skaičių, kai pirma mažesnė už antrą, ir teigiamą skaičių kai pirma eilutė didesnė už antrą:

```
template<typename Comparator>
int compare (const string& a, const string& b)
{
    for (int i = 0; i<a.length() && i<b.length(); i++)
        if (!Comparator::equal(a[i], b[i]))
            return Comparator::less(a[i], b[i]) ? -1 : 1;
    return a.length() - b.length();
}
```

Tokią funkcijos šabloną galime iškviesti tokiu būdu:

```
int result = compare<CompareLT>(lithuanianName, n2);
```

Tokie šablono parametrų panaudojimai algoritmui nusakyti vadinamas "bruožais" (*traits*).

Šablonas gali turėti parametrus su reikšmėmis pagal nutylėjimą:

Pvz., apsirašykime bendrą simbolių palyginimo klasės šabloną, parametrizuojamą simbolio tipu:

```
template<typename T> class Comparator
{
public:
    static bool equal (T a, T b) {return T == T;}
    static bool less  (T a, T b) {return T < T;}
};
```

Jį galime panaudoti, kaip parametą pagal nutylėjimą kitame šablone - funkcijoje *compare*. Atkreipkime dėmesį, kad žemiau esantys tekstinių eilučių šablonai naudoja simbolio tipo parametą *T*. Čia tam atvejui, jei mums reikės vieno baito simbolių (*char*) eilučių arba dviejų baitų simbolių (*wchar_t*) eilučių:

```
template<typename T, typename C = Comparator<T> >
int compare (const String<T>& a, const String<T>& b) {...}
```

Tuomet funkciją *compare* galime naudoti keliais būdais:

```
int result1 = compare(str1, str2);           // Comparator<char>
int result2 = compare<char>(str1, str2);     // Comparator<char>
int result3 = compare<char, ComparatorLT>(str1, str2);
```

Pavyzdžiui, standartinėje bibliotekoje, kuomet naudojami įvairių tipų elementų konteineriai, naudojamas šablono parametras *allocator*, atsakingas už naujų elementų išskyrimą ir sunaikinimą. Pagal nutylėjimą jis realizuotas operatorių *new* ir *delete* pagalba.

Šablonus galima specializuoti:

Pvz., turint steko šabloną su elemento tipo parametru *T*, galime apsirašyti jo specializaciją, kuomet parametras yra rodyklė į *T*. Arba galime pateikti jo efektyvesnę specializaciją konkrečiam tipui, kai *T* yra kažkoks *MyType*:

```
template<typename T> class Stack {...}; // bedras visiems
template<> class Stack<MyType> {...}; // specializacija tipui
// MyType
```

Specializuoti galima ir funkcijų šablonus.

Klasės nariai-šablonai:

Klasės-šablono nevirtualūs metodai patys savo ruožtu gali būti metodai-šablonai su papildomu nuosavu šablono parametru:

```
template<typename T> class MyClass
{
    template<typename W> double myMethod(W w);
    ...
};
```

Tokio metodo realizacija vėliau apipavidalinama taip:

```
template<typename T> template<typename W>
double MyClass<T>::myMethod(W w) { ... }
```

Tokiu būdu klasė turės po vieną egzempliorių kiekvienam tipui T ir dar priedo konkrečiam tipui T po keletą metodo *myMethod* kūnų kiekvienam metodo argumento tipui W .

Išvada ir patarimas:

Bendru atveju, naudojant šablonus, galima užvirti tikrą makalynę. Užtenka vien pažvelgti į standartinės bibliotekos konteinerių *h*-failus. Sakoma, kad šablonų instancijavimas kompiliavimo metu yra ekvivalentus Tiuringo mašinai. Žemiau pateiktas pavyzdys, kaip naudojant šablonus priversti kompiliatorių kompiliavimo metu (o ne programos vykdymo metu!) traukti kvadratinę šaknį:

```
// turing.cpp

const int N = 132 * 132;

template <int Size, int Low = 1, int High = Size> struct Root;

template <int Size, int Mid> struct Root<Size, Mid, Mid>
{
    static const int root = Mid;
};
template <int Size, int Low, int High> struct Root
{
    static const int mean = (Low + High)/2;
    static const bool down = (mean * mean >= Size);
    static const int root =
        Root<Size, (down?Low:mean+1), (down?mean:High)>::root;
};
int main()
{
    cout << "ceil(sqrt(" << N << "))=" << Root<N>::root << endl;
}
```

Ši programa kompiliavimo metu apskaičiuos konstantas ir atspausdins:

```
ceil(sqrt(17424))=132
```

Taigi, šablonus reikėtų pačiam programuoti tik ten, kur jie yra prasmingi. Daug geriau - pasinaudoti kitų parašytais ir gerai ištestuotais šablonais, pvz. stadartine C++ šablonų biblioteka (*STL*).

