

# Въведение в стандартната C++ библиотека

Любомир Чорбаджиев<sup>1</sup>  
lchorbadjiev@elsys-bg.org

<sup>1</sup>Технологическо училище “Електронни системи”  
Технически университет, София

29 март 2009 г.



# Съдържание

- 1 Шаблони (templates) и родово (generic) програмиране
- 2 Контейнери и итератори
- 3 Символни низове `string`

# Шаблони

- Шаблоните обезпечават непосредствената поддръжка на така нареченото *обобщено програмиране*, т.е. програмиране, при което като параметри се използват типове.
- Механизмът на шаблоните в C++ позволява използването на типове в качеството на параметри при дефинирането на функции и класове.
- Шаблонът зависи само от тези свойства на параметъра-тип, които явно използва. Поради това не е необходимо различните типове, които се използват като параметри на шаблона да бъдат свързани по какъвто и да било начин.

# Дефиниране на шаблон

```
template<class T> class Stack {  
    T data_[10];  
public:  
    T pop(void);  
    void push(T val);  
    ...  
};
```

- Префиксът **template<class T>** се използва за дефиниране на шаблон (**template**).
- При използване на шаблона на мястото на “формалния параметър” **class T** се предава фактическият тип.
- В дефиницията на шаблона името на формалния параметър-тип **T** се използва точно по същия начин, по който се използват и имената на другите типове.

# Дефиниране на шаблон

```
template<class T> class Stack {  
    T data_[10];  
public:  
    T pop(void);  
    void push(T val);  
    ...  
};
```

- Областта на видимост за T завършва в края на обявата, започнала с **template<class T>**.
- В дефиницията на шаблона T е име на произволен тип; не е задължително T да бъде име на клас.

# Екземпляри на шаблона

```
Stack<double> doubleStack;  
Stack<int> intStack;
```

- Процесът на генериране на клас от шаблон на клас се нарича **създаване на екземпляр на шаблона (template instantiation)**.
- Генерирането на клас от шаблон на клас се изпълнява от компилатора.
- Класът, генериран от шаблон на клас, е обикновен C++ клас. Използването на шаблони не предполага допълнителни механизми по време на изпълнение на кода.
- Шаблоните обезпечават ефективен начин за генериране на код.

# Пример: стек

```
1 #ifndef STACK_HPP__
2 #define STACK_HPP__
3
4 #include <exception>
5
6 template<class T>
7 class stack {
8     static const unsigned size_=128;
9     T data_[size_];
10    int top_;
11 public:
12    stack(void);
13    const T& top(void) const;
14    void pop(void);
15    void push(const T& val);
16    bool empty(void) const;
17 };
```

## Пример: стек

```
1 template<class T>
2 stack<T>::stack(void)
3   : top_(-1)
4   {}
5
6 template<class T> const T&
7 stack<T>::top(void) const {
8   if (top_ < 0) {
9     throw std::exception();
10  }
11  return data_[top_];
12 }
```



## Пример: стек

```
1 template<class T> void
2 stack<T>::pop(void) {
3     if (top_ < 0){
4         throw std::exception();
5     }
6     top_--;
7 }
8 template<class T> void
9 stack<T>::push(const T& val){
10    if( size_ <= top_+1 ) {
11        throw std::exception();
12    }
13    data_[++top_]=val;
14 }
```

# Пример: стек

```
1 template <class T> bool  
2 stack<T>::empty(void) const {  
3     return top_ <0;  
4 }  
5 #endif
```

## Пример: стек

```
1 #include <iostream>
2 #include "stack.hpp"
3
4 int main(void) {
5     stack<int> si;
6
7     for(int i=0; i<10; ++i){
8         si.push(i);
9     }
10
11    while(! si.empty() ){
12        std::cout << si.top() << "␣";
13        si.pop();
14    }
```

## Пример: стек

```
1  std::cout << std::endl;
2
3  stack<float> sf;
4  for(int i=0; i<10; ++i){
5      sf.push(10.0*i);
6  }
7  while(! sf.empty() ){
8      std::cout << sf.top() << " ";
9      sf.pop();
10 }
```

