Obyektyönlü proqramlaşdırma

C++

ШАГ
компьютерная
АКАДЕМИЯ

Dərs №15

C++ dili ilə obyektyönlü proqramlaşdırma

Mündəricat

C++ stilində tiplərin çevrilməsi

Başlanğıc üçün qeyd edək ki, tip çevirmə operatoru bir verilənlər tipini digərinə çevirməyə imkan verir. C proqramlaşdırma dili kimi C++ proqramlaşdırma dili də tip çevirməsinin bu formasını dəstəkləyir:

```
(tip) ifadə

Məsələn,

double d;
d=(double) 10/3;
```

C++ dilində tip çevirmə operatorları

C stilindəki standart formalardan başqa, C++ əlavə tip çevirmə operatorlarını dəstəkləyir:

const_cast

```
sinfin obyektin_adı.üzvün_adı;
```

dynamic_cast

```
sinfin obyektin_adı.üzvün_adı;
```

reinterpret_cast

```
sinfin obyektin_adı.üzvün_adı;
```

static_cast

```
sinfin obyektin_adı.üzvün_adı;
```

Verilmiş konstruksiyanı daha ətraflı gözdən keçirək:

■■**const_cast** const və/və ya volatile modifikatorlarının aşkar elan edilməsi üçün istifadə edilir. const və ya volatile atributlarının dəyişməsi xaric yeni tip cari tip ilə eyni olmalıdır. Məsələn:

Qeyd: Xatırlamaq lazımdır ki, yalnız const_cast operatoru "sabitliliyə vəd verməkdən" xilas edə bilər, yəni, bu qrupun qalan heç bir operatoru obyektdən const atributunu ala bilməz.

```
#include <iostream>
using namespace std;
//obyektin göstəricisi sabitdir, buna görə
//də onun vasitəsilə obyekti dəyişdirmək olmaz

void test_pow(const int* v){
    int*temp;
    //const modifikatorunu çıxardırıq
    //və indi obyekti dəyişdirə bilərik
    temp=const_cast<int*>(v);
    //obyektin dəyişdirilməsi

    *temp= *v * *v;
}

void main(){
    int x=10;
```



```
//ekrana – 10
    cout<<"Before – "<<x<<"\n\n";
    test_pow(&x);
    //ekrana – 100
    cout<<"After – "<<x<<"\n\n";
}
```

Qeyd: Yeri gəlmişkən, **volatile** nə olduğunu qeyd etmək lazımdır!!!. Bu modifikator dəyişənin qiymətinin qeyri-aşkar dəyişdirilə bildiyini kompilyatorun başa düşməsi imkanını verir. Məsələn, nə isə bir qlobal dəyişən var, onun ünvanı əməliyyat sisteminin qurulmuş taymerinə ötürülür. Gələcəkdə bu konstruksiya real vaxtın hesablanmasında istifadə oluna bilər. Təbii ki, bu dəyişənin qiyməti mənimsətmə operatoru olmadan avtomatik dəyişir. Belə dəyişən volatile açar sözünün istifadəsi ilə elan edilməlidir. Bu onunla əlaqədardır ki, bir çox kompiyatorlar hansısa bir mənimsətmə olmadan dəyişənin heç bir dəyişməsini təsbit etmədən, hesab edir ki, o dəyişmir və ona konkret qiymət verməklə onun ifadəsini optimallaşdırır. Bu zaman dəyişəni yoxlamır. Həqiqətən, əgər dəyişən “dəyişməz” xüsusiyyətindədirsə, bu onun nəyinə lazımdır?! volatile — kompilyatorun bu cür işini əngəlləyəcək.

■■**dynamic_cast verilmiş** tip çevirmə əməliyyatının qanunauyğunluğunu yoxlayır. Əgər belə əməliyyatı yerinə yetirmək mümükün deyilsə, onda ifadə sıfra bərabər edilir. Bu operator əsasən polimorf tiplər üçün istifadə edilir.