8. Daugialypis paveldėjimas

8.1. Daugialypio paveldėjimo pavyzdys

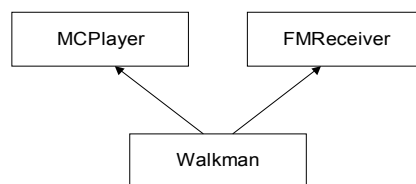
C++ kalba turi daugialypį paveldėjimą: klasė vienu metu gali paveldėti duomenis ir metodus iš keletos bazinių klasių. Pavyzdžiui, nešiojamas kasečių kompaktinių plokštelių grotuvas ("ausinukas", "volkmanas") dažnai turi ir radijo imtuvą, todėl galime sakyti, jog ausinukas yra kasečių grotuvas ir ausinukas taip pat yra radijo imtuvas:

```
class MCPlayer
{
    ...
};

class FMReceiver
{
    ...
};

class Walkman : public MCPlayer, public FMReceiver
{
    ...
};
```

Grafiškai tokį paveldėjimą galime pavaizduoti taip:



Klasės *Walkman* objektus dabar galime naudoti visur ten, kur reikia *MCPlayer* ar *FMReceiver* objektų:

```
void tuneReceiver (FMReceiver* fm)
{
    ...
}
```

```
void playMC (MCPlayer* mc)
{
    ...
}

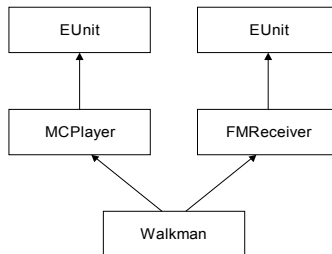
int main ()
{
    Walkman* walkman = new Walkman();
    tuneReceiver (walkman);
    playMC      (walkman);
}
```

8.2. Pasikartojančios bazinės klasės ir virtualus paveldėjimas

Tiek kasečių grotuvas, tiek ir radijo imtuvas yra elektroniniai įrenginiai. Natūralu, kad jie savo ruožtu paveldėtų nuo elektroninio įrenginio klasės *EUnit*:

```
class EUnit {...};
class MCPlayer : public EUnit {...};
class FMReceiver : public EUnit {...};
class Walkman : public MCPlayer, public FMReceiver {...};
```

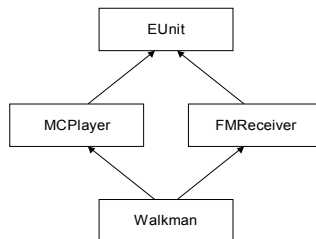
Dabar klasės *Walkman* objektai turės du *EUnit* poobjekčius:



Pasikartojanti bazinė klasė *EUnit* gali sukelti tam tikrų problemų, nes abu jos egzemplioriai klasėje *Walkman* gyvens nepriklausomai, o tipų konversija iš *Walkman* į *EUnit* bus nevienareikšmė. Iš tiesų *Walkman* yra *EUnit* pagal paveldėjimo liniją, nors ir netiesioginę, tačiau kuriuo iš dviejų *EUnit*ų yra *Walkman*'as? Galime kompiliatorių priversti sugeneruoti tik vieną *EUnit* poobjektį klasės *Walkman* objekte. Tuo tikslu klasės *MCPlayer* ir *FMReceiver* turi naudoti virtualų paveldėjimą:

```
class EUnit {...};
class MCPlayer : virtual public EUnit {...};
class FMReceiver : virtual public EUnit {...};
class Walkman : public MCPlayer, public FMReceiver {...};
```

