KLASIŲ BIBLIOTEKOS.

DAUGKARTINIS FUNKCIJŲ IR KLASIŲ PANAUDOJIMAS

Greitas programų kūrimas grindžiamas anksčiau sukurtų programinių komponentų panaudojimu. Su procedūrinėmis kalbomis (FORTRAN, Pascal, C) yra naudojamos funkcijų bibliotekos, kuriose realizuotos tiek atskiros funkcijos (įvedimo-išvedimo, matematinės ir kt.), tiek tipiniai duomenų apdorojimo algoritmai. Kaip jau minėta, C++ kalba suderinta su C kalba, o pastorosios biblioteka įeina į standartinę C++ biblioteką, tad C++ programose galima naudoti C bibliotekines funkcijas.

Funkcijų ir klasių bibliotekos turi iš esmės vienodą paskirtį – daugkartinį jau užprogramuotų sprendimų panaudojimą. Tačiau tokių bibliotekų organizavimo ir panaudojimo principai yra skirtingi, nes bibliotekos palaiko skirtingas programavimo paradigmas.

Standartinės funkcijos adaptuojamos konkrečioje programoje nurodant skirtingus argumentus. Kai pačios funkcijos aiškiai apibrėžtos (pvz., matematinės funkcijos: *exp, sqr, sqrt, sin* ir kt.), tokio adaptavimo mechanizmo visiškai užtenka, o argumentų tipas būna fiksuotas. Kai argumeto tipas iš anksto nežinomas, galima pasinaudoti „universaliu tipu“ *void*. Pvz., standartinės C bibliotekos greito rūšiavimo funkcijoje *qsort*, vienas argumentas (rodyklė į duomenų pradžią) turi reikšmę *void*, o kitas užduoda vieno duomenų atributo (elemento) dydį:

```
void qsort(void base, size_t num, size_t wdth,
        int (*compare) (const void* elem1, const void* elem2));
```

Kadangi kitus standartinių funkcijų adaptavimo būdus būtų gana sudėtinga realizuoti, funkcijų bibliotekų taikymai apsiribuoja matematinėmis ir pagalbinėmis žemio lygio (pvz., įvedimo-išvedimo) funkcijomis, bei, kartias, specializuotomis funkcijų bibliotekomis tam tikros grupės uždavinių sprendimui (pvz., įvairūs diferencialinių lygčių sprendimo metodai).

Galimybės adaptuoti sukurtas klases ir objektus konkrečiam uždaviniui yra žymiai platesnes. Pirmas būdas analogiškas funkcijų adaptavimui. Klasės talpinamos į biblioteką ir derinamos prie konkrečios programos poreikių naudojant parametrus. Toks būdas leidžia lanksčiai įgyvendinti klasėmis, pvz. matematines funkcijas, darbą su datomis ir laiko intervalais bei kitais universaliais objektais.

Bet kartu, objektinis programavimas leidžia esamos bibliotekos pagrindu kurti naujas klases ir klasių sistemas. Dažniausiai tai daroma taikant paveldėjimą. Tokiu atveju bazinės klasės realizuoja pagrindinius duomenų apdorojimo algoritmus, o konkrečioje programoje iš jų kuriamos išvestinės klasės, pritaikytos prie konkretaus uždavinio poreikių. Pavyzdys – *MFC (Microsoft Foundation Classes)*, t.y. bazinių klasių biblioteka.

Kitas klasių adaptavimo būdas – tam tikro interfeiso (sąsajos) realizacija, sukurta taip, kad klasė galėtų funkcionuoti su kitomis klasėmis. Šis būdas taikomas dirbant su šablonais. Pvz., tam kad galima būtų panaudoti tam tikros klasės objektus kaip kintamo dydžio masyvo atributus (elementus), tokia klasė turi turėti standartinį konstrūktorių ir priskyrimo operatorių.

KLASIŲ BIBLIOTEKŲ KŪRIMAS. TAIKOMŲJŲ PROGRAMŲ KARKASAI.

Klasų bibliotekos kūrimas prasideda nuo tikslo ir sudarymo principų apibrėžimo. Pagal tikslą skiriamos universalios ir specializuotos bibliotekos. Pagal sudarymo principus bibliotekos skirstomos į konkrečių klasių bibliotekas, šablonų bibliotekas, abstrakčiųjų klasių bibliotekas ir kūrimo terpes (aplinkas, karkasus).

