# Глава 1. Аналитический обзор

## Оптимизация программ

Под оптимизацией программы подразумевают такие преобразования, в результате которых она становится более эффективной, т.е. становится более экономной по памяти или/и более быстрой по выполнению тех же функций, что и до оптимизации.

Оптимизация проводится по двум частным критериям: время выполнения программы и объём памяти, которую она использует. Но эти два критерия противоречат друг другу, т.к. чтобы уменьшить время работы, необходимо увеличить размер потребляемой памяти и наоборот. В этом случае программист из личных побуждений отдаёт предпочтение одному из критериев.

Частично оптимизацию программы может компилятор. Но в основном этот процесс зависит от квалификации программиста и невозможно дать алгоритм, оптимизирующий любую программу. Можно лишь обратить внимание на те аспекты, где скрыты резервы оптимизации и проиллюстрировать их на примерах.

Существует два подхода к оптимизации программ: «чистка» и перепрограммирование. Оба подхода имеют как достоинства, так и недостатки.

Первый подход заключается в исправлении очевидных неб­режностей в исходной программе. Его достоинство - данный метод требует мало времени. Однако повышение эффективности при этом обычно незначительно.

Второй подход состоит в переделке исходной программы. Можно переделать часть программы, которая, например, рас­ходует наибольшую часть времени. Этот подход обеспечивает обычно наилучший результат, но и самый дорогой. Он приемлем, если оптимизируемая программа подвергалась значительным изменениям.

Оптимизация кода - различные методы преобразования кода ради [улучшения его характеристик](https://www.viva64.com/ru/t/0077/) и повышения эффективности. Среди целей оптимизации можно указать уменьшения объема кода, объема используемой программой оперативной памяти, ускорение работы программы, уменьшение количества операций ввода вывода.

Главное из требований, которые обычно предъявляются к методу оптимизации - оптимизированная программа должна иметь тот же результат и побочные эффекты на том же наборе входных данных, что и неоптимизированная программа. Впрочем, это требование может и не играть особой роли, если выигрыш за счет использования оптимизации может быть сочтен более важным, чем последствия от изменения поведения программы.

## 1.2 Виды оптимизации

Оптимизация кода может проводиться, как и вручную, программистом, так и автоматизировано. В последнем случае оптимизатор может быть как отдельным программным средством, так и быть встроенным в компилятор (т.е. оптимизирующий компилятор). Кроме того, следует отметить, что современные процессоры могут оптимизировать порядок выполнения инструкций кода.

Существуют такие понятия как высокоуровневая и низкоуровневая оптимизация. Высокоуровневые оптимизации в большинстве проводятся программистом, который, оперируя абстрактными сущностями (функциями, процедурами, классами и т.д.) и представляя себе общую модель решения задачи, может оптимизировать дизайн системы. Оптимизации на уровне элементарных структурных блоков исходного кода (циклов, ветвлений и т.д.) тоже обычно относят к высокому уровню; некоторые выделяют их в отдельный ("средний") уровень. Низкоуровневая оптимизация производится на этапе превращения исходного кода в набор машинных команд, и зачастую именно этот этап подвергается автоматизации. Впрочем, программисты на ассемблере считают, что никакая машина не превзойдет в этом хорошего программиста (при этом все согласны, что плохой программист сделает еще хуже и машины).

## 1.3 Методы оптимизация кода C++

1. *Оптимизация работы с массивами*

При работе с массивами использование функций, предназначенных для определённых операций с ними, может существенно увеличить производительность. Так, например, при копировании массивов используется функция memcpy(копирование данных из одного блока памяти в другой) даст заметный прирост производительности по сравнению с поэлементным копированием в цикле for. Пример:

До оптимизации:

double arr1[CNT], arr2[CNT];

for (int i = 0; i< CNT; i++)

arr[i] = 0;

Оптимизированный код:

double arr1[CNT], arr2[CNT];

memcpy(arr1, arr2, sizeof(double) \* CNT);

1. *Замена возврата значения ссылочной переменной*

Оператор return создаёт временный экземпляр и копирует в него переданное в return значение, т.е. вызывается конструктор копий. Для возврата значений используется дополнительный аргумент ссылка.

