КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 19 — Преобразование типов. Исключения.

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.12.15

Оглавление

Стандартная библиотека C++

Состав библиотеки

1) библиотека поддержки языка (Language support library)
2) библиотека диагностики (Diagnostics library)
3) блиотека общих утилит (General utilities library)
4) библиотека строк (Strings library)
5) библиотека локализации (Localization library)
6) библиотека контейнеров (Containers library)
7) библиотека итераторов (Iterators library)
8) библиотека алгоритмов (Algorithms library)
9) библиотека чисел (Numerics library)
10) библиотека ввода/вывода (Input/output library)
11) библиотека регулярных выражений (Regular expressions library)
12) библиотека атомарных операций (Atomic operations library)
13) библиотека поддержки потоков (Thread support library)

Библиотека поддержки языка

Предоставляет компоненты, которые требуются для определенных частей языка C++, такие как выделение памяти (**new**, **delete**) и обработка исключений (**try-catch**).

В **<cstddef>** определены макросы **NULL** и **offsetof**

```
offsetof(type, member-designator)
```

Типы: **ptrdiff_t**, **size_t**, **max_align_t**, **nullptr_t**

Заголовки **<limits>**, **<climits>** и **<cfloat>** предоставляют характеристики арифметических типов, котороые зависят от реализации.

Заголовок **<cstdint>** описывает разного рода арифметические типы, аналогично тому, как это описано в C-библиотеке.

Содержимое заголовка **<cstdlib>** такое же, как и заголовка **<stdlib.h>** из Стандартной библиотеки C за исключением некоторых деталей, специфических для C++.

Например, ф-ция **_Exit(int status)** завершает программу без выполнения каких-либо деструкторов для объектов автоматического, статического времени существования и времени существования потока, а также без вызова функций, зарегистрированных с пом. ф-ции **atexit()**.

То же самое и для **abort()**.

Подробности и отличия читаем в ISO/IEC 14882:2011, 2014, 2017, 2020.

Заголовок **<new>** определяет несколько функций, которые управляют распределением динамической памяти в программе. Он также определяет компоненты для сообщения об ошибках управления памятью разного типа.

Заголовок **<typeinfo>** определяет тип, связанный с информацией о типе, созданном реализацией.

Класс **type_info** описывает информацию о типе. Объекты этого класса эффективно хранят указатель на имя типа и закодированное значение, подходящее для сравнения двух типов на эквивалентность или порядок сортировки. Имена, правила кодирования и последовательность сортировки для типов стандартом не определяются и могут различаться в зависимости от программы.

Здесь также определяется два типа сообщений об ошибках идентификации динамического типа.

Класс **bad_cast** определяет тип объектов, создаваемых реализацией как исключения для сообщения о выполнении недопустимого выражения динамического приведения.

Класс **bad_typeid** определяет тип объектов, создаваемых реализацией как исключения для сообщения о нулевом указателе в выражении **typeid**.

Заголовок **<exception>** определяет несколько типов и функций, связанных с обработкой исключений в программе C++. В частности, определяются такие классы, как:

```
class exception
class bad_exception
class nested_exception
```

а также несколько функций для обработки исключений.

Заголовок **<initializer_list>** определяет один тип, в частности, **initializer_list**, который используется для реализации списков инициализаторов.

Список инициализации — это инициализация объекта или ссылки из списка инициализации в фигурных скобках. Такой инициализатор называется списком инициализаторов, а разделы инициализатора, разделенные запятыми, называются элементами списка инициализаторов. Список инициализаторов может быть пустым.

А также заголовки, предоставляющих поддержку времени выполнения

<csetjmp> — нелокальные переходы);

<csignal> — обработка сигналов);

<cstdalign> — выравнивание);

<cstdarg> — переменное количество аргументов);

<cstdbool> — **bool true false**);

<cstdlib> — среда выполнения **getenv()**, **system()**);

<ctime> — системные часы **clock()**, **time()**).

Все это обеспечивают дополнительную совместимость с кодом на C.

Библиотека диагностики

Обеспечивает согласованную структуру для сообщения об ошибках в программе C++, включая предопределенные классы исключений.

Классы и прочие компоненты в составе этой библиотеки могут использоваться для обнаружения и сообщения об определенных ошибках в программах на C++.

В модели ошибок, отраженной в этих классах, ошибки делятся на две широкие категории — логические ошибки и ошибки времени выполнения.

Отличительной чертой логических ошибок является то, что они возникают из-за ошибок во внутренней логике программы. Теоретически их можно предотвратить.

Напротив, ошибки времени выполнения возникают из-за событий, выходящих за рамки программы. Их нелегко предсказать заранее.

Заголовок **<stdexcept>** определяет несколько типов предопределенных исключений для сообщения об ошибках в программе C++.

Заголовок **<cassert>**, предоставляет макрос **assert** для документирования утверждений в программе на C++ и механизм для отключения проверок утверждений (то же самое, что и в C).