Konkrečių klasių bibliotekos realizuoja visiškai paruoštas konkrečiam taikymui klases. Programoje tokių klasių adaptuoti nereikia, reikia tik sukurti šių klasių objektus. Šablonų bibliotekos apibrėžia klasių šablonus, kuriuos konkrečiose programose reikia dar papildomai apibrėžti nurodant šablonams argumentus (klases), naudojamus šiose programose. Abstrakčiųjų klasių bibliotekos būtinai numato išvestinių klasių kūrimą. Naudojamos ir mišrios bibliotekos, kuriose nors ir apibrėžiamos konkrečios klasės, jos būna gana bendros ir programose reikia kurti išvestines klases (nors žiūrint formaliai šios bibliotekinės klasės nėra abstrakčios).

Kūrimo terpės (aplinkos, karkasai) yra sudėtingiausia bibliotekų forma, nes joje apibrėžiamas ne tik klasių rinkinys, bet ir klasių naudojimo būdai. Pvz., jau minėta Microsoft korporacijos sukurta bazinių klasių biblioteka *MFC* yra būtent tokia terpė, kurioje realizuojamas architektūros modelis „dokumentas-rodinys“ (angl. *Document/View Architecture*). *MFC* sukuria tarsi taikomosios programos skeletą su dokumento klasę ir rodinio (vaizdo) klase. Visas duomenų valdymas vykdomas šių dviejų klasių. Klasė Dokumentas saugo duomenis ir valdo duomenų išvedimą ir

atnaujinimą skirtingose peržiūros languose (rodiniuose-vaizduose). Rodinys rodo duomenis ir valdo vartotojo sąveiką su jais, įskaitant duomenų atrankos ir redagavimo veiksmus.

MFC atsirado jau pirmuose *Microsoft Visual C++* kompiliatoriuose (terpėse), kur C++ klasėse įgyvendino *Windows API* funkcionalumą, t.y. suteikė programuotojui taikomųjų programų kūrimo karkasą. *MFC* klasės buvo apibrėžtos visiems pagrindiniams *Windows* grafines terpės objektams, t.y., langams, valdymo elementams ir pan. Tokiu būdu kurdamas taikomąją programą su *MFC*, programuotojas yra išlaisvintas nuo tiesioginio darbo su Windows *API* funkcijomis. Dabartiniu metu *MFC* (statinė *.lib* arba dinaminė *mfc110.dll* biblioteka, v. 11) įeina į Visual Studio 2005/2008/2010/2012 komercines versijas, bet nėra įtraukta į laisvai platinamas *(Express)* versijas. Jose galima naudoti paprastesnę *Windows Template Library (WTL).*

Populiariuose *Borland* kompanijos produktuose (pvz., Borland Turbo C++) buvo naudojama analogiška *Object Windows Library (OWL)*, vėliau (Borland C++ Builder, Borland Delphi) pakeista į dabar naudojamą vizualinių komponentų biblioteką (karkasą) *VCL (Visual Component Library)*.

Svarbus klasių bibliotekų kūrimo aspektas – klasių, kintamųjų, konstantų ir t.t. vardų konvencija, nes bibliotekų vardai neturi sutapti su programoje naudojamais kitais vardais. Dabartiniu metu patikimiausias būdas yra taikyti vardų kontekstą (sritį). Iki vardų kontekstų atsiradimo įprastas būdas buvo specializuotų vardų priešdėlių panaudiojimas. Pvz., kompanijos *Rogue Wave* bibliotekoje *Tool.h++* visų klasių vardai prasideda priešdėliu *RW (RWDate, RWCString),* bibliotekos *ODMG* vardų sistemoje naudojamas priešdėlis *d_ (d_Time, d_Database),* o *MFC* bibliotekoje daug funkcijų turi priešdėlį *Afx* (pagal seną pavadinimą *Application Framework Extensions*).

STANDARTINĖ ŠABLONŲ BIBLIOTEKA – STL.

Standartinė šablonų biblioteka (angl. *Standard Template Library STL)* sukurta kompanijos Hewlett-Packard tyrimų laboratorijoje kaip bendrųjų algoritmų ir duomenų struktūrų biblioteka C++ kalbai. Biblioteka buvo standartizuota ir pradėta laisvai platinti nuo 1994 m. (Stepanov, Lee, Musser). Kiek vėliau adaptuota STL biblioteka tapo C++ standartinės bibliotekos dalimi, o jos autoriai gavo prestižinę JAV kompiuterininkų sąjungos ACM premiją.

STL biblioteka susidėda iš ortogonalių (statmenų) konteinerių (kolekcijų), iteratorių ir algoritmų realizacijų. **Ortogonalumas** reiškia nepriklausomumą, t. y. šios bibliotekos kolekcijos, iteratoriai ir algoritmai gali būti naudojami bet kokiose kombinacijose, pvz., bet kuris algoritmas nepriklauso nuo naudojamos kolekcijos (konteinerio) tipo. Dar vienas STL bruožas – joje beveik nenaudojamas paveldėjimas.

