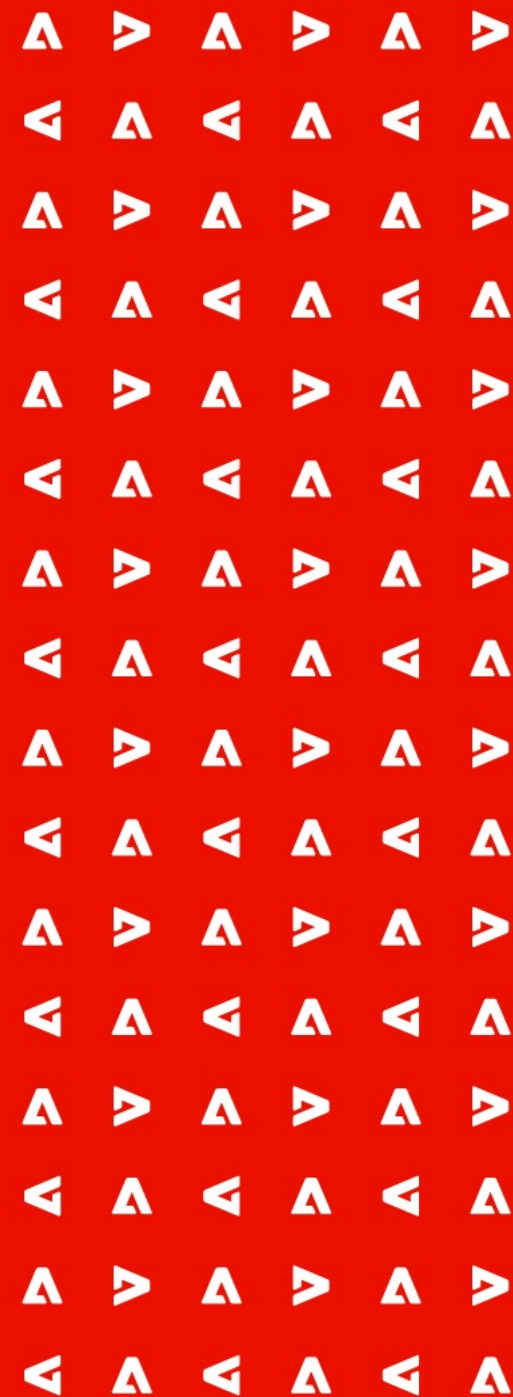




C++ DevCamp
Modulul 4 | Template-uri si
Standard Template Library



De ce?

- STL = Standard Template Library
- STL este o biblioteca generica de C++.
- STL ne ajuta sa dezvoltam rapid implementari scurte si eficiente.
- Tot codul din STL se compune din clase template.

Programarea generica (Generic programming)

- Programarea generica este o metoda de a adaptata implementarea mai multor tipuri de date fără a recurge la duplicarea codului.
- Este baza pentru a implementa algoritmi si structuri de date intr-un mod generic.
- Implementarea este scrisa folosind variabile ale căror tipuri nu sunt inițial cunoscute (care vor fi specificate mai tarziu).
- Programarea generica ne ajuta sa reducem duplicare codului si sa programam mai efficient.
- Codul este reutilizabil si flexibil.

Template-uri



Ce este un template?

- Template-urile sunt solutia in C++ pentru programarea generica
- Practic: o portiune de cod in care anumite tipuri de date sunt lasate intentionat "necompletate".
- Tipurile de date abstractizate se noteaza cu niste nume simbolice care vor fi inlocuite la **compilare** conform cu specializarile si instantierile din restul codului.
Acest process se numeste **instantiere**.
- Astfel, implementarea va putea fi adaptată mai multor tipuri de date fără a recurge la duplicarea codului.

Scurt exemplu

- Vrem o clasa care sa reprezinte o pereche de obiecte. Obiectele pot fi de orice tip.
- Solutia fara template-uri:
 - Folosim pointeri generici (void *) pentru obiecte, dar verificarea tipurilor nu se face la compilare, ceea ce insemna ca este responsabilitatea programatorului sa trateze tipurile.
 - Scriem o clasa separata pentru fiecare combinatie de tipuri de date.
- Solutia in C++ (cu template-uri):

Scurt exemplu

- Solutia in C++
(cu template-uri):

```
1 #include<iostream>
2 using namespace std;
3
4 // Consideram doua tipuri distincte, la care le spunem generic T
5 // U
6 template<class T, class V>
7 class Pair {
8 public:
9     T x;
10    V y;
11
12    Pair<T,V>(T tData, V vData);
13 };
14
15 // Pentru a exemplifica sintaxa, am ales sa nu definesc
16 // constructorul inline
17 template<class T, class V>
18 Pair<T,V>::Pair(T tData, V vData) : x(tData), y(vData)
19 { }
20
21 template<class T, class V>
22 std::ostream& operator<< (std::ostream& out, Pair<T,V>& pair){
23     return out << "(" << pair.x << "," << pair.y << ")\n";
24 }
25
26 int main() {
27     Pair<char, char> a('a', 'b');
28     Pair<int, double> b(1, 2.0);
29
30     std::cout << a;
31     std::cout << b;
32
33     return 0;
34 }
```

