

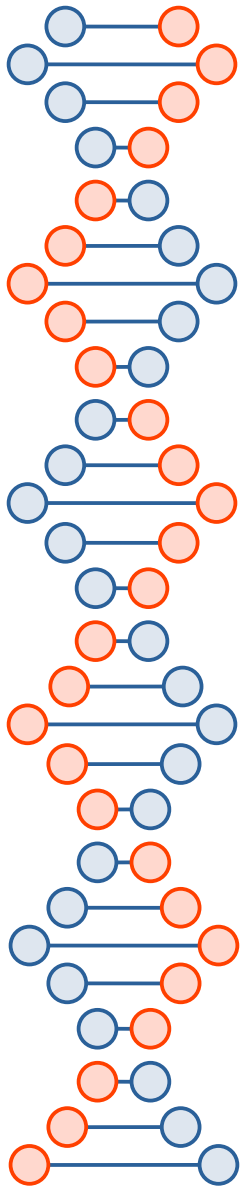
Итераторы и функциональные объекты

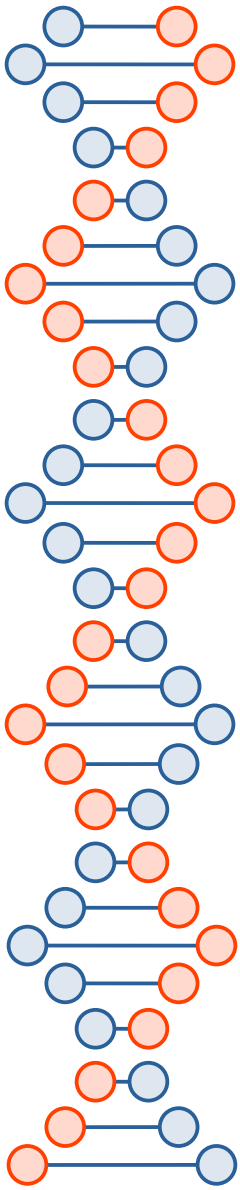
Итератор – аналог указателя, средство поэлементного просмотра набора данных

Функциональный объект – класс, который используется для задания критериев сравнения объектов

Итераторы <iterator>

- В итераторах есть понятия «текущий элемент», «указать на следующий элемент»
- Доступ к текущему элементу * или - >
- Переход к следующему элементу ++
- Определены присваивание, проверка на равенство и неравенство
- `i++` ; `++i`; `i=j`; `i == j`; `i != j`;



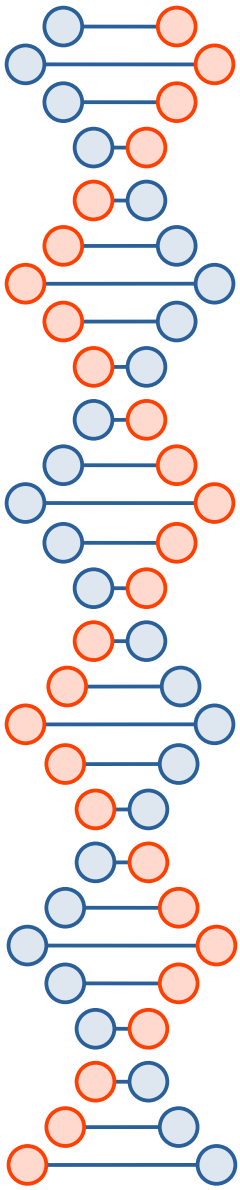


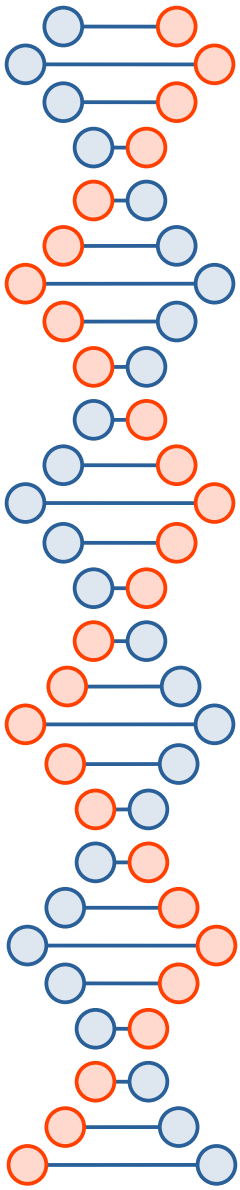
Типы итераторов

Тип итератора	операции	контейнеры
input_iterator	$x=*i$	все
output_iterator	$*i =x$	все
forward_iterator	$x=*i, *i=x$	все
bidirectional_iterator	$x=*i, *i=x, --i, i--$	все
random_access_iterator	$x=*i, *i =x, --i, i--, i+n, i-n, i+=n, i-=n, i<j, i>j, i<=j, i>=j$	Кроме list

