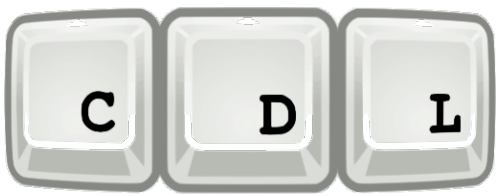
Template-uri si STL

CDL - Cursul 6



Adrian Scoica

adrian.scoica@gmail.com

15 aprilie 2010

ROSEdu

Ce este un template?

Pe scurt, un template class (Ro: clasa sablon) este un mecanism care permite rezolvarea unei probleme generale implementand o singura solutie, care este valabila pentru toate tipurile de date.

Exemplu concret

Vrem sa implementam o clasa care:

Exemplu concret

Vrem sa implementam o clasa care:

- Contine un membru privat de tip intreg

Exemplu concret

Vrem sa implementam o clasa care:

- Contine un membru privat de tip intreg
- Are un constructor care initializeaza acel membru

Exemplu concret

Vrem sa implementam o clasa care:

- Contine un membru privat de tip intreg
- Are un constructor care initializeaza acel membru
- Are o metoda care afiseaza acel membru la iesirea standard

Exemplu concret

Vrem sa implementam o clasa care:

- Contine un membru privat de tip intreg
- Are un constructor care initializeaza acel membru
- Are o metoda care afiseaza acel membru la iesirea standard

```
1  # ifndef __SIMPLEINT_H__
2  # define __SIMPLEINT_H__
3
4  class SimpleInt{
5          private:
6                  int data;
7          public:
8                  SimpleInt(int);
9                  void display();
10 };
11
12 # endif
```

It may work, BUT...

Ce facem daca vrem o clasa similara si pt char, float, short int, unsigned int, etc...?

It may work, BUT...

Ce facem daca vrem o clasa similara si pt char, float, short int, unsigned int, etc...?

- Rescriem inutil codul si facem putine modificari

It may work, BUT...

Ce facem daca vrem o clasa similara si pt char, float, short int, unsigned int, etc...?

- Rescriem inutil codul si facem putine modificari
- Cum ramane cu tipurile definite de utilizator?

It may work, BUT...

Ce facem daca vrem o clasa similara si pt char, float, short int, unsigned int, etc...?

- Rescriem inutil codul si facem putine modificari
- Cum ramane cu tipurile definite de utilizator?
- De cate ori trebuie modificam codul daca am depistat un bug?

It may work, BUT...

Ce facem daca vrem o clasa similara si pt char, float, short int, unsigned int, etc...?

- Rescriem inutil codul si facem putine modificari
- Cum ramane cu tipurile definite de utilizator?
- De cate ori trebuie modificam codul daca am depistat un bug?

$\Rightarrow$ ... it's WRONG!

Solutia - folosim un template

Avantaje:

Solutia - folosim un template

Avantaje:

- Implementam o singura data - efort redus :)

Solutia - folosim un template

Avantaje:

- Implementam o singura data - efort redus :)
- Scapam de sute de "cast"-uri ambigue (hint: void*)

Solutia - folosim un template

Avantaje:

- Implementam o singura data - efort redus :)
- Scapam de sute de "cast"-uri ambigue (hint: void*)
- Merge cu orice tip de date (simplu sau definit de utilizator)

Solutia - folosim un template

Avantaje:

- Implementam o singura data - efort redus :)
- Scapam de sute de "cast"-uri ambigue (hint: void*)
- Merge cu orice tip de date (simplu sau definit de utilizator)
- Codul se intretine foarte usor.

Solutia - folosim un template

Avantaje:

- Implementam o singura data - efort redus :)
- Scapam de sute de "cast"-uri ambigue (hint: void*)
- Merge cu orice tip de date (simplu sau definit de utilizator)
- Codul se intretine foarte usor.

Dezavantaje:

Solutia - folosim un template

Avantaje:

- Implementam o singura data - efort redus :)
- Scapam de sute de "cast"-uri ambigue (hint: void*)
- Merge cu orice tip de date (simplu sau definit de utilizator)
- Codul se intretine foarte usor.

Dezavantaje:

- Compileaza [mai] lent.

Solutia - folosim un template

Avantaje:

- Implementam o singura data - efort redus :)
- Scapam de sute de "cast"-uri ambigue (hint: void*)
- Merge cu orice tip de date (simplu sau definit de utilizator)
- Codul se intretine foarte usor.