Məsələn, əgər iki B və D polimorf siniflər verilmiş olarsa, eyni zamanda D sinfi B sinfindən törəyərsa, onda **dynamic_cast** həmişə D* göstəricisini B* göstəricisinə o vaxt çevirə bilər ki, ünvanlşdırılan obyekt D olsun. Ümumiyyətlə, **dynamic_cast** operatoru o zaman müvəffəqiyyətlə icra olunar ki, tiplərin polimorf çevrilməsinə icazə verilmiş olsun (yəni, bu əməliyyatın baş verdiyi obyekt tipinə yeni tipin qanuni tətbiq edilməsi mümkün olarsa). Bütün yuxarıda deyilənləri nümayiş etdirən nümunəyə baxaq:

```
#include <iostream>
using namespace std;
//baza kursu
class B{
    public:
    //əjdadda növbəti təyin etmə
    //üçün virtual funksiya
     virtual void Test(){
        cout<<"Test B\n\n";
    }
};

//əjdad-sinif
class D:public B{
    public:
    //virtual funksiyanın təyin edilməsi
    void Test(){
        cout<<"Test D\n\n";
    }
};
```



```
void main(){
    //valideyn-sinif göstəricisi
    //valideyn-sinif obyekti
    B *ptr_b, obj_b;
    //əjdad-sinif göstəricisi və əjdad-sinif obyekti
    D *ptr_d, obj_d;
    //obyektin (D*) ünvanını D* tipli göstəriciyə
    //çeviririk
    ptr_d= dynamic_cast<D*> (&obj_d);
    //əgər hər şey müvəffəqiyyətlə baş
    //verərsə,!0 qaytarılır
    //cevrilmə baş verdi
    if(ptr_d){
        cout<<"Good work – ";
        //burada əjdad-sinfin çağırılması
        //ekrada Test
        Test D ptr_d->Test();
    }
    //əgər səhv baş vermişsə, 0 qaytarıldı
    else cout<<"Error work!!!\n\n";

    //(D*) obyektinin ünvanını B* tipində
    //göstəriciyə çeviririk
    ptr_b= dynamic_cast<B*> (&obj_d);
    //əgər hər şey müvəffəqiyyətlə baş verərsə,
    //!0 qaytarılır
    //çevirilmə baş verdi
    if(ptr_b){
        cout<<"Good work – ";
      //Burada əjdad-sinfin funksiyasının çağırılması
        //ekranda – Test
        D ptr_b->Test();
}

    //əgər səhv baş vermişsə, 0 qaytarılıb
    else cout<<"Error work!!!\n\n";
```

```
//(D*)obyektiniin ünvanını B* tipli göstəriciyə
    //çeviririk
    ptr_b= dynamic_cast<B*> (&obj_d); //əgər hər şey
    //müvəffəqiyyətlə baş verərsə, !0 qaytarıldı
    //çevirilmə baş verdi
    if(ptr_b){
        cout<<"Good work – ";
        //burada əjdad-sinfin funksiyası çağırılır
        //Ekranda – Test
        D ptr_b->Test();
    }
    //əgər səhv baş vermişsə, o qaytarıldı
    else cout<<"Error work!!!\n\n";

    //(B*) obyektinin ünvanını B* tipində
    //göstəriciyə çeviririk
    ptr_b= dynamic_cast<B*>(&obj_b); //əgər hər şey
    //müvəffəqiyyətlə baş vermiş olarsa, !0 çağırıldı
    //çevirilmə baş verdi
    if(ptr_b){
        cout<<"Good work – ";
        //burada əjdad-sinfin çağırılması
        //ekranda – Test
        B ptr_b->Test();
    }
    //əgər səhv baş vermişsə, 0 qaytarıldı
    else cout<<"Error work!!!\n\n";

    //DİQQƏT!!! BU MÜMKÜN DEYİL
    //(B*) obyektinin ünvanını D* tipli
    //göstəriciyə çevirmə cəhdi
    ptr_d= dynamic_cast<D*> (&obj_b); //əgər hər şey
    //müvəffəqiyyətlə baş verərsə - !0 qaytarılır
    //çevirilmə baş verdi
    if(ptr_d)
        cout<<"Good work – ";
    //əgər səhv baş vermişsə - 0 qaytarıldı
    cout<<"Error work!!!\n\n";}
```