Итератор может быть

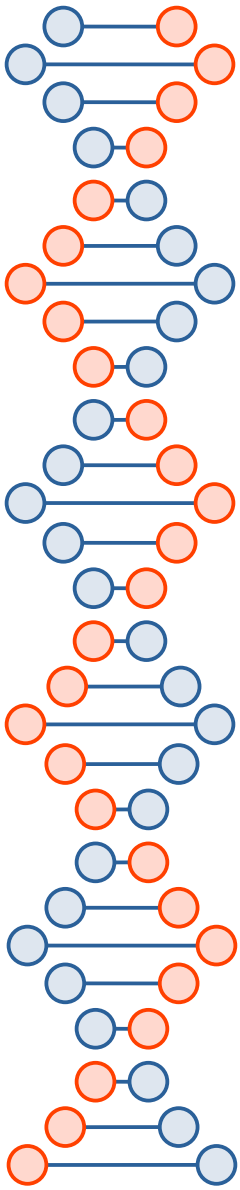
- Действительным или недействительным
- Итератор не был инициализирован
- Контейнер, с которым связан итератор, уничтожен или изменил размеры
- Итератор указывает на конец последовательности





<iterator>

- distance (InputIterator first, InputIterator last);
- Возвращает разность между двумя итераторами
- void advance (InputIterator & i, Distance n);
- i+=n;



reverse_iterator

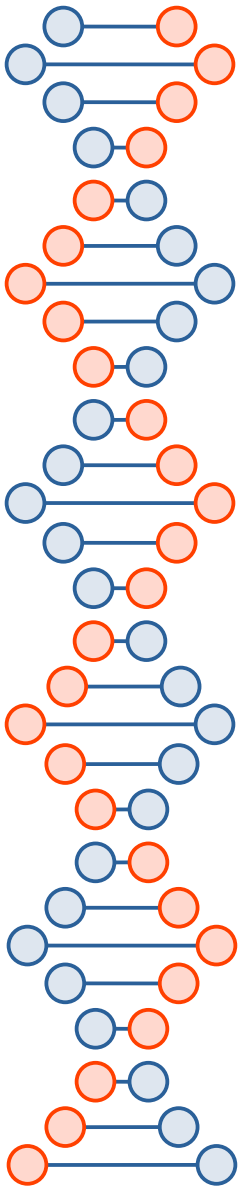
- `*`, `->`, `++`, `--`, `+`, `-`, `+=`, `-=`, `[]`
- `current`
- `==`, `!=`, `<`, `<=`, `>=`
- `rbegin()`, `rend()`
- `vector<int> v;`
- ```
for(vector<int> reverse_iterator i= v.rbegin(); i!= rend(); ++i) cout << *i << « »;
```

# Итераторы вставки

- `back_insert_iterator`
- `front_insert_iterator`
- `insert_iterator`
- 
- `back_inserter( C& x);`
- `front_inserter (C& x);`
- `Inserter (C& x, Iter i);`

# Потоковые итераторы

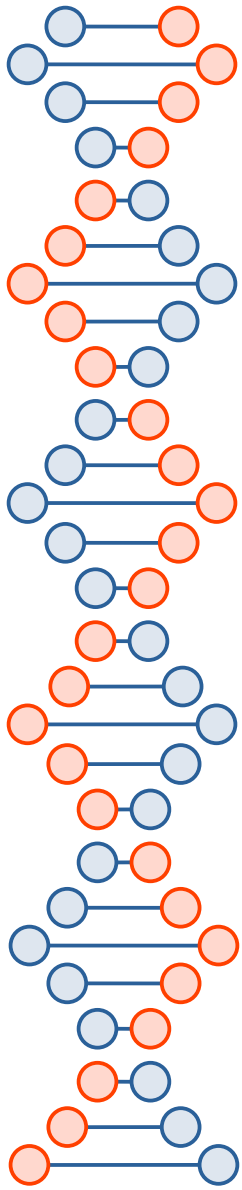
- Итератор входного потока `istream_iterator`
- 
- Итератор выходного потока `ostream_iterator`