Dezavantaje:

- Compileaza [mai] lent.
- Code bloating (more on that later)

Sintaxa unui template

```
1  template<class T>
2  class SimpleClass{
3          private:
4                  T data;
5          public:
6                  SimpleClass(T);
7                  void display();
8  };
9
10 template<class T>
11 SimpleClass<T>::SimpleClass(T data) : data(data){
12 }
13
14 template<class T>
15 void SimpleClass<T>::display(){
16         std::cout << data << "\n";
17 }
18
```

Exemplu de instantiere

```
 1 # include <iostream>
 2 # include <string>
 3
 4 # include "SimpleClass.cpp"
 5
 6 using namespace std;
 7
 8 int main()
 9 {
10         SimpleClass<int> unInt(5);
11         SimpleClass<float> unFloat(12.345);
12         SimpleClass<string> unString("Ana are mere.");
13
14         unInt.display();
15         unFloat.display();
16         unString.display();
17
18         return 0;
19 }
20
```

Exercitiu:

Scrieti un template pentru o clasa Echipa care contine:

- Doua variabile membru de tipuri neprecizate <T> si <V>
- O metoda de tip "void display()" care sa afiseze continutul celor doua variabile membru sub forma (a , b)
- Definiti o structura Persoana cu doi membri de tip std::string :
  - nume
  - prenume
- Instantiati un obiect de tip Echipa<Persoana,Persoana>

ATENTIE! Atunci cand folositi tipuri de date definite de utilizator, va trebui sa definiti modul in care functioneaza asupra lor operatorii folositi in template.

Solutie: Template-ul

```
1  using namespace std;
2
3  template<class T, class V>
4  class Echipa{
5          private:
6                  T Tdata;
7                  V Vdata;
8          public:
9                  Echipa(T,V);
10                 void display();
11 };
12
13 template<class T, class V>
14 Echipa<T,V>::Echipa(T Tdata, V Vdata)
15                         : Tdata(Tdata), Vdata(Vdata){}
16
17 template<class T, class V>
18 void Echipa<T,V>::display(){
19         cout << "( " << Tdata << " , " << Vdata << " )\n";
20 }
21
```

Solutie: Definirea tipului compatibil cu Template-ul

```
 1 # include <iostream>
 2 # include "Echipa.cpp"
 3
 4 using namespace std;
 5
 6 struct Persoana{
 7         string nume, prenume;
 8         Persoana(string nume, string prenume):
 9                                 nume(nume), prenume(prenume){}
10 };
11
12 ostream& operator<< (ostream& output, Persoana& persoana){
13         output << persoana.nume << " " << persoana.prenume;
14         return output;
15 }
16
17 int main(){
18         Echipa<Persoana,Persoana>
team(Persoana("Chuck","Norris"),Persoana("Mr.","McGyver"));
19         team.display();
20         return 0;
21 }
```

Ce este STL?

STL (Standard Template Library) este o biblioteca(!) de template-uri foarte utile, a caror folosire creste siguranta si viteza dezvoltarii de programe.

Ce este STL?

STL (Standard Template Library) este o biblioteca(!) de template-uri foarte utile, a caror folosire creste siguranta si viteza dezvoltarii de programe.
Din biblioteca STL vom mentiona doua tipuri de template-uri:

Ce este STL?

STL (Standard Template Library) este o biblioteca(!) de template-uri foarte utile, a caror folosire creste siguranta si viteza dezvoltarii de programe.
Din biblioteca STL vom mentiona doua tipuri de template-uri:

- Containere
- Algoritmi

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale.

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale. Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale.
Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.
Avantaje:

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale.
Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.
Avantaje:

- Au in general implementari eficiente

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale.
Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.
Avantaje:

- Au in general implementari eficiente
- Gestiunea memoriei este sigura

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale.
Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.
Avantaje:

- Au in general implementari eficiente
- Gestiunea memoriei este sigura (well, sort of...)

