# ООП 2021
# Лекция 10

## Итераторы.
## Аллокаторы.
## <iterator>
## <memory>
## Виртуализация функций

oopCpp@yandex.ru

**Итераторы** — это клей, скрепляющий алгоритмы стандартной библиотеки с их данными. Итераторы можно также назвать механизмом, минимизирующим зависимость алгоритмов от структуры данных, которыми они оперируют. (Б. Страуструп)

Итератор — это аналог указателя, в котором реализованы операции **косвенного доступа** (например, оператор * для разыменования) и **перехода к новому элементу** (например, оператор ++ для перехода к следующему элементу).

Последовательность элементов определяется парой итераторов, задающих полуоткрытый диапазон [begin , end).

Здесь итератор begin указывает на первый элемент последовательности, а итератор end — на элемент, следующий за последним элементом последовательности.

Никогда не считывайте и не записывайте значение *end. Для пустой последовательности всегда выполняется условие begin == end. Другими словами, для любого итератора p последовательность [p:p) является пустой.

Для того чтобы считать последовательность, алгоритм обычно получает пару итераторов [a,b) и перемещается по элементам с помощью оператора ++, пока не достигнет конца.

```
while (a !=b ) { // используйте !=
    ++a;  // переходим к следующему элементу
}
```

Алгоритмы, выполняющие поиск элемента в последовательности, в случае неудачи обычно возвращают итератор, установленный на конец последовательности. Рассмотрим пример.

```
  p = s.find (v. begin () ,v.end() , x) ; // ищем x в последовательности v
if (p != v.end() ) {
  // x найден в итераторе p
}
else {    // x не найден в диапазоне [v.begin(), v.end() )     }
```

Алгоритмы, записывающие элементы последовательности, часто получают только итератор, установленный на ее первый элемент.

```
template< class _Iter> void   f  (_Iter p, int n) {
  while (n>0) *(p++) = --n;
}
  vector<int> v(10);
f (v.begin(), v.size() );       // OK
f (v,begin () ,1000);           // большая проблема
```

**#include <iterator>**

advance ·
back_insert_iterator ·
back_inserter ·
bidirectional_iterator_tag ·
distance ·
forward_iterator_tag ·
front_insert_iterator ·
front_inserter ·
input_iterator_tag ·
insert_iterator · inserter ·
istream_iterator ·
istreambuf_iterator ·

iterator ·
iterator_traits ·
operator!= ·
operator== ·
operator< ·
operator<= ·
operator> ·
operator>= ·
operator+ ·
operator- ·

ostream_iterator ·
ostreambuf_iterator ·
output_iterator_tag ·
random_access_iterator_tag
reverse_bidirectional_iterator ·
reverse_iterator

# Операции над итераторами

++p Префиксная инкрементация: устанавливает итератор p на следующий элемент последовательности или на элемент, следующий за последним ("на один элемент вперед"); результатом является значение p+1

p++ Постфиксная инкрементация; устанавливает итератор p на следующий элемент последовательности или на элемент, следующий за последним ("на один элемент вперед"); результатом является значение p (до инкрементации)

--p  Префиксная декрементация: устанавливает итератор p на предыдущий элемент ("на один элемент назад"); результатом является значение p-1

p++ Постфиксная декрементация: устанавливает итератор p на предыдущий элемент ("на один элемент назад"); результатом является значение p (до декрементации)

*p  Доступ (разыменование): значение *p относится к элементу, на который указывает итератор p

p [x] Доступ (индексирование): значение p [x] относится к элементу, на который указывает итератор p+x; эквивалент выражения  * (p+x)

p->b  Доступ (обращение  к члену); эквивалент выражения (*p) .т

p==q  Сравнение: истина, если итераторы p и q указывают на один и тот же элемент или оба указывают на элемент, следующий за последним

p != q Неравенство: ! (p==q)

p<q Указывает ли итератор p на элемент, расположенный до элемента, на который указывает итератор q?