Virtualus paveldėjimas neturi nieko bendro su virtualiais polimorfiniais metodais, nebent tik tai, jog abiem atvejais kiek sulėtėja programos veikimas. Eilinį kartą taupytas naujo raktinio žodžio įvedimas. Bet koku atveju gauname klasikinį rombą:



8.3. Virtualios bazinės klasės konstravimo problema

Virtualios bazinės klasės išsprendžia pasikartojančios bazinės klasės problemą, tačiau atsineša kitą, bendru atveju neišsprendžiamą, virtualios bazinės klasės konstravimo problemą. Kai mes konstruojame klasės *Walkman* objektą, tai pradžioje yra kviečiami klasių *MCPlayer* ir *FMReceiver* poobjekčių konstruktoriai. Pastarieji savo ruožtu kviečia klasės *EUnit* konstruktorių, perduodami kiekvienas savo parametrus. Tačiau klasės *Walkman* objektas savyje turi tik vieną *EUnit* poobjektį, kuris yra konstruojamas tik vieną kartą naudojant konstruktorių pagal nutylėjimą. Žemiau esantis pavyzdys demonstruoja šią virtualios bazinės klasės konstravimo problemą:

```
// multi_inher.cpp

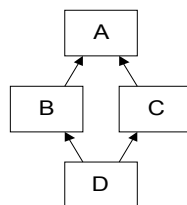
class A { A (int i = 0) {...} };
class B : virtual public A { B () : A(1) {...} };
class C : virtual public A { C () : A(2) {...} };
class D : public B, public C {}

int main ()
{
    A a;
    B b;
    C c;
    D d;
}
```

Kiekvienos klasės konstruktorius atspausdins po vieną eilutę ekrane:

```
A() i=0
A() i=1
B()
A() i=2
C()
A() i=0
B()
C()
D()
```

Grafiškai turime jau matytą rombą:



Matome, jog klasės *B* objektas norėtų, jog jo poobjektis *A* būtų sukonstruotas su argumentu 1, o klasės *C* objektas nori, kad jo poobjektis *A* būtų sukonstruotas su argumentu 2. Tuo tarpu klasės *D* objekto poobjekčiai *B* ir *C* dalinasi vieninteliu poobjekčiu *A*. Kompiliatorius šioje vietoje nutaria nekreipti į objekto *D* poobjekčių *B* ir *C* norus, irkviečia bendro poobjekčio *A* konstruktorių pagal nutylėjimą.

Bendru atveju virtualios bazinės klasės konstravimo problemos išspręsti neįmanoma. Egzistuoja tik dalinis šios problemos sprendimas, kuomet klasės *D* konstruktoriuje galima nurodyti, kurį netiesioginės bazinės klasės *A* konstruktorių iškviešti, kad "daugmaž tikty" ir poobjekčiui *B* ir *C*:

```
class D2 : public B, public C
{
public:
    D2() : A(15) {cout << "D2()" << endl;}
};

int main ()
{
    D2 d2;
}
```

Šis kodo fragmentas atspausdins:

```
A() i=15
B()
C()
D2()
```

Praktikoje yra geriau paveldėti tik iš vienos įprastinės klasės su duomenimis ir iš kiek norime daug "švarių interfeisų". Prisiminkime, jog švarus interfeisas, tai klasė, kuri neturi jokių duomenų ir metodų, o tik švariai virtualius metodus ir tuščią virtualų destruktorių. Kadangi švarus interfeisas neturi duomenų, tai neegzistuoja ir jo konstravimo problema.

8.4. Objektų tipų identifikacija programos vykdymo metu

Objektų tipų identifikacija programos vykdymo metu (*run-time type identification*) galioja tiek vienalypiam, tiek ir daugialypiam paveldėjimui. Prisiminkime, jog klasė, turinti virtualų destruktorių arba bent vieną virtualų metodą, vadinama polimorfine klase. Kaip jau žinome, visų paveldėjusių klasių objektai kartu yra ir bazinės klasės objektai, todėl galime "natūraliai" atlikti tipų konversiją iš paveldėjusios klasės į bazinę:

```
// rtti.cpp

class A { public: virtual ~A() {} };
class B : public A {};

int main ()
{
    A* a = new A();
    B* b = new B();
    a = b; // natūrali tipų konversija
}
```