Заголовок **<cerrno>** описывает коды ошибок и его содержимое такое же, как и содержимое заголовка POSIX **<errno.h>**, за исключением того, что **errno** должен быть определен как макрос.

Для каждого потока существует отдельное значение **errno**.

В заголовке **<system_error>** описываются компоненты, которые могут использоваться для сообщения об ошибках, возникающих из операционной системы или других низкоуровневых прикладных программных интерфейсов. Эти компоненты не изменяют значение **errno**.

Библиотека общих утилит

Включает компоненты, используемые другими элементами библиотеки, такие как предопределенный распределитель памяти для управления динамической памятью (**`operator new`**, **`operator delete`**), и другие компоненты, используемые в качестве инфраструктуры в программах C++, такие как пары, кортежи (**`tuple`**), функциональные объекты и полезный временной функционал

`<utility>`	полезные компоненты (Utility components)
`<utility>`	пары (Pairs)
`<tuple>`	кортежи (Tuples)
`<bitset>`	последовательности битов фиксированного размера[1]
`<memory>` **`<cstdlib>`** **`<cstring>`**	память (Memory)
`<memory>`	умные указатели (Smart pointers)
`<functional>`	функциональные объекты (Function objects)
`<type_traits>`	признаки типов (Type traits)
`<ratio>`	рациональная арифметика времени компиляции[2]
`<chrono>` **`<ctime>`**	утилиты времени (Time utilities)
`<scoped_allocator>`	аллокаторы с заданной областью видимости (Scoped allocators)
`<typeindex>`	индексы типов (Type indexes)

1 Fixed-size sequences of bits
2 Compile-time rational arithmetic

Заголовок **<utility>** определяет несколько типов и шаблонов функций. Он также определяет шаблон **pair** и шаблоны функций, которые работают с объектами типа **pair**.

Пары — это объекты, которые могут содержать два значения разных типов. Например пары **ключ:значение**, где и **ключ** и **значение** могут быть, в принципе, объектами любого типа.

Также тут определены реляционные операторы общего типа — определения реляционных операторов **!=**, **>**, **<=** и **>=** .

Заголовок **<tuples>** описывает библиотеке кортежей, которая предоставляет тип **tuple** как шаблона класса **tuple**, который может быть создан с любым количеством аргументов.

Каждый аргумент шаблона определяет тип элемента в кортеже.

Следовательно, кортежи представляют собой разнородные коллекции значений фиксированного размера. Создание экземпляра кортежа с двумя аргументами аналогично созданию экземпляра пары с теми же двумя аргументами.

Заголовок **<memory>** определяет несколько типов и шаблонных функций, которые описывают свойства указателей и типов, подобных указателям, управляют памятью для контейнеров и других типов шаблонов и создают несколько объектов в неинициализированных буферах памяти.

Заголовок также определяет группу шаблонов т.н. умных указателей (Smart pointers) — шаблоны **unique_ptr**, **shared_ptr**, **weak_ptr** и различные функции шаблонов, которые работают с объектами этих типов.

Заголовки **<cstdlib>** и **<cstring>** предлставляют собой содержимое заголовков C <stdlib.h> и **<string.h>**.

Заголовок **<bitset>** определяет шаблон класса и несколько связанных функций для представления последовательностей битов фиксированного размера.

Тип функционального объекта — это тип объекта, который может иметь тип постфиксного выражения в вызове функции. Такой тип представляет собой указатель на функцию или тип класса, который имеет член **operator()**, или тип класса, который имеет преобразование в указатель на функцию. Это позволяет алгоритмическим шаблонам работать как с указателями на функции, так и с произвольными объектами, в отношении которых допустимо использовать оператор **()**.

В заголовке **<type_traits>** описываются *типажи* (черты/особенности типов — Type traits) — компоненты, используемые в программах на C++, особенно в шаблонах, для поддержки максимально широкого диапазона типов, оптимизации использования кода шаблона, обнаружения ошибок пользователя, связанных с типом, и преобразования типа во время компиляции.

Типаж включает признаки классификации типов, признаки проверки свойств типа и преобразования типов. Признаки классификации типов описывают полную классификацию всех возможных типов в C++ и указывают, к какой части этой классификации относится данный тип. Признаки проверки свойств типа позволяют проверять важные характеристики типов или их комбинаций при программировании шаблонов.

В заголовке **<ratio>** описывается библиотека рациональной арифметики, предоставляющая . шаблоны классов, которые точно представляют любое конечное рациональное число с числителем и знаменателем, представленными константами времени компиляции типа **intmax_t**.

Заголовок **<scoped_allocator>** определяет шаблон класса **scoped_allocator_adaptor** — это шаблон аллокатора (выделителя/распределителя), который выделяет два ресурса памяти — внешний, который будет использоваться контейнером, а также внутренний, который будет передаваться в конструктор каждого элемента в контейнере.