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale.
Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.
Avantaje:

- Au in general implementari eficiente
- Gestiunea memoriei este sigura (well, sort of...)
- Pun la dispozitie multe metode gata implementate

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale.
Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.
Avantaje:

- Au in general implementari eficiente
- Gestiunea memoriei este sigura (well, sort of...)
- Pun la dispozitie multe metode gata implementate

Dezavantaje:

Containerele din STL

Containerele sunt implementari ale unor structuri de date foarte uzuale. Pentru ca o structura de date sa se comporte asemeni unui container, este foarte important ca datele sa fie opace.
Avantaje:

- Au in general implementari eficiente
- Gestiunea memoriei este sigura (well, sort of...)
- Pun la dispozitie multe metode gata implementate

Dezavantaje:

- Set relativ limitat de structuri de date (desi suficient pentru aplicatii practice)
- Adesea sunt mai lente decat implementarile specializate

Principalele structuri de date din STL

Containere:

- vector - Array-uri
- list - Liste dublu inlantuite
- slist - Liste simplu inlantuite
- deque - Double-ended queue
- set - Multimi (in sens matematic)

Adaptori pentru containere:

- queue - Structura FIFO (coada)
- stack - Structuri LIFO
- priority_queue - Coada de prioritati

Toate template-urile sunt definite in namespace-ul "std" si au fisiere-antet de forma <vector>, <stack>, etc

Metode generale comune

Metode generale comune

- push_front / pop_front (nu si pentru "vector")
- push_back / pop_back

Metode generale comune

- push_front / pop_front (nu si pentru "vector")
- push_back / pop_back
- empty / size / clear
- front / back (referinta la primul / ultimul element)

Metode generale comune

- push_front / pop_front (nu si pentru "vector")
- push_back / pop_back
- empty / size / clear
- front / back (referinta la primul / ultimul element)

Pentru "vector" si "deque" sunt definite si metode de acces prin subscripting:

- [] - Acces fara verificarea limitelor
- at - Acces cu verificarea limitelor

Metode generale comune

- push_front / pop_front (nu si pentru "vector")
- push_back / pop_back
- empty / size / clear
- front / back (referinta la primul / ultimul element)

Pentru "vector" si "deque" sunt definite si metode de acces prin subscripting:

- [] - Acces fara verificarea limitelor
- at - Acces cu verificarea limitelor

De ce [] nu verifica limitele?

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi
- Cum sa treci la urmatorul element

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi
- Cum sa treci la urmatorul element
- Unde sa te opresti

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi
- Cum sa treci la urmatorul element
- Unde sa te opresti

Important: valabil INDIFERENT de modul in care e organizat intern containerul.

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi
- Cum sa treci la urmatorul element
- Unde sa te opresti

Important: valabil INDIFERENT de modul in care e organizat intern containerul.
Solutia: iteratori
Iteratorii sunt implementati sa se comporte precum niste pointeri la date.

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi
- Cum sa treci la urmatorul element
- Unde sa te opresti

Important: valabil INDIFERENT de modul in care e organizat intern containerul.
Solutia: iteratori
Iteratorii sunt implementati sa se comporte precum niste pointeri la date.
Exista de doua tipuri:

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi
- Cum sa treci la urmatorul element
- Unde sa te opresti

Important: valabil INDIFERENT de modul in care e organizat intern containerul.
Solutia: iteratori
Iteratorii sunt implementati sa se comporte precum niste pointeri la date.
Exista de doua tipuri:

- iterator - permite sa modifici datele pe masura ce parcurgi

Parcurgerea elementelor unui container

La nivel filozofic, pentru a parcurge elementele unui container ar fi suficient sa stii:

- Unde sa incepi
- Cum sa treci la urmatorul element
- Unde sa te opresti

Important: valabil INDIFERENT de modul in care e organizat intern containerul.
Solutia: iteratori
Iteratorii sunt implementati sa se comporte precum niste pointeri la date.
Exista de doua tipuri:

- iterator - permite sa modifici datele pe masura ce parcurgi
- const_iterator - nu da voie sa modifici datele

Cum se folosesc iteratorii?

In primul rand, containerele trebuie sa puna la dispozitie:

Cum se folosesc iteratorii?

In primul rand, containerele trebuie sa puna la dispozitie:

- Tipul iterator / const_iterator

Cum se folosesc iteratorii?

In primul rand, containerele trebuie sa puna la dispozitie:

- Tipul iterator / const_iterator
- begin() - primul element (exista)

Cum se folosesc iteratorii?

In primul rand, containerele trebuie sa puna la dispozitie:

- Tipul iterator / const_iterator
- begin() - primul element (exista)
- end() - sfarsitul (dupa ultimul element - NU exista!!)

Cum se folosesc iteratorii?