Jau programos kompiliavimo metu yra žinoma, kad rodyklė *b* rodo į klasės *B* tipo objektą, kuris paveldi iš klasės *A*. Todėl priskyrimo sakiny *a = b* yra korektiškai išsprendžiamas dar kompiliavimo metu. Kartais reikia atlikti atvirkščią tipų konversiją: turime rodyklę į *A*, bet iš anksto tikimės, kad ji bus nukreipta į objektą *B*. Ar tai tiesa ar ne galime patikrinti tik programos vykdymo metu. Polimorfinės klasės ar nuo jos paveldėjusių klasių objektams programos vykdymo metu saugoma informacija apie objekto tipą. Šiuo atveju, naudojantis raktiniu žodeliu *dynamic_cast*, mes galime atlikti saugią tipų konversiją iš bazinės klasės *A*, į paveldėjusią klasę *B*, nes klasė *A* turi virtualų destruktorių:

```
void safeCastFromAtoB (A* a)
{
    B* b = dynamic_cast<B*>(a);
    if (b == 0) cout << "cast A -> B failed" << endl;
    else      cout << "cast A -> B succeeded" << endl;
}

int main ()
{
    A* a = new A();
    B* b = new B();
    safeCastFromAtoB(a);
    safeCastFromAtoB(b);
}
```

Šis kodo fragmentas atspausdins:

```
cast A -> B failed
cast A -> B succeeded
```

Raktinis žodelis *dynamic_cast* tarp paprastų skliaustų reikalauja rodyklės (kintamojo ar reiškinių), o tarp ženklų <> - rodyklės tipo, į kurį konvertuosime duotą rodyklę. Jei pavyko, grąžinama saugiai konvertuota rodyklė, jei ne - grąžinamas nulis. *dynamic_cast* galime naudoti ir nuorodų konversijai, tik čia, klaidos atveju, metama standartinė klaida *bad_cast*.

Raktinio žodelio *typeid* pagalba galime gauti informaciją apie polimorfinės klasės objekto tipą. Šis žodelis reikalauja objekto, kaip savo vienintelio argumento, ir grąžina nuorodą į standartinę struktūrą *type_info*, apibrėžtą *h*-faile *typeinfo*. Mes naudosime šios struktūros metodą *name*:

```
void printTypeName (A* a)
{
    cout << "type name: " << typeid(*a).name() << endl;
}

int main ()
{
    A* a = new A();
    B* b = new B();
    printTypeName(a);
    printTypeName(b);
    a = b;
    printTypeName(a);
}
```

Šis kodo fragmentas atspausdins:

```
type name: 1A
type name: 1B
type name: 1B
```


9. OOP receptai

9.1. Objektiškai orientuotos visuomenės folkloras

Ne vienas pradedantis, o dažnai ir toliau pažengęs programuotojas, savo kailiu patyrė, kad konkrečios programavimo kalbos sintaksės žinojimas tik iš dalies padeda spręsti realaus gyvenimo problemas. Galų gale, objektiškai orientuota programavimo kalba, kokia ji bebūtų, yra tik įrankis programoms kurti. Toliau sprendžiami aukštesnio lygio uždaviniai: klasių struktūrų ir tarpusavio sąveikos projektavimas, pusiausvyros tarp kodo lankstumo, vykdymo greičio ir kodo aiškumo ieškojimas. Kadangi objektiškai orientuotas programavimas gyvuoja keletą dešimtmečių, tai ir patirties daugeliui uždavinių spręsti yra sukaupta tikrai daug. Ją galėtume pavadinti objektiškai orientuotos visuomenės folkloru.

Tolesnieji trys skyreliai atitinkamai remiasi trimis puikiomis knygomis:

1. **Erich Gamma, Richard Helm, Raph Johnson, John Vlissides (GOF - Gang Of Four):** *Design Patterns; Elements of Reusable Object-Oriented Software*; Addison Wesley Longman, Inc., 1995
2. **Martin Fowler:** *Refactoring; Improving the Design of Existing Code*; Addison Wesley Longman, Inc., 1999
3. **Kent Beck:** *eXtreme Programming eXplained; Embrace Change*; Addison-Wesley, 2000

Šiose knygose vietoje painių teorinių išvedžiojimų pateikta krūvos praktikoje patikrintų patarimų ir sprendimų. Mes tik trumpai paliesime kai kurias folkloro perliukus. Programuotojui primygtinai rekomenduojama susupažinti su aukščiau išvardintomis knygomis. Sykį perskaite, vėliau dar ne kartą prie jų grįšite. Laikui bėgant augs ir jūsų kaip programuotojų patirtis, tuomet galėsite naujai įvertinti kolegų receptus, palyginti su nuosavais pastebėjimais ir pateikti savų idėjų.

