

# STL and Algorithms (Part 1)

Atanas Semerdzhiev

## ОСНОВНИ ИЗТОЧНИЦИ

- Bjarne Stroustrup (2014) **The C++ Programming Language, 4th Edition**
- Bjarne Stroustrup (2014) **Programming: Principles and Practice Using C++, 2nd Edition**
- Nicolai M. Josuttis (2012) **The C++ Standard Library: A Tutorial and Reference, 2nd Edition**
- David Vandevoorde, Nicolai M. Josuttis (2002) **C++ Templates: The Complete Guide**

## История на стандарта

### First C++ Standard

- C++98: ISO/IEC 14882:1998.
- C++03: “technical corrigendum” (“TC”), ISO/IEC 14882:2003

### Second C++ Standard

- C++11: ISO/IEC 14882:2011
- C++14: ISO/IEC 14882:2014(E)

## За стандартната библиотека

- Като замисъл и имплементация, всяка част от библиотеката има собствена логика, която може да не е консистентна с останалите!
- Standard Library ≠ Standard Template Library

# Big-O

- $O(1)$
- $O(\log N)$
- $O(N)$
- $O(N \log N)$
- $O(N^2)$
- $O(2^N)$
- $O(N!)$

# Big-O in practice

	1	2	3	10	100	1000	1 000 000
$O(1)$	1	1	1	1	1	1	1
$O(\log N)$	1	1	2	4	7	10	20
$O(N)$	1	2	3	10	100	1000	1 000 000
$O(N \log N)$	1	2	5	34	665	9964	19 931 569
$O(N^2)$	1	4	9	100	10 000	1 000 000	$10^{12}$
$O(2^N)$	1	4	8	1024	$> 10^{30}$	$> 10^{301}$	$> 10^{301029}$
$O(N!)$	1	2	6	3 628 800	$> 10^{157}$	$> 10^{2567}$	$> 10^{5565708}$

## Big-O in STL

- Стандартът изисква определена сложност за отделните елементи;
- Той обаче НЕ определя какъв алгоритъм трябва да реализират те, въпреки, че зад повечето от тях стои конкретна идея (напр. динамичен масив за vector).

## Namespaces

Всички идентификатори от стандартната библиотека се намират в пространството (namespace) `std`.

Можем да ги използваме чрез:

1. Fully-qualified names

```
std::cout << "1";
```

2. using-declaration

```
using std::cout;
```

```
cout << "1";
```

3. using-directive

```
using namespace std;
```

```
cout << "1";
```

```
// За предпочитане е
```

```
// да се избягва!
```

## Header Files

1. По конвенция, новите header-файлове нямат разширение.

Например:

```
#include <iostream>
#include <utility>
#include <algorithm>
```

Идентификаторите, които се дефинират в тях са в namespace std.

2. За поддържане на съвместимост, с компилатора ви вероятно идват две версии на header-файловете. Например:

```
#include <string.h>
#include <stdlib.h>
```

и

```
#include <cstring>
#include <cstdlib>
```

## Обработка на изключения

## Пример: хвърляне на изключение

```
try
{
    p = new int[SIZE];
}
catch (std::bad_alloc& e)
{
    std::cerr << "ERROR: allocation failed (exception: "
                << e.what()
                << ")\n";
}
```

По-детайлен пример можете да намерите тук: <https://github.com/semerdzhiev/oop-samples/tree/master/Exceptions>