p<=q p<q || p==q

p>q Указывает ли итератор p на элемент, расположенный после элемента, на который указывает, итератор q?

p>=q p>q || p==q

p +=a Вперед на a элементов: устанавливает итератор p на a-й элемент, считая вперед от элемента, на который он ссылается в данный момент

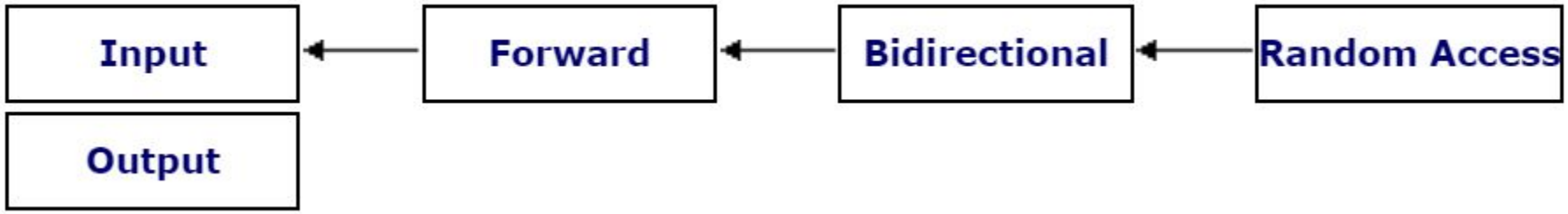
p -=a Вперед на -a элементов: устанавливает итератор p на a-й элемент, считая назад от элемента, на который он ссылается в данный момент

q= p + n Итератор q ссылается на n-й элемент считая вперед от элемента, на который ссылается итератор p

q= p - n Итератор q ссылается n-й элемент, считая назад от элемента, на который ссылается итератор p; после его выполнения q+n==p

advance (p,n) Перемещение вперед: аналог выражения p+=n; функцию advance() можно использовать, даже если итератор p не является итератором произвольного доступа

x = difference (p,q) Разность: аналог выражения q-p




Итераторы **ввода** и **вывода** являются наиболее ограниченными типами итераторов: они могут выполнять последовательные однопроходные операции ввода или вывода.

Прямые (**Forward**) итераторы имеют все функциональные возможности **входных** итераторов и - если они не являются константными итераторами – поддерживают также функциональные возможности **выходных** итераторов , хотя они ограничены одним направлением, в котором выполняется итерация в диапазоне (вперед). Все стандартные контейнеры поддерживают, по крайней мере, типы Forward - итератора.

**Двунаправленные** итераторы похожи на Forward- итераторы, но также могут действовать в обратном направлении. Т.е., осуществлять чтение и запись элементов контейнера в прямом и обратном направлениях

**Итераторы с произвольным доступом** реализуют все функциональные возможности двунаправленных итераторов, а также имеют возможность обращаться к диапазонам **не последовательно**: удаленные элементы могут быть доступны непосредственно путем применения значения смещения к итератору без перебора всех элементов между ними. Эти итераторы имеют аналогичную функциональность со стандартными указателями ( указатели являются итераторами этой категории ).

**input iterator** - Можем перемещаться вперед с помощью оператора ++ и считывать каждый элемент только один раз с помощью оператора *. Итераторы можно сравнивать с помощью операторов == и !=. Этот вид итераторов реализован в классе istream

**output iterator** - Можем перемещаться вперед с помощью оператора ++ и записывать каждый элемент только один раз с помощью оператора *. Этот вид итераторов реализован в классе ostream

**forward iterator** - Можем перемещаться вперед, применяя оператор ++ повторно, а также считывать и записывать элементы (если они не константные), с помощью оператора *. Если итератор указывает на объект класса, то для доступа к его члену можно использовать оператор ->

**bidirectional iterator** - Можем перемещаться вперед ( используя оператор ++) и назад (используя оператор -- ), а также считывать и записывать элементы (если они не константные) с помощью оператора *. Этот вид итераторов реализован в классах list, map и set