(a,b)

(1,2)

Template-uri

- Sunt un fel de pattern pe care le completeaza compilatorul.
- Compilatorul instantiaza template-ul pentru fiecare combinatie de tipuri de date depistata in cod.
- Codul template-ului se compileaza separat pentru fiecare specializare in parte (particularizare a template-ului pentru anumite tipuri de date).
- Acest lucru incetinesc compilarea si poate genera erori la compilare neasteptate daca tipurile de date nu implementeaza toate metodele invocate in template.
- **code bloating**
 - = copierea repetata a codului;
 - incetinesc compilarea foarte mult si duce la binare foarte voluminoase.

Tipuri de template-uri

- **Function** Template
- **Class** Template
- **Variable** Template (C++14)

Template-urile sunt initializate prin keyword-ul *template*.

Sintaxa este:

```
template <class identifier> declarație;  
template <typename identifier> declarație;
```

Parametrii template-urilor

3 tipuri diferite de parametri pentru template-uri

- **Type** – tip concret de date: int, char, double, o class, etc. (Este cel mai folosit tip.)
- **None-type** – valori: 3, 24, etc. Incepand cu C++20, ele pot fi si floating-points sau string-uri.
- **Template-template** – atunci cand pasam un template ca parametru al altui template.

Function Template

```
template <typename T>
T getMax(const T& a, const T& b) {
    return a > b ? a : b;
}
```

```
int main() {
    int intMax = getMax<int>(0, 1); // intMax este 1
    float doubleMax = getMax<double>(0.5, 1.0); // doubleMax este 1.0
    return 0;
}
```

- Compilatorul va genera două variante ale funcției getMax():
 - una în care parametrul de template T a fost înlocuit cu int,
 - alta în care a fost înlocuit cu float.
- Procesul de “rescriere” a funcției este transparent programatorului.

Function Template

```
template <typename T>
T getMax(const T& a, const T& b) {
    return a > b ? a : b;
}

int main() {
    int intMax = getMax<int>(0, 1); // intMax este 1
    float doubleMax = getMax<double>(0.5, 1.0); // doubleMax este 1.0
    return 0;
}
```

- Compilatorul va genera două variante ale funcției getMax():
 - una în care parametrul de template T a fost înlocuit cu int,
 - alta în care a fost înlocuit cu float.
- Procesul de “rescriere” a funcției este transparent programatorului.
- Nu au loc conversii implicite de tip.

```
template <typename T>
T getMax(const T& a, const T& b) {
    return a > b ? a : b;
}

int main() {
    int intMax = getMax(0, 1);
    float floatMax = getMax(0.5, 1.0);
    return 0;
}
```








- Dacă valorile de tip T sunt date ca parametri pentru funcție, atunci compilatorul poate infera tipul de date pentru care să apeleze funcția.
- Nu mai este nevoie de menționarea explicită a tipului în apelul funcției.

Instantierea: cppinsights.io

- Putem sa vedem instantierile pe care le face compilatorul in spate.

Inst

■ Pu



[C++ Standard: C++ 17] ▼ D... ▼ More

[Support the project on Patreon](#) ×

Made by [Andreas Fertig](#)
Powered by [Flask](#) and [CodeMirror](#)

Source:

```
1 template <typename T>
2 T getMax(const T& a, const T& b) {
3     return a > b ? a : b;
4 }
5
6 int main() {
7     int intMax = getMax<int>(0, 1); // intMax este 1
8     float doubleMax = getMax<double>(0.5, 1.0); // doubleMax este 1.0
9
10    // int intMax = getMax(0, 1);
11    // float floatMax = getMax(0.5, 1.0);
12
13    return 0;
14 }
15
16
```

Insight:

```
1 template <typename T>
2 T getMax(const T& a, const T& b) {
3     return a > b ? a : b;
4 }
5
6 /* First instantiated from: insights.cpp:7 */
7 #ifdef INSIGHTS_USE_TEMPLATE
8 template<>
9 int getMax<int>(const int & a, const int & b)
10 {
11     return a > b ? a : b;
12 }
13 #endif
14
15
16 /* First instantiated from: insights.cpp:8 */
17 #ifdef INSIGHTS_USE_TEMPLATE
18 template<>
19 double getMax<double>(const double & a, const double & b)
20 {
21     return a > b ? a : b;
22 }
23 #endif
24
25
26 int main()
27 {
28     int intMax = getMax<int>(0, 1);
29     float doubleMax = static_cast<float>(getMax<double>(0.5, 1.0));
30     return 0;
31 }
32
33
34
```

Specializari

- O specializare este o particularizare a template-ului pentru anumite tipuri de date concrete.

```
1  using namespace std;
2
3  template <typename T>
4  T equal(const T& a, const T& b) {
5      return a == b;
6  }
7
8  template <typename T>
9  T equal(const double& a, const double& b) {
10     return std::abs(a - b) < 0.0001;
11 }
12
13 int main() {
14     equal(0, 1);
15     equal(0.5, 1.0);
16     return 0;
17 }
```

Class Template

- Majoritatea regulilor de la funcțiilor template se aplica si aici.
- Metodele pot fi implementate in interiorul clasei sau in afara ei.
- Metodele implementate in afara clasei au nevoie sa fie precedate de template-head.
- Implementarea nu poate fi separata in .h si .cpp.