In primul rand, containerele trebuie sa puna la dispozitie:

- Tipul iterator / const_iterator
- begin() - primul element (exista)
- end() - sfarsitul (dupa ultimul element - NU exista!!)
- Optional:
  - reverse_iterator / reverse_const_iterator
  - rbegin() / rend()
  - rend() - sfarsitul (inainte de primul elem - NU exista!!)

Exemplu de parcurgere a unui vector

```
 1 # include <iostream>
 2 # include <vector>
 3
 4 using namespace std;
 5
 6 int main()
 7 {
 8         vector<int> v;
 9         for (int i=1; i<= 10; i++)
10                 v.push_back(i);
11         vector<int>::iterator it;
12         for (it = v.begin(); it != v.end(); ++it)
13                 cout << *it << " ";
14         cout << "\n";
15         return 0;
16 }
```

Exemplu de parcurgere a unui vector

```
 1 # include <iostream>
 2 # include <vector>
 3
 4 using namespace std;
 5
 6 int main()
 7 {
 8         vector<int> v;
 9         for (int i=1; i<= 10; i++)
10                 v.push_back(i);
11         vector<int>::iterator it;
12         for (it = v.begin(); it != v.end(); ++it)
13                 cout << *it << " ";
14         cout << "\n";
15         return 0;
16 }
```

Ce trebuie sa schimbam pentru o lista?

Exemplu de parcurgere a unui vector

```
 1 # include <iostream>
 2 # include <vector>
 3
 4 using namespace std;
 5
 6 int main()
 7 {
 8         vector<int> v;
 9         for (int i=1; i<= 10; i++)
10                 v.push_back(i);
11         vector<int>::iterator it;
12         for (it = v.begin(); it != v.end(); ++it)
13                 cout << *it << " ";
14         cout << "\n";
15         return 0;
16 }
```

Ce trebuie sa schimbam pentru o lista? Dar pentru o multime?

Practice

1. Scrieti un program simplu care sa adauge intr-o multime(= set) restul impartirii primelor 100 numere la 15
2. Afisati in ordine inversa acele numere din multime divizibile la 3.

Solutie:

```
 1 # include <iostream>
 2 # include <set>
 3
 4 using namespace std;
 5
 6 int main()
 7 {
 8         set<int> m;
 9         for (int i=1; i<= 100; i++)
10                 m.insert(i%15);
11         set<int>::const_reverse_iterator it;
12         for (it = m.rbegin(); it != m.rend(); ++it){
13                 if ( *it % 3 == 0)
14                         cout << *it << " ";
15         }
16         cout << "\n";
17         return 0;
18 }
```

Algoritmii din STL

Algoritmii din STL

- Sunt generici (nu depind de tipul de date)

Algoritmii din STL

- Sunt generici (nu depind de tipul de date)
- In general, implementati optim

Algoritmii din STL

- Sunt generici (nu depind de tipul de date)
- In general, implementati optim
- Again... mai lenti decat o implementare custom-made

Algoritmii din STL

- Sunt generici (nu depind de tipul de date)
- In general, implementati optim
- Again... mai lenti decat o implementare custom-made
- Sunt definiti in antetul <algorithm>

Exemplu: pentru a sorta un vector de elemente, este suficient sa stim cum sa comparam elementele intre ele.
Algoritmul in sine este acelasi.

Algoritmul sort

Algoritmul sort

- Este o implementare a lui Quick Sort ($O(n * log(n))$ mediu, dar $O(n^2)$ maxim)

Algoritmul sort

- Este o implementare a lui Quick Sort ($O(n * log(n))$ mediu, dar $O(n^2)$ maxim)
- Cea mai uzuala definitie este: sort(RandomAccessIterator, RandomAccessIterator)

Algoritmul sort

- Este o implementare a lui Quick Sort ($O(n * log(n))$ mediu, dar $O(n^2)$ maxim)
- Cea mai uzuala definitie este: sort(RandomAccessIterator, RandomAccessIterator)
- Nu uitati, exista si alte definitii (specificara criteriului de comparatie, de ex.)

Algoritmul sort

- Este o implementare a lui Quick Sort ($O(n * log(n))$ mediu, dar $O(n^2)$ maxim)
- Cea mai uzuala definitie este: sort(RandomAccessIterator, RandomAccessIterator)
- Nu uitati, exista si alte definitii (specificara criteriului de comparatie, de ex.)
- Implicit, se foloseste pt comparatie operatorul "$<$". Trebuie definit daca nu exista.