1. *Префиксный и постфиксный оператор*

Префиксный оператор предпочтительнее постфиксного. При работе с примитивными типами, что префиксные, что постфиксные арифметические операции, с большей вероятностью, будут иметь одинаковые результаты производительности. Однако с объектами, операторы постфикса могут заставить объект создавать собственную копию, чтобы сохранить своё начальное значение(которое должно возвращаться в результате операции), а также вызвать возможный побочный эффект операции. Рассмотримпример:

class IntegerIncreaser {

int m\_Value;

public:

*/\* Postfix operator. \*/*

Integer Increaser operator++ (int) {

Integer Increaser tmp(\*this);

++m\_Value; return tmp; };

*/\* Prefix operator. \*/*

Integer Increaser operator++ () {

++m\_Value;

return \*this;

};

};

Поскольку операторы постфикса обязаны возвращать неизменённую версию значения, которое увеличивается (или уменьшается) – независимо от того, используется ли результат на самом деле – скорее всего, он сделает копию. Например, итераторы STL более эффективны при изменении с помощью префиксных операторов.

1. *Оптимизация передачи аргументов в функции*

Данный вид оптимизации делится на 2 метода:

1. Метод макрозамен – даёт выигрыш в производительности, благодаря двум составляющим:

* Время на передачу управления в функцию и возврат из неё – это время складывается из ­времени выполнения оператора *call*, а также времени на сохранении в стеке основных регистров процессора.
* Время на копирование данных из внешних переменных в локальные переменные.

Метод макрозамен заключается в преобразовании функции в макрос.

Макрос – участок кода, который так же, как и функция, имеет имя и аргументы, но в отличие от функции тело макроса вставляется во все места исходного кода, где он вызывается, при этом размер используемого кода программы вырастит пропорционально количеству вызовов макросов.

В языках C/C++ существует поддержка на уровне международного стандарта ключевого слова inline, которое информирует компилятор о необходимости замены функции на макрос. В этом случае, если компилятор обнаружит препятствие для такой замены, то ключевое слово будет проигнорировано.

Для того чтобы принудительно преобразовать функцию в web-макрос, в компилятор фирмы «Microsoft» включена поддержка ключевого слова \_\_forceinline, который игнорирует все условия потенциально препятствующие макрозамене.

Начиная со стандарта C++ 11, для модификации функции в макрос недостаточно простого добавления ключевого слова inlineили \_\_forceinline.

Тело модифицируемой функции необходимо полностью перенести в заголовочный файл.

Пример макрозамены с помощью inline:

До оптимизации:

double f(double x, double y) {

return (sin(x) \* cos(y)) / (x \* y);

}

int main(){

double k = 5, y = 7;

int z = f(k, y);

return 0;

}

После добавления ключевого слова inline:

Inline double f(double x, double y) {

return (sin(x) \* cos(y)) / (x \* y);

}

int main(){

double k = 5, y = 7;

int z = f(k, y);

return 0;

}

1. Альтернативой использования стандартных средств языка является прямая вставка кода. Данный способ потенциально позволяет получить максимальный выигрыш в производительности, однако на практике является наиболее трудоёмким.

Самой важной проблемой является возможный конфликт имён локальных переменных исходной функции, т.е. при прямой вставке может возникнуть необходимость изменить наименования переменных, модифицируемых функцией.

1. *Замена параметров значений на параметры ссылки*

Один из способов повышения производительности в функции – замена параметров значений на параметры ссылки. В этом случае в функцию передается не копия значений аргументов, а адреса передаваемых значений. Это даст выигрыш в том случае, если размер типа аргумента больше чем размер указателя на текущей платформе. В следующем примере оптимизация даст выигрыш в производительности для приложения на платформе Win 32, т.к. на этой платформе указатель является 32-х битным, а тип double является 64-х разрядным.

До оптимизации:

double f(double x, double y) {

return (sin(x) \* cos(y)) / (x \* y);

}

int main(){

double k = 5, y = 7;

int z = f(k, y);

return 0;

}

После оптимизации:

double f(double &x, double &y) {

return (sin(x) \* cos(y)) / (x \* y);

}

int main(){

double k = 5, y = 7;

int z = f(k, y);

return 0;

}

## 1.4 Выбор участка кода для оптимизации

При оптимизации кода вручную существует ещё одна проблема: нужно знать не только, каким образом проводить оптимизацию, но и в каком месте. На оптимизацию всей программы будет затрачено колоссальное количество ресурсов, поэтому рациональнее всего выбрать участок кода, который называется «критическим». Такой фрагмент называют узким местом или бутылочным горлышком, и для их определения используют специальные программы – профайлер, которые позволяют замерять время работы различных частей программы.