# Пример: стек

```
1  std::cout << std::endl << std::endl;  
2  
3  stack<stack<int> > ssi;  
4  for(int i=0;i<5;++i){  
5      stack<int> temp;  
6      for(int j=0;j<10;++j){  
7          temp.push(i);
```

# Пример: стек

```
1     ssi.push(temp);
2 }
3
4 while(!ssi.empty()){
5     stack<int> ts=ssi.top();
6     while(!ts.empty()){
7         std::cout << ts.top() << "␣";
8         ts.pop();
9     }
10    std::cout << std::endl;
11    ssi.pop();
12 }
13
14 return 0;
15 }
```

# Проверка на типовете

- Проверка в точката на дефиниция: проверка за синтактични грешки и грешки, които не зависят от фактическите параметри-типове на шаблона.
- Проверка при създаване на екземпляр на шаблона: проверка за съответствие на фактическите типове, предадени на шаблона.

# Проверка на типовете: пример

```
1 class Foo {
2     int bar_;
3 public:
4     Foo(int bar) {
5         bar_=bar;
6     }
7 };
```

```
1 Stack<Foo> fooStack; // грешка!!
```



# Проверка на типовете: пример

```
1 template<class T> class Stack {  
2 ...  
3   void print_all(void) {  
4     for(int i=0;i<top_;i++) {  
5       cout << data_[i] << "\n";  
6     }  
7     cout << endl;  
8   }  
9 ...  
10 };
```

# Проверка на типовете: пример

```
1 class Bar {  
2     int foo_;  
3 public:  
4     Bar(int foo=0) {  
5         foo_=foo;  
6     }  
7 };
```

```
1 Stack<Bar> barStack;  
2 barStack.print_all(); // грешка!!
```

# Въведение в стандартната шаблонна библиотека

- STL (Standard Template Library) - стандартна шаблонна библиотека.
- Съдържа мощни компоненти, базирани върху шаблони:
  - Контейнери — шаблонни реализации на основните структури от данни.
  - Итератори — аналог на указателите. Предоставят достъп до елементите, които се съхраняват в контейнерите.
  - Алгоритми — търсене, сортиране, манипулиране на данните и т.н.
- Предоставя голямо количество компоненти, които са много мощни и лесни за използване.

# Контейнери

- Стандартната шаблонна библиотека предоставя три типа контейнери:
  - Последователни контейнери — линейни структури от данни (вектори, свързани списъци).
  - Асоциативни контейнери — нелинейни структури;
    - Дават възможност за бързо търсене на елементи;
    - Съхраняват се двойки ключ/стойност.
  - Адаптери — класове, които позволяват модифициране на поведението на други контейнери.
- Контейнерите имат ред общи функции.

# Контейнери

- Последователни контейнери:
  - `vector`
  - `deque`
  - `list`
- Асоциативни контейнери:
  - `set`
  - `map`
  - `multiset`
  - `multimap`
- Адаптери:
  - `stack`
  - `queue`
  - `priority_queue`

# Последователен контейнер `vector`

- Дефиниран е в заглавния файл `<vector>` в стандартното пространство от имена `std`.
- Структура от данни, при която елементите са разположени последователно в една област от паметта.
- Има достъп до елементите на вектора чрез оператора за индексирание `[]`.
- Използва се когато данните трябва да се сортират и да са лесно достъпни.
- Векторът `std::vector` е динамична структура, т.е. при необходимост неговият размер може да се променя по време на изпълнение на програмата. Промяната на размера на вектора обаче, е тежка операция.

# Член-функции на класа `vector`

Нека е дефинирана променливата `v` от типа `vector`. Тогава:

- `v.size()` — връща текущия размер на масива.
- `v.capacity()` — връща капацитета на масива, т.е. колко елемента могат да се добавят към масива, преди да се наложи ново заделяне на памет.
- `v.front()` — връща първия елемент на масива.
- `v.back()` — връща последния елемент на масива.
- `v.push_back(<value>)` — добавя елемент в края на масива.
- `v.pop_back()` — изтрива последния елемент в масива.