Этот адаптер создается с одним внешним, нулевым или несколькими внутренними ресурсами. Если создается экземпляр только с одним типом (внешним), используется один и тот же ресурс памяти и для контейнера и для каждого элемента в контейнере. Если сами элементы являются контейнерами, каждый из элементов этих контейнеров будет использовать соответсвующий аллокатор рекурсивно.

Библиотека строк

Обеспечивает поддержку для манипулирования текстом, представленным в виде последовательностей типа **char**, **char16_t**, **char32_t**, **wchar_t** и последовательностей любого другого символьного типа.

Библиотека предоставляет компоненты для управления последовательностями любого POD[3]-типа, не являющегося массивом. В контексте этой библиотеки такие типы называются символьными типами, а объекты символьных типов называются символьными объектами или просто символами. Заголовки описывают следующее:

- класс характеристик символа (character traits) — **<string>**
- классы строк (string classes) — **<string>**
- последовательности с завершающим нулем (C-строки) — **<cctype>**, **<cwctype>**, **<cstring>**, **<cwchar>**, **<cstdlib>**, **<cuchar>.**

3 Plain Old Data

Библиотека локализации

Обеспечивает расширенную поддержку интернационализации для обработки текста.

Библиотека контейнеров

Предоставляет возможности для организации информации в виде коллекций — массивы, вектора, списки, деки, стеки и очереди (последовательностные), множества и словари (ассоциативные).

Библиотека итераторов

Предоставляет возможности для упорядоченного доступа к элементам контейнеров.

Библиотека алгоритмов

Предоставляет возможности для выполнения часто используемых алгоритмов над контейнерами и их элементами.

Библиотека числовых данных

Предоставляет числовые алгоритмы и компоненты комплексных чисел, которые расширяют поддержку числовой обработки.

Компонент **`valarray`** обеспечивает поддержку одновременной параллельной обработки однородных операций на платформах, которые такую обработку поддерживают.

Компонент случайных чисел предоставляет средства для генерации псевдослучайных чисел.

Библиотека ввода/вывода

Предоставляет компоненты **`iostream`**, которые являются основным механизмом ввода и вывода программ на C++. Их можно использовать с другими элементами библиотеки, в частности, со строками, локалями и итераторами.

Библиотека регулярных выражений

Обеспечивает сопоставление и поиск регулярных выражений.

Библиотека атомарных операций

Обеспечивает более детальный параллельный доступ к совместно используемым данным, чем это возможно при использовании блокировок.

Библиотека поддержки потоков (threads)

Предоставляет компоненты для создания потоков и управления ими, включая примитивы взаимного исключения и межпоточного взаимодействия.

Стандартная библиотека C

Стандартная библиотека C++ также предоставляет возможности стандартной библиотеки C, настроенные соответствующим образом, чтобы обеспечить безопасную статическую типизацию.

Описание многих библиотечных функций зависит от стандартной библиотеки C для семантики этих функций.

В некоторых случаях сигнатуры, используемые в стандартной библиотеке C++, могут отличаться от сигнатур в стандартной библиотеке C, но поведение и требования к вызовам, включая любые предварительные условия, предполагаемые при использовании ISO C квалификатора **`restrict`**, остаются такими же, если не указано иное.

Класс initializer_list — Список инициализации

```
#include <initializer_list>
template <class T> class initializer_list;
```

Этот тип используется для доступа к значениям в списке инициализации C++, который представляет собой список элементов типа **const T**.

Объекты этого типа автоматически создаются компилятором из объявлений списка инициализации, который представляет собой список элементов, разделенных запятыми, заключенных в фигурные скобки:

```
std::initializer_list il = {10, 20, 30}; // тип объекта il - initializer_list
```

Этот шаблонный класс неявно не определен, и для доступа к нему должен быть включен заголовок **<initializer_list>**, даже если тип используется неявно.

Объекты класса **initializer_list** создаются автоматически, как если бы был выделен массив элементов типа **T**, при этом каждый из элементов в списке инициализируется копией соответствующего элемента в массиве с использованием любых необходимых не сужающих неявных преобразований.

Объект **initializer_list** ссылается на элементы этого массива, но не содержит их — копирование объекта с типом **initializer_list** создает другой объект, ссылающийся на те же базовые элементы, а не на их новые копии (ссылочная семантика).

Время жизни этого временного массива такое же, как у объекта с типом **initializer_list**.

Конструкторы, принимающие только один аргумент этого типа, представляют собой особый вид конструкторов, называемых конструктором из списка инициализаторов.

```
struct myclass {
  myclass(int,int);
  myclass(initializer_list<int>);
  /* definitions ... */
};
myclass foo {10, 20};  // calls initializer_list constructor
myclass bar (10, 20);  // calls first constructor
```

Конструктор

```
initializer_list() noexcept;
```

Создает пустой объект типа **initializer_list**.