Beje, aukščiau išvardintos knygos turi omenyje, jog skaitytojas jau įgijo pakankamai galias programavimo kalbos žinias (C++ arba Java). Jei trūksta kasdienių programavimo įgūdžių, rekomenduojama paskaityti labai šaunią Majerso knygą:

- **Scott Meyers:** *Effective C++, Second Edition; 50 Specific Ways to Improve Your Programs and Designs*; Addison Wesley, 1998
- **Scott Meyers:** *More Effective C++: 35 New Ways to Improve Your Programs and Designs*; Addison Wesley, 1995

9.2. Projektavimo šablonai (*design patterns*)

Erich Gamma, Richard Helm, Raph Johnson, John Vlissides (GOF - Gang Of Four):
Design Patterns; Elements of Reusable Object-Oriented Software; Addison Wesley Longman, Inc., 1995

Objektiškai orientuotų (OO) programų kūrimas yra sunkus uždavinys, o daug kartų panaudojamų programinių komponentų kūrimas - dar sunkesnis. Patyrę OO projektuotojai žino, kad lankstaus ir ateiityje naudingo dizaino sukūrimas iš pat pirmo karto yra sudėtingas, o dažniausiai ir neįmanomas dalykas. Ir visgi ekspertai sumeistrąja puikius programų modelius. Ką gi žino ekspertai, ko nežinotu naujokai?

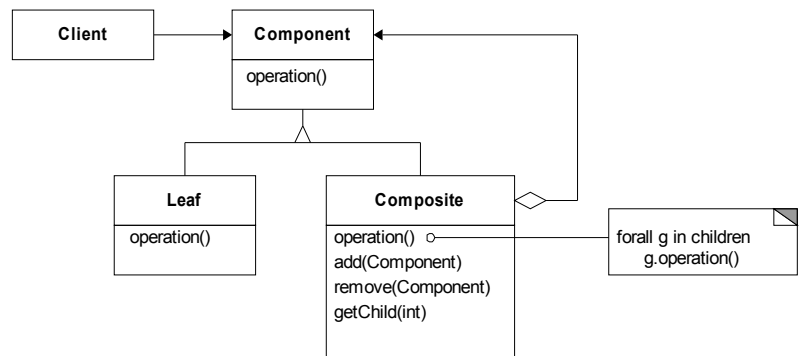
Vienas dalykas, kurio tikrai nedaro ekspertai, yra programų kūrimas nuo nulio. Jie verčiau pasieško gatavų sprendimų, kurie puikiai veikė praeityje, ir, labai tikėtina, toliau puikiai veiks. Projektavimo šablonai - tai būdas nuosekliai dokumentuoti projektuotojų patirtį, t.y. programuotojų folklorą - legendas ir padavimus. Prie kiekvieno projektavimo šablono, be kitų dalykų, yra pažymima:

- problema, kurią norime išspręsti,
- sprendimas, susidedantis iš keletos klasių ir jų tarpusavio sąveikos,
- sprendimo pritaikymo pasėkmės, t.y. jo privalumai ir trūkumai.

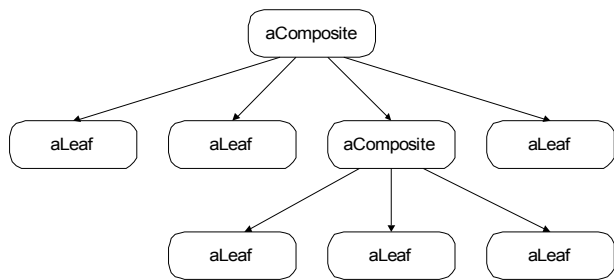
Autorių kolektyvas, ketverto gauja, sudarė 23-jų projektavimo šablonų katalogą. Kiekvienas šablonas turi pavadinimą su užuomina apie jo paskirtį. Šiame kurse mes jau palietėme mažiausiai tris projektavimo šablonus:

- *Singleton* - užtikrina, jog klasė turės tik vieną egzempliorių, kurį galima pasiekti globaliai
- *Composite* - komponuoja objektus į medžio pavidalo hierarchijas; pavieniai ir sudėtiniai objektai traktuojami vienodai
- *Iterator* - leidžia paeiliui perbėgti konteinerio elementus, kartu paslepianč vidinę konteinerio realizaciją

Composite šablonas yra vienas populiariausių. Grafinės langų sistemos yra puikus panaudojimo pavyzdys: langai susideda iš grafinių komponentų, kurių kiekvienas savo ruožtu gali susidėti iš smulkesnių ir t.t.. Žemiau pateikta apibendrinta *Composite* šablono klasių hierarchija:



Tipinė *Composite* objektų struktūra:



9.3. Programų pertvarkymas (*refactoring*)

Martin Fowler: *Refactoring: Improving the Design of Existing Code*; Addison Wesley Longman, Inc., 1999

Tiek dideli, tiek ir maži projektai dažnai susiduria su tokia problema: kodas nepaliaujamai auga ir pučiasi, pridedant naujų funkcijų pradeda griūvinėti anksčiau parašyti moduliai, niekas normaliai nesusigauja, kurios dalys įtakoja viena kitą ir pan.. Tokiu atveju pakvimpa programų pertvarkymu (*refactoring*): tenka pertvarkyti (sakoma, "išvalyti") egzistuojantį kodą, išsaugant ankstesnį funkcionalumą. T.y. programos pertvarkymo metu nėra diegiamas naujas ar keičiamas senas funkcionalumas, tiesiog išeities tekstai tampa tvarkingesni, geriau struktūrizuoti ir labiau suprantami.

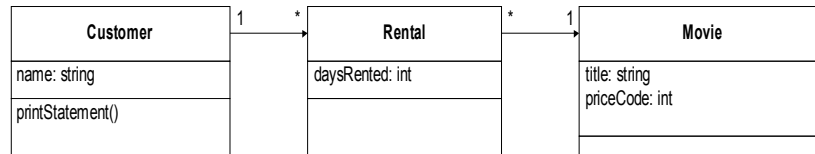
Nereikia nė sakyti, jog dar kartą perrašyti tą patį kodą mėgsta nedaugelis. Labai nedaugelis. Tačiau nebūtina keisti viską iš karto. Martinas Fowleris moko, jog viską reikia daryti mažyčiais, aiškiai apibrėžtais ir gerai kontroliuojamais žingsneliais. Galbūt vienas modulis bus pertvarkytas per valandą, o kitam prireiks kelių mėnesių.

Kada reikia programą pertvarkyti? Paprastai, palaipsniui projektuojant ir programuojant, patyręs programuotojas kartu pertvarkinėja ir neseniai parašytą kodą. Kitas atvejis, kai reikia įdiegti naujų funkcijų, tačiau egzistuojantis kodas joms nepritaikytas. Kaip žinoti, kurias programos vietas reikia keisti? O gi tas, kurios "nemaloniai kvepia". Martinas parodo, kaip aptikti visą eilę "nemalonių kvapų" ir kaip juos pažingsniui pašalinti. Rezultate gauname programą, kuri daro tą patį, tik "maloniau kvepia".

Pertvarkant svarbu prisilaikyti bent dviejų principų:

- turėti pasirašius automatinius programos testus, kurie patikrina, ar pertvarkyta programa elgiasi taip, kaip ir anksčiau
- pertvarkyti tik labai mažai žingsneliais ir atlikti automatinius testus - jei kuris neveiks, bus aišku, kad klaida slypi naujausiuose pakeitimuose

Šiame skyrelyje apsiribosime vienu pavyzdžiu: video nuoma. Tai programa, kuri duotam klientui atspausdina, kokius jis filmus yra išsinuomavęs ir kiek jis turi mokėti. Mokestis priklauso nuo to, kuriam laikui filmas yra išnuomotas ir nuo filmo tipo. Tėra trys filmų tipai: reguliarūs, vaikiški ir nauji filmai. Tuo tikslu konstruojama paprastutė klasių diagrama, leidžianti atspausdinti tekstinę ataskaitą apie klientą:



Smulkesnės detalės pateikiamos Faulerio knygoje. Beje, tokiai paprastutei programėlei šis dizainas puikiausiai tinka. Ir nors iš klasių diagramos sunku pamatyti, tačiau visgi atsiranda tam tikrų problemų, kuomet mes užsimanome naujų dalykų:

- ataskaitos ne tik tekstiniame, bet ir HTML formate
- laisvai kaitaliojamos filmų kainų politikos ir nuolaidų skaičiavimo dažniams videotekos klientams nuomuojančios naują filmą (uždirbtų taškų skaičius)

Fauleris parodo, kaip žingsnis po žingnio pertvarkyti šią diagramą lydintį kodą ir gauti naują klasių hierarchiją:



Kaip sako Fauleris: svarbiausia pertvarkymo ritmas - mažas pakeitimas, testas, mažas pakeitimas, testas ir t.t..

9.4. Ekstremalus programavimas (*eXtreme programming*)

Kent Beck: *eXtreme Programming eXplained; Embrace Change*; Addison-Wesley, 2000 (<http://www.xprogramming.com>)

Ekstremalus programavimas - tai receptų rinkinys viso programavimo proceso organizavimui. Jame nėra nieko ekstremalaus, nebent tai, jog jis gerai veikia tik tuomet, kai didesnė dalis receptų yra naudojama ir turi mažai prasmės, kai naudojamas atmetinai. Be to, jame yra keletas patarimų, su kuriais ne kiekvienas programuotojas iš karto sutiks - ekstremalumas slypi tame, jog programuotojui tenka priversti save keisti netikusius įpročius. Žemiau pateikti kai kurie bendrai pripažinti geri įpročiai, tik jie yra hiperbolizuoti iki ekstremalumo:

- jei **kodo peržiūrėjimas** yra gerai, tai mes jį peržiūrėsime nepertraukiamai (*pair programming*: programavimas poromis - du žmonės prie kiekvieno kompiuterio);
- jei **testavimas** yra gerai, tai mes testuosime nuolatos (*unit testing*: automatiniai modulių testai, kurie grąžina arba OK, arba išvardina rastas klaidas);
- jei **projektavimas** yra gerai, tai tuomet mes jį darysime kasdien (*refactoring*: nuolatinis programų pertvarkymas);
- jei paprastumas yra gerai, mes visuomet paliksime patį paprasčiausią programos dizainą, kuris palaiko visas reikiamas funkcijas;
- jei atskirų modulių tarpusavio integracija ir testavimas yra gerai, mes apjunginėsime savo modulius kelis kartus per dieną;
- jei darbų iteracijos yra gerai, mes planuosime labai mažus darbus net valandomis, o ne savaitėmis ar mėnesiais.

Tai nėra visi receptai. Plačiau apie juos pačius bei jų argumentaciją galima rasti Kento Beko knygoje. Taigi, programuotojai sėdi poromis, parašo testus klasėms anksčiau, negu kad pačias klases, jie daug kalbasi, keletą kartu per dieną padeda savo kodą į bendrą serverį ir pasiima iš jo kitų programuotojų darbą, perkompiluoja ir testuoja, vėl kalbasi, keičiasi poromis, keičia kitų parašytą kodą ir t.t.. Gal tik ne taip chaotiškai.

Reikėtų pastebėti, jog ekstremalus programavimas tinka tik mažiems ar vidutiniams projektams. Jis reikalauja aukštos programuotojo, kaip kolektyvo nario, kultūros ir disciplinos.

Labai svarbi darbo atmosfera. Visuomet privalo būti pakankamai ramu, gaivu ir prikrauta pakankamai užkandžių. Sykį Kentas Bekas konsultavo vienos firmos programuotojus. Primokė visokių receptų. Vėliau praktika parodė, jog didžiausią įtaka programuotojams padarė stalų perstumdymas: anksčiau visi keturi stalai buvo prie skirtingų sienų, o po to - visi kambario viduryje. Kiekvienas matė kito veidą ir bendravimas vyko mieliau ir efektyviau.

Pats Kentas Bekas apie save sako: aš nesu fantastiškas programuotojas, aš tiesiog geras programuotojas su fantastiškais įpročiais.