Тема вежбе: Туторијал – C++ STL

БИБЛИОТЕКА СТАНДАРДНИХ ШАБЛОНА (STANDARD TEMPLATE LIBRARY)

Једно од каснијих проширења стандарда језика C++ је библиотека стандардних шаблона (STL – Standard Template Library). STL је скуп апстрактних типова података, функција и алгоритама пројектованих да подрже типове података дефинисане од стране корисника. Сваки апстрактни тип такође садржи скуп функција и преклопљених оператора прилагођених за приступ конкретном типу податка. Идеја је да се имплементирају алгоритми и структуре података независне од конкретно употребљеног типа податка. На пример, апстрактни тип ***vector*** може се употребити за складиштење низа променљивих било ког типа података. Коришћење стандардних шаблона такође доноси и аутоматско заузимање потребне меморије за податке, као и механизме заштите од приступа ван опсега структуре.

Употреба стандардних шаблона упрошћава дизајн пројекта, у смислу да нема потребе за посебном имплементацијом структура као што су низови, редови, листе и слично. Осим тога, садржи и моћне алгоритме независне од типа података, укључујући ту сортирање и претраге. Те структуре уносе релативно мало успорења приликом извршавања, док са друге стране њихова употреба најчешће значајно смањује број грешака у програмском коду.

Иако логички није део библиотеке стандардних шаблона (јер није шаблон), у њу је укључена и ***string*** класа. Ова класа доноси знатна олакшања у раду са словним низовима.

STL Vector Class – шаблонска класа вектора

Једна од основних класа из STL-а је ***vector*** класа. У основи, вектор је низ променљиве величине, са омогућеним директним приступом преко оператора [].

Мора се обратити пажња да у оператор [], као и код приступа обичним низовима, не садржи контролу опсега. Међутим, вектор класа поседује методу ***at()***, која има ову контролу (али уз смањену брзину приступа). У наставку је дата листа неких од функција садржаних у класи вектор:

```
unsigned int size();          број елемената структуре
push_back(type element);      додаје елемент на крај вектора
bool empty();                 провера да ли је вектор празан
void clear();                 брише све елементе вектора
type at(int n);               враћа елемент на позицији n, са провером опсега
```

Такође, ту је и листа неколико основних оператора класе вектор:

=	Додела замењује елементе вектора са елементима другог вектора

==	Поређење два вектора елемент по елемент
[]	Приступ елементу вектора као код класичног низа, без провере опсега

Напомена: Пошто је STL део стандардног C++ окружења, информације о класама (атрибути, методе, детаљи око имплементације, итд.) могу се наћи у документацији која прати Visual Studio окружење. Приступање тој помоћној документацији (тзв. „Help“) је брзо и једноставно: притиском на тастер F1. Корисна ствар коју окружење подржава је могућност да у свом програмском коду обележите део текста (име функције, променљиве, резервисане речи, итд.) у вези са којим желите сазнати више, и онда притиском на F1 ће се отворити део документације која се бави управо обележеним појмом. За вежбу, пронађите у помоћној документацији информације о вектор класи. Употреба помоћне документације у баратању са STL-ом (али и иначе) се изузетно препоручује. Немојте се устручавати да јој често приступате у потрази за одговорима.

Дат је једноставан пример употребе шаблон класе вектор.

```
#include <iostream>
#include <vector>

using namespace std;
int main()
{
    vector<int> example;          //Вектор за целобројни тип података
    example.push_back(3);         //Додај 3 на крају
    example.push_back(10);        //Додај 10 на крају
    example.push_back(33);        //Додај 33 на крају
    for (int x = 0; x < example.size(); x++)
    {
        cout << example[x] << " ";    //Испис на екрану: 3 10 33
    }
    if (!example.empty())          //Провера да ли није празан
        example.clear();           //Брише садржај
    vector<int> another_vector;    //Још један вектор целобројног типа
    another_vector.push_back(10);  //Додаје 10 на крај
    example.push_back(10);         //
    if (example == another_vector) //Провера једнакости два вектора
    {
        example.push_back(20);
    }
    for (int y = 0; y < example.size(); y++)
    {
        cout << example[y] << " ";    //Испис на екрану: 10 20
    }
    return 0;
}
```

Итератори библиотеке стандардних шаблона

Концепт итератора је фундаментални концепт STL-а, јер омогућава једнак начин приступа појединачним елементима различитих структура података као што су низови, листе, вектори, пресликавања, и тако даље.