# Итератор входного потока

- Читает элементы из потока
- `istream in («temp»);`
- `istream_iterator <int> i(in);`
- `int buf= *i;`
- `++i; int buf1= *i;`
- `while ( i !=istream_iterator<int> ())`
- `cout << *i;`



# Особенность итераторов входного потока

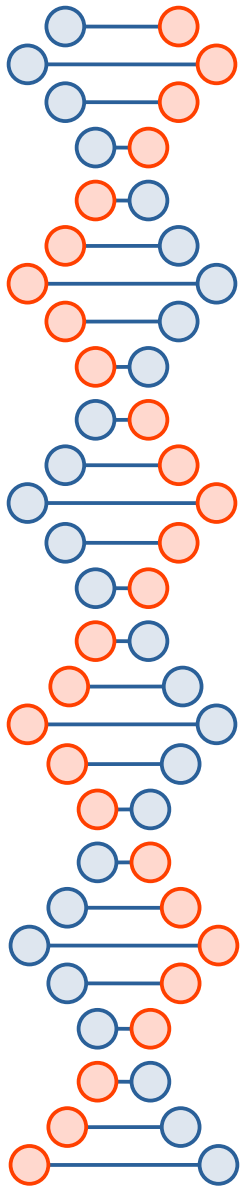
- Из  $i == j$  не следует  $++i == ++j$

# Итераторы выходного потока

- Записывают с помощью `<<` элементы в выходной поток
- `ostream_iterator <int> os (cout, «кг»);`
- `*os=100;`
- `++os; *os= 2;`

# Функциональные объекты

- Класс, в котором определена операция вызова функции
- Используются в качестве параметров стандартных алгоритмов для задания критериев сравнения или способов их обработки
- `<functional>`
- Предикат – функциональный объект или обычная функция, возвращающая `bool`

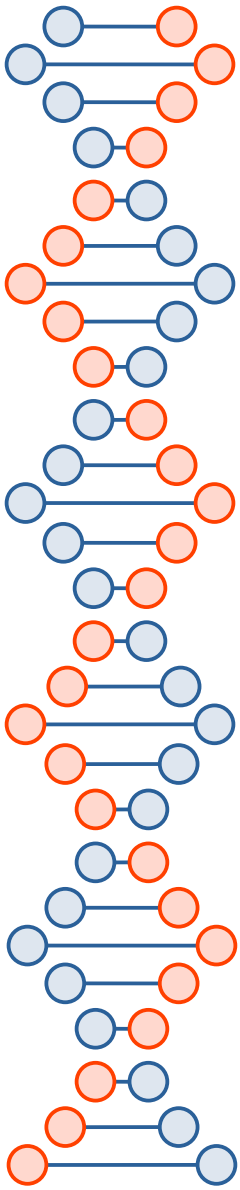


# Шаблоны базовых классов

- `template <class Arg, class Result>`
- `struct unary_function {`
- `typedef Arg argument_type;`
- `typedef Result result_type; };`
- 
- `template<class Arg1, class Arg2, class Result>`
- `typedef Arg1 first_argument_type;`
- `typedef Arg2 second_argument_type;`
- `typedef Result result_type; };`

# Адаптеры функций

- Функция, которая получает в качестве аргумента функцию и конструирует из нее другую функцию.
- Связыватели
- Отрицатели
- Адаптеры указателей на функцию
- Адаптеры методов

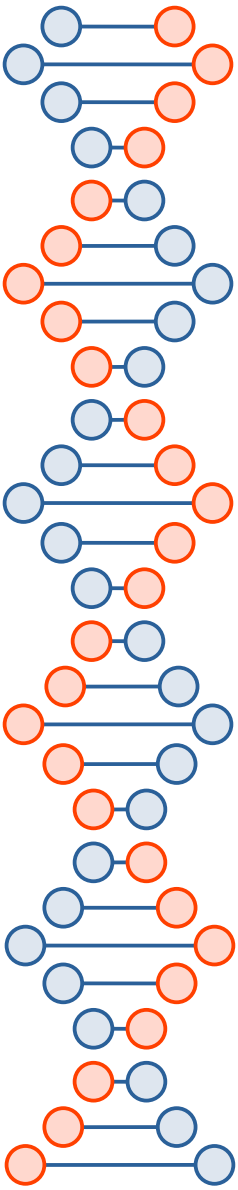


# Арифметические функциональные объекты

| имя        | тип      | результат |
|------------|----------|-----------|
| plus       | бинарный | $x+y$     |
| minus      | бинарный | $x-y$     |
| multiplies | бинарный | $x*y$     |
| divides    | бинарный | $x/y$     |
| modulus    | бинарный | $x\%y$    |
| negate     | унарный  | $-x$      |