Proqramın işinin nəticəsi:

```
Good work – Test D
Good work – Test D
Good work – Test B
Error work!!!
```

■■ **static_cast** poliforf olmayan tip çevirilməsini icra edir. Onu istənilən standart çevirilmələr üçün istifadə etmək olar. Bu halda proqramın icrası zamanı heç bir yoxlama baş vermir. Faktik olaraq, static_cast — bu C üslubunda standart çevirmə əməliyyatının analoqudur. Məsələn, növbəti şəkildə:

```
#include <iostream>
using namespace std;
void main(){
    int i;
    for(i=0;i<10;i++)
        //i dəyişəninin double tipinə çevirilməsi
     //bölmənin nəticəsi ekranda, təbiiki həqiqi tip

        cout<<static_cast<double>(i)/3<<"\t";
}
```

■■**reinterpret_cast** bir tipi tamamilə başqasına çevirir. Məsələn, onu göstəricini tam tipə və əksinə çevirmək üçün istifadə etmək olar. reinterpret_cast operatorunu öz təbiəti etibari ilə uyğun olmayan göstəricilər tipini çevirmək üçün istifadə edə bilərsiniz. Nümunəyə baxaq:

```
#include <iostream>
using namespace std;
void main(){
```

```
    //tamtipli dəyişən
    int x;
    //sətir (char tipində göstərici)
    char*str="This is string!!!";
    //sətri ekranda nümayiş etdirirk
    cout<<str<<"\n\n"; //ekranda – This is string!!!
    //char tipini ədədə çeviririk
    x=reinterpret_cast<int>(str);
    //nəticəni nümayiş etdiririk
    cout<<x<<"\n\n"; //ekranda – 4286208
}
```

Beləliklə, biz C++ üslübunda çevrilmə ilə tanış olduq. Birincisi, çevrilmə C-nin hətda ehtimal etmədiyi işləri görə bilir: const modifikatorunu çıxartmaq və ya verilənlər tipini radikal dəyişdirmək. İkincisi, C++ üslubu çevirilmə üzərində nəzarəti artırır və onlarla bağlı səhvləri ilkin mərhələdə təsbit edir.

Standart şablonlar kitabxanası (STL)

Əsas anlayışlar

(konteyner, iterator, alqoritm, funktor, predikat, allokator)

Hazırda C++ kompilyatorlarının demək olar ki, hamısı xüsusi əlavə edilmiş STL kitabxanası ehtiva edir. Verilmiş kitabxana tez-tez istifadə edilən alqoritmlərin reallaşdırılması üçün siniflər və funksiyalar dəsti ehtiva edir. Belə ki, kitabxana müxtəlif verilənlər tipləri ilə işləmək üçün nəzərdə tutulduğu üçün onda olan bütün siniflər və funksiyalar şablondur. Bu gözəl vasitənin ətraflı gösdən keçirilməsi ilə biz yaxın zamanda məşğul olacağıq.

STL kitabxanasını gözdən keçirmək üçün onun əsaslandığı əsas anlayışlarla tanış olmalıyıq. Beləliklə, başlayaq.

Bizim kitabxanamız onun daxili strukturunu təşkil edən dörd komponent ehtiva edir:

- **Konteyner** — verilənlərin saxlanılması, onların idarə edilməsi və yerləşdirilməsi üçün blok. Başqa sözlə, bu başqa elementlərin saxlanılması və istifadəsi üçün nəzərdə tutulmuş obyektdir.
- **Alqoritm** — konteynerde saxlanılan verilənlər ilə işləmək üçün xüsusi funksiya.

- **İterator** — alqoritmlərə konkret konteynerdə verilənlərə müraciət imkanı verən xüsusi göstəricidir.
- **Funktorlar** — konkret obyektdə funksiyanın onu başqa komponentlərlə istifadə etmək üçün inkapsulyasiya mexanizmidir.