На практике оптимизация зачастую проводится после этапа "хаотического" программирование, поэтому представляет собой смесь из собственно оптимизации, рефакторинга и исправления ошибок: упрощение "причудливых" конструкций – вроде strlen(path.c\_str()), логических условий (a.x != 0 && a.x != 0) и т.п. Для такого рода оптимизации профайлеры не пригодны. Однако для обнаружения таких мест используются программы статического анализа – средства поиска семантических ошибок на основе глубоко анализа исходного кода - ведь, как видно из второго примера, неэффективный код может быть следствием ошибок (как, например, опечатки в данном примере - скорее всего, имелось ввиду a.x != 0 && a.y != 0). Хороший статический анализатор обнаружит подобный код, и выведет предупреждающее сообщение.

1.5 Вред и польза оптимизации

Во всём надо относиться рационально, и оптимизация – не исключение. Считается, что новичок в программировании, написавший код на ассемблере, будет работать медленнее, чем код сгенерированный компилятором.

К оптимизации, проводимой оптимизатором, почти нет претензий, причём иногда некоторые оптимизации являются фактическими стандартными и обязательными.

Однако следует понимать, что многочисленные сложные оптимизации на уровне машинного кода могут сильно замедлить процесс компиляции. Причем выигрыш от них может быть чрезвычайно мал по сравнению с оптимизациями общего дизайна системы.

Таким образом, не стоит забывать проводить оптимизацию кода, при этом использовать специальные программные средства, но это стоит делать в меру и с осторожностью.

## 1.6 STL – стандартная библиотека шаблонов

Под термином библиотека стандартных шаблонов (STL, **S**tandard **T**emplate **L**ibrary) понимают набор интерфейсов и компонентов, первоначально разработанных Александром Степановым, Менг Ли и другими сотрудниками AT&T BellLaboratories и Hewlett-PackardResearchLaboratories в начале 90-х годов (хотя и позже ещё весьма многие приложили руку к тому, что стало на сегодня стандартным компонентом C++). Далее библиотека STL перешла в собственность компании SGI, а также была включена как компонент в набор библиотек Boost. И наконец, библиотека STL вошла в стандарты C++ 1998 и 2003 годов (ISO/IEC 14882:1998 и ISO/IEC 14882:2003) и с тех пор считается одной из составных частей стандартной библиотек C++.

Первоначальной целью STL (это хорошо видно из хронологии комментариев в заголовочных файлах) было создание боле гибкой модели регулярных контейнеров по сравнению с массивами и обобщение на них некоторых широко используемых алгоритмов (таких как поиск, сортировка и некоторых других). Но затея оказалась плодотворнее первоначальных намерений, и была существенно расширена. STL вводит ряд понятий и структур данных, которые почти во всех случаях позволяют сильно упростить программный код.

Вводятся следующие категории понятий:

1. Контейнер – способ хранения набора объектов в памяти.
2. Итератор – средство доступа к содержимому отдельных объектов в контейнере.
3. Алгоритм – определение наиболее стандартных вычислительных процедур на контейнерах.
4. Адаптер – адаптация основных категорий для обеспечения наиболее употребляемых интерфейсов (таких как стек или очередь).
5. Функтор (функциональный объект) – сокрытие функции в объекте для использования её другими категориями.

## 1.7 STL.Контейнеры

Центральным понятием STL, вокруг которого крутится всё остальное, это контейнер (ещё используют термин коллекция). Контейнер — это набор некоторого количества обязательно однотипных элементов, упакованных в контейнер определённым образом. Простейшим прототипом контейнера в классическом языке C++ является массив. Тот способ, которым элементы упаковываются в контейнер, и определяет тип контейнера и особенности работы с элементами в таком контейнере.

Стандартная библиотека предоставляет различные типобезопасные контейнеры для хранения коллекций связанных объектов. При объявлении переменной контейнера указывается тип элементов, которые будет содержать контейнер. Контейнеры могут создаваться с использованием списков инициализаторов. Они содержат функции-члены для добавления и удаления элементов и выполнения других операций.

Итерация элементов в контейнере и доступ к отдельным элементам осуществляются с помощью итераторов. Вы можете использовать итераторы явно, с помощью их функций-членов и операторов, а также глобальных функций. Вы можете также использовать их неявно, например, с помощью цикла range-for. Итераторы для всех контейнеров стандартной библиотеки C++ имеют общий интерфейс, но каждый контейнер определяет собственные специализированные итераторы.