Exemplu de sortare

```
 1 # include <iostream>
 2 # include <algorithm>
 3 # include <vector>
 4
 5 using namespace std;
 6
 7 int main()
 8 {
 9         int junk[] = {1,102,13,24,2};
10         vector<int> v(junk, junk+sizeof(junk)/sizeof(int));
11         sort(v.begin()/*+2*/, v.end());
12         vector<int>::iterator it;
13         for (it = v.begin(); it != v.end(); ++it)
14                 cout << *it << " ";
15         cout << "\n";
16         return 0;
17 }
```

Sortarea dupa alte criterii

Am vazut ca daca vrem sa sortam structuri, trebuie sa supraincarcam operatorul "$<$".

Sortarea dupa alte criterii

Am vazut ca daca vrem sa sortam structuri, trebuie sa supraincarcam operatorul "$<$".
Ce se intampla daca vrem sa sortam intai crescator si apoi descrescator?

Sortarea dupa alte criterii

Am vazut ca daca vrem sa sortam structuri, trebuie sa supraincarcam operatorul "$<$".
Ce se intampla daca vrem sa sortam intai crescator si apoi descrescator?

Solutie: Folosim comparatori! Un comparator poate sa fie:

Sortarea dupa alte criterii

Am vazut ca daca vrem sa sortam structuri, trebuie sa supraincarcam operatorul "$<$".
Ce se intampla daca vrem sa sortam intai crescator si apoi descrescator?

Solutie: Folosim comparatori! Un comparator poate sa fie:

- Un pointer la o functie (C-style)
- Un function object (numit si Functor)

Sortarea dupa alte criterii

Am vazut ca daca vrem sa sortam structuri, trebuie sa supraincarcam operatorul "$<$".
Ce se intampla daca vrem sa sortam intai crescator si apoi descrescator?

Solutie: Folosim comparatori! Un comparator poate sa fie:

- Un pointer la o functie (C-style)
- Un function object (numit si Functor)

Observatie: un Functor este un obiect pentru care este definit operatorul "()".

Sortarea dupa alte criterii: exemplu

```
1  using namespace std;
2
3  struct CriteriuMaiMicInModul{
4    bool operator() (int a, int b) {
5      return abs(a)<abs(b);
6    }
7  };
8
9  int main()
10 {
11   int junk[] = {1,-102,13,-24,2};
12   vector<int> v(junk, junk+sizeof(junk)/sizeof(int));
13   vector<int>::iterator it;
14   sort(v.begin(), v.end());
15   for (it = v.begin(); it != v.end(); ++it) cout << *it << " ";
16   sort(v.begin(), v.end(), CriteriuMaiMicInModul());
17   for (it = v.begin(); it != v.end(); ++it) cout << *it << " ";
18   CriteriuMaiMicInModul comparator;
19   cout << comparator(-1,-7) << "\n";
20   cout << comparator(12,-100) << "\n";
21   return 0;
22 }
```

Cautarea unui obiect intr-o colectie

Pentru gasire, se poate folosi cel mai rapid algoritmul STL "find". Acest algoritm primeste ca parametru:

Cautarea unui obiect intr-o colectie

Pentru gasire, se poate folosi cel mai rapid algoritmul STL "find". Acest algoritm primeste ca parametru:

- Doi iteratori din colectie: first si last

Cautarea unui obiect intr-o colectie

Pentru gasire, se poate folosi cel mai rapid algoritmul STL "find". Acest algoritm primeste ca parametru:

- Doi iteratori din colectie: first si last
- Un obiect egal (NU identic!) cu obiectul cautat.

Cautarea unui obiect intr-o colectie

Pentru gasire, se poate folosi cel mai rapid algoritmul STL "find". Acest algoritm primeste ca parametru:

- Doi iteratori din colectie: first si last
- Un obiect egal (NU identic!) cu obiectul cautat.

Si returneaza un iterator:

Cautarea unui obiect intr-o colectie

Pentru gasire, se poate folosi cel mai rapid algoritmul STL "find". Acest algoritm primeste ca parametru:

- Doi iteratori din colectie: first si last
- Un obiect egal (NU identic!) cu obiectul cautat.

Si returneaza un iterator:

- Care indica la elementul cautat din colectie, daca a fost gasit un obiect egal.

