Последовательные контейнеры

Контейнер	Описание
vector	Массив переменного размера. Быстрый произвольный доступ.
deque	Двусторонняя очередь. Быстрый произвольный доступ. Быстрая вставка в начало и конец
list	Двусвязный список. двунаправленный последовательный доступ. Быстрая вставка и удаление в любую позицию
forward_list	Односвязный список Быстрая вставка и удаление в любую позицию
array	Массив фиксированного размера. Быстрый произвольный доступ. не позволяет добавлять или удалять элементы
string	Специализированный контейнер, подобный вектору, содержащий символы. Быстрый произвольный доступ. Быстрая вставка в начало и конец

Конструкторы контейнеров

```
/* Конструктор по умолчанию, создающий пустой
контейнер */
Container c;

/* Создает контейнер c1 как копию c2 */
Container c1(c2);

/* Копирует элементы из диапазона,
обозначенного
   итераторами begin и end */
Container c(begin, end);

/* Списочная инициализация  */
Container c{a, b, c...}
```

```
string c;
string c2 = "1";
string c1(c2);

string c3(c2.begin(),
c2.end());

string c4{'a', 'b'};
```

assign

- Заменяет содержимое контейнера.
  - заменяет содержимое с count копии значения value
  - заменяет содержимое с копиями тех, кто в диапазоне [first, last)

```
void assign( size_type count, const T& value );

template< class InputIt >
void assign( InputIt first, InputIt last );
```

assign

```
vector<int> v1 = {7, 9};

vector<int> v2 = {1, 2 , 3, 4};

v1.assign(v2.begin(), v2.begin() + 2);
```

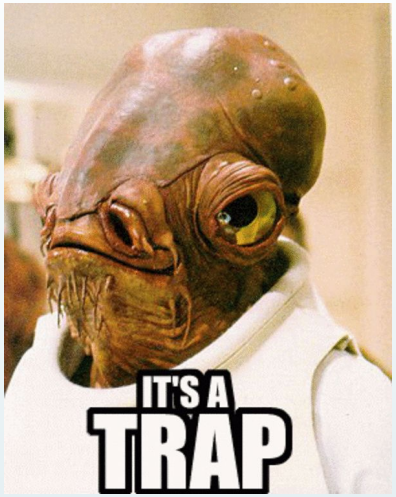
```
vector<char> v1;

v1.assign(5, 'a');

vector<int> v2 = {1, 2 , 3, 4};

v1.assign(v2.begin(), v2.begin() + 2);
```

insert

- Вставляет элементы в указанную позицию в контейнере.
  - Вставляет value перед элементом, на который указывает pos.
  - Вставляет count копий value перед элементом, на который указывает pos.
  - Вставляет элементы из диапазона [first, last) перед элементом, на который указывает pos.
  - Вставляет элементы из списка инициализации ilist.
- Вызывает реаллокацию если новый size() больше, чем старый capacity().
- Если новый size() больше, чем capacity(), все итераторы и указатели становятся нерабочими. В противном случае, нерабочими становятся только итераторы и указатели на элементы, идущие после вставленных.

insert

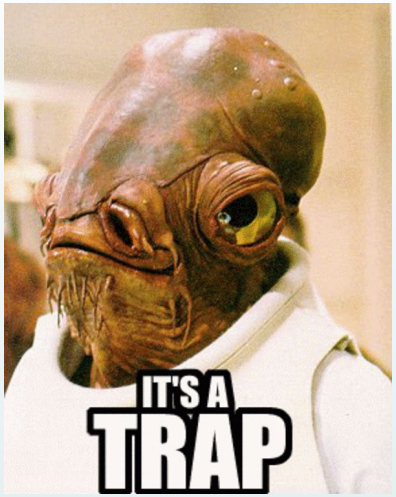
```
vector<int> container;

container.insert(container.begin(), 2);

vector<int>::iterator it;

it = container.begin();
it++;

container.insert(it, 200);


container.insert(it, 5, 200);
```



после выполнения этой операции it будет не валидным

insert

```
vector<int> container;

container.insert(container.begin(), 2);

vector<int>::iterator it;

it = container.begin();
it++;

container.insert(it, 200);

it = container.begin();

container.insert(it, 5, 200);
```



обновление итератора it

erase

- Удаляет указанные элементы из контейнера.
  - Удаляет элемент в позиции pos.
  - Удаляет элементы в диапазоне [first; last).