Библиотека контейнеров является универсальной коллекцией шаблонов классов и алгоритмов, позволяющих программистам легко реализовывать общие структуры данных, такие как очереди, списки и стеки.

Контейнер управляет выделяемой для его элементов памятью и предоставляет функции-члены для доступа к ним, либо непосредственного, либо через итераторы (объекты, обладающие схожими с указателями свойствами).

Большинство контейнеров обладают, по крайней мере, несколькими общими функциями-членами и общей функциональностью. Выбор оптимального контейнера для конкретного случая зависит не только от предоставляемой функциональности, но и от его эффективности при различных рабочих нагрузках.

Контейнеры можно разделить на три категории: последовательные контейнеры, ассоциативные контейнеры и контейнеры-адаптеры.

Последовательные контейнеры поддерживают указанный пользователем порядок вставляемых элементов. К ним относятся:

Контейнер «*vector»* ведет себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину. По этим и многим другим причинам контейнер «vector» является наиболее предпочтительным последовательным контейнером для большинства областей применения. Если вы сомневаетесь в выборе вида последовательного контейнера, начните с использования вектора.

Контейнер «*array»* обладает некоторыми преимуществами контейнера «vector», однако его длина не обладает такой гибкостью.

Контейнер «*deque»* (двусторонняя очередь) обеспечивает быструю вставку и удаление в начале и в конце контейнера. Он, как и контейнер «vector», обладает преимуществами прямого доступа и гибкой длины, но не обеспечивает связанное хранение.

Контейнер «*list»* — это двунаправленный список, который обеспечивает двунаправленный доступ, быструю вставку и удаления в любом месте контейнера, но не поддерживает прямой доступ к элементам контейнера.

В ассоциативных контейнерах элементы вставляются в предварительно определенном порядке — например, с сортировкой по возрастанию. Также доступны неупорядоченные ассоциативные контейнеры. Ассоциативные контейнеры можно объединить в два подмножества: сопоставления («set») и наборы («map»).

Контейнер «map», который иногда называют словарем, состоит из пар "ключ-значение". Ключ используется для упорядочивания последовательности, а значение связано с ключом. Например, «map» может содержать ключи, представляющие каждое уникальное ключевое слово в тексте, и соответствующие значения, которые обозначают количество повторений каждого слова в тексте. «map» — это неупорядоченная версия unordered\_map.

«set» — это контейнер уникальных элементов, упорядоченных по возрастанию. Каждое его значение также является и ключом. Set — это неупорядоченная версия unordered\_set.

Контейнеры «map» и «set» разрешают вставку только одного экземпляра ключа или элемента. Если необходимо включить несколько экземпляров элемента, следует использовать контейнер «multimap» или «multiset». Неупорядоченные версии этих контейнеров — unordered\_multimap и unordered\_multiset.

Упорядоченные контейнеры «map» и «set» поддерживают двунаправленные итераторы, а их неупорядоченный аналоги — итераторы с перебором в прямом направлении.

Контейнер-адаптер — это разновидность последовательного или ассоциативного контейнера, который ограничивает интерфейс для простоты и ясности. Контейнеры-адаптеры не поддерживают итераторы.

Контейнер «queue» соответствует семантике FIFO (первым поступил — первым обслужен). Первый элемент, которыйотправляется, то есть вставляется, в очередь, должен быть первым элементом, извлекаемым из очереди.

Контейнер priority\_queue упорядочен таким образом, что первым в очереди всегда оказывается элемент с наибольшим значением.

Контейнер stack соответствует семантике LIFO (последним поступил — первым обслужен). Последний элемент, отправленный в стек, становится первым извлекаемым элементом.

Поскольку контейнеры-адаптеры не поддерживают итераторы, их невозможно использовать в алгоритмах стандартной библиотеки C++.

## 1.8 Контейнер Vector

Использование класса vector является альтернативой применению встроенных массивов. Этот класс предоставляет гораздо больше возможностей, поэтому его использование предпочтительней.

Шаблон «vector» расположен в заголовочном файле <vector>. Как и все стандартные компоненты, он расположен в [пространстве имён](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BC%D1%91%D0%BD_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) std. Данный интерфейс эмулирует работу стандартного массива [C](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) (например, быстрый [произвольный доступ](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF) к элементам), а также некоторые дополнительные возможности, вроде автоматического изменения размера вектора при вставке или удалении элементов.