Yuxarıda göstərilən konstruksiyalardan başqa STL kitabxanası daha bir neçə əlavə edilmiş komponentləri dəstəkləyir:

- **Allokator** — yaddaş ayırıcisi. Bu cür yaddaş ayırıcısı hər bir konkret konteynerdə var. Bu konstruksiya sadəcə olaraq konteyner üçün yaddaşın ayırılması prosesini idarə edir. Qeyd etmək lazımdır ki, susmaya görə yadaş ayırıcısı allocator sinfinin obyektidir. Belə ki, özümüzün xüsusi yaddaş ayırıcımızı təyin edə bilərik.
- **Predikat** — konteynerdə istifadə edilən standart olmayan funksiya. Predikat unar və binar olur. Məntiqi qiymət (doğru və ya yanlış) qaytara bilər.

Dərsin bu hissəsində biz STL-ə aid yalnız əsas təyinlərə baxdıq. Sonra biz bu mexanizmlərin hər birini ayrılıqda analiz edəcəyik.

auto_ptr sinfi

C++ proqramlaşdırma dilində növbəti fikirdə üzə çıxan anlayış var — **"resursun ayırılması — bu qiymətləndirmədi (inisializasiyadır)"**. Bu fikir istənilən tip resurs üçün mütləq istifadə edilir. Bu proqramın işini, proqramın icrası zamanı istisna meydana gəldiyi halda daha da etibarlı edir.

Məsələn, proqram bir faylı açır və yaddaş ayırır. İşin sonunda bu proqram təbii ki, faylı bağlamalı və ya yaddaşı azad etməlidir. Lakin, yuxarıda verilmiş əməliyyatlardan biri yerinə yetirilməzdən əvvəl istisna hal meydana gələ bilər ki, nəticədə də əməliyyat yerinə yetirilməz.

```
void f(){
    FILE*f;

    if (!(f = fopen("test.txt", "rt")))
    {
        //faylı açmaq mümükün olmadı - çıxırıq
        exit(0);
    }
    //alındı, faylla işləyirik, lakin burada istisna
    //meydana gələ bilər və növbəti sətrə biz
    //çatmarıq, uyğun olaraq da fayl bağlanmayacaq


    fclose(f);
}
```

Bu problemin həlli növbəti əməliyyatlar ardıcıllığıdır:

1. Sinif yaratmaq.
2. Faylın açılacağı konstruktor reallaşdırmaq.
3. Faylın bağlanacağı konstruktor reallaşdırmaq.
4. Faylla işə başlamaq üçün bu sinfin obyektini yaratmaq lazımdır (fayl konstruktorda avtomatik açılacaq).

İndi isə əsas.

Proqramın icrası zamanı istisna hal meydana gəldiyində obyekt təbii ki, ləğv ediləcəkdir, lakin bu zaman onun üçün onun destruktoru çağırılacaq ki, bu da faylı bağlayacaq (yaddaşı boşaldacaq). Əgər proqram icrası düzgün tamamlanarsa, onda bu halda sizin faylın aşkar bağlanması haqqında düşünməyinizə ehtiyac yoxdur, belə ki, yaradılmış obyekt görünmə oblastından çıxdıqdan sonra avtomatik ləğv ediləcək və fayl yenə də sinfin destruktoru vasitəsilə bağlanacaq.

```
class FileOpen
{
    FILE* f;

public:

    FileOpen(char* filename, char* mode)
    {
        if (!(f = fopen(filename, mode)))
        {
```



```
            exit(0);
        }
    }
    ~FileOpen(){
        fclose(f);
    }
};

void f(){
    FileOpen MyFile("test.txt", "r+");
    //burada faylla lazım olan işi yerinə yetiririk
}
```

Bizim prinsipin həyata keçirilməsi üçün biz **auto_ptr** sinfindən (automatic pointer — avtomatik-göstərici) istifadə edə bilərik. Bu sinif standart C++ kitabxanası tərəfindən təqdim edilir və adətən aşkar silinmək lazım gələn obyektlər ilə iş üçün nəzərdə tutulub (məsələn, new operatoru vasitəsilə dinamik yaradılan obyektlər).

auto_ptr sinfinin obyektinin yaradılması üçün konstruktorun parametri dinamik yaradılan obyekt göstəricisi olmalıdır. Sonra **auto_ptr** ilə, cari göstəricinin göstərdiyi eyni bir dinamik obyekti göstərən, demək olar ki, adi göstərici kimi işləmək olar. Biz obyektin aşkar ləğv edilməsi haqqında düşünməliyik, o **auto_ptr** sinfinin destruktoru vasitəsilə avtomatik silinəcək.