**randomaccess iterator** - Можем перемещаться вперед (с помощью операторов ++ и +=) и назад (с помощью операторов -- и -=), а также считывать и записывать элементы (если они не константные) с помощью оператора * или [ ]. Мы можем применять индексацию, добавлять к итератору произвольного доступа целое число с помощью оператора +, а также вычитать из него целое число с помощью итератора -. Можем вычислить расстояние между двумя итераторами произвольного доступа, установленными на одну и ту же последовательность, вычитая один из другого. Итераторы произвольного доступа можно сравнивать с помощью операторов <, <=, > и >=. Этот вид итераторов реализован в классе vector

```
template<class C, class T, class Dist = ptrdiff_t>
struct iterator {
    typedef C       iterator_category;
    typedef T       value_type;
    typedef Dist    distance_type;
};
```

Категории итераторов <u>не являются классами</u>, эту иерархию нельзя считать иерархией классов, реализованной с помощью наследования.

# Примеры реализаций

```
#define     _STRING_ITERATOR(ptr)        iterator(ptr, this)


iterator begin()
{       // return iterator for beginning of mutable sequence
     return (_STRING_ITERATOR( this->_Myptr() ) );
}
end()
{      // return iterator for end of mutable sequence
     return (_STRING_ITERATOR(this->_Myptr() + this->_Mysize));
}
value_type *_Myptr()
{    // determine current pointer to buffer for mutable string
     return ( this->_BUF_SIZE <=   this->_Myres  ?
                          _STD addressof ( *this->_Bx._Ptr)
                                                  : this->_Bx._Buf);
}
```

Итераторы можно определять отдельными переменными, а можно использовать анонимные непосредственно в качестве аргументов при вызове.

```
vector<int> v(3);
istream_iterator<int> Read(cin);// входной итератор
istream_iterator<int> end;          // итератор конца потока
  copy (Read, end, inserter(v, v.begin() ) ); // вставка в вектор

 //  вставка в вектор без предварительного объявления  итераторов
copy(     istream_iterator<int>(cin),   // входной итератор
            istream_iterator<int>(),       // итератор конца потока
            inserter (v, v.begin()));        // вставка в вектор  (адаптер: inserter )

  // чтение последовательности строк из файла
ifstream infile("integers.out") ;    // входной файл
istream_iterator<int> is (infile) ; // входной итератор связан с файлом
istream_iterator<int> end;  // итератор конца потока

     vector<int> tt;
copy(is, end. inserter( tt, tt.begin() ));       // ввод чисел в вектор
```

Вспомогательные функции для итераторов

Для большего удобства при работе с итераторами в библиотеке реализованы две вспомогательные функции:

// передвинуть итератор

template <class InputIterator, class Distance>

void **advance** (InputIterator& i, Distance n):

// вычислить расстояние между итераторами

tempiate<class InputIterator>

typename iterator_traits<InputIterator>::difference_type **distance** (InputIterator & first, InputIterator last);

# <iterator>
# Примеры: advance

```
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
int main () {
     vector <int>  v;
  for (int i=0; i<10; i++)   {
     v.push_back (i*10);
    }

 vector<int>::iterator   it = v.begin();
   advance (it,5);
    cout << "The sixth element in v  is: " << *it << '\n';
 return 0;
}
```

**Output**:

The sixth element in v  is: 50

# <iterator>
# Примеры: distance

```
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
int main () {
  vector <int>  v;
  for (int i=0; i<10; i++) mylist.push_back (i*10);
 vector <int>::iterator first = v.begin();
 vector <int>::iterator last = v.end();
  std::cout << "The distance is: " <<  distance (first , last) << '\n';
  return 0;
}
```