Компилятор автоматически создает непустой объект этого типа шаблона класса всякий раз, когда необходимо передать или скопировать выражение списка инициализатора.

Это единственный способ установить значения объекта типа **initializer_list**.

```
#include <iostream>          // std::cout
#include <initializer_list>  // std::initializer_list
int main () {
  std::initializer_list<int> mylist;
  mylist = { 10, 20, 30 };
  std::cout << "mylist contains:";
  for (int x: mylist) std::cout << ' ' << x;
  std::cout << '\n';
  return 0;
}
```

Библиотека контейнеров

Предоставляет возможности для организации информации в виде коллекций (массивы, вектора, списки, деки, стеки, множества, очереди и отображения).

Контейнер — это специфическая структура данных, которая содержит данные, обычно в неограниченном количестве. У каждого типа контейнера есть ограничения на то, как фактически получать доступ, добавлять или удалять данные.

Контейнерные классы — это классы, которые в состоянии хранить в себе элементы различных типов данных. Почти все контейнерные классы реализованы как шаблонные и, таким образом, могут хранить данные любого типа. Основная идея шаблона состоит в создании родового класса, который определяется при создании объекта этого класса.

Классы контейнеров могут включать целые серии других объектов, которые, в свою очередь, тоже могут являться контейнерами.

Чтобы правильно подобрать контейнер для конкретного случая, очень важно правильно понимать различия разновидностей контейнеров. От этого в значительной степени зависит скорость работы кода и эффективность использования памяти.

Предоставляет две категории разновидностей классов контейнеров:

- последовательные (sequence containers)
- ассоциативные (associative containters).

Последовательные контейнеры — это упорядоченные коллекции, где каждый элемент занимает определенную позицию.

Позиция элемента зависит от места его вставки.

Контейнеры подразделяются на последовательные и ассоциативные.

Последовательные контейнеры

В последовательных контейнерах данные упорядочены. Однако, данные в таких контейнерах сами по себе не сортируются — для этого используются соответсвующие алгоритмы.

К последовательным контейнерам относятся:

- массив (array);
- дек (deque);
- однонаправленный список (forward list);
- список (list);
- вектор (vector),
- стек (stack)
- очередь (queue).

Массив (array)

Массивы представляют собой последовательные контейнеры фиксированного размера — они содержат определенное количество элементов, упорядоченных в строгой линейной последовательности. Массивы обладают произвольным доступом, что означает, что доступ к любому элементу с целочисленным индексом выполняется за постоянное время.

Вектор (vector)

Векторы — это последовательные контейнеры, представляющие массивы, размер которых может изменяться. Векторы обладают произвольным доступом, что означает, что доступ к любому элементу с целочисленным индексом выполняется за постоянное время (точно так же, как массив).

Однонаправленный список (Forward list)

Однонаправленные списки — это последовательные контейнеры, которые позволяют выполнять операции вставки и удаления в любом месте последовательности за постоянное время.

Список (list)

Списки представляют собой последовательные контейнеры, которые позволяют выполнять операции вставки и удаления с постоянным временем в любом месте последовательности и просмотр в обоих направлениях.

Стек (LIFO stack)

Стеки — это контейнеры-адаптеры, специально разработанный для работы в контексте LIFO (последним пришел - первым вышел), когда элементы вставляются и извлекаются только с одного конца контейнера.

Очередь (FIFO queue)

Очереди — это контейнеры-адаптеры, специально разработанный для работы в контексте FIFO (first-in first-out), где элементы вставляются в один конец контейнера и извлекаются из другого.

Дек (deque — double ended queue)

deque — нестандартное сокращение от «двусторонней очереди». Двусторонние очереди — это последовательные контейнеры с динамическими размерами, которые можно расширять или ужимать с обоих концов (как в начале, так и в конце).

Ассоциативные контейнеры

Ассоциативные контейнеры — это такие коллекции, в которых позиция элемента зависит от его значения, то есть после занесения элементов в коллекцию порядок их следования будет задаваться их значениями. К ассоциативным контейнерам относятся:

-набор (set);

- словарь (map).

Набор (set)

Наборы — это контейнеры, в которых хранятся уникальные элементы в определенном порядке.

Неупорядоченный набор (unordered set)

Неупорядоченные наборы — это контейнеры, в которых уникальные элементы хранятся в произвольном порядке и которые позволяют быстро извлекать отдельные элементы на основе их значения.

Словарь (map)

Словари — это ассоциативные контейнеры, в которых хранятся элементы, сформированные комбинацией значения ключа и сопоставленного ему значения в определенном порядке.

Неупорядоченный словарь (unordered map)

Неупорядоченные словари — это ассоциативные контейнеры, в которых хранятся элементы, сформированные комбинацией ключевого значения и значения, сопоставленного ему. Они позволяют быстро извлекать отдельные элементы на основе их ключей.