Свака структура има дефинисану функцију ***begin()***, која враћа итератор који показује на први елемент те структуре, као и функцију ***end()***, која враћа итератор еквивалентан итератору који је дошао на крај структуре (обратите пажњу да то **није** итератор који показује на крај структуре, у смислу да показује на последњи, тј. крајњи, елемент структуре, већ се може замислити као да показује „иза“ последњег елемента структуре). Елементу структуре се приступа преко дереференцирања итератора, као да је у питању обично показивач.

Итератор неке класе шаблона се дефинише на следећи начин:

```
std::class_name<template_parameters>::iterator name
```

где ***name*** представља име итератора, ***class_name*** је име шаблон класе за коју дефинишемо итератор (нпр. ***vector***), а ***template_parameters*** су параметри шаблона коришћени при дефиницији објекта са којем ће приступати итератор (нпр ***int***).

Пример дефиниције вектора и итератора за класу вектор над целобројним типом:

```
std::vector<int> myIntVector;
std::vector<int>::iterator myIntVectorIterator;
```

Постоји више различитих класа итератора, свака са незнатно другачијим својствима. Основна разлика је да ли се итератор користи за читање или упис елемената. Неки итератори дозвољавају обе операције, али не у исто време. Неки од најважнијих типова итератора су: за приступ елементима од првог ка последњем (forward), од последњег ка првом (backward) и двосмерни. За кретање напред може се користити оператор ++. Аналогно, за кретање у назад може се користити оператор --.

Укратко, за приступ елементима неке STL структуре, уместо класичног C++ начина програмирања, најбоље је користити итераторе. Позивом ***begin()*** функције добија се итератор, употребом ++ оператора креће се кроз структуру, * се користи за приступ елементу, а крај итерација се детектује провером једнакости са итератором који враћа функција ***end()***.

За испис елемената вектора могла би се написати оваква функција:

```
using namespace std;

vector<int> intVect;
unsigned int i;

intVect.push_back(1);
intVect.push_back(4);
intVect.push_back(8);

for (i = 0; i < intVest.size(); i++)
{
    cout << intVect[i] << " ";
    //исписује на екран: 1 4 8
}
```

Иста та фукција би са употребом итератора изгледала овако:

```
using namespace std;

vector<int> intVect;
vector<int>::iterator intVectIt;

intVect.push_back(1);
intVect.push_back(4);
intVect.push_back(8);

for (intVectIt = intVect.begin(); intVectIt != intVect.end(); intVectIt++)
{
    cout << *intVectIt << " ";
    //исписује на екран: 1 4 8
}
```

У објекат класе вектор могуће је и уметнути елемент на одређеној позицији. То се остварује на следећи начин:

```
intVect.insert(intVectIt, 5);
```

У том смислу, метода ***push_back*** је еквиалентна са:

```
intVect.insert(intVect.end(), 5);
```

Пошто се елементи вектора налазе редом један до другог у меморији, као код обичног низа, треба имати у виду да додавање елемента било где осим на крај вектора подразумева извршење додатног кода ради прерасподеле података у меморији.

Итератори су посебно погодни када је потребно дефинисати одређени опсег над којим се жели радити. Итератори, који раде над структурама које омогућавају директан приступ елементима као што су вектори, имају могућност померања за одређени број елемената у структури. Ова операција изводи се једноставним сабирањем итератора и константе. Резултат је итератор померен за константу у односу на тренутни елемент.

Напомена: Са овим треба бити пажљив јер се може прекорачити опсег структуре.

Тако на пример ако желимо да обришемо елементе из вектора, помоћу итератора можемо задати опсег елемената које желимо избрисати. Сваки елемент између два итератора ће бити обрисан. Брисање свих елемената је приказано у следећем примеру:

```
vector<int> myIntVector;
myIntVector.erase(myIntVector.begin(), myIntVector.end());
```

Ако желимо да обришемо само прва два елемента, то можемо урадити на следећи начин:

```
myIntVector.erase(myIntVector.begin(), myIntVector.begin() + 2);
```

STL List Class – шаблонска класа листе

Класа листе у STL-у представља шаблонску изведбу двоструко спрегнуте листе. Декларација листе целобројних елемената изгледа овако:

```
std::list<int> intList;
```

Класа листе поседује методу ***push_back***, као и вектор, али поседује и ***push_front***, с обзиром да је у питању двоструко спрегнута листа.