Standart kitabxanada **auto_ptr** sinfinin sintaksisi növbəti şəkildədir:

```
template<class X>
class Std::auto_ptr
{
```

```
X* ptr;
public:

    //konstruktor və destruktor
    explicit auto_ptr(X* p = 0)throw()
    {
        ptr = p;
    }

    ~auto_ptr() throw()
    {
        delete ptr;
    }

//Adlandırma operatoru obyekt əldə etməyə imkan verir
     X& operator*()const throw() {

        return *ptr;
    }

// -> operatoru göstərici əldə etməyə imkan veririr
    X* operator->()const throw()
    {
        return ptr;
    }
};
```

Qeyd: -> və * operatorlarının təyinindən başqa, auto_ptr sinfi iki üzv-funksiya ehtiva edir: **X* get () const**; — X şablonu altında saxlanılan sinfin obyektinin göstəricisini qaytarır.

X* release () const; — X şablonu altında saxlanılan, sinfin obyektinin göstəricisini qaytarır, lakin bu zaman auto_ptr-dən bu obyektə müraciət haqqını alır. DİQQƏT!!! Obyektin özü ləğv eilmir!!!

İndi, tamamilə təbiidir ki, auto_ptr-in ümumi kostruksiyasına baxaraq onun işdə necə göründüyü maraqlıdır:

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

class TEMP{
    public:
    TEMP(){
        cout<<"TEMP\n\n";
    }
    ~TEMP(){
        cout<<"~TEMP\n\n";
    }
    void TEST(){
        cout<<"TEST\n\n";
    }
};
void main(){
    //iki avtomatik göstərici təyin edirik
    //onlardan biri üçün TEMP tipində yaddaş ayırırıq

    auto_ptr<TEMP>ptr1(new TEMP), ptr2;

    //mənsubolma haqqının ötürülməsi
    ptr2=ptr1;
    //funsiyanın avtomatik göstərici vasitəsilə
    //çağırılması
    ptr2->TEST();

    //avtomatik göstəricinin sinfin obyektinin adi

    //göstəricisinə mənimsədilməsi

    TEMP*ptr=ptr2.get();

    //Funksiyanın adi göstərici vasitəsilə
    //çağırılması
    ptr->TEST();
}
```

string sinfinin analizi və istifadəsi

İndi isə biz STL kitabxanasının siniflərinə ətraflı baxmağa başlayaq. Başlanğıc üçün string (sətir) sinfinə baxaq. Bu sinif simvollarla işləmək üçün metodlar tədim edir və ixtiyari uzunluqda sətri dəstəkləyən simvollar massivini təmsil edir.

string sinfinin ən çox istifadə edilən funksiyaları:

- **operator[]** — yazmaq və oxumaq üçün sətirdə müəyyən simvola müraciət.
- **c_str()** — string ilə işləyə bilməyən funksiyaların istifadə edilməsi üçün sətrin const char*-a çevrilməsi.
- **append** — sətrin sonuna simvolların əlavə edilməsi.
- **operator=** — sətrə digər sətirlərin, simvol massivlərini, hətta ədədlərin mənimsədilməsi.
- **insert** — string tipində dəyişənə simvolların və ya sətirlərin yerləşdirilməsi.
- **erase** — verilmiş sətrin verilmiş mövqeyindən başlayaraq bir və ya bir neçə simvolun silinməsi.
- **replace** — Verilmiş mövqedə bir və ya bir neçə simvolun silinməsi.
- **length** (və **size**) — sətirdəki simvolların sayınnın qaytarılması.

- **empty** — sətirdə simvolların olmasının təyini.
- **find** — verilmiş sətirdə simvolun və ya alt sətrin ilk rast gəlindiyi mövqeyinin tapılması.
- **rfind** — find funksiyasının analoqu, lakin axtarış sətrin sonundan əvvələ doğrudur.
- **find_first_of** — verilmiş sətirdə simvollar dəstindən olan hər hasnı bir simvolun ilk rast gəlindiyi mövqenin tapılması.
- **substr** — altsətri qaytarır.
- **find_first_not_of** — verilmiş dəstə aid olmayan sətirdə birinci simvolun tapılması.
- **compare** — sətirlərin müqayisəsi (həmçinin!=, <, > operatorlarını dəstəkləyir)

string sinfi ən çox istifadə edilən siniflərdəndir. Gəlin onunla işə aid nümunəyə baxaq:

```
#include <stdio.h>
#include <stdlib.h>
#include <string>
using namespace std;

void main()
{//string tipli obyektdə simvollar
//sətrini mənimsətmək
    string s = "Hello world";
    //Sətirdəki birinci simvolu əldə etmək
    int nWordEnd = s.find(' ');
    string sub_string = s.substr(0,nWordEnd);
   //nəticəni vermək
    printf("String: %s\n", s.c_str());
    printf("Sub String: %s\n", sub_string.c_str());
}
```

Nümunənin şərhi

- Siz string obyektinə birbaşa simvollar sətrini mənimsədə bilərsiniz. Bu verilənlər sətrinin istifadə edilməsini sadələşdirir.
- find metodu simvolun sətirdə ilk rast gəlmə mövqeyini tapır və tapılmış simvolun mövqeyini (0-dan başlayaraq) qaytarır və ya simvol sətrə daxil deyilsə, -1 qaytarır.
- Substr metodu sətrin metodunun birinci parametrində verilmiş hissəsinin, ikinci parametrdə verilmiş uzunluqda nüsxəsini qaytarır. Əgər ikinci arqument buraxılarsa, onda cari sətrin verilmiş mövqeyindən sonuna qədər simvollar sətri qaytarılacaq.
- Sətrlərin ekrana simvollar massivi kimi verilməsi üçün printf funksiyasından istifadə edilir. c_str() metodu string obyektini simvollar massivinə çevirmək üçün isitifadə edilir.

İteratorun ətraflı analizi

Beləliklə yaxından tanış olma vaxtı gəldi.

İterator — ümumi təyinatlı özünəməxsus göstəricidir.

Əslində sadə göstəricilər müxtəlif verilənlərlə universial üsulla işləməyə imkan verən iteratorların xüsusi halıdır. İstənilən funksiya parametrlər kimi iteratorları qəbul edərək onların emalı zamanı bu iteratorların göstərdikləri verilənlərin tipinə diqqət eləmir. İteratorlar göstəricilərin massivin elementlərinə müraciət etdikləri kimi konteynerin məzmununa müraciət etməyə imkan verirlər. İteratorların istifadəsi üçün <iterator> kitabxanasını qoşmaq lazımdır.

İteratorların tiplərinə və onların hər birinin təyinatına baxaq:

İteratorların tipi

- ***Giriş*** (*input*) — ünvalaşdırılan verilənlərin oxunmasına xidmət edir. Bərabərlik, uyğunlaşdırma və inkrement operatorlarını dəstəkləyir: ==, !=, *i, ++i, i++, *i++.
- ***Çıxış*** (*output*) — verilənlərin yazılacağı obyektləri ünvanlaşdırır. Mənimsətmənin yalnız sol tərəfinə uyğunlaşdırmaya imkan verir və inkrement operatorlarını dəstəkləyir: ++i, i++, *i = t, *i++ = t.

- ***Biristiqamətli*** (*forward*) — giriş və çıxışın bütün xüsusuyyətlərinə malikdir, həmçinin ünvanlaşdırılan verilənlərin sonuna qədər yerdəyişdirə bilir. Giriş/çıxış iteratorlarının bütün əməliyyatlarını dəstəkləyir, bundan başqa, mənimsətməni məhdudiyyətsiz istifadə etməyə imkan verir: ==, !=, =, *i, ++i, i++, *i++.
- ***İkiistiqamətli*** (*bidirectional*) — biristiqamətlilərin xassələrinə malikdir, lakin verilənlər zəncirində istənilən istiqamətdə: həm irəli, həm də geriyə yerdəyişmə etməyə imkan verir. Əlavə dekrement əməliyyatına malikdir (--i, i--, *i--).
- ***İxtiyari müraciətli iteratorlar*** (*random access*) — bütün digər iteratoların funksiyalarına sahibdirlər. Müqayisə və ünvan, yəni indeksə dolayı müraciət əməliyyatlarını dəstəkləyirlər. i += n, i + n, i –= n, i — n, i1 — i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2.

Qeyd: Yeri gəlmişkən, C++ dilində göstəricilər həmçinin ixtiyari müraciətli iteratorlardır.

Diqqətinizə çatdırmaq istəyirik ki, yuxarıda göstərilən təsnifləndirmə qəti deyildir. Bir qayda olaraq, daha geniş xassələrə malik iteratorları daha aşağı səviyyəli iteratorların yerinə istifadə etmək olar. Məsələn, giriş iteratorunun yerinə birbaşa iteratordan istifadə edə bilərsiniz.

Bundan başqa, STL kitabxanası əks iteratorlar da (*reverse iterator*) ehtiva edir. Bu cür iteratorlar ya ikistiqamətli olurlar, ya da ixtiyari müraciətli iteratoru təmsil edirlər.

İstənilən təzahürdə konteyner boyunca əks istiqamətdə yerdəyişmə xüsusiyyətinə sahib olmalıdırlar. Yəni, əgər bu cür iterator hansısa bir ardıcıllığın sonunu göstərirsə, onda onun uzanması əvvəlki elementin mövqeləşməsinə gətirib çıxarır.

Və nəhayət, əsas təsnifləndirməyə daxil olmayan daha iki iterator vardır, bunlar məhz:

Axın iteratorlar – axında dolaşmağa imkan verən iteratorlar.

Daxiletmə iteratoru — elementlərin müəyyən konteynerə yerləşdirilməsini sadələşdirən iteratorlar.

Yuxarıda deyilənlərdən aydın göründüyü kimi, iteratorlar aləmi kifayət qədər genişdir. STL-n öyrənilməsi prosesində biz sizinlə tamamilə müxtəlif standart siniflər üçün bütün təsnifatın iteratorları ilə tanış olacağıq. Lakin bütün iteratorların kifayət qədər ümumi xüsiyyətləri vardır və məsələ yalnız onların təyinində deyil, həmçinin onlarla işlədir. İteratorlarla işləmək üçün iki funksiyaya baxaq:

advance() и distance()

