

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Стандартная библиотека C++

ВОПРОСЫ

- ◎ Какие контейнеры входят в состав STL?
- ◎ Назовите контейнеры, которые не обеспечивают доступ к своим элементам по индексу?
- ◎ Какой контейнер необходимо выбрать, если вам нужно необходимо быстро узнавать о присутствии или отсутствии определенного элемента в контейнере?

АЛГОРИТМЫ СТАНДАРТНОЙ БИБЛИОТЕКИ C++

АЛГОРИТМЫ STL

- ⊙ Алгоритмы предназначены для манипулирования элементами контейнера.
- ⊙ Любой алгоритм рассматривает содержимое контейнера как последовательность, задаваемую итераторами первого и следующего за последним элементов. Итераторы обеспечивают интерфейс между контейнерами и алгоритмами, благодаря чему и достигается гибкость и универсальность библиотеки STL.
- ⊙ Каждый алгоритм использует итераторы определенного типа.
- ⊙ Вместо менее функционального итератора можно передать алгоритму более функциональный, но не наоборот.
- ⊙ Все стандартные алгоритмы описаны в файле `algorithm`, в пространстве имен `std`.

АЛГОРИТМЫ STL

- ◎ Основные виды алгоритмов:
 - ◎ Математические (расчет сумм, произведений, генерация случайных значений)
 - ◎ Поиска (минимальное значение, поиск последовательности, подсчет числа значений)
 - ◎ Сортировки
 - ◎ Работы с последовательностями (объединение последовательностей, сравнения, обработки последовательности типовой операцией)

АЛГОРИТМЫ STL

STL - алгоритмы представляют набор готовых функций, которые могут быть применены к STL коллекциям и могут быть подразделены на три основных группы:

1) Функции для перебора всех членов коллекции и выполнения определенных действий над каждым из них:

`count, find, for_each, search, copy, swap, replace, transform, remove, unique, reverse, random_shuffle, partition` и др.

2) Функции для сортировки членов коллекции:

`sort, stable_sort, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, includes, min, max, min_element, max_element, lexicographical_compare` и др.

3) Функции для выполнения определенных арифметических действий над членами коллекции:

`accumulate, inner_product, partial_sum, adjacent_difference`

FOR_EACH, ACCUMULATE, TRANSFORM

- ◎ `for_each(iterator1, iterator2, function);`
- ◎ `list<int> L(..);`
`int sum = accumulate(L.begin(), L.end(),`
`0, [plus<int>()]);`
- ◎ `transform (first.begin(), first.end(),`
`second.begin(), op_increase);`
- ◎ `transform (first.begin(), first.end(),`
`second.begin(), result.begin(), op_sum);`

ПРИМЕР TRANSFORM

```
// transform algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int op_increase (int i) { return ++i; }
int op_sum (int i, int j) { return i+j; }

int main () {
    vector<int> first;
    vector<int> second;
    vector<int>::iterator it;

    // set some values:
    for (int i=1; i<6; i++) first.push_back (i*10); // first: 10 20 30 40 50

    second.resize(first.size()); // allocate space
    transform (first.begin(), first.end(), second.begin(), op_increase);
                                                // second: 11 21 31 41 51

    transform (first.begin(), first.end(), second.begin(), first.begin(), op_sum);
                                                // first: 21 41 61 81 101

    return 0;
}
```


АЛГОРИТМЫ ПОИСКА

- ◎ `find` (`iterator1`, `iterator2`, `value`)
 - ◎ возвращает итератор, указывающий на первый элемент, имеющий значение `value`
- ◎ `find_if` (`iterator1`, `iterator2`, `function`)
 - ◎ аналогично `find`, но возвращает итератор на элемент, при котором функция возвращает `true`.
- ◎ `binary_search` (`iterator1`, `iterator2`, `value`)
 - ◎ поиск в отсортированном по возрастанию списке с использованием бинарного поиска