Элементы контейнера «vector» хранятся непрерывно, а значит, доступны не только через итераторы, но и через смещения, добавляемые к указателям на элементы ([data()](https://ru.cppreference.com/w/cpp/container/vector/data) или же, для непустых массивов, — &vect[0]). Это означает, что указатель на элемент вектора может передаваться в любую функцию, ожидающую указатель на элемент массива.

Хранилище вектора обрабатывается автоматически, расширяясь и сужаясь по мере необходимости. Векторы обычно занимают больше места, чем статические массивы, поскольку некоторое количество памяти выделяется про запас на обработку будущего роста. Таким образом, память для вектора требуется выделять не при каждой вставке элемента, а только после исчерпания резервов. Общий объём выделенной памяти можно получить с помощью функции [capacity()](https://ru.cppreference.com/w/cpp/container/vector/capacity). Резервная память может быть возвращена системе через вызов [shrink\_to\_fit()](https://ru.cppreference.com/w/cpp/container/vector/shrink_to_fit).

Перераспределения обычно являются дорогостоящими операциями в плане производительности. Функция [reserve()](https://ru.cppreference.com/w/cpp/container/vector/reserve) может использоваться для предварительного выделения памяти и устранения перераспределений, если заранее известно количество элементов.

В дополнение к функциям прямого доступа к элементам, описанным выше, элементы вектора можно получить посредством [итераторов](https://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Итераторы обычно используются парами, один из которых используется для указания текущей итерации, а второй служит для обозначения конца контейнера. Итераторы создаются при помощи таких стандартных методов как begin() и end(). Функция begin() возвращает указатель на первый элемент, а end() — на воображаемый несуществующий элемент, следующий за последним.

Вектор использует наиболее функционально богатый тип итераторов — RandomAccessIterator (итератор произвольного доступа), который позволяет обходить контейнер в любом порядке, а также изменять содержимое вектора в процессе обхода. Однако, при изменении вектора итератор может стать недействительным.

Пример подсчёта суммы элементов вектора при помощи итераторов:

#include"stdafx.h"

#include<vector>

#include<iterator>

#include<iostream>

using namespace std;

int main()

{

vector<int>the\_vector;

vector<int>::iteratorthe\_iterator;

for (int i = 0; i< 10; i++) {

the\_vector.push\_back(i);

}

int total = 0;

the\_iterator=the\_vector.begin();

while (the\_iterator!=the\_vector.end()) {

total += \*the\_iterator;

++the\_iterator;

}

cout<<"summa= "<< total <<endl;

system("pause");

return 0;

}

Вектор сохраняет определённый порядок его элементов, так, что при вставке нового элемента в начале или в середине вектора, последующие элементы перемещаются в обратном направлении с точки зрения их оператора присваивания и конструктора копии. Следовательно, ссылки и итераторы элементов после места вставки становятся недействительным. Пример:

#include"stdafx.h"

#include<vector>

#include<iterator>

#include<iostream>

using namespace std;

int main()

{

std::vector<int>v(2); // Создаём вектор, состоящий из двух элементов типа Int

// Создаём ссылки на оба элемента

int &first = v.front();

int &last = v.back();

v.insert(v.begin() + 1, 1, 1); // Добавляем новые элементы в середину вектора

int i = first; // Неопределённое поведение, если вставка вызвала перераспределение памяти

int j = last; // Неопределённое поведение, согласно стандарту C++

}

Сложность (эффективность) обычных операций над векторами следующая:

* Произвольный доступ — постоянная *O(1)*
* Вставка и удаление элементов в конце — амортизированная постоянная *O(1)*
* Вставка и удаление элементов — линейная по расстоянию до конца вектора *O(n)*

Плюсы и минусы вектора(vector):

* Как и все реализации [динамического массива](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), вектор не использует дополнительных структур данных, данные расположены в памяти рядом, за счёт чего они хорошо [кэшируются](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88).
* Вектор может быстро выделять память, необходимую для хранения конкретных данных. Это особенно полезно для хранения данных в списках, длина которых может быть не известна до создания списка, а удаление (за исключением, быть может, в конце) необходимо редко.
* Как и другие контейнеры STL, может содержать примитивные типы данных, сложные или определённые пользователем.
* Вектор разрешает [произвольный доступ](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF); то есть на элемент вектора можно ссылаться так же, как на элемент массива (по индексу). [Связанные списки](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA) и множества, напротив, не поддерживают произвольный доступ и арифметические операции над указателями.
* Удаление элемента из вектора или даже очистка вектора совершенно не обязательно освободит память, связанную с этим элементом. Это потому, что максимальный размер вектора с момента его создания является хорошей оценкой размера для нового вектора.
* Векторы являются неэффективными для вставки элементов в любые места, кроме конца. Такая операция имеет О(n) (см. [O-нотация](https://ru.wikipedia.org/wiki/O-%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D0%B8%D1%8F)) сложность по сравнению с O(1) для [связанных списков](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA). Удаление элемента из произвольного места также имеет сложность O(n) (необходимо сдвинуть к началу все элементы, располагающиеся после удаляемого, что в худшем случае даст n-1 перемещений). Это компенсируется скоростью доступа. Доступ к произвольному элементу вектора имеет сложность O(1) по сравнению с О(n) для [связанного списка](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA) и O(log n) для сбалансированного [двоичного дерева поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0).