Питање: Зашто вектор нема методу ***push_front***? (Одговор је имплицитно дат у наредном пасусу)

Класе вектора и листе се са становишта спреге према кориснику врло мало разликују (основна разлика је то што листа нема могућност директног приступа произвољном елементу, док се код вектора то постиже индексирањем). Њихова изведба, међутим, значајно је другачија. То се пре свега одражава на брзину извршења појединих акција над објектима тих класа. Тако на пример, иако се елементи у објекте обе класе умећу на исти начин:

```
intList.insert(intListIt, 5);
intVect.insert(intVectIt, 5);
```

брзина којом ће се те две акције извршити може се разликовати поприлично. Логично, уметање у листу је далеко брже, јер при уметању елемента у вектор, сви елементи на позицијама после позиције уметања се морају померити. Са друге стране, величина листе и вектора се проверава методом идентичног назива:

```
intList.size();
intVect.size();
```

али се провера величине листе обавља побројавањем њених елемената, док се код вектора своди на једноставну акцију. Због тога, на пример, није најповољније проверавати да ли је листа празна на следећи начин:

```
if (intList.size() == 0)
```

Уместо тога, боље је користити методу ***empty***, која ту проверу обавља далеко ефикасније.

```
if (intList.empty())
```

STL листа поседује још неколико корисних метода, као што су ***unique***, ***reverse***, и тако даље. Погледајте у помоћној документацији шта оне раде.

Испис свих елемената листе може се урадити на следећи начин:

```
using namespace std;

list<int> intList;
list<int>::iterator intListIt;

intList.push_back(1);
intList.push_back(4);
intList.push_back(8);

for (intListIt = intList.begin(); intListIt != intList.end(); intListIt++)
{
    cout << *intListIt << " ";
    //исписује на екран: 1 4 8
}
```

Приметите да је разлика између исписа елемената вектора и листе коришћењем итератора само у класи која је декларисана. Због тога се, што се тиче неке основне функционалности, може лако прећи са употребе вектора на листу, или обрнуто; уколико се јави потреба за тим. Овде се јасније увиђа предност рада са итераторима. Међутим, предност није толико у томе што се може једноставније прећи са употребе једне класе на

другу (то је случај са ове две класе, али не важи и за остале класе из STL-а, јер се њихова употреба логички знато разликује). Предност је у томе што су одређени поступци у својој форми еквивалентни и једном научени поступак је једноставно применљив и на остале класе из STL-а.

STL Map Class – шаблонска класа мапе

Под појмом мапа, овде се подразумева асоцијативни низ, то јест, ради се о пресликавању (енг. mapping, па отуд и назив „мапа“) једног скупа елемената у други.

Претпоставимо да радимо са подацима који за сваку вредност имају придружен знаковни низ, односно стринг. На пример, уз свако име студента придружимо његову оцену, или број индекса. Како бисмо овакву структуру описали језиком C++? Један начин би био да самостално израдимо неку врсту табеле (или пак хеш, енг. ***hash***, табеле). Тај приступ подразумева и писање сопствене функције за претраживање табеле, бригу о граничним ситуацијама као и много испитивања како би били сигурни да све ради како треба. Са друге стране, библиотека стандардних шаблона већ садржи класу која одговара управо овом случају: класу ***map***.

Питање: Шта је хеш табела?

Мапе нам омогућавају да асоцирамо било која два типа података. За дефиницију променљиве типа класе мапе, потребно је задати два, односно три типа података:

```
std::map<key_type, data_type, [comparison_function]>
```

Уочавамо да је функција поређења (***comparison_function***) у угластим заградама, што значи да овај параметар није обавезан докле год класа која дефинише кључ (***key_type***) има дефинисан оператор „мање од“, тј. <. Оператор < обезбеђује да елементи у мапи буду поређани по реду. То значи да ће и приступ елементима помоћу итератора бити по редоследу кључева.

У следећем примеру показано је чување података о студентским оценама:

```
std::map<std::string, int> grade_list;
```

Сада се за приступ студентским оценама структура може третирати као низ и користити оператор []. Једина разлика је што се сада за индекс низа користи тип кључа при дефиницији структуре.