Итераторы и указатели к удалённым элементам и к элементам, идущим за ними, становятся нерабочими.

```
// до C++11
iterator erase( iterator pos );
//начиная с C++11
iterator erase( const_iterator pos );

// до C++11
iterator erase( iterator first, iterator last );
//начиная с C++11
iterator erase( const_iterator first, const_iterator last );
```

erase

Что будет выведено?

```
std::vector<int> c{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
c.erase(c.begin());

for (auto &i : c) {
    std::cout << i << " ";
}
std::cout << '\n';

c.erase(c.begin()+2, c.begin()+5);

for (auto &i : c) {
    std::cout << i << " ";
}
std::cout << '\n';
```

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 6 7 8 9
```

Аргументы по умолчанию

- Аргумент передаваемый в функцию по умолчанию
  - автоматически подставляется, если не задан явным образом

Аргументы по умолчанию

- Функции могут обладать параметрами, значения которых во многих случаях совпадают

```
string window(int w = 1, int h = 1, char background = ' ');
```

```
1.    window();
2.    window(3);
3.    window(7, 7);
4.    window(, , '#');
5.    window(4, 6, '*');
```

```
1.  w = 1, h = 1, background = ' ';
2.  w = 3, h = 1, background = ' ';
3.  w = 7, h = 7, background = ' ';
4.  Ошибка, нельзя пропустить
    аргументы слева
5.  w = 4, h = 6, background = '*';
```

Аргументы по умолчанию

- Функции могут обладать параметрами, значения которых во многих случаях совпадают

```
string window(int w = 1, int h = 1, char background = ' ');
```

```
window('?');
```

```
w = 63, h = 1, background = ' ';
```

char ‘?’ неявно преобразуется к типу string::size_type, который является беззнаковым целым. Символ ‘?’ имеет код 0x3F (63)

Аргументы по умолчанию

- Функция может объявляться многократно (но лучше делать это только один раз)
- У каждого параметра может быть свое значение по умолчанию:
  - определенное **однажды** в данной **области видимости**

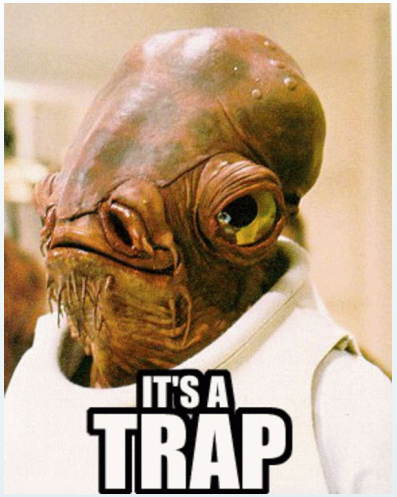
- Любое последующее объявление может добавлять значения по умолчанию для параметров, которые не имели значений по умолчанию

Аргументы по умолчанию

```
string window(int w, int h, char background = ' ');

//Ошибка, background уже имеет значение по умолчанию
string window(int w, int h, char background = '*');
```

```
string window(int w, int h, char background = ' ');

string window(int w, int h = 50, char background)
{
    //реализация
}
```

Что будет выведено?




```
class InitParam
{
public:
    int get_arg1()
    {
        return arg1;
    }
private:
    int arg1;
};

float arg2;

void defParamFunc(float x = arg2)
{
    cout << x << endl;
}
```

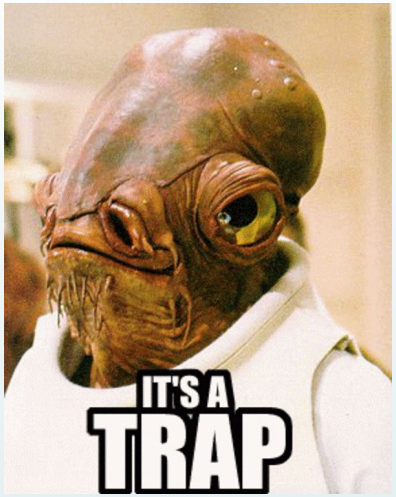
```
int main()
{
defParamFunc();
InitParam param;
defParamFunc(param.get_arg1());

}
```

Constexpr (C++11)

- Позволяет создавать объекты, которые будут рассчитаны на этапе компиляции

- constexpr - функция:
  - если значение параметров возможно посчитать на этапе компиляции, то возвращаемое значение также должно посчитаться на этапе компиляции
  - если значение хотя бы одного параметра будет неизвестно на этапе компиляции, то функция будет запущена в runtime (ошибки компиляции не будет)
- constexpr - переменная
  - создание константы
  - значение должно быть известно на этапе компиляции

Что произойдет?