**Output**:

The distance is: 10

# <iterator>
# Примеры: begin и end

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
 int foo[ ] = {10,20,30,40,50};
  std::vector<int> bar;
for (auto it =  begin (foo); it != std::end(foo);  ++it ){
    bar.push_back(*it);
 }
  std::cout << "bar contains:";
  for (auto it = std::begin( bar); it != std::end(bar); ++it )
    std::cout << ' ' << *it;
  std::cout << '\n';
 return 0;
}
```

**Output**:

bar contains: 10 20 30 40 50

# iterator_traits

```
#include <iostream>
#include <iterator>
#include <typeinfo>

int main() {
    typedef std::iterator_traits<int*> traits;
  if ( typeid (traits::iterator_category) == typeid (std::random_access_iterator_tag))
    std::cout << "int* is a random-access iterator"<<'\n';
  return 0;
}
```

**Output**:

int*  is a random-access iterator

# std::prev

```
#include <iostream>     // std::cout
#include <iterator>     // std::next
#include <list>         // std::list
#include <algorithm>    // std::for_each

int main () {
  std::list<int> mylist;
  for (int i=0; i<10; i++) mylist.push_back (i*10);

  std::cout << "The last element is " << *std::prev (mylist.end()) << '\n';

  return 0;
}
```

Output: The last element is 90

# std::next

```
#include <iostream>     // std::cout
#include <iterator>     // std::next
#include <list>         // std::list
#include <algorithm>    // std::for_each
int main () {
  std::list<int> mylist;
  for (int i=0; i<10; i++) mylist.push_back (i*10);

  std::cout << "mylist:";
  std::for_each (mylist.begin(),
                 std::next (mylist.begin(),5),
                 [ ](int x) {std::cout << ' ' << x;} ); // не пугайтесь – это лямбда )

  std::cout << '\n';
  return 0;
}
```

Output: mylist: 0 10 20 30 40

# iterator_traits

```
#include <iostream>
#include <iterator>
#include <vector>
#include <list>
using namespace std;

template< class it >
void   function( it   i1, it    i2 ) {
   iterator_traits<it>::iterator_category cat;
     cout << typeid( cat ).name( ) << endl;
   while ( i1 != i2 )     {
      iterator_traits<it>::value_type  x;
        x = *i1;
      cout << x << " ";
        i1++;
   };
   cout << endl;
};
```

```
int main( )
{
   vector<char> vc( 10,'a' );
   list<int> li( 10 );
   function( vc.begin( ), vc.end( ) );
   function( li.begin( ), li.end( ) );
}


/* Output:
struct std::random_access_iterator_tag
a a a a a a a a a a
struct std::bidirectional_iterator_tag
0 0 0 0 0 0 0 0 0 0
*/
```

# Создаем свой итератор