# Предикаты

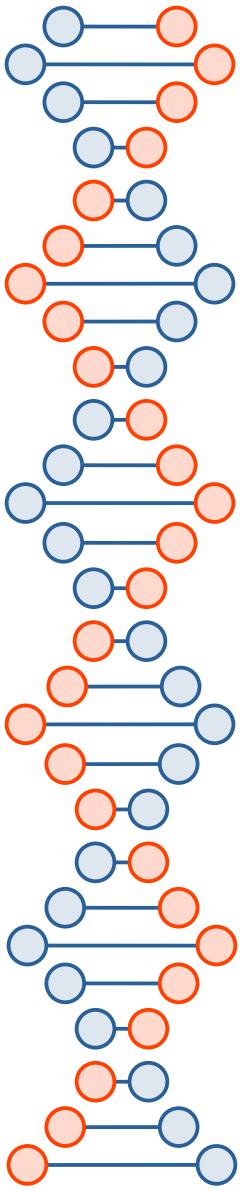
| имя           | тип      | результат   |
|---------------|----------|-------------|
| equal_to      | бинарный | $x == y$    |
| not_equal_to  | бинарный | $x != y$    |
| greater       | бинарный | $x > y$     |
| less          | бинарный | $x < y$     |
| greater_equal | бинарный | $x >= y$    |
| less_equal    | бинарный | $x <= y$    |
| logical_and   | бинарный | $x \& \& y$ |
| logical_or    | бинарный | $x \& \& y$ |
| logical_not   | унарный  | $! x$       |





# Предикат equal\_to

- `template<class T> struct equal_to:`
- `binary_function <T, bool>`
- `{ bool operator () (const T& x, const T& y) const`
- `{ return x == y;`
- `}`
- `};`

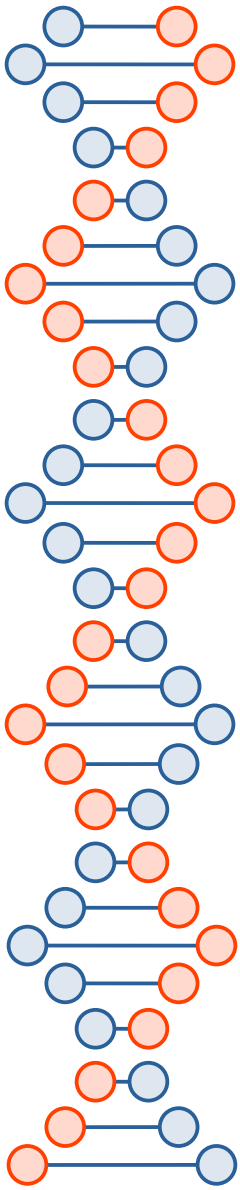




# Пример пользовательского предиката

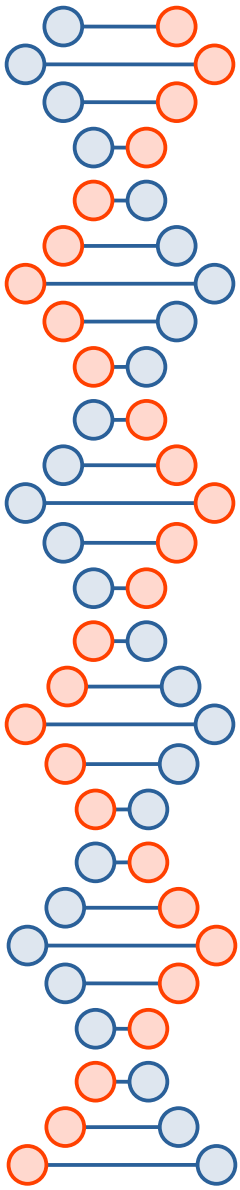
- `struct monstr_less_ammo:`
- `public binary_function <monstr, monstr, bool>(`
- `bool operator () ( monstr &m1, monstr &m2)`
- `{`
- `return m1.get_ammo () < m2.get_ammo();`
- `}`
- `};`

Предикат сравнивает двух монстров  
по значению поля ammo



# Отрицатели

- not1 , not2
- Применяются для получения противоположного предиката
- not2 (less <int>())    greater\_equal<int>