Библиотека итераторов

Предоставляет возможности для упорядоченного доступа к элементам контейнеров.

Определения итератора

Итератор — это любой объект, который, указывая на некоторый элемент в некотором диапазоне элементов, например, массива или контейнера, имеет возможность перебирать элементы этого диапазона, используя набор операторов, таких, как оператор инкремента (**++**) и оператор разыменования (*).

Наиболее очевидной формой итератора является указатель — указатель может указывать на элементы в массиве и может выполнять итерацию по ним с помощью оператора инкремента (**++**).

Но возможны и другие виды итераторов. Например, каждый тип контейнера (например, список) имеет определенный тип итератора, предназначенный для перебора его элементов.

Следует заметить, что, хотя указатель является формой итератора, не все итераторы обладают той же функциональностью, что и указатели. В зависимости от свойств, поддерживаемых итераторами, они делятся на пять различных категорий.

Категории итераторов

Итераторы подразделяются на пять категорий в зависимости от реализуемой ими функциональности:

1) ввода (Input);
2) вывода (Output);
3) однонаправленные (Forward);
4) двунаправленные (Bidirectional);
5) с произвольным доступом (Random Access).

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

Однонаправленные итераторы имеют все функции итераторов ввода и, если они не являются константными итераторами, имеют также функциональные возможности итераторов вывода.

Хотя они ограничены одним направлением для прохода по диапазону (только вперед).

Все стандартные контейнеры поддерживают как минимум итераторы данного типа.

Двунаправленные итераторы подобны однонаправленным итераторам, но также могут выполнять проход и в обратном направлении.

Итераторы с произвольным доступом реализуют все функции двунаправленных итераторов, а также имеют возможность доступа к диапазонам непоследовательно — к дальним (относительно текущего положения) элементам можно получить доступ напрямую, применяя значение смещения к итератору без прохода по всем элементам между ними. Эти итераторы имеют функциональность, аналогичную стандартным указателя (указатели являются итераторами этой категории).

Свойства каждой категории итераторов::

<table>
<tr><th colspan="4">Категория</th><th>Свойства</th><th>Выражение</th></tr>
<tr><td colspan="4" rowspan="2">Все категории</td><td>конструктор копирования
присвоение копированием
деструкторы</td><td>X b(a);
b = a;</td></tr>
<tr><td>может инкрементироваться</td><td>++ a
a++</td></tr>
<tr><td rowspan="7">Произвольного доступа</td><td rowspan="6">Двунаправленные</td><td rowspan="5">Однонаправленные</td><td rowspan="2">Ввода</td><td>поддерживает сравнение на равенство/неравенство</td><td>a == b
a != b</td></tr>
<tr><td>может быть разыменован как rvalue</td><td>*a
a->m</td></tr>
<tr><td>Вывода</td><td>может быть разыменован как lvalue (только для изменяемых типов итераторов)</td><td>*a = t
*a++ = t</td></tr>
<tr><td rowspan="2"></td><td>конструктор по умолчанию</td><td>X a;
X()</td></tr>
<tr><td>многопроходный — ни разыменование, ни инкремент не влияют на разыменование</td><td>{ b=a;
*a++;
*b; }</td></tr>
<tr><td colspan="2"></td><td>может декрементироваться</td><td>--a
a--
*a--</td></tr>
<tr><td colspan="3"></td><td>поддерживают арифметические операторы + и -</td><td>a + n
n + a
a - n
a - b</td></tr>
</table>

		Поддерживают сравнение на неравенство (**<**, **>**, **<=** и **>=**) между итераторами	**a < b** **a > b** **a <= b** **a >= b**
		Поддерживают составные операции присваивания **+=** и **-=**	**a += n** **a -= n**
		Поддерживает оператор разыменования для смещения (**[ ]**)	**a[n]**

Функциональные шаблоны

Итераторные операции:

begin() — возвращает итератор, указывающий на первый элемент в последовательности
end() — возвращает итератор, указывающий на последний элемент в последовательности

Конструкторы

из контейнера (1)

```
template <class Container>
auto begin(Container& cont)->decltype(cont.begin());

template <class Container>
auto begin(const Container& cont)->decltype(cont.begin());
```

из массива(2)

```
template <class T, size_t N>
constexpr T* begin (T(&arr)[N]) noexcept;
```

prev() — возвращает итератор, указывающий на предыдущий элемент в последовательности

next() — возвращает итератор, указывающий на следующий элемент в последовательности

advance() — перемещает итератор на **n** элементов.

```
#include <iterator>

template <class InputIterator, class Distance>
void advance(InputIterator& it, Distance n);
```

Если данная операция относится к итератору с произвольным доступом, функция использует **operator+** или **operator-** только один раз.