```
#include <iostream>
#include <iterator>
class MyIterator : public  std::iterator<std::input_iterator_tag, int> {
  int* p;
public:
  MyIterator(int* x) :p(x) { }
  MyIterator(const MyIterator& mit) : p (mit.p)  { }
  MyIterator& operator++() {++p;  return *this;}
  MyIterator operator++(int) {MyIterator tmp(*this); operator++(); return tmp;}
  bool operator==(const MyIterator& rhs) const {return p==rhs.p;}
  bool operator!=(const MyIterator& rhs) const {return p!=rhs.p;}
  int& operator*() {return *p;}
};
void   main () {
  int numbers[ ]={10,20,30,40,50};
  MyIterator from (numbers);
  MyIterator until( numbers+5);
  for (MyIterator it=from; it != until;  it++)
    std::cout << *it << ' ';    }
```

Output:

10 20 30 40 50

# Распределители памяти

Три стратегии управления памятью в C++:

1. Распределение общего назначения, или универсальное распределение может обеспечить блок памяти любого размера, который может запросить вызывающая программа (размер запроса, или размер блока). Такое распределение очень гибкое, но имеет ряд недостатков, основными из которых являются пониженная из-за необходимости выполнения большего количества работы производительность и фрагментация памяти, вызванная тем, что при постоянном выделении и освобождении блоков памяти разного размера образуется большое количество небольших по размеру несмежных участков свободной памяти.
2. Распределение фиксированного размера всегда выделяет блоки памяти одного и того же фиксированного размера. Очевидно, что такая стратегия менее гибкая, чем универсальная, но зато она работает существенно быстрее и не приводит к фрагментации памяти.
3. Третья важная стратегия, распределение со сборкой мусора, не полностью совместима с указателями C и C++, функциями типа malloc, new.

(по книге Саттера «Новые сложные задачи на C++»)

На практике мы часто сталкиваемся с комбинацией этих стратегий. Например, возможно, ваш диспетчер памяти использует схему общего назначения для всех запросов размером больше некоторого значения S, а в качестве оптимизации для всех запросов размером меньше S используется выделение блоков памяти фиксированного размера.

Обычно достаточно неудобно иметь отдельные области памяти для запросов размером 1 байт, 2 байта и так далее, так что большинство диспетчеров используют отдельные области для выделения блоков, размер которых кратен некоторому числу, скажем, 16 байтам.

Если вы запрашиваете блок размером 16 байтов, все отлично; но если вы запросите 17 байтов, то память будет выделена из области для 32-байтовых блоков, и 15 байтов памяти пропадут впустую. Это источник дополнительных расходов памяти.

# Выбор стратегии

В чем состоит отличие различных уровней управления памятью в контексте STL и типичных средах, в которых используются реализации этой библиотеки? Что можно сказать об их взаимоотношениях, как они взаимодействуют друг с другом и как между ними распределяются обязанности?

Имеется 4 возможных уровней управления памятью, каждый из которых может **скрывать** предыдущий уровень.

- Ядро операционной системы предоставляет базовые услуги по распределению памяти. Эта базовая стратегия распределения памяти и ее свойства могут изменяться от одной операционной системы к другой, и на этот уровень в наибольшей степени влияет используемое аппаратное обеспечение.
- Библиотека времени выполнения компилятора, используемая по умолчанию, содержит собственные средства работы с памятью, такие как оператор **new** в C++ или функция **malloc** в C.
- Стандартные контейнеры и распределители используют сервисы, предоставляемые компилятором и, в свою очередь, могут перекрывать их путем реализации собственных стратегий и оптимизаций.
- Пользовательские контейнеры и/или пользовательские распределители могут использовать любой из сервисов более низкого уровня.

# Распределитель по умолчанию (The default allocator)

```
template <class T>
class allocator {
    public:
typedef T* pointer;
typedef const T* const_pointer;
typedef T& reference;
typedef const T& const_reference;
typedef T value_type;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
allocator();
~allocator();
pointer address(reference x);
const_pointer const_address(const_reference x)
pointer allocate(size_type n);
void deallocate(pointer p);
size_type init_page_size();
size_type max_size();
}; // allocator

 // специализация типом void
class allocator< void> {
public:
typedef void* pointer;
allocator();
~allocator();
};
```

# Пример создания аллокатора

(взято на stackoverflow.com)

```
#include <cassert>
#include <memory>
#include <vector>
#include <deque>
class Arena {
public:
    Arena() { }
    ~Arena() {
        assert (m_allocations == 0);
    }
 void* allocate(std::size_t n) {
        if (n > m_available) {
            m_chunks.emplace_back(100500);
            m_available = m_chunks.back().size();
            m_memory = &m_chunks.back().front();
        }
```

```
        auto mem = m_memory;
        m_available -= n;
        m_memory += n;
        ++m_allocations;
        return mem;
    }
    void deallocate(void* p, std::size_t n) {
        --m_allocations;
        auto mem = (unsigned char*)p;
        if (mem + n == m_memory) {
            m_memory = mem;
            m_available += n;
        }
    }

private:
    std::deque<std::vector<unsigned char>> m_chunks;
    std::size_t m_available = 0;
    unsigned char* m_memory;
    int m_allocations = 0;
};  // конец класса Arena
```

Предполагается, что контейнеры обязаны обращаться к аллокатору не напрямую, а через шаблон std::allocator_traits, который предоставляет значения по-умолчанию, такие как typedef T* pointer;

**std::allocator_traits <T>**

allocator_traits шаблон класса обеспечивает стандартный интерфейс к различным свойствам распределителей.

Стандартные контейнеры и другие стандартные компоненты библиотеки обращаются к распределителям через этот шаблон.

Это позволяет использовать любой тип класса в качестве распределителя, если предоставленный пользователем allocator_traits реализует все необходимые функциональные возможности.