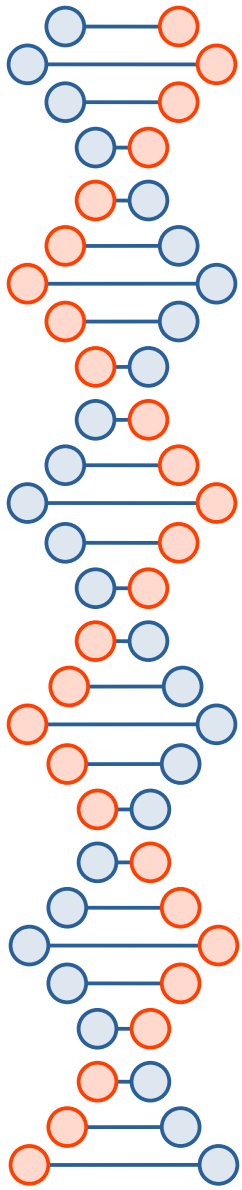


# Связыватели

- `bind2nd`
- `bind1st`
- Позволяют связать с конкретным значением второй и первый аргумент бинарной функции

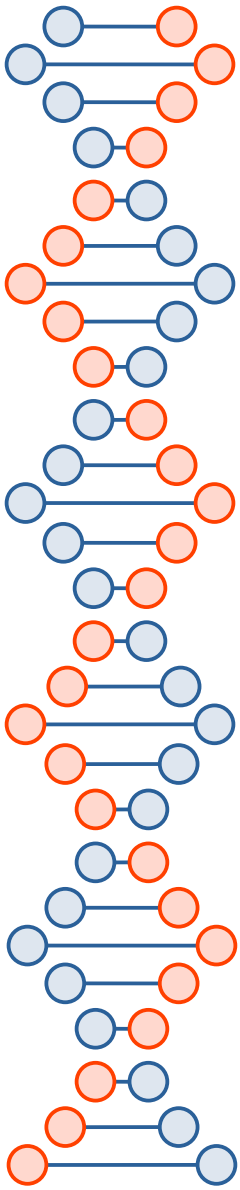
# Пример со связывателем

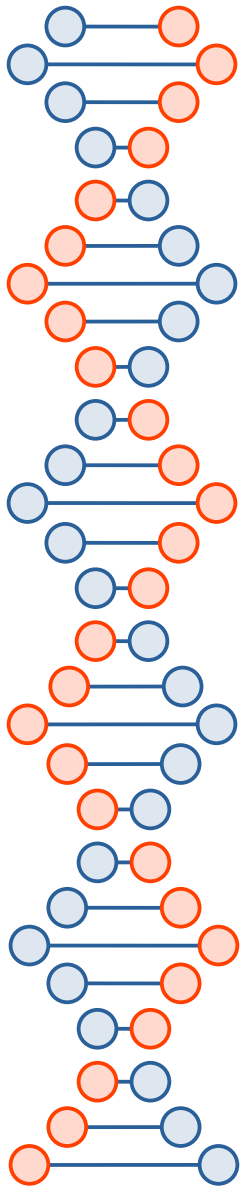
- Пусть требуется вычислить количество элементов целочисленного массива, меньших 40
- `#include<iostream> #include<functional>`
- `#include< algorithm> using namespace std;`
- `int main()`
- `{`
- `int m[8]= {45,65, 36, 25, 674,2,13,35};`
- `cout << count_if ( m, m+8, bind2nd (less<int>(), 40));`
- `return 0;`
- `}`



# Адаптеры указателей на функцию

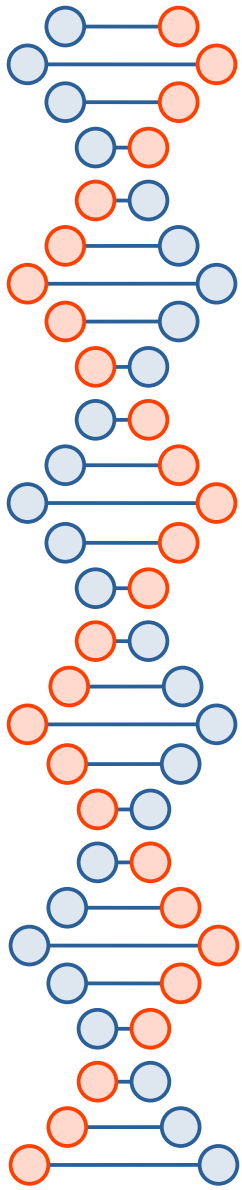
- `pointer_to_unary_function`
- `pointer_to_binary_function`
- Нужны, чтобы применять связыватели к обычным указателям на функции
- `ptr_fun`
- функции\_адаптеры, которые преобразуют переданный им в качестве параметра указатель на функцию в функциональный объект