## Видове изключения

### 1. Language support

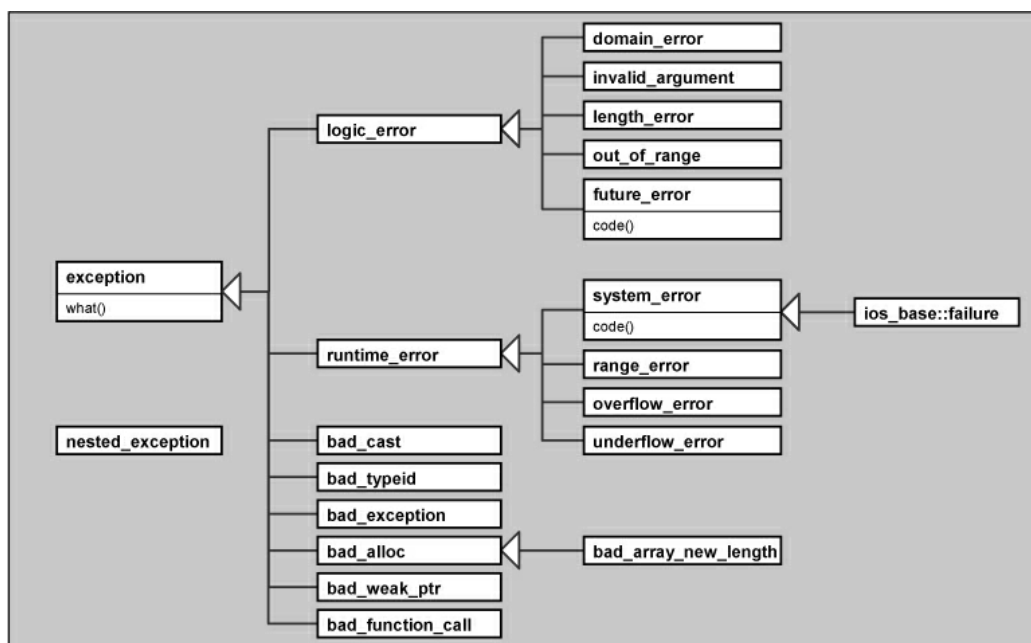
- Използват се от средствата на езика C++;
- Например `bad_cast`, което се хвърля от `dynamic_cast`.

### 2. Logic errors

- Възникват от нарушения в логиката на програмата;
- Могат да се избегнат ако напишем програмата „по-добре“.

### 3. Runtime errors

- Възникват по време на изпълнение;
- Не са в резултат от грешки в програмната логика. Например: няма достатъчно памет, липсващ конфигурационен файл и т.н.



Източник: Nicolai M. Josuttis (2012) *The C++ Standard Library: A Tutorial and Reference*, 2nd Edition

## Информация за грешката

В общия случай, информация за това какво се е объркало в програмата можете да получите по два начина:

1. От **типа на изключението**, което сте хванали;
2. От символния низ, който връща функцията `what()`.

`what()` е виртуална функция дефинирана в `exception` по следния начин:

```
virtual const char* what() const noexcept;
```

## Код на грешка

- Някои изключения, като например `system_error` носят допълнително и код на грешката, която е възникнала;
- Той може да се получи от член-функцията `code()`;
- Възможните кодове са изброени в `std::errc` (за повече информация вижте <http://en.cppreference.com/w/cpp/error/errc>).

## Кое изключение да хвърлим?

Добра (но не винаги удачна) практика е:

1. Ако е възможно, да използвате някой от стандартните класове;
2. Ако те не ви вършат работа, да наследите някой от тях и да реализирате нужното поведение;



## Добавяне на съобщение за грешка

```
if (i > SIZE)
    throw std::out_of_range("Invalid array index");
else
    std::cout << array[i];
```

Не всички класове от йерархията имат конструктор, който приема параметри!

Например `bad_alloc` няма.

`nullptr`

## За указателите и нулите

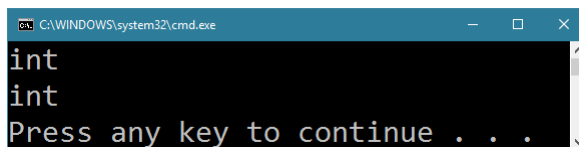
- Какво се случва, когато присвоим нула (0) на указател?
- Какво се случва, когато сравним два null-указателя?

```
int* p = 0;  
int* q = 0;  
  
if (p == q)  
{  
    //...  
}
```

## Проблем: разлика между null и 0

```
void f(int)  
{  
    std::cout << "int\n";  
}  
  
void f(char*)  
{  
    std::cout << "char*\n";  
}
```

```
int main()  
{  
    f(0);  
    f(NULL);  
    return 0;  
}
```



## Проблем: ориентиране в кода

```
int* pData = 0;  
int Index = 0;  
double Income = 0.0;  
char c = '\\0';  
char d = 0;  
f(0);  
std::cout << 0 << "\\n" << pData;
```

## Примери за употреба на nullptr

```
char *p = nullptr;  
char *q = 0;  
  
p == q;           // true  
p == nullptr;    // true  
q == nullptr;    // true  
  
f(nullptr);      // извежда  
                  // "char*"
```

```
int(*pf)(int);  
  
int (MyClass::*pm)(int);  
  
pf = nullptr;    // OK  
pm = nullptr;    // OK
```