# Член-функции на класа `vector`

Нека дефинирана променливата `v` от типа `vector`. Тогава:

- `v[<index>]=<value>` — присвоява нова стойност `<value>` на елемента с индекс `<index>`. Не проверява индекса за коректност.
- `v.at(<index>)=<value>` — присвоява нова стойност `<value>` на елемента с индекс `<index>`. Проверява индекса за коректност и ако той е извън границите на масива генерира изключение `out_of_range`.



# Пример за използване на `vector`

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main(int argc, char* argv[]) {
6     vector<int> array;
7
8     cout << "Начален размер на масива: "
9           << array.size() << endl
10          << "Начален капацитет на масива: "
11          << array.capacity() << endl;
```

# Пример за използване на `vector`

```
1 array.push_back(2);
2 array.push_back(3);
3 array.push_back(4);
4
5 cout << "Размер на масива: "
6     << array.size() << endl
7     << "Капацитет на масива: "
8     << array.capacity() << endl;
9
10 return 0;
11 }
```

# Пример за използване на `vector`

```
lubo@kid ~/school/cpp/notes $ ./a.out
Начален размер на масива: 0
Начален капацитет на масива: 0
Размер на масива: 3
Капацитет на масива: 4
lubo@kid ~/school/cpp/notes $
```

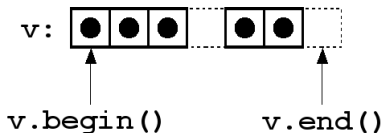
# Пример за обхождане на елементите на `vector`

```
1 vector<int> v;  
2 ...  
3 int sum=0;  
4 for(int i=0;i<v.size();i++) {  
5     cout << v[i] << endl;  
6     sum+=v.at(i);  
7 }  
8 cout << "sum=" << sum << endl;
```

# Итератори

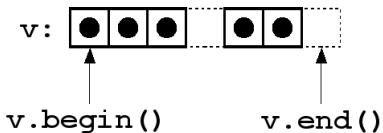
- Всеки контейнер от стандартната библиотека дефинира спомагателен тип `iterator`.
- Обектите от типа `iterator` се използват за последователно обхождане на елементите на контейнера.
- Итераторите могат да се разглеждат като “указатели” към елементите на контейнера.
- За итераторите е дефиниран операторът `*`. Този оператор връща елемента, към който “сочи” итераторът.
- За всички итератори е дефиниран операторът `++`, който премества итератора към следващия елемент от контейнера. Итераторите на някои контейнери поддържат и операцията `--`.

# Итератори



- Всеки контейнер има член-функция `begin()`, която връща итератор, насочен към първия елемент на контейнера.
- Всеки контейнер има член-функция `end()`, която връща итератор, насочен с едно след последния елемент на контейнера.

# Пример за използване на `iterator`



```
1 vector<int> v;  
2 ...  
3 int sum=0;  
4 for(vector<int>::iterator it=v.begin();  
5     it!=v.end();it++) {  
6     cout << *it << endl;  
7     sum+=*it;  
8 }  
9 cout << "sum=" << sum << endl;
```

# Итератори и контейнери

Голяма част от операциите с контейнери очакват като параметри итератори.

- `v.insert(<iterator>,<value>)` — вмъква елемент със стойност `<value>` преди елемента, сочен от итератора `<iterator>`.
- `v.erase(<iterator>)` — изтрива елемента, сочен от итератора `<iterator>`.
- `v.erase(<iterator1>, <iterator2>)` — изтрива елементите, които се намират между `<iterator1>` и `<iterator2>`. Например `v.erase(v.begin(),v.end())` изтрива всички елементи в контейнера.
- `v.clear()` — изтрива всички елементи в контейнера.



# Конструктори на `vector`

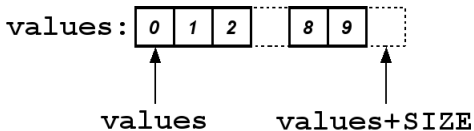
- `vector<type> v(<size>,<initial value>)`