## Пример применения адаптера функции

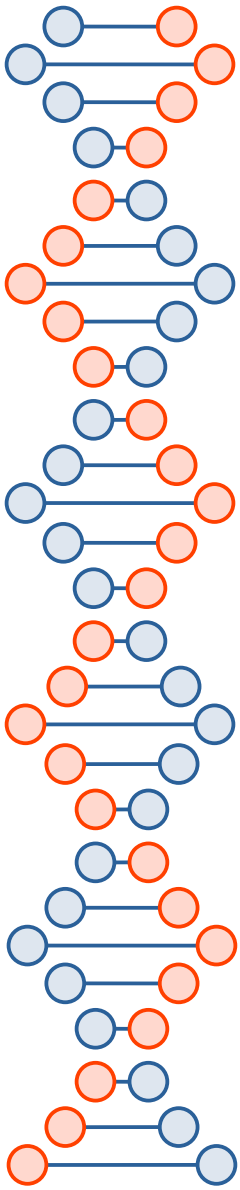
- `#include<iostream> #include<functional>`
- `#include<algorithm> using namespace std;`
- `struct A { int x, int y; };`
- `bool lss (A a1, A a2){ return a1.x<a2.x;}`
- `int main ()`
- `{ A ma[5] ={{2,4}, {3,1}, {2,2}, {1,2}, {1,2}};`
- `A elem= {3,0};`
- `cout << count_if (ma, ma+5, bind2nd( ptr_fun( lss),`  
`elem);`
- `return 0;}`



## Пример применения адаптера функции

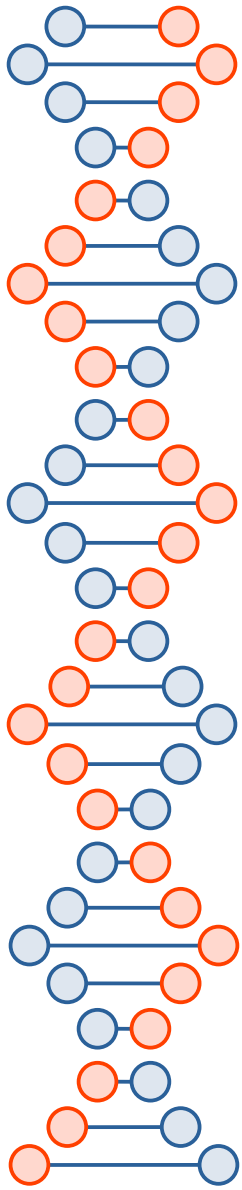
- `#include<iostream> #include<functional>`
- `#include<algorithm> #include<vector>`
- `using namespace std;`
- `enum color {red, green, blue};`
- `class monstr {`
- `int health, ammo; color skin; char * name;`
- `public: monstr ( int he=100, int am=10);`
- `monstr (color sk); monstr ( const *nam);`
- `monstr (const monstr &M);`





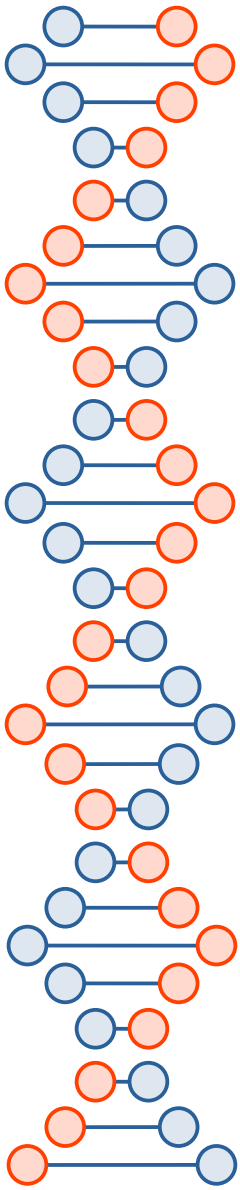
## Пример применения адаптера функции

- `~monstr () { delete [ ] name; }`
- `operator int() { return health; }`
- `Int get_health() { return health(); }`
- `friend ostream&`
- `operator << (ostream & out, monstr & m)`
  - `{ return out << «monstr: « << «ammo= «`
  - `<<m.ammo << «health =» << m.health << endl;`
  - `};`