## Допълнителни източници

- Herb Sutter, Bjarne Stroustrup (2003) **A name for the null pointer: nullptr**. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1488.pdf>
- Herb Sutter, Bjarne Stroustrup (2007) **A name for the null pointer: nullptr (revision 4)**. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2431.pdf>

## Наредени двойки

```
#include <utility>
```

## pair

```
std::pair<int, char> p1;  
std::pair<int, char> p2(2, 'b');  
  
p1.first = 1;  
p1.second = 'a';  
  
p1.swap(p2); // Алтернатива: swap(p1, p2)  
  
std::cout << "p1(" << p1.first << ", " << p1.second << ")\n";  
std::cout << "p2(" << p2.first << ", " << p2.second << ")\n";
```

За повече информация: <http://www.cplusplus.com/reference/utility/pair/>

## pair

```
std::pair<int, char> p1(10, 'a');  
std::pair<int, char> p2(20, 'a');  
std::pair<int, char> p3(10, 'b');  
  
p1 < p2; // true  
p1 < p3; // true  
p1 == p1; // true  
p1 != p2; // true
```

pair

```
std::pair<int, char> p1(10, 'a');
```

```
std::get<0>(p1) = 1;
```

```
std::get<1>(p1) = 'a';
```

```
std::cout << "p1(" << std::get<0>(p1)  
          << ", " << std::get<1>(p1) << ")\n";
```

## Наредени n-орки

```
#include <tuple>
```

## tuple

```
std::tuple<int, double, char> t1;  
std::tuple<int, double, char> t2(10, 5.0, 'b');  
  
std::get<0>(t1) = 10;  
std::get<1>(t1) = 1.5;  
std::get<2>(t1) = 'a';  
  
t1 < t2;  
t1 < std::make_tuple(20, 7.0, 'c');  
t1 < std::make_tuple<int, double, char>(20, 7.0, 'c');
```

## ВАЖНО!

Въпреки, че работата с наредени n-орки (tuples) може на пръв поглед да изглежда лесна, тяхната реализация се основава върху няколко механизма на езика, които не са тривиални.

За повече информация прочетете главите от основните източници, които третират:

- Variadic Templates
- Move Semantics

Полезен може да ви бъде и:

- Eli Bendersky (2014) **Variadic templates in C++**.  
<http://eli.thegreenplace.net/2014/variadic-templates-in-c/>

## tuple

```
typedef std::tuple<int, double, char> IntDoubleChar;

IntDoubleChar t1(10, 5.0, 'c');

auto t2 = std::tuple_cat(t1, std::make_tuple(20, 30));

std::cout << "Size of t1 is "
            << std::tuple_size<IntDoubleChar>::value
            << "\nSize of t2 is "
            << std::tuple_size<decltype(t2)>::value
            << "\n";
```

## Помощни функции

```
#include <algorithm>
```



## Намиране на минимум

```
int value;

value = std::min(10, 20); // 10
value = std::min(10, 10.0); // грешка!

std::pair<int,int> m = std::minmax(20, 10);
std::cout << m.first << " <= " << m.second << "\n";
```

## Когато стандартното сравнение не ни върши работа

```
bool SmallerLastBit(int a, int b)
{
    return (a & 1) < (b & 1);
}

int main()
{
    std::cout << std::min(500, 1, SmallerLastBit)
               << " has smaller last bit\n";
}
```

## Разменяне на стойностите на два елемента

```
int x = 1, y = 2;

std::swap(x, y);

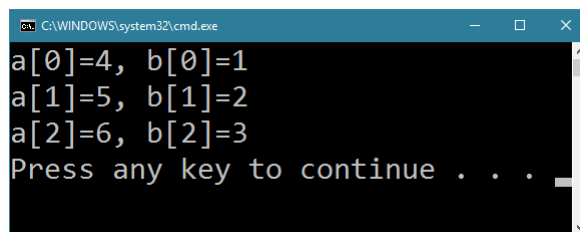
std::cout << "x = " << x
          << ", y = " << y << "\n";
```

