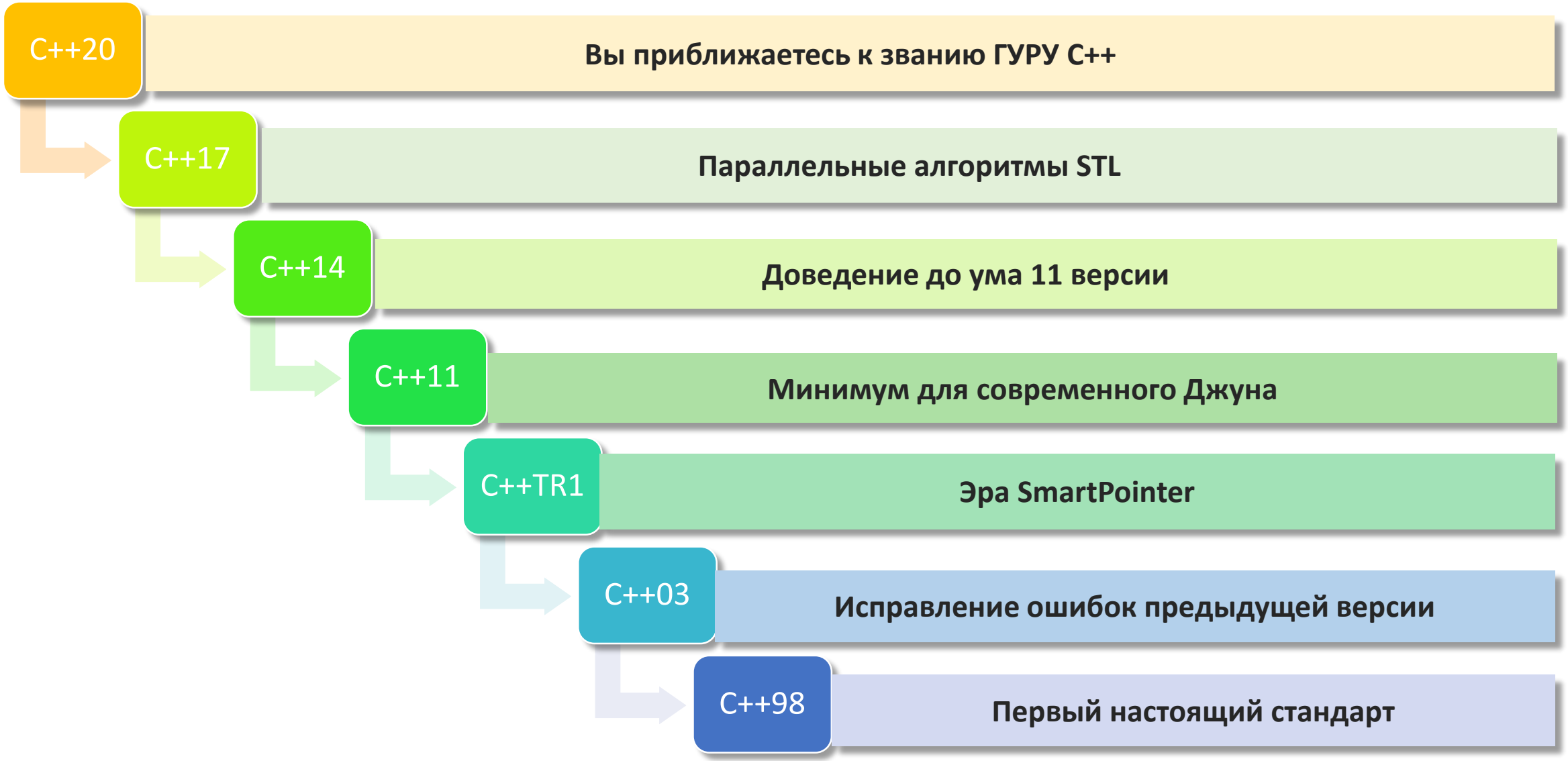
Стандарты C++

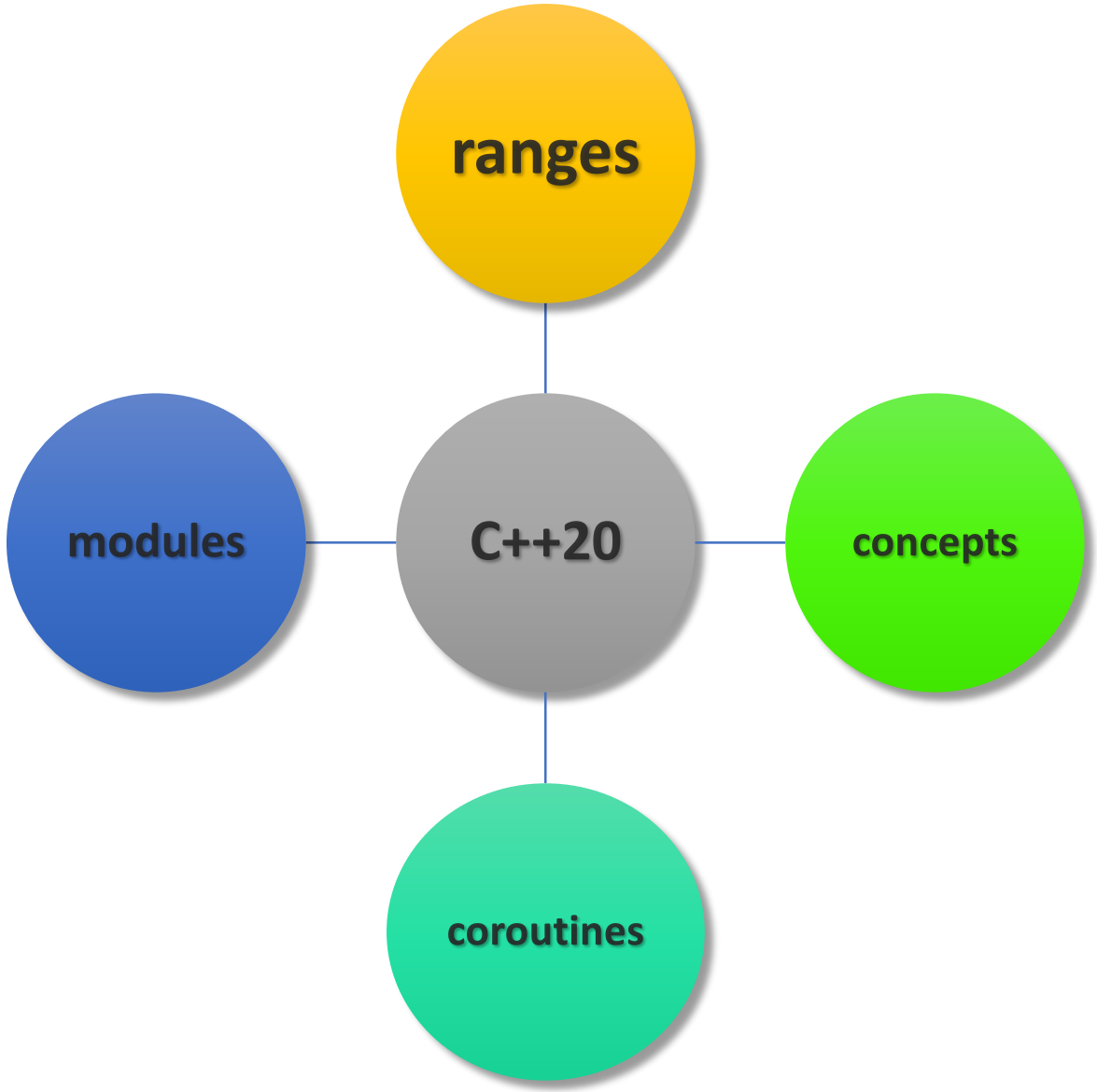
Лекция

Ретроспектива стандартов



C++20

«Большая четверка» стандарта:



C++20. Concepts

Введение **концепций** (**concepts**) связывают с дальнейшим развитием в языке C++ инструментария, основанного на парадигме обобщенного программирования. Они позволяют вам выразить свое намерение непосредственно через систему типов данных, обеспечивают соответствие используемых в шаблоне данных указанному набору критериев и проверяют это в начале процесса компиляции. Если что-то пойдет не так, вы получите короткое, но в то же время значимое сообщение об ошибке, вместо огромного списка сообщений об ошибках и предупреждений, ведущих куда-то вглубь шаблона.

https://habr.com/ru/companies/yandex_praktikum/articles/556816/
https://metanit.com/cpp/tutorial/17.2.php

Concepts позволяют установить ограничения для параметров шаблонов (как шаблонов функций, так и шаблонов класса).

Concepts фактически представляет шаблон для именованного набора ограничений, где каждое ограничение предписывает одно или несколько требований для одного или нескольких параметров шаблона. В общем случае он имеет следующий вид:

```
template <параметры>
concept имя_концепта = ограничения;
```

```
template <typename T>
concept size = sizeof(T) <= sizeof(int);
```

В данном случае определен *concept* **size**. Его смысл в том, что тип, который будет передаваться через параметр **T**, должен удовлетворять условию **sizeof(T) <= sizeof(int)**. То есть физический размер объектов типа T не должен быть больше размера значений типа int.