```
int sum (int a, int b)
{
    return a + b;
}

constexpr int new_sum (int a, int b)
{
    return a + b;
}

void func()
{
    constexpr int a1 = new_sum (5, 12);
    constexpr int a2 = sum (5, 12);
    int a3 = new_sum (5, 12);
    int a4 = sum (5, 12);
}
```

OK: constexpr-переменная

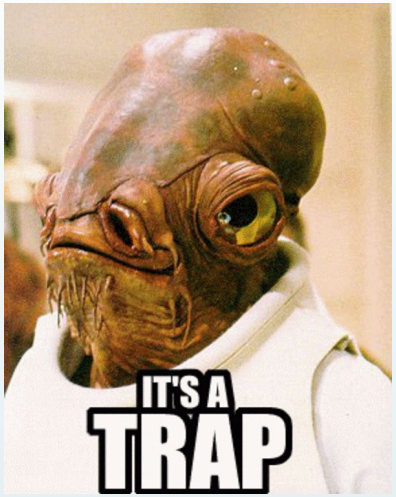
ошибка: функция sum не является constexp-выражением

OK: функция будет вызвана на этапе компиляции

OK

Что произойдет?

```
constexpr int inc (int a)
{
    return a + 1;
}

void func()
{
    int a = inc (3);
    constexpr int b = inc (a);
}
```



ошибка: a не является constexpr-выражением, из-за чего возвращаемое значение не имеет спецификатор constexpr

Constexpr-переменная

- Тип constexpr должен быть литеральным:
  - Скалярный тип
  - Указатель
  - Массив скалярных типов

Класс, который удовлетворяет следующим условиям:

- Имеет деструктор по умолчанию
- Все нестатические члены класса должны быть литеральными типами
- Класс должен иметь хотя бы один constexpr-конструктор (но не конструктор копирования и перемещения) или не иметь конструкторов вовсе

Скалярный тип - это тип, который имеет встроенные функции для оператора сложения без перегрузок (арифметика, указатель, указатель элемента, перечисление и std::nullptr_t).

Constexpr-переменная

- constexpr-переменная должна удовлетворять следующим условиям:
  - Ее тип должен быть литеральным
  - Ей должно быть сразу присвоено значение или вызван constexpr-конструктор
  - Параметры конструктора или присвоенное значение могут содержать только литералы или constexpr-переменные и constexpr-функции

Constexpr-функция

- Она не может быть виртуальной (virtual)
- Она должна возвращать литеральный тип (void вернуть нельзя*)
- Все параметры должны иметь литеральный тип
- Тело функции должно содержать только следующее:
  - static_assert
  - typedef или using, которые объявляют все типы, кроме классов и перечислений (enum)
  - using для указания видимости имен или пространств имен (namespace)
  - Ровно один return, который может содержать только литералы или constexpr-переменные и constexpr-функции

Указатель на функцию

Exercise: any_of_odd

Реализуйте эту функцию, которая проверяет наличие каких-либо нечетных элементов в последовательности.

```
// Возвращает true если хотя бы один элемент нечетный
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {

  // реализация

}
```

```
int main() {
  List<int> list;
  cout << any_of_odd(list.begin(), list.end());
}
```

Solution: any_of_odd

Реализуйте эту функцию, которая проверяет наличие каких-либо нечетных элементов в последовательности.

```
// Возвращает true если хотя бы один элемент нечетный
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  for (Iter_type it = begin; it != end; ++it) {
    if (*it % 2 != 0) { return true; }
  }
  return false;
}
```

```
int main() {
  List<int> list; // заполнен числами
  cout << any_of_odd(list.begin(), list.end());
}
```

Функция any_of

- Если захотим реализовать функцию, которая проверяет наличие четных объектов.
  - Получим дублирование кода!
- Единственный кусок кода который нужно изменить это проверку на четность/нечетность.