Придруживање оцене 9 студенту „Djura“ ради се на следећи начин:

```
grade_list["Djura"] = 9;
```

Ако је у међувремену студент поправио оцену, измена се врши на идентичан начин:

```
grade_list["Djura"] = 9;
// Ђура се поправио
grade_list["Djura"] = 10;
```

Са друге стране, брисање елемента се обавља позивом функције ***erase()*** која припада класи мапе.

Брисање студента „Djura" обавиће се следећом линијом:

```
grade_list.erase("Djura");
```

Следи још неколико корисних метода класе мапа. Број елемената добија се позивом функције ***size()***, која је такође припадник класе мапа. Још једна корисна функција је ***empty()*** и користи се за проверу да ли је структура мапе празна. Брисање свих елемената обавља се позивом функције ***clear()***. Приметите да и листа и вектор имају ове методе.

Уколико је потребно проверити присутност неког елемента у структури, употреба оператора [] није препоручљива, јер нисмо сигурни који податак ће бити враћен ако елемент не постоји. У том случају користи се функција ***find()*** која враћа итератор траженог елемента уколико он постоји, или исти итератор који враћа функција ***end()***. уколико тај елемент не постоји. Следећи пример показује употребу функције ***find()***:

```
std::map<std::string, int> grade_list;
grade_list["Djura"] = 10;
if (grade_list.find("Pera") == grade_list.end())
{
    std::cout << "Pera se ne nalazi u mapi!" << endl;
}
```

За приступ елементима се такође могу користити итератори као што је већ било показано. Елементу на који показује итератор класе мапа се приступа на мало другачији начин. У овом случају итератор показује на пар – кључ и упарену вредност. Овим елементима итератора се приступа преко променљивих ***first*** и ***second***. Пошто се итератор третира као показивач, овим елементима се приступа преко оператора ->, што се види из следећег примера:

```
std::map<std::string, int> grade_list;
grade_list["Djura"] = 10;
// Исписаће "Djura"
std::cout << grade_list.begin()->first << endl;
// Исписаће "10"
std::cout << grade_list.begin()->second << endl;
```

Када је у питању брзина употребе класе мапа, треба имати на уму да је трајање уноса новог елемента у структуру или претрага постојећег сразмерна log(n), где је n тренутни број елемената у структури. Овакав приступ је потенцијално знатно спорији од употребе еквивалентне хеш табеле са брзом хеш функцијом. Међутим, за разлику од хеш табле елементи у мапи су сортирани по кључу.

STL string class – класа знаковног низа

Класа знаковног низа (илити стринг; енг. **string**) замишљена је да унапреди рад са знаковним низовима тако што ће учинити рад са њима сличан раду са основним типовима података. Следи низ елементарних акција које се могу обавити над објектима класе знаковног низа, дате у паралели са истим тим акцијама обављеним над променљивом целобројног типа.

```
using namespace std;

int i = 5;
string s = "pet";

...

cin >> i;
cin >> s;

...

cout << i;
cout << s;

...

if (i == 7) ...
if (s == "sedam") ...

...

i += 7;
s += "sedam"; // шта је резултат?

...

int i1, i2;
i = i1 + i2 + 8;
string s1, s2;
s = s1 + s2 + "osam";
```

Знаковни низ има много истих метода као и остали елементи STL-а. Поседује методе ***size*** (али има и методу ***length*** која ради исту ствар), ***push_back***, ***begin***, ***end***, ***erase***, ***insert*** и тако даље. Његови елементи (знакови) могу се индексирати и оператором [] и функцијом ***at***. Поседује итераторе, и кроз њега се може итеритати као и кроз остале

елементе STL-а. Има и методу ***find***, која проналази поднизове, али и још неколико варијанти методе find са додатним могућностима и условима. Из знаковног низа се могу вадити поднизови методом ***substr***.

```
using namespace std;

string str = "C:\\windows\\winhelp.exe";

int pos = str.find_last_of("/\\");

string path = str.substr(0, pos); //шта садржи променљива path?

string file = str.substr(pos + 1, str.size()); //шта садржи променљива file?
```

У случају да је потребно на неком месту употребити знаковни низ у форми показивача типа **char** на почетну адресу у меморији (како се класично представља знаковни низ у C-у), ту је метода ***c_str*** која враћа ***char****. Но, треба увек имати на уму да то није конверзија типа и да се тако добијени „цеовски“ знаковни низ не може мењати.

Вежбање

У прилогу се налази пројекат са једном изворном датотеком. Отворите га и крените редом да попуњавате делове који су означени са TODO.