АЛГОРИТМЫ СОРТИРОВКИ

- ◎ `sort(begin, end)`
- ◎ `partial_sort(begin, begin+N, end)`
 - ◎ выдает первые N элементов в отсортированной последовательности. Остальные – как попало
- ◎ `nth_element(begin, begin+N, end)`
 - ◎ получить один элемент в корректной n-й позиции, в которой он бы находился при полной сортировке всего диапазона.
- ◎ `partition(begin, end, function)`
 - ◎ разделить на 2 части: удовлетворяющие функции и не удовлетворяющие функции
- ◎ `stable_sort, stable_partition`
- ◎ Замечания
 - ◎ Опционально, все могут принимать функцию сравнения в качестве аргумента
 - ◎ `std::sort` быстрее чем `clib sort`

EQUAL, MISMATCH, LEXICOGRAPHICAL_COMPARE

- ⦿ Функции производят сравнение последовательностей значений
- ⦿ **equal** (first1, last1, first2);
 - ⦿ возвращает true если последовательности равны (==)
 - ⦿ возвращает false если разной длины
 - ⦿ сравнивает последовательность от first1 до last1 с последовательностью, начинающейся с first2
 - ⦿ Контейнеры могут быть различного типа.
- ⦿ **mismatch** (first1, last1, first2)
 - ⦿ возвращает указатель на первый элемент, который не равен соответствующему в последовательности
- ⦿ **lexicographical_compare** (first1, last1, first2, last2)
 - ⦿ возвращает true если первая последовательность лексикографически меньше, чем вторая.

FILL, FILL_N, GENERATE, GENERATE_N

- ⦿ Заполняют контейнеры
- ⦿ **fill** – меняет последовательность элементов (от iterator1 до iterator2) на значение value
 - ⦿ `fill (iterator1, iterator2, value);`
- ⦿ **fill_n** – меняет N элементов, начиная с указанного
 - ⦿ `fill_n (iterator1, quantity, value);`
- ⦿ **generate** – как fill, но для генерации вызывается функция, которая не принимает аргументов
 - ⦿ `generate (iterator1, iterator2, function);`
- ⦿ **generate_n**
 - ⦿ `generate (iterator1, quantity, value)`

SWAP, ITER_SWAP, SWAP_RANGES

- ◎ **swap** (element1, element2) – обмен двух элементов
 - ◎ `swap(a[0], a[1]);`
- ◎ **iter_swap** (iterator1, iterator2) - обмен элементов, на которые указывают итераторы
- ◎ **swap_ranges** (iterator1, iterator2, iterator3) - обмен элементами от iterator1 до iterator2 с элементами, начинающимися с iterator3

COPY_BACKWARD, MERGE, UNIQUE, REVERSE

- ◎ **copy_backward**(iterator1, iterator2, iterator3)
 - ◎ скопировать элементы от iterator1 до iterator2 в iterator3, но в обратном порядке.
- ◎ **merge**(first1, last1, first2, last2, result)
 - ◎ отрезки first1-last1 и first2-last2 должны быть отсортированы по возрастанию.
 - ◎ объединяет эти отрезки в result.
- ◎ **unique**(iter1, iter2) – удалить дубликаты из отсортированного списка и вернуть итератор на новый конец списка.
- ◎ **reverse**(iter1, iter2)

ПРИВЕДЕНИЕ ТИПОВ В C++

ПРИВЕДЕНИЕ ТИПОВ В С

- ⊙ Тип (выражение)
- ⊙ (тип) выражение

```
int a = 5;  
float b = 6.8;  
a = b;  
a = (int) b;
```

Недостатки:

- ⊙ неуправляемое преобразование (всё – во всё)
- ⊙ тяжело найти блоки преобразования в исходном тексте программы

ПРИВЕДЕНИЕ ТИПОВ В C++

- ◎ `const_cast`
- ◎ `dynamic_cast`
- ◎ `static_cast`
- ◎ `reinterpret_cast`

CONST_CAST

- ◎ Самое простое приведение типов.
- ◎ Убирает так называемые cv спецификаторы (cv qualifiers), то есть const и volatile. volatile встречается не очень часто, так что более известно как приведение типов, предназначенное для убирания const.
- ◎ Если приведение типов не удалось, выдается ошибка на этапе компиляции.
- ◎ **НО** если снимать const с переменной, которая изначально была const, то дальнейшее её изменение приведёт к undefined behaviour.