```
if (*it % 2 != 0) { return true; }
```

Избавление от дублирования

Как не нужно

```
int times2(int x) {
  return x * 2;
}

int times3(int x) {
  return x * 3;
}

int times4(int x) {
  return x * 4;
}

int main() {
  cout << times2(42);
  cout << times3(42);
  cout << times4(42);
}
```

Как нужно

```
int times(int x, int n) {
  return x * n;
}

int main() {
  times(42, 2);
  times(42, 3);
  times(42, 4);

  for (int i=0; i<10; ++i) {
    cout << times(42, i);
  }

}
```

Функция как параметр

```
bool is_even(int x);
bool is_odd(int x);
bool is_prime(int x);

template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end, ____ fn) {

  for (Iter_type it = begin; it != end; ++it) {
    if ( fn(*it) ) { return true; }
  }

  return false;
}

int main() {
  List<int> list; // Fill with numbers
  cout << any_of(list.begin(), list.end(), is_prime);
}
```

Функция как параметр

проверка возвращаемого результата

Передаем нужную функцию.

Указатель на функцию

- Содержит адрес функции.
- Указатель имеет определенный тип
- Тип указателя на функцию определяется типом возвращаемого значения и списком параметров

fn указывает на функцию, получающую два параметра и возвращающую int

```
int max(int x, int y) { return x > y ? x : y; }
int min(int x, int y) { return x < y ? x : y; }

int (*fn)(int, int) = max; // fn указывает на max
fn = min; // fn указывает на min
cout << fn(2, 5); // используется min, выведется 2
```

Объявление

- Объявление читается “изнутри наружу”.
- Постфиксные операторы ([], ()) имеют более высокий приоритет чем префиксные (*, &).
- Например: `double (*func)(int)`

```
double (*func)(int)   func это
double (*func)(int)   указатель на
double (*func)(int)   функцию
double (*func)(int)   принимающую int
double (*func)(int)   и возвращающую double
```

Указатель на ф-цию any_of

"Переменная содержащая ф-цию"

```
bool is_odd(int x) {
  return x % 2 != 0;
}
```

fn указатель на функцию принимающую int и возвращающую bool

```
// возвращает true в случае если fn вернет true
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end,
            bool (*fn)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (fn(*it)) { return true; }
  }
  return false;
}
```

```
int main() {
  List<int> list;
  cout << any_of(list.begin(), list.end(), is_odd);
}
```

Тип функции is_odd должен совпадать с типом fn.

Указатель на перегруженные функции

```
void func1(int *);
void func1(unsigned int);
```

```
void   (*ptr1_func)(unsigned int) = func;

void   (*ptr2_func)(int) = func;

double (*ptr3_func)(int *) = func;
```

OK

Ошибка: нет версии с такими параметрами

Ошибка: тип возвращаемого значения не совпадает

Предикаты

- Предикат - функция возвращая true или false в зависимости от входных аргументов
  - `is_odd`, `is_even`, `is_prime`.

```
bool any_of(Iter_type begin, Iter_type end,
            bool (*pred)(int));
```

- Можем использовать любой унарный предикат как параметр функции.

Создание предикатов

- Что если хотим написать функцию для проверки на превышение значения?
  - Больше 0?

```
bool greater0(int x) {
  return x > 0;
}
```

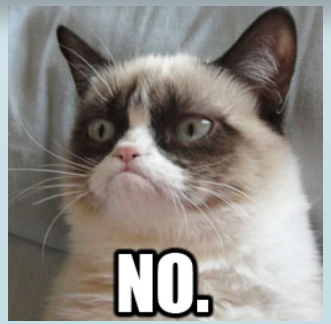
  - Больше 32?

```
bool greater32(int x) {
  return x > 32;
}
```

  - Больше 212?

```
bool greater212(int x) {
  return x > 212;
}
```

- Хороший подход?



Объекты первого класса

- Объекты первого класса могут…
  - …быть сохранены в переменную
  - …создаваться на этапе выполнения.
  - …передаваться в качестве аргумента в функцию.
  - …быть возвращены из функции как результат.
- В C++ функции не являются объектами первого класса.
  - Функции не могут в себе хранить информацию.
  - Не могут создаваться на этапе выполнения.

Функторы

- Функтор - класс, реализующий оператор()
  - Экземпляр функтора действует как функция
  - Может хранить состояние

- **Итератор** объект действующий как **указатель**.
- **Функтор** объект действующий как **функция**.

Реализация GreaterN

```
class GreaterN {
private:
  int threshold;

public:
  GreaterN(int threshold_in)
    : threshold(threshold_in) { }

  bool operator()(int n) const {
    return n > threshold;
  }
};
```

Хранит порог как член-класса.

Порог передается в конструктор

Перегрузка оператор ()

```
int main() {
  GreaterN g0(0);
  GreaterN g32(32);
  GreaterN g212(212);

  cout << g0(-5); // false
  cout << g0(3);  // true

  cout << g32(9);// false
  cout << g32(45); // true
}
```

```
int main() {
  List<int> list; //   GreaterN g0(0);
  cout << any_of(list.begin(), list.end(), g0);
  cout << any_of(list.begin(), list.end(), GreaterN(32));
}
```

Функтор как параметр

Реализуем функцию any_of как шаблон