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T, // A type parameter
5         size_t SIZE> // A NTTP
6 struct Array {
7     const T* data() const {
8         return std::addressof(mData[0]);
9     }
10    T* data();
11    void print();
12
13    T mData[SIZ];
14 };
15
16 template <typename T, size_t SIZE>
17 T* Array<T, SIZE>::data() {
18     return std::addressof(mData[0]);
19 }
20
21 template <typename T, size_t SIZE>
22 void Array<T, SIZE>::print() {
23     for (int i = 0; i < SIZE; ++i) {
24         cout << mData[i] << " ";
25     }
26     cout<<endl;
27 }
28
29 int main() {
30     Array<int, 5> a{1, 2, 3, 4, 5};
31     a.print();
32     return 0;
33 }
```


Source:

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T, // A type parameter
5         size_t SIZE> // A NTTP
6 struct Array {
7     const T* data() const {
8         return std::addressof(mData[0]);
9     }
10    T* data();
11    void print();
12
13    T mData[SIZE];
14 };
15
16 template <typename T, size_t SIZE>
17 T* Array<T, SIZE>::data() {
18     return std::addressof(mData[0]);
19 }
20
21 template <typename T, size_t SIZE>
22 void Array<T, SIZE>::print() {
23     for (int i = 0; i < SIZE; ++i) {
24         cout << mData[i] << " ";
25     }
26     cout<<endl;
27 }
28
29 int main() {
30     Array<int, 5> a{1, 2, 3, 4, 5};
31     a.print();
32
33     Array<double, 2> b{1.0};
34
35     return 0;
36 }
37
38
```

Insight:

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T, // A type parameter
5         size_t SIZE> // A NTTP
6 struct Array {
7     const T* data() const {
8         return std::addressof(mData[0]);
9     }
10    T* data();
11    void print();
12
13    T mData[SIZE];
14 };
15
16 /* First instantiated from: insights.cpp:30 */
17 #ifdef INSIGHTS_USE_TEMPLATE
18 template<
19 struct Array<int, 5>
20 {
21     inline const int * data() const;
22
23     int * data();
24
25     void print()
26     {
27         for(int i = 0; static_cast<unsigned long>(i) < 5UL; ++i) {
28             std::operator<<(std::cout.operator<<(this->mData[i]), " ");
29         }
30
31         std::cout.operator<<(std::endl);
32     }
33
34     int mData[5];
35 };
36
37 #endif
38
```

```

1 #include <iostream>
2 using namespace std;
3
4 template <typename T, // A type parameter
5         size_t SIZE> // A NTTP
6 struct Array {
7     const T* data() const {
8         return std::addressof(mData[0]);
9     }
10    T* data();
11    void print();
12
13    T mData[SIZ];
14 };
15
16 template <typename T, size_t SIZE>
17 T* Array<T, SIZE>::data() {
18     return std::addressof(mData[0]);
19 }
20
21 template <typename T, size_t SIZE>
22 void Array<T, SIZE>::print() {
23     for (int i = 0; i < SIZE; ++i) {
24         cout << mData[i] << " ";
25     }
26     cout<<endl;
27 }
28
29 int main() {
30     Array<int, 5> a{1, 2, 3, 4, 5};
31     a.print();
32
33     Array<double, 2> b{1.0};
34
35     return 0;
36 }
37
38

```

```

39
40 /* First instantiated from: insights.cpp:33 */
41 #ifdef INSIGHTS_USE_TEMPLATE
42 template<
43 struct Array<double, 2>
44 {
45     inline const double * data() const;
46
47     double * data();
48
49     void print();
50
51     double mData[2];
52 };
53
54 #endif
55
56
57 template <typename T, size_t SIZE>
58 T* Array<T, SIZE>::data() {
59     return std::addressof(mData[0]);
60 }
61
62 template <typename T, size_t SIZE>
63 void Array<T, SIZE>::print() {
64     for (int i = 0; i < SIZE; ++i) {
65         cout << mData[i] << " ";
66     }
67     cout<<endl;
68 }
69
70 int main()
71 {
72     Array<int, 5> a = {{1, 2, 3, 4, 5}};
73     a.print();
74     Array<double, 2> b = {{1.0, 0}};
75     return 0;
76 }

```

Class Template

Exceptie 1:

- Pentru Class Template compilatorul nu poate infera tipul de date pentru care să instantieze clasa.