STL bibliotekoje yra ir kitų komponentų: atminties skirstytojai, predikatai ir lyginimo funkcijos. STL predikatas – tai funkcija, kurios reikšmė loginio arba sveikojo tipo (*false*, 0 ir *true*, ≠0). Predikatai būna unariniai ir binariniai, jie naudojami kaip universalių algoritmų parametrai. Specialus binarinio predikato atvejis – lyginimo funkcija, kurios reikšmė yra *true*, jei pirmas argumentas didesnis už antrąjį.

Konteineriai ir algoritmai.

Konteineris – tai klasė, kurios objektuose saugomas organizuotas elementų (duomenų) rinkinys. Prieiga prie elementų yra reglamentuota tam tikromis apibrėžtomis taisyklėmis. STL konteineriai realizuoja įvairias sudėtines duomenų struktūras: sąrašus, eiles, stekus, dekus, vektorius, reikšmių masyvus, dėstymo lenteles, aibes ir kt.

Konteineriai skirtsomi į dvi pagrindines grupes: nuoseklūs ir asociatyvūs konteineriai. **Nuoseklūs konteineriai** – tai vektorius (*vector*), sąrašas (*list*) ir dėkas (*deque*). **Asociatyvūs konteineriai**: aibė (set), multiaibė (*multiset*), atvaizdas (*map*) ir multiatvaizdas (*multimap*). Taip pat yra vadinamieji **konteinerių adapteriai**, kuriems apibrėžtas specializuotas interfeisas ir kurie įgyvendinami naudojant kitus konteinerius: eilė (*queue*), prioretizuota eilė (*priority_queue*) ir stekas (*stack*). Kadangi visos šios klasės yra šablonai, jais galima generuoti konteinerius, susidedančius iš bet kokio tipo objektų. Pvz., sveikųjų skaičių vektorių galima naudoti vietoj sveikaskaitinio masyvo nesirūpinant dinaminės atminties skirstymo klausimais:

```
vector<int> vec(3);
vec[0] =10;
vec[1] = vec[0] – 5;
vec[2] = vec[0]+ vec[1];
```

STL bibliotekoje yra platus algoritmų, skirtų konteinerių duomenims apdoruoti, rinkinys. Šie algoritmai realizuoti šablonėmis funkcijomis. Pvz., naudojant bibliotekos algoritmą *reverse* galima aukščiau pateiktoje vektoriaus kolekcijoje išdėstyti elementus atvirkštine tvarka:

reverse(vec.begin(), vec.end());

Tokia funkcija nėra klasės narė, o yra globalinė ir operuoja ne su atskirais elementais, bet su konteinerio elementais iš tam tikro intervalo. Būtent taip pasiekiamas minėtas STL bibliotekos algoritmų ortogonalumas, t.y. algoritmai atskirti nuo klasių. Pvz., funkciją *reverse()* galima naudoti su bet kokiu konteineriu, kuriam galima nurodyti intervalą (iš esmės su visais STL konteineriais). Galima taikyti tokias funkcijas netgi su standartiniais masyvais:

```
double A[5] = {2, 3, 5, 8, 13};
reverse(A, A+5);
for (int i=0; i<5; ++i)
        cout << "A(" << i  << ") = " << A[i];
```

Čia pirmas argumentas nurodo intervalo pradžią, o antras – intervalo pabaigą (į patį intervalą neįeina).

Iteratoriai. Iš tikrųjų funkcijos *reverse()* argumentai yra specialaus pavidalo objektai, vadinami iteratoriais. Iteratorius apibendrina rodyklės sąvoką ir saugo elemento, su kuriuo dirbama, padėtį konteineryje. Su iteratoriais apibrėžtos analogiškos rodyklėms operacijos: perėjimas prie kito konteinerio elemento ++ (--) ir elemento, į kurį rodo iteratorius, reikšmės gavimas * .

Iteratoriai būna tiesioginiai ir atvirkštiniai: tiesioginiai iteratoriai leidžia nuosekliai eiti per konteinerį nuo jo pradžios link pabaigos, o atvirkštiniai – nuo pabaigos iki pradžios. Reikėtų pabrėžti, kad ir tiesioginiai, ir atvirkštinai iteratoriai naudoja inkrementinę operaciją ++. Taip pat atskiriami tokie iteratorių tipai: tiesinis, dvikryptis, atsitiktinis. Tiesinis iteratorius skirtas judėti per konteinerį tik vieną kryptimi nuo vieno elemento prie kito vykdant operaciją ++. Skirtingai nuo jo dvikryptis iteratorius leidžia judėti dviem kryptim ir atitinkamai vykdyti ir operaciją ++ ir operaciją --. Atsitiktinis iteratorius leidžia judėti laisvai, t.y. nurodyti bet kurį konteinerio elementą ir eiti abejomis kryptimis.