## swap е предефинирана за масиви

```
int a[] = { 1, 2, 3 };
int b[] = { 4, 5, 6 };

std::swap(a, b);

for (int i = 0; i < 3; i++)
{
    std::cout
        << "a[" << i << "]= " << a[i]
        << ", b[" << i << "]= " << b[i] << "\n";
}
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the output of the program: "a[0]=4, b[0]=1", "a[1]=5, b[1]=2", "a[2]=6, b[2]=3", followed by "Press any key to continue . . .". The text is displayed in a monospaced font on a black background.

## Аргументи на swap

```
// Типовете на аргументите на swap()
```

```
// трябва да съвпадат!
```

```
int a[] = { 1, 2, 3 };
```

```
int c[] = { 7, 8, 9, 0 };
```

```
std::swap(a, c); // Грешка!
```

```
int x = 1;
```

```
long y = 1;
```

```
std::swap(x, y); // Грешка!
```

## Оптимизиране на разменянето

- swap() може да се предефинира за произволен тип.
- Тя може да оптимизира разменянето на стойности, като използва някакво свойство на типа – например за контейнерен тип може да размени указатели, вместо да копира съдържание.
- Така swap() може да работи по-бързо от еквивалентния код:

```
Temp = A;
```

```
    A = B;
```

```
    B = Temp;
```

- Това е направено за стандартните контейнери

# STL

## Увод

- STL = Standard Template Library (Стандартна библиотека с шаблони)
- Бележка: template може да се произнася като „темплейт“ (UK) или „темплит“ (USA).
- На следния адрес е направено нагледно сравнение между контейнерите и операциите, които те поддържат:  
<http://www.cplusplus.com/reference/stl/>
- Важно: стандартът не фиксира алгоритмите и структурите от данни, които контейнерите реализират! Те следват от изисквания върху поведението и сложността им.

## Видове контейнери

- Sequence containers
  - наредени колекции, в които всеки елемент има уникална позиция
  - array, vector, deque, list, forward\_list
- Associative containers
  - сортирани колекции; позицията на елементите зависи от техния ключ.
  - set, multiset, map, multimap.
- Unordered (associative) containers
  - колекции без наредба; позицията на елементите няма значение и може да се променя.
  - unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap.

## Типична реализация

- Sequence containers
  - Масив или свързан списък
- Associative containers
  - Двоично дърво
- Unordered (associative) containers
  - Хеш

## Основни операции с контейнерите

Операция	Коментар
<code>empty()</code>	Може да бъде по-бързо от <code>size() == 0</code>
<code>size()</code>	
<code>max_size()</code>	
<code>swap()</code>	Работи по-различно за класа <code>array</code>
<code>clear()</code>	
Оператори за сравнение	За наредените контейнери; По лексикографската наредба

## Итератори

- Основна идея
- Основни операции:

Операция	Описание
<code>operator *</code>	Текущ елемент
<code>operator ++</code>	Придвижва итератора напред
<code>operator --</code>	Придвижва итератора назад
<code>operator ==</code> <code>operator !=</code>	Проверява дали два итератора сочат една и съща позиция
<code>operator =</code>	Присвоява позиция на итератор

## Итератори

Контейнерите, които могат да бъдат итерирани предлагат следните операции (но може да реализират и допълнителни):

- `begin()` – връща итератор към първия елемент
- `end()` – връща итератор към последния елемент

Освен това те предлагат и два типа итератори: `iterator` и `const_iterator`. Например:

```
std::vector<int>::iterator
```

## std::vector

- По идея моделира динамичен масив
- Основни свойства
- Предимства и недостатъци
- Примерна реализация:  
<https://github.com/semerdzhiev/sdp-samples/tree/master/Dynamic%20Array>
- Източници:  
<http://www.cplusplus.com/reference/vector/vector/>  
<http://en.cppreference.com/w/cpp/container/vector>

## std::vector

Гарантирано във вектора елементите са подредени в непрекъснат блок в паметта

Това значи, че:

$\&v[i] == \&v[0] + i$

## Примери за употреба

```
std::vector<int> v1;
```

```
v1.push_back(0);
```

```
v1.push_back(1);
```

```
v1.push_back(2);
```

```
std::vector<int> v2(v1);
```

```
for (int i = 0; i < v2.size(); i++)  
    std::cout << v2[i] << " "  
              << v2.at(i) << "\n";
```

```
v2.clear();
```

```
v1.pop_back();
```

```
v1.pop_back();
```

```
v1.shrink_to_fit();
```



## Тип на променлива, по която обхождаме

```
// работи, но не е коректно
for (int i = 0; i < v2.size(); i++)
    std::cout << v2[i] << "\n";