```
template <class T>
struct ArenaAllocator {
    using value_type = T;
    using Traits = std::allocator_traits<ArenaAllocator<T>>;
    explicit ArenaAllocator(Arena& arena) : m_arena(  &arena ) { }
    template<class U> ArenaAllocator( const ArenaAllocator<U>&  other) :
       m_arena(other.m_arena) { }
    T* allocate( std::size_t n) { return (T*) m_arena->allocate(n * sizeof(T) ); }
    void deallocate(T* p, std::size_t n) { m_arena->deallocate(p, n * sizeof(T)); }

    // требуется в VC++ и libstdc++
    template<class U, class... Args> void construct (U* p, Args&&... args) {
       std::allocator<T>().construct(p, std::forward<Args>(args)...);
     }
    template<class U> void destroy (U* p) {    std::allocator<T>().destroy ( p);    }
    template<class U> struct rebind {  using other = ArenaAllocator<U>; };

    Arena* m_arena;  // член – данных (указатель на начало блока хранения)
};
```

Операторы сравнения:

```
template<class T, class U> bool operator==(const ArenaAllocator<T>& lhs, const
    ArenaAllocator<U>& rhs) { return lhs.m_arena == rhs.m_arena; }

template<class T, class U> bool operator!=(const ArenaAllocator<T>& lhs, const
    ArenaAllocator<U>& rhs) { return !(lhs == rhs); }
```

```
  // применение с разными  контейнерами
#include <deque>
#include <functional>
#include <list>
#include <map>
#include <memory>
#include <set>
#include <string>
#include <unordered_set>
#include <unordered_map>
#include <vector>

using a_string = std::basic_string<char, std::char_traits<char>,
    ArenaAllocator<char>>;
template <class T> using a_vector = std::vector<T, ArenaAllocator<T>>;
template <class T> using a_deque= std::deque<T, ArenaAllocator<T>>;
template <class T> using a_list = std::list<T, ArenaAllocator<T>>;
```

```
template <class K> using a_set = std::set<K, std::less<K>, ArenaAllocator<K>>;

template <class K, class V> using a_map = std::map<K, V, std::less<K>,
    ArenaAllocator<std::pair<const K, V>>>;

template <class K> using a_unordered_set = std::unordered_set<K,
    std::hash<K>, std::equal_to<K>, ArenaAllocator<K>>;

template <class K, class V> using a_unordered_map = std::unordered_map<K,
    std::hash<K>, std::equal_to<K>, ArenaAllocator<std::pair<const K, V>>>;

struct X { };
```

```
int main() {
    Arena arena;
    ArenaAllocator<char> arena_allocator( arena);

    a_string s_empty (arena_allocator);
    a_string s_123("123", arena_allocator);

    a_vector<int> v_int ( {1, 2, 3}, arena_allocator);
    a_vector<X> v_x(42, X{ }, arena_allocator);
    a_vector<a_string> v_str ( {s_empty, s_123}, arena_allocator);
    a_vector<a_string> v_str_copy(v_str, arena_allocator);
    a_deque<int> d_int({1, 2, 3}, arena_allocator);
    a_list<int> l_int({1, 2, 3}, arena_allocator);
    a_set<int> s_int({1, 2, 3}, std::less<int>{}, arena_allocator);
    a_map<a_string, int> m_str_int(arena_allocator);
    a_unordered_set<int> us_int(arena_allocator);

    auto p = std::allocate_shared<int>(arena_allocator, 123);
}
```

# Виртуализация функций – не членов класса.

По книги С. Мейерса «Наиболее эффективное использование С++» (пр. 25)