Paprastai iteratoriai surišami su konteineriais, pvz., *vec.begin()* ir *vec.end()* gražina objektus tipo *vector<int>::iterator* , t.y. atitinkamai metodas *begin* gražina iteratorių, kuris nurodo pirmą konteinerio elementą, o *end* – paskutinį. Iteratoriai padeda atskirti algoritmus nuo konteinerių.

Algoritmai – tai šabloninės funkcijos, parametrizuotos iteratoriaus klase. Pvz., elemento paieškos konteineryje algoritmo apibrėžimas galėtų atrodyti taip:

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
        while (first !=last  && *first !=value) ++first;
        return first;
}
```

Čia pirmais dviem parametrais užduodamas intervalas ir trečiu - jame ieškoma reikšmė. Šabloninės funkcijos parametras – tai *InputIterator* tipas, vietoj jo, generuojant konkrečią šablono realizaciją, ir bus panaudotas konkretus tipas.

Koncepcijos.

Svarbu numatyti kokie tipai gali būti šablonų parametrais. Pvz., minėtoje šabloninėje funkcijoje *find* gali būti panaudotas tipas **double*, bet negali būti panaudotas tipas *double*. Galimiems šablonų tipams apibrėžti STL bibliotekoje naudojama **koncepcijos** sąvoka. Jei tam tikras tipas remiasi (modeliuoja) koncepciją *InputIterator*, jame turi būti apibrėžtas tam tikras metodų ir operatorių rinkinys, pvz., toks tipas turi realizuoti minėtus ++ ir * operatorius.

Koncepciją galima patikslinti. Pvz., funkcijai *reverse()* reikalingas ne tik ++, bet ir -- operatorius. Reiškia funkcijos parametras turi modeliuoti koncepciją *BidirectionalIterator*. Ši koncepcija yra koncepcijos *InputIterator* patikslinimas, nes ji turi viską, ką turi *InputIterator* ir dar operatorių --.

STL BIBLIOTEKOS TAIKYMAS.

Vektoriai. Vektorius – tai konteineris, kuriame prieiga prie elementų įgyvendinama pagal indeksą. Vektorius praplečia standartinio masyvo galimybes ir yra saugesnis, nes jį galima papildyti (arba sumažinti) naujais elementais ir kontroliuoti indeksų diapazoną. Konteinerio klasė *vector* apibrėžta vardų srities *std* antraštiniame faile *<vector>:*

```
template <class T, class A =allocator<T>>
class std::vector
{
//tipų, iteratorių, metodų, konstruktorių, lyginimo operatorių ir kt.
//apibrėžimas
}
```

Pirmas argumentas – vektoriaus elementų tipas, antras (nebūtinas) – apibrėžia klasę, atsakingą už atminties skirstymą vektoriaus elementams.

Pagal nutylėjimą, atmintis skirstoma globaliais operatoriais *new()* ir *delete()*, klasė-šablonas *allocate* taip pat turi konstruktorių, destruktorių ir tipų apibrėžimą (pvz., rodyklė ir nuoroda į vektoriaus elementus, vektoriaus dydžio tipas ir t.t.). Naujam vektoriui sukurti iškviečiamas klasės T konstruktorius. Jei naudojami standartiniai C++ tipai, vektorių galima apibrėžti taip:

vector <int> vek_sveik;
vector <float> vek_real;

Jei žinomas elementų skaičius, galima rezervuoti atmintį iš karto:

vector <int> vek_sveik(100);

Sužinoti elementų skaičių galima naudojant funkciją-narę *size()*:

size_type size() const;

o pakeisti vektoriaus dydį funkcija *resize()*:

void resize(size_type newsize);

Nauji elementai pridedami į vektoriaus galą (jei dabartinis dydis < *newsize*), mažinamas vektorius taip pat nuo galo (jei dabartinis dydis > *newsize*). Kreiptis į vektoriaus elementus galima pagal indeksą, elementai numeruojami nuo 0, pvz., *vek_sveik[0], vek_sveik[25]*. Klasėje *vector* taip pat apibrėžtos visos standartinės lyginimo operacijos.
Kitos dažnai naudojamos funkcijos:

void pop_back()	Šalina paskutinį vektoriaus elementą
void push_back(const &T x)	Prideda elementą į vektoriaus galą; elementų-objektų klasė turi turėti kopijavimo konstruktorių
bool empty() const	Patikrina ar konteineris tuščias (jei taip - true)
void clear()	Pašalina visus elementus
front()	Nuoroda į pirmą elementą
back()	Nuoroda į paskutinį elementą
at()	Saugus analogas indeksavimo operatoriaus [ ]
insert()	Įterpia vieną ar daugiau elementų, pradedant nurodyta pozicija.

Pavyzdys 3.27. Konteineriaus <vector> panaudojimas.