```
vector<int> v1(10,0);
```

Векторът `v1` има десет елемента, като всеки елемент има стойност, равна на 0.

- `vector<type> v(<iterator1>,<iterator2>)`

```
const int SIZE=10;  
int values[SIZE]={0,1,2,3,4,5,6,7,8,9};  
vector<int> v2(values,values+SIZE);
```



# Пример за използване на `vector`

```
1 #include <iostream>
2 #include <vector>
3 #include <stdexcept>
4 using namespace std;
5
6 void print_all(vector<int>& v){
7     cout << "v.size()=" << v.size() << endl;
8     for(vector<int>::iterator it=v.begin();
9         it!=v.end(); ++it) {
10        cout << (*it) << " ";
11    }
12    cout << endl;
13 }
```

# Пример за използване на `vector`

```
1 int main(int argc, char* argv[]) {  
2  
3     const int SIZE=10;  
4     int values[SIZE]={0,1,2,3,4,5,6,7,8,9};  
5     vector<int> v(values, values+SIZE);  
6     print_all(v);  
7  
8     v.insert(v.begin(), -1);  
9     v.insert(v.end(), 10);  
10    print_all(v);  
}
```

# Пример за използване на `vector`

```
1  v.erase(v.begin());
2  v.erase(v.end()-1);
3  print_all(v);
4
5  v.erase(v.begin(),v.end());
6  cout << "v.empty()="
7      << boolalpha << v.empty() << "="
8      << noboolalpha << v.empty() << endl;
```

# Пример за използване на `vector`

```
1  try {  
2      v.at(1)=10;  
3  } catch(out_of_range e) {  
4      cout << "out_of_range_exception_catched: "  
5          << e.what() << endl;  
6  }  
7  return 0;  
8 }
```

```
lubo@kid ~/school/cpp/notes $ ./code/vector-example-05
v.size()=10
0 1 2 3 4 5 6 7 8 9
v.size()=12
-1 0 1 2 3 4 5 6 7 8 9 10
v.size()=10
0 1 2 3 4 5 6 7 8 9
v.empty()=true=1
out_of_range exception caught: vector [] access out of range
```

# Последователен контейнер `list`

- В стандартната библиотека е дефиниран последователен контейнер `list`. Дефиниран е в заглавния файл `<list>`.
- Списъкът в стандартната библиотека е реализиран като двусвързан списък.
- Списъкът е контейнер, при който операциите вмъкване и изтриване на елемент са бързи.
- За разлика от вектора, при списъка няма операция за достъп до елементите по индекс.

# Пример за използване на `list`

```
1 #include <iostream>
2 #include <list>
3 #include <stdexcept>
4 using namespace std;
5
6 void print_all(list<int>& l){
7     for(list<int>::iterator it=l.begin();
8         it!=l.end(); ++it) {
9         cout << (*it) << " ";
10    }
11    cout << endl;
12 }
```



# Пример за използване на `list`

```
1 int main(int argc, char* argv[]) {
2
3     const int SIZE=10;
4     int values[SIZE]={0,1,2,3,4,5,6,7,8,9};
5     list<int> l(values, values+SIZE);
6     print_all(l);
7
8     l.insert(l.begin(), -1);
9     l.insert(l.end(), 10);
10    print_all(l);
11
12    l.erase(l.begin());
13    list<int>::iterator last=l.end();
14    l.erase(--last);
15    print_all(l);
}
```

## Пример за използване на `list`

```
1 l.erase(l.begin(),l.end());
2 cout << "l.empty()="
3     << boolalpha << l.empty() << "="
4     << noboolalpha << l.empty() << endl;
5
6 l.push_back(10);
7 l.push_front(1);
8 l.push_front(0);
9 print_all(l);
10
11 l.pop_front();
12 print_all(l);
13 return 0;
14 }
```

# Символни низове `string`

- Дефинирани са в заглавния файл `<string>`.
- Поддържат основните операции със стрингове — копиране, търсене и т.н.
- Класът `string` поддържа автоматично управление на паметта, използвана от стринговете.
- Конструктори:

```
string str1("Hello_");  
string str2="world!";  
string str;
```

# Основни операции със `string`

Нека `str` е обект от типа `string`. Тогава:

- `str.length()` — връща дължината на стринга.
- `str.empty()` — връща **true**, ако стринга е празен.
- `str[<index>]` — връща символа с индекс `<index>`. Валидните стойности на индекса са в интервала от 0 до `(str.length()-1)`.
- `str.at(<index>)` — връща символа с индекс `<index>`. Валидните стойности на индекса са в интервала от 0 до `(str.length()-1)`. Ако индексът е извън допустимия интервал генерира изключение `out_of_range`.
- `str.c_str()` — връща терминиран с `'\0'` символен низ от типа **char\***

# Присвояване и конкатенация

Нека `str1` и `str2` са обекти от типа `string`. Тогава:

- `str2=str1` — копира стойността на стринга `str1` като стойност на `str2`.
- `str2.assign(str1)` — същото като `str2=str1`.
- `str2.assign(str1,<start>,<count>)` — копира `<count>` на брой символа започвайки от символа с индекс `<start>` от стринга `str1`.

```
string str1, str2;  
str1="Hello";  
str2.assign(str1, 0, 4);
```

# Присвояване и конкатенация

Нека `str1`, `str2` и `str3` са обекти от типа `string`. Тогава:

- `str2.append(str1)` — добавя стринга `str1` в края на стринга `str2`.
- `str2+=str1` — същото като `str2.append(str1)`.
- `str2.append(str1, <start>, <count>)` — добавя `<count>` на брой символа от стринга `str1`, като започва от символа с индекс `<start>`.
- `str3=str1+str2` — създава нов стринг, който е резултат от добавянето на `str2` към края на `str1`, и го присвоява като стойност на `str1`.

# Входно-изходни операции

- Обектите от типа `string` могат да участват във входно-изходни операции.

```
1  string str;
2  while(cin >> str)
3      cout << str << endl;
```

- Когато обект от типа стринг се чете от входния поток `cin >> str` интервалите се разглеждат като разделители.
- Когато е необходимо да се прочете цял ред от входния поток се използва функцията `getline(cint, str)`:

```
1  string str;
2  while(cin) {
3      getline(cin, str);
4      cout << str << endl;
5  }
```

# Изходни операции в паметта

- Потоците от типа `ostream` се асоциират с област от паметта, в която се записват резултатите от извършените изходни операции. Тази област от паметта е достъпна като стринг.
- Потоците от типа `ostream` са дефинирани в заглавния файл `<sstream>`.
- Потокът `ostream` е изходен поток. Върху него могат да се извършват всички изходни операции, допустими за изходен поток. Резултатът от тези операции се записва в специално създадена област от паметта.



# Изходни операции в паметта

```
1 #include <iostream>
2 #include <string>
3 #include <sstream>
4 #include <iomanip>
5 using namespace std;
6
7 int main(int argc, char* argv[]) {
8     ostringstream ostr;
9     ostr << "|" << setw(5) << 42
10         << "|" << setw(5) << 42 << "|";
11     string result=ostr.str();
12     cout << "result:␣" << result
13         << ">..." << endl;
14     return 0;
15 }
```

# Входни операции от паметта

- В заглавния файл `<sstream>` е дефиниран и потокът `istringstream`. Това е входен поток, който е асоцииран с област от паметта.
- При създаване на обект от типа `istringstream` на конструктора се предава обект от типа `string`. Операциите за четене от входния поток използват символи от предадения стринг.
- Потоците от типа `istringstream` са входни потоци. Върху тях могат да се изпълняват всички входни операции.

# Изходни операции в паметта

```
1 #include <iostream>
2 #include <string>
3 #include <sstream>
4 using namespace std;
5
6 int main(int argc, char* argv[]) {
7     string input("42_4.2");
8     istringstream istr(input);
9     int i;
10    double d;
11    istr >> i >> d;
12    cout << "i=" << i << ";_d=" << d << endl;
13    return 0;
14 }
```