// коректно
for (std::vector<int>::size_type i = 0; i < v2.size(); i++)
    std::cout << v2[i] << "\n";
```

За повече информация вижте: <http://stackoverflow.com/questions/409348/iteration-over-vector-in-c>

## Обхождане с итератор

```
// в един for
std::vector<int>::iterator it;

for (it = v2.begin(); it != v2.end(); ++it)
    std::cout << *it << "\n";

// в един for... ако се събере на екрана :- )
for (std::vector<int>::iterator it = v2.begin();
     it != v2.end();
     ++it)
    std::cout << *it << "\n";
```

За повече информация вижте: <http://stackoverflow.com/questions/409348/iteration-over-vector-in-c>

## Константен итератор

```
std::vector<int>::const_iterator cit;

for (cit = v2.cbegin(); cit != v2.cend(); ++cit)
{
    std::cout << *cit << "\n"; // OK
    *cit = 0; // грешка! (би работило за iterator)
}
```

## Преобразуване (cast) на итератори

- iterator се преобразува до const\_iterator, но не и обратното:

```
std::vector<int>::iterator it;
std::vector<int>::const_iterator cit;

cit = it; // OK
it = cit; // грешка!
```

## Следствие

```
std::vector<int>::iterator it;
std::vector<int>::const_iterator cit;

// OK
for (cit = v2.begin(); cit != v2.end(); ++cit)
    std::cout << *cit << "\n";

// Грешка!
for (it = v2.cbegin(); it != v2.cend(); ++it)
    std::cout << *it << "\n";
```

## Упражнение: работа с вектор

Напишете програма, която получава на стандартния вход поредица от цели числа и след това:

1. Съхранява числата във вектор от подходящ тип
2. Извежда съдържанието на вектора на екрана
3. Извежда средното аритметично на числата на екрана

Реализирайте поне една от стъпките 2 и 3 с итератор

**Съвет:** за конвертиране на числата от символен низ към `int`, използвайте наготово функцията `atoi()`.

## Инициализиране (1)

```
// ръчно запълване
std::vector<int> v;
v.push_back(0);
v.push_back(1);
v.push_back(2);
v.push_back(3);
```

## Инициализиране (2)

```
// копиране на друг вектор
std::vector<int> vectorCopy(v);

// копиране с итератори...
std::vector<int> vectorRange(v.cbegin(), v.cend());

// ...то работи за различни колекции
std::list<int> l(v.cbegin(), v.cend());

// запълване с fill constructor - четири единици
std::vector<int> vectorFill(4, 1);
```

## Извеждане (1)

```
template <class Iter>
void Print(Iter start, Iter end)
{
    for (; start != end; ++start)
        std::cout << *start;

    std::cout << "\n";
}
```

## Извеждане (2)

```
Print(v.begin(), v.end());
Print(vectorCopy.begin(), vectorCopy.end());
Print(vectorRange.begin(), vectorRange.end());
Print(vectorFill.cbegin(), vectorFill.cend());
Print(l.begin(), l.end());
```

## Обратно итериране

```
// Извежда 3210
for (std::vector<int>::reverse_iterator it = v.rbegin();
     it != v.rend();
     ++it)
{
    std::cout << *it;
}
```

## Random Access Iterator

```
std::vector<int>::iterator it = v.begin();

// ... запълваме вектора с 0, 1, 2, 3, 4, 5, 6

int elem;
elem = *it;           // elem == 0
elem = *(it + 5);     // elem == 5
elem = it[5];         // elem == 5

std::advance(it, 5);  // функцията работи за всички итератори
elem = *it;          // elem == 5
```

## Категории итератори в STL

- **Input Iterator:** може да се използва за извличане на стойностите на елементите в колекцията, но не и да ги променя;
- **Output Iterator:** може да се използва за променяне на стойностите на елементите в колекцията, но не и да ги извлича;
- **Forward Iterator:** може и да чете и да променя;
- **Bidirectional Iterator:** може да се движи и напред и назад;
- **Random Access Iterator:** позволяват да се достъпи произволен друг елемент, който се намира на някакъв отстъп (offset) от текущата им позиция.

## Категории итератори в STL

Итератор	Достъп до елементите	Посока на движение	Произволен достъп (random access)
Input	Read	Напред	Не
Output	Write	Напред	Не
Forward	Read/Write	Напред	Не
Bidirectional	Read/Write	Напред и назад	Не
Random Access	Read/Write	Напред и назад	Да

## Итератори и колекции

Ненаредените колекции предлагат най-малкото forward итератори, но създателите на имплементацията могат да ги реализират и с bidirectional итератори.

list, set, multiset, map и multimap предлагат bidirectional итератори.  
vector, deque и array предлагат random-access итератори.

end() не ни дава валидна позиция!