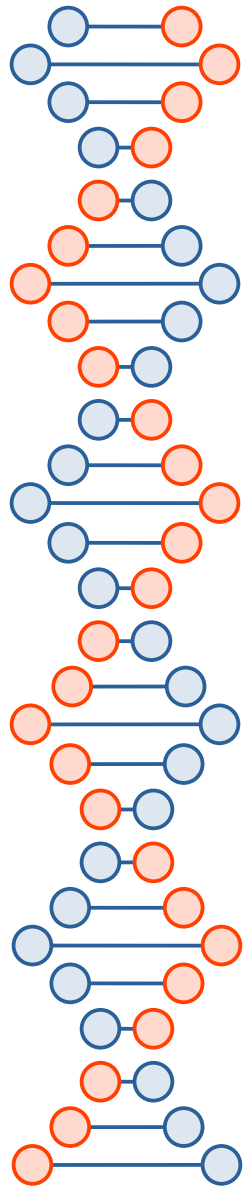
## Пример применения адаптера функции

- `monstr :: monstr (int he, int am):`
- `health(he), ammo(am), skin(red), name(0){}`
- 
- `monstr :: monstr (const monstr &M)`
- `{ if (M.name){ name= new char [strlen( M,name)+1];`
- `strcpy( name, M.name);}`
- `else name=0;`
- `health= M.health; ammo= M.ammo; skin= M.skin;}`
-



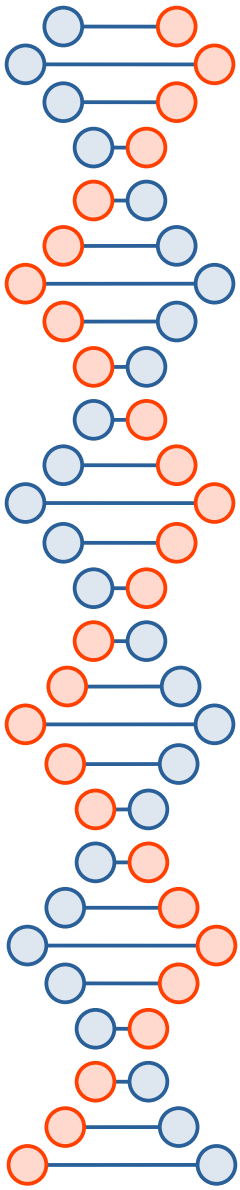
## Пример применения адаптера функции

- `monstr:: monstr (color sk)`
- `{`
- `switch(sk) {`
- `case red: health=1; ammo=10; skin=red; name=0;`  
`break;`
- `case green: health=2; ammo=20; skin=green;`  
`name=0; break;`
- `case blue: health=3; ammo=40; skin=blue; name=0;`  
`break;`
- `}}`



## Пример применения адаптера функции

- `bool less_health (monstr m1, monstr m2)`
- `{ return m1.get_health() <m2.get_health(); }`
- 
- `int main()`
- `{ vector<monstr> m;`
- `monstr M(10,30);`
- `m.push_back(M);`
- `m.push_back( monstr(«Vasia»);`
- `m.push_back(monstr(red));`

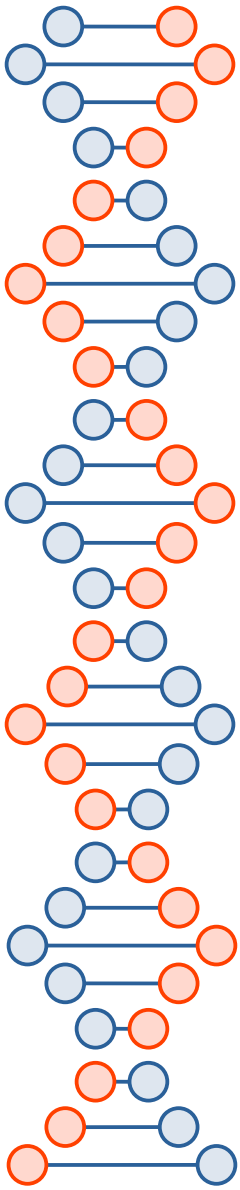


## Пример применения адаптера функции

- `cout << «Monstry:» << endl;`
- `for (int i=0; i<m.size();i++) cout << m[i] << « »;`
- `cout << endl;`
- `cout << «Count_if :»;`
- `cout << count_if (m.begin(), m.end(),`
- `bind2nd (ptr_fun (less_health),20));`
- `return 0;`
- `}`

# Результаты работы программы

- Monstry:
- monstr : ammo=30 health =10
- monstr : ammo=10 health=200
- monstr: ammo=10 health=1
- Count\_if : 2