```
Array<int, 5> a{1, 2, 3, 4, 5};
```

- Fiecare argument trebuie sa fie explicit specificat.

Exceptie 2

- In C++17 compilatorul poate infera automat tipul de date pentru care să instantieze clasa, doar daca ea are un singur parametru.

```
Array<int> a{1, 2, 3, 4, 5};
```

```
Array a{1, 2, 3, 4, 5};
```

Class Template: Metode Template

Metodele unei clase template pot avea un template al lor.

Exceptie:

- Copy constructorul si destructorul nu pot fi template.

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename T>
5  class Foo {
6  public:
7      Foo(const T& x) : mX{x} {}
8
9      template<typename U>
10     Foo<T>& operator=(const U& u) {
11         mX = static_cast<T>(u);
12         return *this;
13     }
14 private:
15     T mX;
16 };
17
18 int main() {
19     Foo<int> fi{3};
20     fi = 2.5;
21     return 0;
22 }
```

Class Template: Mostenirea

Class Template pot mostenii alte Class Template sau alte clase si vice versa.

Cand o clasa template e derivate, in clasa derivate metodele si membrii acesteia nu sunt automat vizibili.

Solutie:

- `this`
- `using Base<T>::functie`
- Apel direct specificand namespace-ul:
`Base<T>::functie`

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename T>
5  class Foo {
6  public:
7      void func() {}
8  };
9
10 template <typename T>
11 class Bar : public Foo<T> {
12 public:
13     void barFunc() {
14         // func();
15         this->func();
16         Foo<T>::func();
17     }
18 };
19
20 int main() {
21     Bar<int> b{};
22     b.barFunc();
23     return 0;
24 }
```

Alias Template

- Un fel de sinonime pentru template-uri.
- Utile atunci când avem de scris în multiple locuri o definiție mai lungă.
- Permit specializare parțială a template-urilor.
- Permit abstractizarea diferențelor introduse de platformele unde e compilat codul

```
1  #include <array>
2
3  template<size_t N>
4  using CharArray = std::array<char, N>;
5
6  int main() {
7      CharArray<24> ar;
8      return 0;
9  }
```

```
1  #include <array>
2
3  template<size_t N>
4  using CharArray =
5  #ifdef PLATFORM_WIN
6      Array<char, N>;
7  #else
8      std::array<char, N>;
9  #endif
10
11 int main() {
12     CharArray<24> ar;
13     return 0;
14 }
```

Probleme

- Template-urile ne ajuta sa generam cod ca si cum am face copy-paste si am schimba tipurile variabilelor.
- In functie de compiler si optimizer, acest lucru poate duce la binare mari (code bloat).

Bune practici



Span (C++20)

- Dezavantaj: code bloat!

```
1 template<size_t N>
2 bool Send(const std::array<char, N>& data)
3 {
4     return write(data.data(), data.size());
5 }
6
7 template<size_t N>
8 void Read(std::array<char, N>& data)
9 {
10     // fill buffer with data
11 }
12
13 void Main()
14 {
15     std::array<char, 1'024> buffer{};
16
17     Read(buffer);
18     Send(buffer);
19
20     std::array<char, 2'048> buffer2{};
21
22     Read(buffer2);
23     Send(buffer2);
24 }
```

Span (C++20)

Solutie: span (c++20)

- Actioneaza ca un wrapper peste array-uri.
- Intern foloseste un pointer pentru lungime.
- => Folosim niste biti in plus ca sa stocam lungimea si evitam sa cream mai multe instantiari ale functiei.

```
1 bool Send(const span<char>& data)
2 {
3     return write(data.data(), data.size());
4 }
5
6 void Read(span<char> data)
7 {
8     int i = 1;
9     // fill buffer with data
10    for(auto& c : data) {
11        c = i;
12        ++i;
13    }
14 }
15
16 void Main()
17 {
18     std::array<char, 1'024> buffer{};
19
20     Read(buffer);
21     Send(buffer);
22
23     char buffer2[2'048]{};
24
25     Read(buffer2);
26     Send(buffer2);
27 }
```

Bune practici pentru Class Template

- Codul care este comun pentru toate instantiarile ar trebui mutat intr-o clasa de baza pentru a evita definirea multipla a acestuia la instantierea template-ului.
- Determina care este preferabil sa stochezi valori aditionale in loc sa le pasezi in template pentru ca reduce numarul de instantieri ale template-ului in detrimentul memoriei RAM.

Bune practici pentru Function Template

- Foloseste-le doar ca un API, iar cand vine vorba de calculul computational apeleaza o functie non-template.

Compile time evaluation

- De obicei ne gandim la obiecte ca sunt evaluate la run-time.
- Dar tipurile sunt evaluate la compile-time.