Cautarea unui obiect intr-o colectie

Pentru gasire, se poate folosi cel mai rapid algoritmul STL "find". Acest algoritm primeste ca parametru:

- Doi iteratori din colectie: first si last
- Un obiect egal (NU identic!) cu obiectul cautat.

Si returneaza un iterator:

- Care indica la elementul cautat din colectie, daca a fost gasit un obiect egal.
- Egal cu "last" daca nu a fost gasit un obiect egal.

Cautarea unui obiect intr-o colectie

Pentru gasire, se poate folosi cel mai rapid algoritmul STL "find". Acest algoritm primeste ca parametru:

- Doi iteratori din colectie: first si last
- Un obiect egal (NU identic!) cu obiectul cautat.

Si returneaza un iterator:

- Care indica la elementul cautat din colectie, daca a fost gasit un obiect egal.
- Egal cu "last" daca nu a fost gasit un obiect egal.

Observatie: Cautarea se face in intervalul [first, last). Asta pentru ca in mod normal, "last" trebuie sa fie prima pozitie imediat de _dupa_ ultima pozitie din container.

Cautarea binara a unui element intr-o colectie

Observatie: Cautarea binara are sens doar pentru colectii sortate (hint: sort()).

Cautarea binara a unui element intr-o colectie

Observatie: Cautarea binara are sens doar pentru colectii sortate (hint: sort()). Exista mai multe variante ale cautarii binare:

Cautarea binara a unui element intr-o colectie

Observatie: Cautarea binara are sens doar pentru colectii sortate (hint: sort()). Exista mai multe variante ale cautarii binare:

- binary_seach(ForwardIterator first, ForwardIterator last, const T& value);

Cautarea binara a unui element intr-o colectie

Observatie: Cautarea binara are sens doar pentru colectii sortate (hint: sort()). Exista mai multe variante ale cautarii binare:

- binary_seach(ForwardIterator first, ForwardIterator last, const T& value);

Sau cu un criteriu de comparatie custom-made:

Cautarea binara a unui element intr-o colectie

Observatie: Cautarea binara are sens doar pentru colectii sortate (hint: sort()). Exista mai multe variante ale cautarii binare:

- binary_seach(ForwardIterator first, ForwardIterator last, const T& value);

Sau cu un criteriu de comparatie custom-made:

- binary_seach(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

Cautarea binara a unui element intr-o colectie

Observatie: Cautarea binara are sens doar pentru colectii sortate (hint: sort()). Exista mai multe variante ale cautarii binare:

- binary_seach(ForwardIterator first, ForwardIterator last, const T& value);

Sau cu un criteriu de comparatie custom-made:

- binary_seach(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

Observatie: tipul intors este "bool" (deci putem afla doar daca exista sau nu, nu putem avea acces la acel element gasit).

Alte template-uri de functii utile

- replace(ForwardIterator, ForwardIterator, const T&, const T&)
- replace_if(ForwardIterator, ForwardIterator, Predicate, const T&)
- count(ForwardIterator, ForwardIterator, const T&)
- swap(ForwardIterator, ForwardIterator)

etc...

Facts about <vector>

Acest template este probabil cel mai folosit template din STL.

Facts about <vector>

Acest template este probabil cel mai folosit template din STL.

- vector<bool> este o specializare pe biti $\Rightarrow$ eficienta timp / spatiu

Facts about <vector>

Acest template este probabil cel mai folosit template din STL.

- vector<bool> este o specializare pe biti $\Rightarrow$ eficienta timp / spatiu
- Folositi vector::swap pentru a interschimba doi vectori (Complexitate $O(1)$!)

Facts about <vector>

Acest template este probabil cel mai folosit template din STL.

- vector<bool> este o specializare pe biti $\Rightarrow$ eficienta timp / spatiu
- Folositi vector::swap pentru a interschimba doi vectori (Complexitate $O(1)$!)
- Daca adaugati la vector in timp ce il parcurgeti cu un iteratori, riscati *Segfault*. Posibila solutie: vector::reserve

Facts about container adapters

Desi la inceput exista impresia ca

- stack
- queue
- priority_queue

sunt containere, acest lucru este FALS!!