```
#include <iostream>
#include <list>
using namespace std;
class NLComponent { public:
    virtual ostream& operator<< (ostream& str) const = 0;  // проба
};

class TextBlock: public NLComponent { public:
    virtual ostream& operator<< (ostream& str) const;
} ;
class Graphic: public NLComponent { public:
  virtual ostream& operator<< (ostream& str) const;
} ;
```

```
class NewsLetter {// газета: текст и иллюстрации
public:
    NewsLetter (istream& str){
      while (str) {
         // Добавить новое сообщение
       }
      }
private:
    list<NLComponent*> components;
} ;

int _tmain(int argc, _TCHAR* argv[])
{
TextBlock t; Graphic g;
t << cout;  // Вывести t в cout при помощи  виртуального operator<<
g << cout;// Вывести g в cout при помощи виртуального operator<<
return 0;
}
```

error LNK2001: unresolved external symbol "public: virtual class std::basic_ostream<char,struct std::char_traits<char> > & __thiscall TextBlock::operator<<(class std::basic_ostream<char,struct std::char_traits<char> > &)const " (??6TextBlock@@UBEAAV?$basic_ostream@DU?$char_traits@D@std@@@std@@AAV12@@Z)

Клиенты должны помещать объект потока **справа** от символа , что противоречит соглашению для операторов вывода.

Чтобы вернуться к обычному синтаксису, придется вывести operator из классов TextBlock и Graphic, но если делать это, нельзя будет объявлять его как виртуальный.

Альтернативный подход - объявить виртуальную функцию для вывода (например, функцию print) и определить ее в классах TextBlock и Graphic.

Но в этом случае синтаксис вывода объектов TextBlock и Graphic будет не совпадать с синтаксисом остальных типов языка, использующих operator в качестве оператора вывода.

Ни одно из предложенных решений не является удовлетворительным. Нужно, чтобы функция - не член класса вызывала operator<< , который вел бы себя **подобно** виртуальной функции, такой как **printf**.

Обратите внимание: описание того, **что** нужно, очень близко к описанию того, **как** это можно сделать. Необходимо определить обе функции operator<< и print и вызвать вторую из первой!

```
class NLComponent { public:
virtual ostream&  print(ostream& s) const = 0;
};
class TextBlock: public NLComponent { public:
  virtual ostream&  print(ostream& s) const {return s<<"TextBlock\n"; };
} ;
class Graphic: public NLComponent { public:
virtual ostream&  print(ostream&  s) const { s<<"Graph\n"; return s;};
} ;
inline
ostream& operator<<(ostream&  s, const  NLComponent& c) {
return c.print(s);
}

int _tmain(int argc, _TCHAR* argv[ ]) {
TextBlock t; Graphic g;
    cout<<t;
  operator<< ( cout,t1);// Вывести t в cout при помощи
   cout<<g<<g;
    return 0;
}
```

# Домашнее задание на неделю

**Проект 33**.

- С любого сайта (например - новостного) взять текст (три-четыре абзаца) и, использовав notepad, записать этот текст в файл.
- В главной функции, используя **файловый поток** прочитать этот файл (getline не использовать), распределяя слова прочитанной информации по двум классам – Заглавный (Title ) и Прописной (Uppercase ) (каждый из них наследует от базового класса Word), - в зависимости от качества первой буквы слова. Для каждого слова надо создать экземпляр определенного класса и положить его в любым известным способом организованное хранилище (БД).
- Создать еще одно хранилище того же типа и скопировать туда только объекты Заглавного типа.
- Распечатать информацию из второго хранилища **в новый файл** построчно.

# Контрольная работа

В полиморфной иерархии из классов – One, Two и Three, наследуемых друг от друга

для класса **Three**, в котором есть 2 члена данных:

```
list < vector <string * >* > _list;
```

**и**

```
double* d;
```

реализовать конструктор копирования для класса **Three**.