## 1.9 Список List

Контейнер std::list широко применяется в системах управления данными ввиду высокой скорости добавления новых элементов.

Список представляет собой контейнер, который поддерживает быструю вставку и удаление элементов из любой позиции в контейнере. Быстрый произвольный доступ не поддерживается. Он реализован в виде двусвязного списка. В отличие от [std::forward\_list](https://ru.cppreference.com/w/cpp/container/forward_list) этот контейнер обеспечивает возможность двунаправленного итерирования, являясь при этом менее эффективным в отношении используемой памяти.

В отличие от других контейнеров для типа «list» не определена операция обращения по индексу или функция at(), которая выполняет похожую задачу.

Тем не менее для контейнера «list» можно использовать функции front() и back(), которые возвращают соответственно первый и последний элементы.

Чтобы обратиться к элементам, которые находятся в середине (после первого и до последнего элементов), придется выполнять перебор элементов с помощью циклов или итераторов:

#include"stdafx.h"

#include<list>

#include<iterator>

#include<iostream>

using namespace std;

int main()

{

std::list<int> numbers = { 1, 2, 3, 4, 5 };

int first = numbers.front(); // 1

int last = numbers.back(); // 5

// переборвцикле

for (int n : numbers)

std::cout<< n <<"\t";

std::cout<<std::endl;

// переборспомощьюитераторов

for (auto iter = numbers.begin(); iter!=numbers.end(); iter++)

{

std::cout<<\*iter<<"\t";

}

std::cout<<std::endl;

return 0;

}

## 1.10 Контейнер map

std::map — отсортированный ассоциативный контейнер, который содержит пары ключ-значение с неповторяющимися ключами. Контейнер map, очень похож на остальные контейнеры, такие как [vector](http://cppstudio.com/post/8453/), [list](http://cppstudio.com/post/8482/), [deque](http://cppstudio.com/post/8469/), но с небольшим отличием. В этот контейнер можно помещать сразу два значения.

Контейнер **map<>** (таблица, отображение):

* Содержит упорядоченные пары <ключ, значение>, где ключ и значение могут принадлежать к произвольным типам. Для типа ключа должна быть либо предопределена, либо определена пользователем операция сравнения;
* Элементы с любым значением ключа должны быть уникальны;
* Попытка добавить (метод **insert()**) к таблице новую пару с уже имеющимся значением ключа завершится неудачей;
* Операция добавления новой пары в таблицу возвращает пару типа **<итератор, bool>**, у которой второй компонент (логический **second**) указывает на успешность операции. Если он **true**, то первый компонент возвращаемого результата (**first**) даёт итератор добавленного элемента. Если же он **false**, то возвращается итератор существующего элемента с тем же ключом;
* Операции индексации таблицы ( **[ ]** или **at()** ) требуют в качестве ключа любое значение типа, определённого для ключа;
* Операция индексации **at()**, при задании ключа-параметра, отсутствующего в составе элементов таблицы, вызывает исключение;
* Напротив, операция индексации **[ ]**, при задании ключа-параметра, отсутствующего в составе элементов таблицы, исключение не вызывает. (наоборот, даже если индексация запрошена только по чтению, добавляет к контейнеру новый элемент  с требуемым значением ключа, но с нулевым полем значения);

Чтобы подключить map, нужно подключить заголовочный файл map и, собственно, простейший код:

#include<iostream>

#include<map>

using namespace std;

int main() {

map<string, int> m;

m["Вася"] = 0;

m["Петя"] = 1;

m["Федор"] = 2;

cout<< m["Вася"] <<"\n";

cout<< m["Петя"] <<"\n";

cout<< m["Федор"] <<"\n";

return 0;

}