Facts about container adapters

Desi la inceput exista impresia ca

- stack
- queue
- priority_queue

sunt containere, acest lucru este FALS!! Ele sunt adaptori de containere ("container adapters"). Rolul lor este de a limita accesul la containerul de baza (ca un filtru)

Facts about container adapters

Desi la inceput exista impresia ca

- stack
- queue
- priority_queue

sunt containere, acest lucru este FALS!! Ele sunt adaptori de containere ("container adapters"). Rolul lor este de a limita accesul la containerul de baza (ca un filtru) De exemplu:

Facts about container adapters

Desi la inceput exista impresia ca

- stack
- queue
- priority_queue

sunt containere, acest lucru este FALS!! Ele sunt adaptori de containere ("container adapters"). Rolul lor este de a limita accesul la containerul de baza (ca un filtru) De exemplu:

- stack<int> - declaratie implicita. Se instantiaza un deque<int>! (+ adaptare)

Facts about container adapters

Desi la inceput exista impresia ca

- stack
- queue
- priority_queue

sunt containere, acest lucru este FALS!! Ele sunt adaptori de containere ("container adapters"). Rolul lor este de a limita accesul la containerul de baza (ca un filtru) De exemplu:

- stack<int> - declaratie implicita. Se instantiaza un deque<int>! (+ adaptare)
- stack< int, vector<int> > - declaratie explicita. Se instantiaza un vector<int>! (+ adaptare)

Observatie!!

Facts about container adapters

Desi la inceput exista impresia ca

- stack
- queue
- priority_queue

sunt containere, acest lucru este FALS!! Ele sunt adaptori de containere ("container adapters"). Rolul lor este de a limita accesul la containerul de baza (ca un filtru) De exemplu:

- stack<int> - declaratie implicita. Se instantiaza un deque<int>! (+ adaptare)
- stack< int, vector<int> > - declaratie explicita. Se instantiaza un vector<int>! (+ adaptare)

Observatie!!

- Atunci cand instantiati template-uri, este OBLIGATORIU sa puneti spatii intre simboluri < sau > succesive.

Facts about container adapters

Desi la inceput exista impresia ca

- stack
- queue
- priority_queue

sunt containere, acest lucru este FALS!! Ele sunt adaptori de containere ("container adapters"). Rolul lor este de a limita accesul la containerul de baza (ca un filtru) De exemplu:

- stack<int> - declaratie implicita. Se instantiaza un deque<int>! (+ adaptare)
- stack< int, vector<int> > - declaratie explicita. Se instantiaza un vector<int>! (+ adaptare)

Observatie!!

- Atunci cand instantiati template-uri, este OBLIGATORIU sa puneti spatii intre simboluri < sau > succesive.
- << si >> sunt operatori! $\Rightarrow$ eroare de compilare.

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

De exemplu, daca dorim sa implementam o structura de date care retine o versiune "arhivata" a datelor, suntem tentati sa derivam o specializare a unui container din STL. (Java-style)

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

De exemplu, daca dorim sa implementam o structura de date care retine o versiune "arhivata" a datelor, suntem tentati sa derivam o specializare a unui container din STL. (Java-style)
Desi posibil, acest lucru este in general RAU!!

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

De exemplu, daca dorim sa implementam o structura de date care retine o versiune "arhivata" a datelor, suntem tentati sa derivam o specializare a unui container din STL. (Java-style)
Desi posibil, acest lucru este in general RAU!!
Motivul?

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

De exemplu, daca dorim sa implementam o structura de date care retine o versiune "arhivata" a datelor, suntem tentati sa derivam o specializare a unui container din STL. (Java-style)
Desi posibil, acest lucru este in general RAU!!
Motivul? Destructorii containerelor STL NU sunt virtuali. (ce inseamna asta?)

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

De exemplu, daca dorim sa implementam o structura de date care retine o versiune "arhivata" a datelor, suntem tentati sa derivam o specializare a unui container din STL. (Java-style)
Desi posibil, acest lucru este in general RAU!!
Motivul? Destructorii containerelor STL NU sunt virtuali. (ce inseamna asta?)
Solutia corecta?

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

De exemplu, daca dorim sa implementam o structura de date care retine o versiune "arhivata" a datelor, suntem tentati sa derivam o specializare a unui container din STL. (Java-style)
Desi posibil, acest lucru este in general RAU!!
Motivul? Destructorii containerelor STL NU sunt virtuali. (ce inseamna asta?)
Solutia corecta? Facem un adaptor (un wrapper pentru containerul care ne intereseaza).