```
1 template<typename T, size_t SIZE>
2 struct Array
3 {
4     A Added a check that T is not a pointer
5     static_assert(not std::is_pointer<T>::value);
6
7     T*      data() { return std::addressof(mData[0]); }
8     const T* data() const
9     {
10         return std::addressof(mData[0]);
11     }
12     constexpr size_t size() const { return SIZE; }
13     T*      begin() { return data(); }
14     T*      end() { return data() + size(); }
15     T& operator[](size_t idx) { return mData[idx]; }
16
17     T mData[SIZE];
18 };
19
20 void Main()
21 {
22     int x{22};
23
24     // Array<int*, 2> invalid{&x}; B This will not compile
25     Array<int, 2> valid{x};
26 }
```

Compile time evaluation

- De obicei ne gandim la obiecte ca sunt evaluate la run-time.
- Dar tipurile sunt evaluate la compile-time.

C++20

```
1 template<typename T, size_t SIZE>
2 A C++20 require T to not be a pointer
3 requires(not std::is_pointer_v<T>) struct Array
4 {
5     T*      data() { return std::addressof(mData[0]); }
6     const T* data() const
7     {
8         return std::addressof(mData[0]);
9     }
10    constexpr size_t size() const { return SIZE; }
11    T*          begin() { return data(); }
12    T*          end() { return data() + size(); }
13    T& operator[](size_t idx) { return mData[idx]; }
14
15    T mData[SIZE];
16 };
17
18 void Main()
19 {
20     int x{22};
21
22     // Array<int*, 2> invalid{&x};
23     Array<int, 2> valid{x};
24 }
```


Standard Template Library (STL)



Standard Template Library (STL)

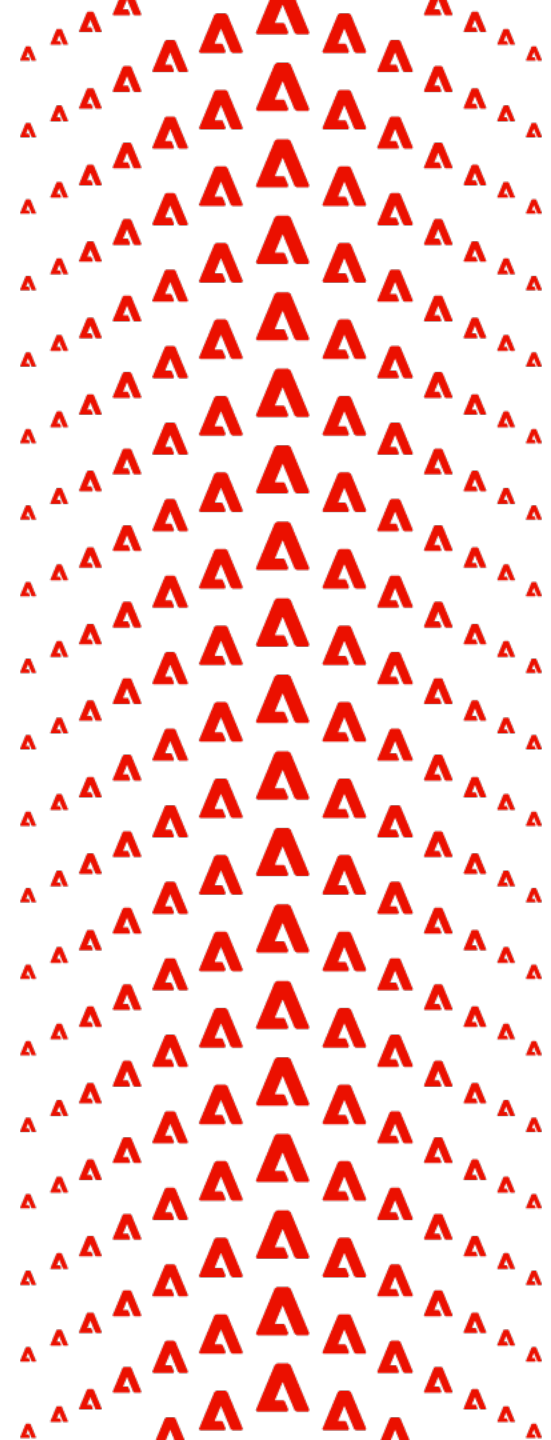
- Standard Template Library (STL) este o bibliotecă generică de C++, parțial inclusă în C++ Standard Library.
- Reprezintă una din modalitățile de a folosi cod gata implementat și conține implementări de template-uri pentru o serie de structuri de date și algoritmi uzuali.
- STL ne ajută să dezvoltăm rapid implementări scurte și eficiente.

Componente STL

STL pune la dispozitie 3 categorii de implementari care pot fi folosite de-a gata:

- Containere
- Adaptori
- Algoritmi

***Container: vector, list, slist,
deque, set, map***



`std::vector<T>`

`std::vector` este un wrapper peste un array alocat dinamic.

- **`push back()`** – Face insertia la sfarsit.

Inercarea de a insera elemente pe pozitii nealocate poate duce la SegFault, pentru ca nu se garanteaza nicaieri capacitatea array-ului din interior (folosim `resize()`).

- **operatorul `[]` si `at()`** – Fac accesul la elemente:

- operatorul `[]` fara verificarea limitelor
- functia membru `at()` cu verificarea limitelor

Avantajul functiei `at()` este ca nu primiti SegFault daca accesati indecsi invalizi.

In schimb , operatorul `[]` ofera acces direct si rapid la array-ul din interiorul clasei, dar raspunderea ii revine programatorului.

std::vector<T>

std::vector este un wrapper peste un array alocat dinamic.

- **resize()** sau **reserve()** – manipuleaza memoria
 - **resize()** redimensioneaza capacitatea array-ului la capacitatea data ca parametru. Daca noua dimensiune este mai mare decat vechea dimensiune, se invoca constructorul tipului pentru noile obiecte de la sfarsit, astfel incat acestea pot fi folosite direct!
 - **reserve()** nu redimensioneaza neaparat array-ul alocat, ci mai degraba se asigura ca acesta are o capacitate cel putin egala cu parametrul. Daca este nevoie de o realocare pentru a spori capacitatea vectorului, NU se invoca constructorul in spatiul de memorie de la sfarsit! Daca veti incerca sa folositi acele obiecte, veti avea tot felul de surprize neplacute.

`std::vector<T>`

`std::vector` este un wrapper peste un array alocat dinamic.

- **Atribuirea** dintre doi vectori se traduce prin dezalocarea memoriei primului vector si clonarea tuturor obiectelor din vectorul atribuit in vectorul la care se atribuie. Chiar daca STL face aceste operatii in batch, interschimbarea a doi vectori prin trei atribuirii dureaza mai mult sau mai putin o eternitate in termeni de timp de executie.
- Solutia inteligenta este sa folosim metoda **`swap()`**, care interschimba in mod inteligent doar pointerii catre zonele de memorie din interior.

std::vector<bool>

Exista specializari de vector in STL care sunt reimplementate mai eficient.

`std::vector<bool>` care tine elementele pe biti nu pe char-uri. Teoretic, acest lucru duce la o economie de memorie, dar practic alocarea se face in pasi incrementali de `sizeof(int)` din motive de performanta de viteza.