```
// возвращает true в случае если fn вернет true
template <typename Iter_type, typename Predicate>
bool any_of(Iter_type begin, Iter_type end,
            Predicate pred) {
  for (Iter_type it = begin; it != end; ++it) {
    if (pred(*it)) { return true; }
  }
  return false;
}
```

```
int main() {
  List<int> list;
  GreaterN g0(0);
  cout << any_of(list.begin(), list.end(), g0);
  cout << any_of(list.begin(), list.end(), GreaterN(32));
}
```

Recall: max_element

```
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {

  Iter_type maxIt = begin;

  for (Iter_type it = begin; it != end; ++it) {
    if (*it > *maxIt) {
      maxIt = it;
    }
  }
  return maxIt;
}

int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;
}
```

Считаем первый элемент максимальным

Обход по итератору

Если находим элемент больше обновляем maxIt

С каким типом элементов можно использовать эту функцию?
С теми которые реализуют > operator.

Разыменование итератора для обращения к элементу

max_element для производных типов

- Использование функции max_element
- для контейнера Ducks не работает, как как оператор > не перегружен для класса Duck.
- Но мы все еще хотим найти "максимум" для Duck.
  - Имя ближе всего к концу алфавита.
  - Наибольшее количество утят.

Функция сравнивающая Duck по именам

```
int main() {
  vector<Duck> vec;
  max_element(vec.begin(), vec.end(), DuckNameLess());
}
```

Компараторы

- Компаратор - это функция, которая сравнивает два аргумента и возвращает true / false в зависимости от их порядка
- Обычно используют "less" компараторы, которые возвращают true, если первый аргумент меньше второго аргумента.
  - `DuckNameLess`, `DuckDucklingsLess`

```
class DuckNameLess {
public:
  bool operator()(const Duck &d1, const Duck &d2) const {
    return d1.getName() < d2.getName();
  }
};
```

max_element для Ducks

```
template <typename Iter_type, typename Comparator>
Iter_type max_element(Iter_type begin, Iter_type end,
                      Comparator less) {
  Iter_type maxIt = begin;
  for (Iter_type it = begin; it != end; ++it) {
    if (less(*maxIt,*it)) {
      maxIt = it;
    }
  }
  return maxIt;
}

int main() {
  vector<Duck> vec;
  max_element(vec.begin(), vec.end(), DuckNameLess());
}
```

for_each

- Различные функции выполняют операции применимые к каждому элементу в последовательности

```
// функция func применяется для каждого элемента
// последовательности [begin, end)

template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;
}
```

Вывод каждого элемента for_each

```
template <typename T>
class Printer {
public:
  void operator()(const T &n) const {
    cout << n;
  }
};

template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;
}

int main() {
  List<int> list;
  for_each(list.begin(), list.end(), Printer());
}
```

Exercise: Printer

- Реализуйте функцию таким образом чтобы она могла работать с любым потоком

```
template <typename T>
class Printer {

public:
  void operator()(const T &x) const { os << x; }
};


int main() {
  List<int> list; // Fill with numbers
  ofstream fout("list.out");
  for_each(list.begin(), list.end(), Printer(fout));
}
```

Solution: Printer

```
template <typename T>
class Printer {
private:
  ostream &os;
public:
  Printer(ostream &os_in) : os(os_in) { }

  void operator()(const T &x) const { os << x; }
};

int main() {
  List<int> list; // Fill with numbers
  ofstream fout("list.out");
  for_each(list.begin(), list.end(), Printer(fout));
}
```

Exercise: for_each поиск среднего

```
class Averager {
private:
  // поля
public:
  void operator()(double x) {
    // реализация
  }
  // возвращает среднее
  double get() const {
    // реализация
  }
};

int main() {
  List<int> list; //
  double avg = for_each(list.begin(), list.end(),
                        Averager()).get();
}
```

Solution: Averaging with for_each

```
class Averager {
private:
  int count; double total;
public:
  Averager() : count(0), total(0) {}
  void operator()(double x) {
    ++count;
    total += x;
  }
  // возвращает среднее
  double get() const {
    return total / count;
  }
};

int main() {
  List<int> list; // Fill with numbers
  double avg = for_each(list.begin(), list.end(),
                        Averager()).get();
}
```