Template-urile si mostenirea

Este important atunci cand lucram cu template-urile sa retinem urmatoarele ecuatii:

$$\text{Template Class} + \text{Tipuri} = \text{Class}$$

$$\text{Class} + \text{"imbunatatiri"} = \text{Shortcut} >:)$$

De exemplu, daca dorim sa implementam o structura de date care retine o versiune "arhivata" a datelor, suntem tentati sa derivam o specializare a unui container din STL. (Java-style)
Desi posibil, acest lucru este in general RAU!!
Motivul? Destructorii containerelor STL NU sunt virtuali. (ce inseamna asta?)
Solutia corecta? Facem un adaptor (un wrapper pentru containerul care ne intereseaza).
P.S. Care e diferenta intre un destructor si o alta functie membru supradefinita?

O stiva care retine datele in format arhivat - part 1

```
 1 class ArchivedPairStack {
 2     private: //Containerul "wrapped"
 3         stack<char> s;
 4     public:  //Functii de acces
 5         void push(pair<int,int> p){
 6             if (p.first >= 16 || p.second >= 16 || p.first<0 ||
p.second<0)
 7                 return;
 8             char c = ((char)p.first<<4) | ((char)p.second);
 9             s.push(c);
10         }
11
12         pair<int,int> top(){
13             char sol = s.top();
14             return pair<int,int>(sol>>4, sol&0x0F);
15         }
16
17         bool empty(){ return s.empty(); }
18
19         void pop(){ s.pop(); }
20 };
```

O stiva care retine datele in format arhivat - part 2

```
1  ostream& operator<< (ostream& out, pair<int,int> p){
2          cout << "(" << p.first << "," << p.second << ")";
3          return out;
4  }
5
6  int main(){
7          ArchivedPairStack trail;
8          trail.push(pair<int,int>(1,2));
9          trail.push(pair<int,int>(4,2));
10         trail.push(pair<int,int>(23,1));
11         while (!trail.empty()){
12                 cout << trail.top() << " ";
13                 trail.pop();
14         }
15         cout << "\n";
16         return 0;
17 }
```

Concluzie

Cand folosim template-uri ?

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge
- Cand nu vrem/stim sa implementam de mana :D

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge
- Cand nu vrem/stim sa implementam de mana :D

Cand nu folosim template-uri?

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge
- Cand nu vrem/stim sa implementam de mana :D

Cand nu folosim template-uri?

- Cand putem scoate implementari mai bune daca tinem cont de tipul de date.

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge
- Cand nu vrem/stim sa implementam de mana :D

Cand nu folosim template-uri?

- Cand putem scoate implementari mai bune daca tinem cont de tipul de date. (exemple?)

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge
- Cand nu vrem/stim sa implementam de mana :D

Cand nu folosim template-uri?

- Cand putem scoate implementari mai bune daca tinem cont de tipul de date. (exemple?)
- Cand nu vom avea nevoie sa refolosim aceeasi clasa, dar pentru alt tip de date.

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge
- Cand nu vrem/stim sa implementam de mana :D

Cand nu folosim template-uri?

- Cand putem scoate implementari mai bune daca tinem cont de tipul de date. (exemple?)
- Cand nu vom avea nevoie sa refolosim aceeasi clasa, dar pentru alt tip de date.
- Cand vrem sa evitam "code bloating".

Concluzie

Cand folosim template-uri ?

- Cand avem de scris cod care nu tine cont de tipurile de date inglobate SI altfel am avea de rescris
- Cand vrem sa folosim cod care sigur merge
- Cand nu vrem/stim sa implementam de mana :D

Cand nu folosim template-uri?

- Cand putem scoate implementari mai bune daca tinem cont de tipul de date. (exemple?)
- Cand nu vom avea nevoie sa refolosim aceeasi clasa, dar pentru alt tip de date.
- Cand vrem sa evitam "code bloating".
- Cand proiectului nostru ii trebuiesc deja 12 minute sa compileze si folosirea unor template-uri nu se justifica in mod deosebit :D

Further Reading

Link-uri

- Sursa oficiala de documentatie pt C++
- Un tutorial de baza pentru STL
- Un tutorial ceva mai serios despre STL
- Site-ul oficial al bibliotecii BOOST

Carti

- "The C++ Programming Language - Special Edition", Bjarne Stroustrup
- "C++", Danny Kalev, Michael J. Tobler, Jan Walter