В противном случае функция многократно использует оператор инкрементирования/декрементирования (**operator++** или **operator--**) до тех пор, пока не будет выполнено перемещение на **n** элементов.

it — итератор, подлежащий продвижению. **InputIterator** должен быть как минимум итератором ввода.

n — количество позиций элемента для продвижения.

Количество позиций для продвижения должно быть отрицательным только для итераторов с произвольным доступом и двунаправленных итераторов.

Distance должно быть числовым типом, способным представлять расстояния между итераторами этого типа.

distance() — возвращает расстояние (в элементах) между итераторами.

```
#include <iterator>

template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Вычисляет количество элементов между **first** и **last**.

Если это итератор с произвольным доступом, функция использует оператор **-** (минус) для его вычисления. В противном случае функция кратно использует оператор инкрементирования (**operator++**).

Генераторы итераторов

back_inserter — создать итератор вставки в конец
front_inserter — создать итератор вставки в начало
inserter — создать итератор вставки
make_move_iterator — построить итератор перемещения

Шаблоны классов

iterator — базовый класс итератора
iterator_traits — свойства итератора

Шаблоны классов предопределенных итераторов

reverse_iterator — обратный итератор
move_iterator — адаптирует итератор так, чтобы тот при разыменовании производил ссылки на rvalue
back_insert_iterator — итератор вставки в конец
front_insert_iterator — итератор вставки в начало
insert_iterator — итератор вставки
istream_iterator — итератор входного потока istream
ostream_iterator — итератор выходного потока ostream
istreambuf_iterator — итератор буфера входного потока
ostreambuf_iterator — итератор буфера выходного потока

Классы тегов категорий (пустые классы для определения катагрии итератора)

input_iterator_tag — категория итератора ввода
output_iterator_tag — категория итератора вывода
forward_iterator_tag — категория однонаправленного итератора
bidirectional_iterator_tag — категория двунаправленного итератора
random_access_iterator_tag — категория итератора произвольного доступа

Представляют собой следующие структуры:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};
```

Шаблон строк — <string>

Заголовок представляет типы строк, характеристики символов (типажи) и набор функций преобразования:

Шаблоны классов (Class templates)

basic_string	Общий строковый класс (шаблон класса)
char_traits	Типаж (характеристики) символов (шаблон класса)

Создание экземпляров класса

string	Строковый класс (класс)
u16string	Строка 16-битных символов (класс)
u32string	Строка 32-битных символов (класс)
wstring	Широкая строка (класс)

Функции преобразования из строки в число

stoi	Преобразование строки в **int** (шаблон функции)
stol	Преобразование строки в **long int** (шаблон функции)
stoul	Преобразование строки в **unsigned int** (шаблон функции)
stoll	Преобразование строки в **long long** (шаблон функции)
stoull	Преобразование строки в **unsigned long long** (шаблон функции)
stof	Преобразование строки во **float** (шаблон функции)
stod	Преобразование строки в **double** (шаблон функции)
stold	Преобразование строки в **long double** (шаблон функции)

Функции преобразования из числа в строку

to_string	Преобразование числового значения в строку (функция)
to_wstring	Преобразование числового значения в широкую строку (функция)

Доступ к диапазону

begin	Итератор в начало (шаблон функции)
end	Итератор до конца (шаблон функции)

std::string — строковый клас

```
#include <string>
typedef basic_string<char> string;
```

Строки — это объекты, которые представляют собой последовательности символов.

Стандартный строковый класс обеспечивает поддержку таких объектов с интерфейсом, аналогичным интерфейсу стандартного контейнера байтов, но с добавлением функций, специально разработанных для работы со строками однобайтовых символов.

Класс **`string`** является экземпляром шаблона класса **`basic_string`**, который использует **`char`** (т.е. байты) в качестве его *типа символа*, с его типажами **`char_traits`** и аллокатором по умолчанию.

Следует обратить внимание, что этот класс обрабатывает байты независимо от используемой кодировки — если он используется для обработки последовательностей многобайтовых символов или символов переменной длины (например, UTF-8), все члены этого класса (например, длина или размер), а также его итераторы по-прежнему будут работать в байтах (а не в кодированных символах).

Члены типа

Члены типа	Определение
value_type	char
traits_type	char_traits<char>
allocator_type	allocator<char>
reference	char&
const_reference	const char&
pointer	char*
const_pointer	const char*
iterator	итератор произвольного доступа к char (с возможностью преобразования в const_iterator)
const_iterator	итератор произвольного доступа к const char
reverse_iterator	reverse_iterator<iterator>
const_reverse_iterator	reverse_iterator<const_iterator>
difference_type	ptrdiff_t
size_type	size_t

Что это такое — члены типа?

Для каждого шаблона класса определяется ряд членов, которые характеризуют класс

value_type — определяет тип шаблона

size_type — определяет переменную, способную удержать тип шаблона

...