CONST_CAST

- ◎ Ответственность за неизменность параметра в функции несет программист

```
void f(int * p )
{
    cout << *p;
}
int main(int argc, char* argv[])
{
    const int * p = new int(5);
    f(p);
    delete p;
    return 0;
}
```



```
void f(int * p )
{
    cout << *p;
}
int main(int argc, char* argv[])
{
    const int * p = new int(5);
    f(const_cast <int *> (p));
    delete p;
    return 0;
}
```

error C2664: 'f' : cannot convert parameter 1 from 'const int *' to 'int *'

DYNAMIC_CAST

- ⦿ Безопасное приведение по иерархии наследования, в том числе и для виртуального наследования.

`dynamic_cast<deriv_class *>(base_class_ptr_expr)`

- ⦿ Используется RTTI (Runtime Type Information), чтобы привести один указатель на объект класса к другому указателю на объект класса.
- ⦿ Возможно провести:
 - ⦿ Понижающее преобразование – из базового в производный
 - ⦿ Повышающее преобразование – из производного в базовый
 - ⦿ Перекрестное преобразование – между производными классами с одним базовым или между базовыми одного производного

DYNAMIC_CAST

- ⊙ Классы должны быть полиморфными, то есть в базовом классе должна быть хотя бы одна виртуальная функция.
- ⊙ Если эти условие не соблюдено, ошибка возникнет на этапе компиляции.
- ⊙ Если приведение невозможно, то об этом станет ясно только на этапе выполнения программы и будет возвращен NULL.

`dynamic_cast`<deriv_class &>(base_class_ref_expr)

- ⊙ Работа со ссылками происходит почти как с указателями, но в случае ошибки во время исполнения будет выброшено исключение `bad_cast`.

STATIC_CAST

- ⦿ Может быть использован для приведения одного типа к другому.
- ⦿ Если это встроенные типы, то будут использованы встроенные в C++ правила их приведения.
- ⦿ Если это типы, определенные программистом, то будут использованы правила приведения, определенные программистом.
- ⦿ Если приведение не удалось, возникнет ошибка на этапе компиляции => при работе с полиморфными классами очень опасно, т.к. не проверяется работа в механизмами виртуализации.

STATIC_CAST

```
int k;  
float p=123.5;  
k=static_cast<int> (p);    // k присваивается значение 123  
class B {};  
class C: public B{};  
C *c = new C();  
B *bp = static_cast <B*> ( c );    //ничего хорошего  
B b;  
C &cp = static_cast <C&> ( b );    //ничего хорошего
```

REINTERPRET_CAST

- ⊙ Оператор `reinterpret_cast` используется для небезопасных преобразований, зависящих от реализации. С его помощью проводят преобразования между указателями разных типов, между интегральными типами и указателями. Действует как Си-приведение.
- ⊙ Считается, что вы лучше компилятора знаете как на самом деле обстоят дела, а он тихо подчиняется.
- ⊙ Не может быть приведено одно значение к другому значению. Обычно используется, чтобы привести **указатель к указателю, указатель к целому, целое к указателю**. Умеет также работать со ссылками.

REINTERPRET_CAST

- ◎ `reinterpret_cast<whatever *>(some *)`
- ◎ `reinterpret_cast<integer_expression>(some *)`
- ◎ `reinterpret_cast<whatever *>(integer_expression)`

- ◎ Чтобы использовать `reinterpret_cast` нужны очень и очень веские причины. Используется, например, при приведении указателей на функции.

ДИНАМИЧЕСКАЯ ИДЕНТИФИКАЦИЯ ТИПОВ (RUN-TIME TYPE IDENTIFICATION, RTTI)

- ◎ `typeid` (выражение)
- ◎ `typeid` (имя типа)
- ◎ Операция возвращает ссылку на объект `typeinfo`.
При ошибке порождается исключение `bad_typeid`.

```
#include <iostream>
#include <typeinfo>
void main()  {
char C;
double D;
cout << "type of C is " << typeid (C).name() << endl;
cout << "type of C+D is " << typeid(C+D).name() << endl;
}
```