```

1 #include <iostream>
2 #include <string>
3
4 // Headerul care contine template-ul pentru vector
5 #include <vector>
6
7 int main()
8 {
9     int arrayInt[4] = { 0, 1, 2, 3};
10    // Cream un vector din intregii intre doua adrese dintr-un array
11    std::vector<int> vInt(arrayInt, arrayInt+4);
12
13    // Cream un vector de stringuri
14    std::vector<std::string> v;
15
16    char cifre[2] = { 0, 0 };
17    for (int i = 0; i < 10; i++){
18        cifre[0] = i + '0';
19        // Inseram la sfarsit instante de stringuri create din "cifre"
20        v.push_back(std::string(cifre));
21    }
22
23    // Afisam "0", in doua moduri: cu si fara bounds checking
24    std::cout << v[0] << " " << v.at(0) << "\n";
25
26    // Redimensionam vectorul v la 3 elemente
27    // (celelalte 7 se distrug automat)
28    v.resize(3);
29
30    // Il reextindem la 8 elemente cu reserve()
31    // (cele 5 elemente in plus nu se instantiaza automat)
32    // Reserve se foloseste pentru a evita in avans realocari in cascada
33    v.reserve(8);
34
35    // Cream un vector nou, gol, si il interschimbam cu cel vechi
36    std::vector<std::string> w;
37    w.swap(v);
38
39    // O sa cauzeze exceptie pentru ca elementul de pe pozitia 6 nu este instantiat
40    std::cout << w.at(6) << "\n";
41
42    return 0;
43 }

```


std::list<T> si std::slist<T>

- **std::list** si **std::slist** implementeaza liste dublue, respectiv simplu inlantuite de obiecte generice.
- Metodele de insertie sunt **push_back()** si **push_front()**.
- Metodele de eliminare de elemente sunt **pop_back()** si **pop_front()**.
- Accesul la elemente se face prin elementele de la capete listelor prin metodele **front()** si **back()**.