Общедоступные (public) функции-члены

Конструкторы

по умолчанию (1)	`string();`
копирования (2)	`string(const string& str);`
из подстроки(3)	`string(const string& str, size_t pos, size_t len = npos);`
из с-строки (4)	`string(const char* s);`
из буфера (5)	`string(const char* s, size_t n);`
с заполнением (6)	`string(size_t n, char c);`
диапазонный (7)	`template <class InputIterator>` `string(InputIterator first, InputIterator last);`
из списка инициализации (8)	`string(initializer_list<char> il);`
перемещения (9)	`string(string&& str) noexcept;`

Деструктор `~string();`

Оператор присваивания

`operator=` — Присваивает строке новое значение, заменяя ее текущее содержимое

строки (1)	`string& operator= (const string& str);`
с-строки (2)	`string& operator= (const char* s);`
символа (3)	`string& operator= (char c);`
списка инициализации (4)	`string& operator= (initializer_list<char> il);`
перемещением (5)	`string& operator= (string&& str) noexcept;`

Пример

```
// string assigning
#include <iostream>
#include <string>

int main () {

  std::string str1, str2, str3;

  str1 = "Test string: ";   // c-string
  str2 = 'x';               // из символов
  str3 = str1 + str2;       // копирование конкатенации строк

  std::cout << str3  << '\n';
  return 0;
}
```

Выход

```
Test string: x
```

Итераторы

begin	Вернуть итератор в начало
end	Вернуть итератор в конец
rbegin	Вернуть реверсивный итератор в реверсивное начало
rend	Вернуть реверсивный итератор в реверсивный конец
cbegin	Вернуть **const_iterator** в начало
cend	Вернуть **const_iterator** в конец
crbegin	Вернуть **const_reverse_iterator** в реверсивное начало
crend	Вернуть **const_reverse_iterator** в реверсивный конец

Емкость

size	Вернуть длину строки
length	Вернуть длину строки
max_size	Вернуть максимальный размер строки
resize	Изменить размер строки
capacity	Вернуть размер выделенной памяти под строку
reserve	Запросить изменение объема
clear	Очистить строку
empty	Проверить, пуста ли строка
shrink_to_fit	Сократить память до размеров содержимого

Доступ к элементам

operator[]	Получить символ строки
at()	Получить символ в строке
back()	Доступ к последнему символу
front ()	Доступ к первому символу

std::string::operator[]

```
      char& operator[] (size_t pos);
const char& operator[] (size_t pos) const;
```

Возвращает ссылку на символ в позиции **pos** в строке.

Если **pos** равно длине строки, функция возвращает ссылку на нулевой символ, следующий за последним символом в строке.

Пример string::operator[]

```
#include <iostream>
#include <string>

int main () {
  std::string str("Test string");
  for (int i = 0; i < str.length(); ++i) {
    std::cout << str[i];
  }
  return 0;
}
```

`std::string::at()` — Получить символ в строке

```
      char& at(size_t pos);
const char& at(size_t pos) const;
```

Возвращает ссылку на символ в позиции **`pos`** в строке.

Функция автоматически проверяет, является ли **`pos`** допустимой позицией символа в строке (т.е. меньше ли **`pos`**, чем длина строки), генерируя исключение **`out_of_range`**, если это не так.

Пример `std::string::at()`

```
#include <iostream>
#include <string>

int main () {

  std::string str ("Test string");
  for (unsigned i = 0; i < str.length(); ++i) {
    std::cout << str.at(i);
  }
  return 0;
}
```

Выход

```
Test string
```

`std::string::back/front()` — Доступ к последнему/первому символу

```
      char& back();
const char& back() const;
      char& front();
const char& front() const;
```

Возвращает ссылку на последний/первый символ строки.
Эти функции не должны вызываться для пустых строк (неопределеное поведение).

Пример `std::string::back()`

```
#include <iostream>
#include <string>

int main () {

  std::string str ("hello world.");
  str.back() = '!';
  std::cout << str << '\n';
  return 0;
}
```

Выход

```
hello world!
```

Модификаторы

operator+=	Добавить к строке с конца
append()	Добавить к строке
push_back()	Добавить символ к строке с конца
assign()	Присвоить новое значение строке, заменив текущее
insert()	Вставить в строку в указанной позиции
erase()	Удалить часть строки указанной длины с указанной позиции
replace()	Заменить часть строки указанной длины с указанной позиции
swap()	Поменять местами значения строк, возможно, разной длины
pop_back()	Удалить последний символ (откусить)

Строковые операции

c_str — получить эквивалент строки C.
data — получить строковые данные
Обе возвращают указатель на массив, содержащий последовательность символов с завершающим нулем (то есть C-строку), представляющую текущее значение строкового объекта.

get_allocator	Получить копию объекта аллокатора

copy — копировать последовательность символов из строки
Копирует подстроку указанной длины с указаной позиции текущего значения строкового объекта в указанный массив char. Функция не добавляет нулевой символ в конец скопированного содержимого.