```
template <class InIter, class Dist>
    void advance(InIter &itr, Dist d);
template <class InIter> ptrdiff_t
                distance(InIter &start, InIter end);
```

Qeyd: ptrdiff_t — iki iteratorun fərqini təmsil etməyə imkan verən verilənlər tipi.

advance() funksiyası itr iteratorunu d qədər artırır.

distance() funksiyası *start* və *end* iteratorları arasında yerləşən elementlərin sayını qaytarır.

Bu iki funksiya vacibdirlər, belə ki, yalnız ixtiyari müraciətli itertorlar iteratorlara əlavə etməyə və onlardan hansısa bir qiyməti çıxmağa imkan verirlər. **advance()** və **distance()** funksiyaları bu məhdudiyyətlərdən yan keçməyə imkan verirlər. Lakin qeyd etmək lazımdır ki, bəzi iteratorlar bu funksiyaların səmərəli reallaşdırılmasını təmin etmirlər.

Beləliklə, biz STL kitabxanasının daha bir ayrılmaz elementini gödən keçirdik. Yekunlaşdıraq. İteratorlar — ölçüsüz konstruksiyanın ayr-ayrı elementlərinə müraciəti həyata keçirməyə imkan verən mexanizmdir. Bu konsturksiya rolunda bizin hazırda və gələcəkdə öyrənəcəyimiz konteyner-siniflər çıxış edirlər.

Ev tapşırıqları

1. Riyazi ifadələr (məsələn, (2+3)*4+1 formasında) ehtiva edən sətirlərin analizi üçün string sinfi istifadə edən proqram yazın. Sətir klaviaturadan daxil edilir. Proqram ifadənin hesablanmasının nəticəsini verir.