```
1 #include <iostream>
2 #include <string>
3
4 // Headerul care contine template-urile pentru liste
5 #include <list>
6
7 int main()
8 {
9     std::list<std::string> l;
10
11     char cifre[2] = { 0, 0 };
12     for (int i = 0; i < 10; i++){
13         cifre[0] = i + '0';
14         l.push_back(std::string(cifre));
15     }
16
17     // Afisam elementele de la inceputul si sfarsitul listei
18     std::cout << l.front() << " ... " << l.back() << "\n";
19
20     // Sortam lista
21     l.sort();
22
23     // Golim lista de la coada la cap si afisam elementele
24     // din ea pe masura ce le scoatem
25     while (!l.empty()){
26         std::cout << l.back() << " ";
27         l.pop_back();
28     }
29     std::cout << "\n";
30
31     return 0;
32 }
```

std::deque<T>

- std::deque functioneaza ca o lista dublu inlantuita, dar permite niste operatii suplimentare datorita modului intern de reprezentare.
- Cea mai importanta operatie suplimentara este accesul la elemente prin subscripting: **operatorul []**.

`std::set<T>`

- **`std::set`** este implementare a unei multimi in sens matematic.
- Tipul de date cu care se specializeaza un template trebuie sa suporte comparatie (**operatorul `<`** sa fie bine definit intre doua obiecte de tip `T`).
- Inserarea se realizeaza cu ajutorul metodei **`insert()`**.
- Cautarea se face cu ajutorul metodei **`find()`** care intoarce un iterator.
- Ambele operatii se fac in timp amortizat $O(\log N)$.
- Seturile sunt implementate cu ajutorul arborilor red-black.

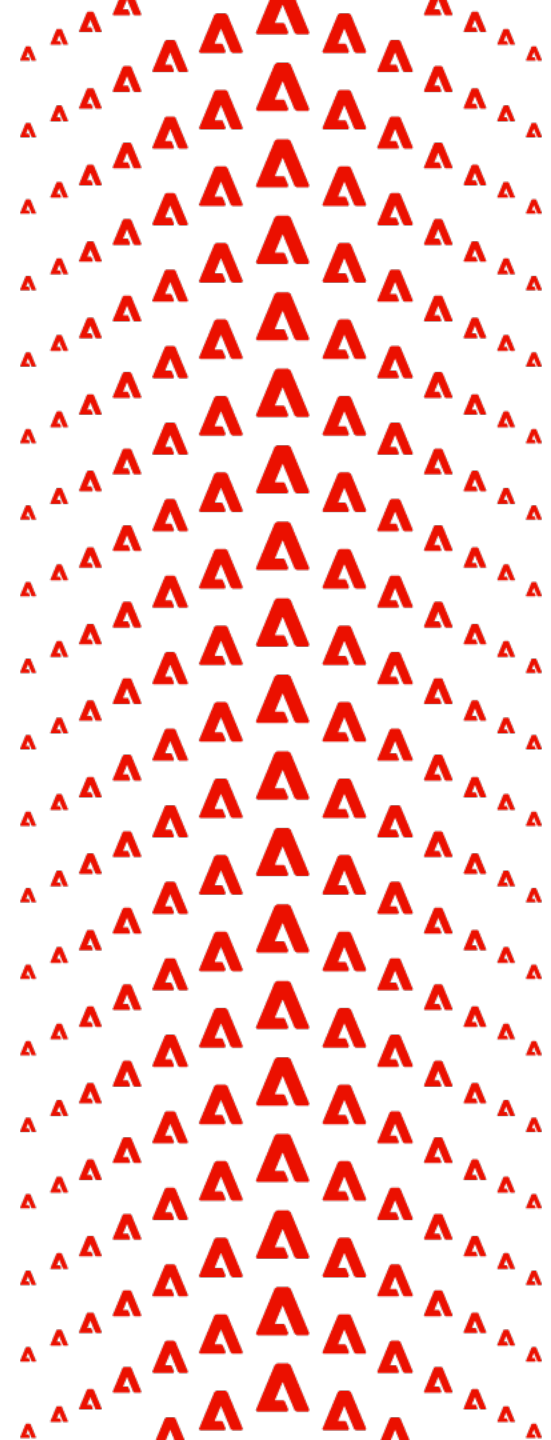
```
2
3 // Headerul care contine template-ul pentru set
4 #include <set>
5
6 int main()
7 {
8     // Cream un set de intregi. Comparatorul este implicit "<"
9     std::set<int> s;
10
11     // Inseram un element
12     s.insert(1);
13
14     // Verificam existenta lui. find() intoarce un iterator
15     if (s.find(1) != s.end()){
16         std::cout << "Da!\n";
17     } else {
18         std::cout << "Nu!\n";
19     }
20
21     return 0;
22 }
```

std::map<T, V>

- `std::map` este un template cu doi parametri de instantiere: tipul cheilor si tipul valorilor asociate.
- Ca si `std::vector`, `map` implementeaza **operatorul de indexare []**, dar folosirea operatorului conduce automat la inserarea in `map` a cheii.
- Functia alternativa la inserare se numeste **`insert()`**, iar functia de stergere se numeste **`erase()`**.