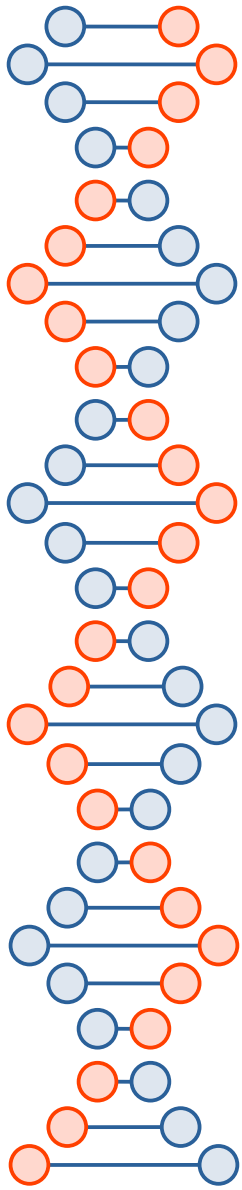


# Адаптеры методов

- `#include<iostream> #include<algorithm>`
- `using namespace std;`
- `void show (int a){ cout << a<< endl;}`
- `int main()`
- `{ int m[4]={3,5,9,6};`
- `for_each( m,m+4, show);`
- `return 0;`
- `}`

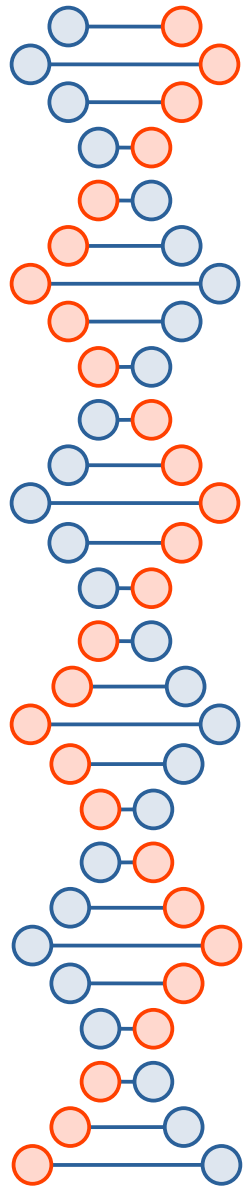
# Адаптеры методов

- Позволяют использовать методы классов в качестве аргументов стандартных алгоритмов
- Адаптер получает функцию и конструирует из нее другую функцию

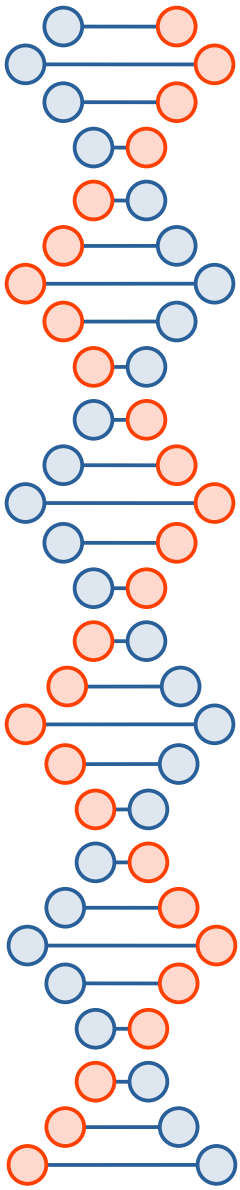




# Стандартные адаптеры методов



| имя         | Тип объекта          | действие                                                  |
|-------------|----------------------|-----------------------------------------------------------|
| mem_fun     | mem_fun_t            | Вызывает безаргументный метод через указатель             |
| mem_fun     | const_mem_fun_t      | Вызывает безаргументный константный метод через указатель |
| mem_fun     | mem_fun1_t           | Вызывает унарный метод через указатель                    |
| mem_fun_ref | mem_fun_ref_t        | Вызывает безаргументный метод через ссылку                |
| mem_fun_ref | const_mem_fun_ref_t  |                                                           |
| mem_fun_ref | mem_fun1_ref_t       | Вызывает унарный метод через ссылку                       |
| mem_fun_ref | const_mem_fun1_ref_t |                                                           |
| mem_fun_ref | const_mem_fun1_t     |                                                           |



## Пример

- Опишем в классе `monstr` метод `dead`
- `bool dead(){ return !health;}`
- `vector< monstr> ostrich(100);`
- Вычислим число потерь в векторе
- `cout << count_if( ostrich.begin(), ostrich.end(), mem_fun_ref (&monstr ::dead));`