```
std::vector<int> v;  
  
// ...запълваме вектора...  
  
std::vector<int>::iterator it;  
  
it = v.end();  
  
std::cout << *it // Грешка!!!  
          << std::endl;
```



end() не е като NULL!

```
std::vector<int> v;  
  
// ...запълваме вектора...  
  
std::vector<int>::iterator it;  
  
it = v.end();  
--it; // OK  
  
std::cout << *it // OK  
          << std::endl;
```

## Други операции с вектори

Операция	Описание
insert(elem,pos)	Вмъква elem на дадена позиция pos
insert(elem,n,pos)	Вмъква n-копия на elem на дадена позиция
erase(pos)	Води до shift
reserve(n)	Резервира поне капацитет n
resize(n)	Може да намали или увеличи размера на вектора; Ако го увеличава, новите елементи се създават с default constructor;

## std::list

- Моделира работата на свързан списък
- Основни свойства
- Предимства и недостатъци
- Примерна реализация (вкл. итератор):  
<https://github.com/semerdzhiev/sdp-samples/tree/master/LinkedList>
- Източници:  
<http://www.cplusplus.com/reference/list/list/>  
<http://en.cppreference.com/w/cpp/container/list>

## Разлики на list спрямо vector

- Имплементира двусвързан списък
- Няма достъп до произволен елемент
- Вмъкването и изтриването на елементи е бързо, стига да сте на нужната позиция (в противен случай  $O(n)$  за достигането ѝ)
- Изтриването и вмъкването на инвалидират указатели и итератори към други елементи.
- Поддържа операции front(), push\_front(), and pop\_front()
- Няма операции за преоразмеряване или капацитет (но поддържа resize())
- Много допълнителни операции, като например remove, remove\_if, splice, reverse и др. Поддържа обратно-вървящи итератори.

## list::remove\_if

```
bool isOdd(int n) { return n & 1; }

int main()
{
    std::list<int> l;

    // ...запълваме списъка с елементи...

    l.remove_if(isOdd); // Премахва нечетните числа от l
}
```

## Други контейнери: Array и Deque

- Основни свойства
- Типична имплементация
- За повече информация:
  - Array: <http://www.cplusplus.com/reference/array/array/>
  - Deque: <http://www.cplusplus.com/reference/deque/deque/>

## Кога какво да използваме

Как да изберем между четирите контейнера:

- Array
- Vector
- List
- Deque

Полезна статия:

- Bjarne Stroustrup - Are lists evil?  
[http://www.stroustrup.com/bs\\_faq.html#list](http://www.stroustrup.com/bs_faq.html#list)

## Стек и опашка

```
#include <stack>
```

```
#include <queue>
```

## std::stack

Основните операции на структурата от данни се реализират от:

- `top()`
- `pop()`
- `push()`

Оптимизиран е за бързодействие, като е направен компромис с удобството на работа и безопасността.

`pop()` не връща стойност и не може да се използва в цикъл в конструкция от тип `while(s.pop(x))`.

За проверки за размера на стека се използват `empty()` и `size()`.

## Пример

```
std::stack<int> st;

st.push(10);
st.push(20);

std::cout << st.top() << "\n";
st.pop();
std::cout << st.top() << "\n";
st.pop();

st.pop();           // runtime error!
std::cout << st.top(); // runtime error!
```

## Стек и опашка като адаптори

- Стекът и опашката в STL са реализирани като адаптер върху някой от другите контейнери:
  - За стек може да се използват vector, deque или list;
  - За опашка може да се използват deque или list.
- Освен ако не е указано друго, по подразбиране и за двете се използва deque.
- За повече информация вижте:  
<http://www.cplusplus.com/reference/queue/queue/>  
<http://www.cplusplus.com/reference/stack/stack/>

## Указване на контейнер

Ако искате да използвате друг контейнер, можете да го укажете, когато създавате стека/опашката:

```
#include <stack>
```

```
#include <vector>
```

```
int main()
```

```
{
```

```
    std::stack<int, std::vector<int> > st;
```

```
}
```

## std::priority\_queue

STL поддържа и клас за приоритетна опашка.

Той също се реализира като адаптор, но трябва да бъде върху клас, който поддържа достъп до произволен елемент (например vector или deque).

Контейнер по подразбиране е deque.

Вътрешно класът използва произволния достъп, за да поддържа пирамида (heap).

За повече информация вижте:

[http://www.cplusplus.com/reference/queue/priority\\_queue/](http://www.cplusplus.com/reference/queue/priority_queue/)

## От рекурсивен към итеративен процес

- Типичен случай на употреба на стек е за преминаване от рекурсивен към итеративен процес (елиминиране на рекурсията).
- Важна техника в случаи, когато работим с рекурсивен алгоритъм, в който дълбочината на рекурсията е голяма и има риск от препълване на системния стек.
- Пример: преминаване от рекурсивна към итеративна версия на бързото сортиране (quicksort):

<https://github.com/semerdzhiev/sdp-samples/tree/master/Sorting>

## Упражнение: DFS със стек

Реализирайте алгоритъма за търсене в дълбочина (DFS), който проверява дали има път между две стаи в лабиринт.

В решението не може да използвате рекурсия.

Лабиринта представете като двумерен масив от тип `char`:

- Диез ( '#' ) – стена
- Интервал ( ' ' ) – проходима клетка

## Упражнение: BFS с опашка

Реализирайте задачата от предишното упражнение, но този път използвайте алгоритъма за търсене в широчина (BFS).



## Упражнение: сравнение между DFS и BFS

- Сравнете двете реализации, които написахте в предишните две упражнения.
- Какви прилики и разлики откривате в тях?
- Можем ли да ги опишем (на високо ниво, като се абстрахираме от използваната структура от данни) с едни и същи стъпки?

## Упражнение: намиране на стойност на израз

Напишете програма, която получава като параметър от командния ред аритметичен израз и извежда на екрана неговата стойност. Например:

```
> calc.exe "1+2*2-8/2^3"
4
```

В изразите:

- Няма интервали;
- Числата са едноцифрени;
- Всички операции са ляво-асоциативни и са както следва:
  - събиране (+)
  - изваждане (-)
  - умножение (\*)
  - деление (/)
  - степенуване (^)

## Упражнение: обратен полски запис

Напишете програма, която получава израз като този описван в предишното упражнение и:

- Преобразува израза до обратен полски запис (RPN) със Shunting-yard алгоритъма;
- Пресмята стойността на така получения RPN-израз.

## Алгоритми

## Разделяй и владей (Divide and conquer)

Рекурсивно на всяка стъпка:

- Задачата се разделя на по-малки подзадачи (divide);
- За всяка от тях се намира решение (conquer);
- Решенията се обединяват, за да се получи решението (combine).

Възможно е при рекурсията една и съща задача да се реши многократно (по-голяма времева сложност).

Всяка от подзадачите се решава независимо от останалите.

## Динамично оптимизиране/програмиране (dynamic programming)

В итеративен процес:

- Задачата се разделя на серия от подзадачи;
- Когато една подзадача се реши, решението се запазва;
- Решението на една подзадача в серия се получава от решените преди нея.

Възможно е при решаването на една подзадача да се използва решението на друга подзадача;

Всяка подзадача се решава точно по веднъж;

Необходима е допълнителна памет, в която да пазим решенията на задачите (по-голяма пространствена сложност)

## Упражнение: облепяне на писмо с марки

Нека разполагаме с марки от 1, 2, 3, 4 и 5 лева. По колко различни начина можем да облепим с марки един пощенски плик на стойност 10 лева? При облепянето всяка марка може да се използва само по веднъж.

Например три различни начина са:

1+2+3+4

2+3+5

1+4+5

## Упражнение: задача за раницата

Дадени са  $N$  броя предмети  $a_1, a_2, \dots, a_N$ .

Всеки от тях има цена  $p_1, p_2, \dots, p_N$ .

Имаме и раница с капацитет от максимум  $K$ -килограма.

Можем да вземем колкото и които поискаме от предметите, като единственото ограничение е те да се поберат в раницата.

Кои от предмети да вземем, така че да получим най-голяма стойност.