find — найти указанное содержимое в строке
rfind — найти последнее вхождение указанного содержимого в строке
find_first_of — найти символ из набора в строке
find_last_of — найти символ из набора в строке с конца
find_first_not_of — найти в строке отсутствующий в наборе символ
find_last_not_of — найти в строке отсутствующий в наборе символ с конца
substr — создать новую подстроку
compare — сравнить строки

Перегруженные функции-не-члены

operator+ — объединить строки
Возвращает новый строковый объект, созданный из двух строковых объектов, значение которого является конкатенацией символов из первого, за которыми следуют символы второго.

relational operators — реляционные операторы для строк
Выполняет операцию сравнения (**==**, **!=**, **<**, **<=**, **>**, **>=**) между двумя строковыми объектами.

swap — меняет значения двух строк

operator>> — извлечь строку из потока
operator<< — вставить строку в поток
getline — Получить строку из потока в строку **<string>**

std::basic_string — общий строковый класс

basic_string — это обобщение класса **string** для любого типа символа.

```
#include <string>

template < class charT,
           class traits = char_traits<charT>,  // basic_string::traits_type
           class Alloc = allocator<charT>      // basic_string::allocator_type
           > class basic_string;
```

Параметры шаблона

charT — Тип символа.
Строка состоит из последовательности символов этого типа.
Это должен быть POD-тип, не являющийся типом массива.

traits — класс типажа, который определяет основные свойства символов, используемых объектами **basic_string**.
traits::char_type должен быть таким же, как **charT**.
Имеет псевдоним как члена типа **basic_string::traits_type**.

Alloc
Тип объекта-аллокатора, используемого для определения модели распределения памяти. По умолчанию используется шаблон класса аллокатора, который определяет простейшую модель распределения памяти.
Имеет псевдоним как члена типа **basic_string::allocator_type**.

std::char_traits — шаблон класса типажей символов

```
#include <string>

template <class charT> struct char_traits;

template <> struct char_traits<char>;     // специализация для char
template <> struct char_traits<wchar_t>;  // специализация для wchar_t
template <> struct char_traits<char16_t>; // специализация для char16_t
template <> struct char_traits<char32_t>; // специализация для char32_t
```

Типажи (характеристики) символов

Классы типажей символов определяют свойства символов и обеспечивают конкретную семантику (порядок и способы использования) для определенных операций с символами и последовательностями символов.

Стандартная библиотека включает стандартный набор классов типажей символов, которые могут быть созданы из шаблона **`char_traits`** и которые используются по умолчанию как для объектов **`basic_string`**, так и для объектов потока ввода/вывода. Но вместо этого можно использовать любой другой класс, который соответствует требованиям класса характеристик символа.

Параметры шаблона

`charT` — Тип символа.

Класс определяет стандартные характеристики символов для этого типа символов.

Это должен быть один из типов, для которых предусмотрена специализация.

Специализации шаблона

Стандартный шаблон **char_traits** поддерживает создание экземпляров, по крайней мере, со следующими типами символов:

char — базовый набор символов (размер 1 байт);

wchar_t — самый широкий набор символов (тот же размер, знаковость (signedness) и выравнивание, что и у подобного интегрального типа);

char16_t — представляет 16-битные единицы кода (тот же размер, знаковость и выравнивание, что и **uint_least16_t**);

char32_t — представляет любую из 32-битных кодовых точек (тот же размер, подписи и выравнивания, что и **uint_least32_t**).

Типы-члены

member type	Описание для типов character traits	definition			
		char	**wchar_t**	**char16_t**	**char32_t**
char_type	Параметр шаблона (**charT**)	**char**	**wchar_t**	**char16_t**	**char32_t**
int_type	Целочисленный тип, котороый может представлять все значения типа **charT**, а также и eof()	**int**	**wint_t**	**uint_least16_t**	**uint_least32_t**
off_type	Тип с поведением streamoff	**streamoff**	**streamoff**	**streamoff**	**streamoff**
pos_type	Тип с поведением streampos	**streampos**	**wstreampos**	**u16streampos**	**u32streampos**
state_type	Состояние мультибайтных преобразований, аналогичное mbstate_t	**mbstate_t**	**mbstate_t**	**mbstate_t**	**mbstate_t**

Общедоступные (public) статические функции-члены

eq	Сравнить символы на равенство
lt	Сравнить символы на предмет неравенства
length	Получить длину строки с завершающим нулем
assign	Присвоить символ
compare	Сравнить последовательности символов
find	Найти первое вхождение символа
move	Переместить последовательность символов
copy	Копировать последовательность символов
eof	Символ конца файла
not_eof	Не символ конца файла
to_char_type	Преобразовать к типу char
to_int_type	Преобразовать к типу int
eq_int_type	Сравнить значения типа int_type