```
1 #include <iostream>
2 #include <string>
3
4 // In acest header se afla template-ul pentru map
5 #include<map>
6
7 int main()
8 {
9     std::map<std::string, int> numere;
10
11     // Inseram implicit un numar prin indexare
12     numere[std::string("unu")] = 1;
13
14     // Inseram prin apel de functie
15     numere.insert(std::pair<std::string, int>(std::string("doi"), 2));
16
17     // Stergem un numar din map
18     numere.erase(std::string("unu"));
19
20     // Afisam un numar din map
21     std::cout << numere[std::string("doi")] << "\n";
22
23     return 0;
24 }
```

*Adaptori: queue, stack, priority
queue*



Adaptori

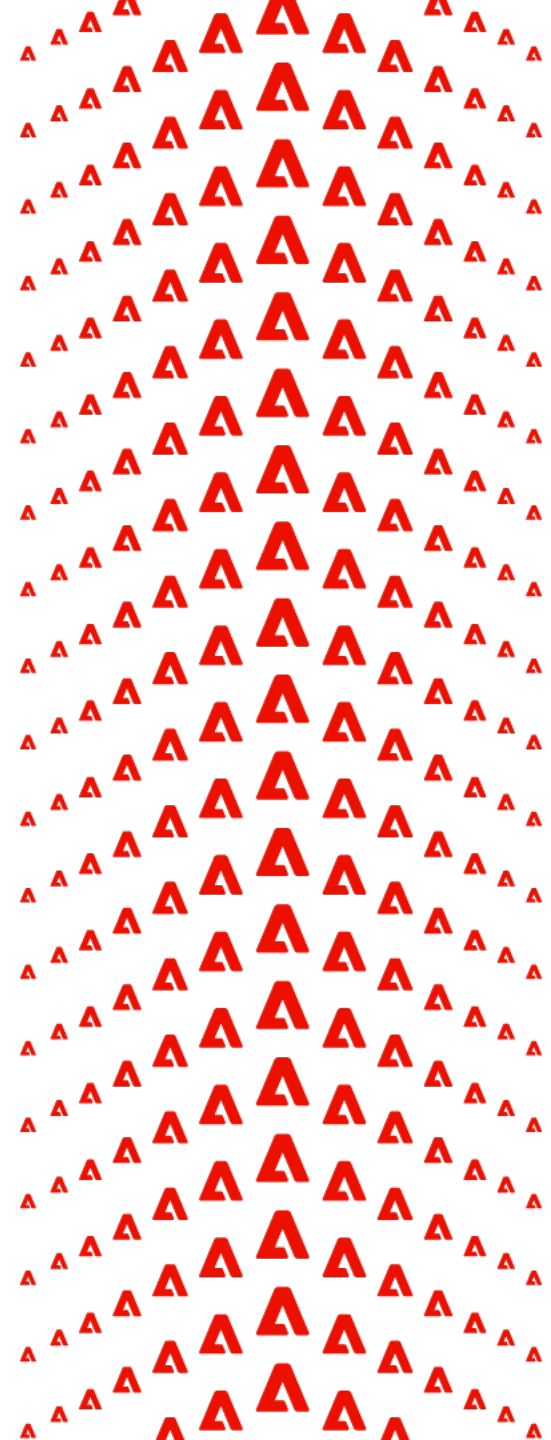
Adaptorii sunt clase care au rolul de a incapsula in interior alte containere mai generalizate, cu scopul de a restrictiona accesul la ele.

Solutia oferita de adaptori: cream o clasa noua care sa contina in interior un membru privat de alt tip (implicit este `std::deque`), iar apelurile la obiectul exterior sa fie directinoate catre apeluri de la obiectul intern. Astfel ramane vizibila doar functionalitatea care ne intereseaza.

Principalii adaptori din C++ sunt:

- **queue** - reprezinta o structura de tip FIFO (first in, first out)
- **stack** - reprezinta o structura de tip LIFO (last in, first out)
- **priority_queue** - reprezinta o structura de tip HEAP (cel mai mic element este primul care este scos)

Algoritmi



Algoritmi

STL pune la dispozitie algoritmi pentru:

- **cautare si statistici**: find, count, mismatch, equal, search
- **modificarea ordinii elementelor**: swap, copy, reverse, replace, rotate, partition
- **sortare**: sort, partial_sort, nth_element
- **cautare binara**: binary_search, lower_bound, upper_bound
- **interclasare**: merge, set_intersection, set_difference
- **gestiunea datelor**: make_heap, sort_heap, push_heap, pop_heap
- **minim si maxim**: min, min_element, lexicographical_compare

Possibile greseli

- Derivarea claselor rezultate din specializarea template-urilor de STL poate cauza memory leakuri si comportamente neasteptate, deoarece destructorii containerelor din C++ NU sunt virtuali.
- In containere, insertia se face prin copiere! Adica `v.push back(someObject)` va instantia un obiect nou care se va depune in container, iar `someObject` va ramane neatins.
- Un container sortat nu se reorganizeaza automat atunci cand modificam un obiect din interior prin intermediul unui pointer. Modul corect pentru a pastra containerul sortat este sa stergem din container obiectul pe care vrem sa il modificam si sa reintroducem versiunea modificata cu `insert()`.
- Containerele din C++ NU sunt thread-safe! Daca sunt accesate de mai multe thread-uri, atunci ele trebuie protejate explicit cu mutex-uri, semafoare, etc.