```
#include <iostream>
```

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

int main()
 {
      vector<int> v;              //sudaromas nulinio ilgio vektorius
      int i;
      for (i = 0; i<5; i++)
            v.push_back(i);       // reikšmės įrašomos į vektorių
      cout<<"Vektoriaus dydis v="<<v.size() << endl;
      for (i=0; i<v.size(); i++)
            cout<< v[ i ]<<endl; //išvedamas vektoriaus turinys
      return 0;
 }
```

Klasėje *vector* apibrėžti iteratoriai, kurie atlieka rodyklės vaidmenį ir leidžia atlikti greitą navigaciją tarp elementų. Šių iteratorių tipas yra *iterator*, kuriame apibrėžtos inkrementinės adresinės operacijos ir reikšmės gavimo operacija * . Taip pat naudojamos funkcijos

iterator begin(); ir *iterator end();*

kurių reikšmės – tai rodyklės į pirmą ir sekantį už paskutinio elementus. Įterpimo funkcija įterpia elementą *x* prieš elementą, nurodyta *position* parametru:

iterator insert(iterator position, const T& x);

Atvirkštinis iteratorius turi tipą *reverse_iterator* ir atlieka navigaciją priešinga kryptimi, nuo vektoriaus galo, pvz., funkcija

reverse_iterator rend();

gražins rodyklę į pirmą vektoriaus elementą.

Pavyzdys 3.28. Darbas su vektoriumi naudojant iteratorius.

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

int main()
 {
```

```
    vector<int> v;              //sudaromas nulinio ilgio vektorius
    int i;
    for (i = 0; i<5; i++)
        v.push_back(i);         //reikšmės įrašomos į vektorių
    cout<<“Vektoriaus dydis v=”<<v.size() << endl;

//Išvedamas vektoriaus turinys taikant iteratorius prieigai prie elementų
    vector<int>::iterator p= v.begin();
    while (p!=v.end())
    {
        cout<< *p << endl; //Išveda vektoriaus turinį
        p++;
    }
    return 0;
}
```

Šių programų (3.27 ir 3.28) vykdymo rezultatai yra vienodi.
Pateiksime dar vieną STL vektorių ir iteratorių panaudojimo pavyzdį – rūšiavimo algoritmo realizaciją.

Pavyzdys 3.29. Rūšiavimas naudojant vektorių ir iteratorius.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>

using namespace std;

int main()
{
   vector<int> record;
   vector<int>::const_iterator iter;
   int j;
   srand(static_cast<unsigned int>(time(0)));

   cout<<"\t\t Rikiavimo algoritmo realizacija naudojant STL vektorius\n\n";
   cout<<"Iveskite rikiuojamo duomenu masyvo dydi: ";
   cin>>j;

cout<<"\n Rikiuojami duomenys: ";
   for (int i=0;i<j;i++){
```

```
        int random=rand()%100;
        record.push_back(random);
        cout<<random<<" ";
        }
        cout<<"\n\n";
   cout<<"\tVykdomas Algoritmas\n\n";
   for(int i=0;i<record.size();i++){
        int band=i+1;
        int lyg=1;
       for(int j=0;j<record.size()-1;j++){
          if(record[j]>record[j+1]) {
              int temp=record[j];
              record[j]=record[j+1];
              record[j+1]=temp;
       }
      else if(record[j]<record[j+1]) lyg++;
      }
      if(lyg==record.size()) break;
      cout<<"Bandymas "<<band<<":  ";
      for(iter=record.begin(); iter!=record.end(); iter++) {cout<<*iter<<" ";}
      cout<<"\n\n" ;
   }
   getchar(); getchar();
   return 0;
}
```

Iš tikrųjų, pagal savo prigimtį iteratoriai daugiau skirti (ir dažniau naudojami) darbui su tokiomis duomenų struktūromis, kurios apibrėžiamos naudojant rodykles. STL bibliotekoje – tai tokie nuoseklūs konteineriai kaip sąrašas (*list*) ir dėkas (*deque*).