C++20. Ranges

Новая библиотека **диапазонов** (**ranges**): Позволяет алгоритмам напрямую работать с контейнерами, комбинировать алгоритм с символом конвейера | и применять их к бесконечным потокам данных.

```
#include <iostream>
#include <ranges>
#include <list>
#include <algorithm>
int main() {
    std::list<int> dt = {1, 4, 2, 3};
    std::ranges::sort(dt.begin(), dt.end());
    std::ranges::copy(dt.begin(), dt.end(),
            std::ostream_iterator<int>(std::cout, ","));
}
```

Диапазоны (**ranges**) по сути являются итераторами, которые охватывают последовательность значений в коллекциях, таких как списки или векторы. Но вместо того, чтобы постоянно перемещать начало и конец итератора, диапазоны просто сохраняют их внутри.
Диапазоны перешли из экспериментального состояния в стандарт языка. Диапазоны зависят от концептов и используют их для улучшения обработки старого итератора, позволяя добавлять ограничения к обработанным значениям. Помимо типов, ограничивающих значения, диапазоны воспринимают представления (*Views*) как особую форму диапазона, позволяющую манипулировать данными или фильтровать их, возвращая изменённую версию данных исходного диапазона в качестве ещё одного диапазона. Например, есть вектор целых чисел, и необходимо получить все чётные значения в квадрате — диапазоны и представления помогут вам в этом.
Со всеми этими изменениями компилятор окажет гораздо большую помощь при проверке типов и представит более полезные сообщения об ошибках.

https://habr.com/ru/articles/546996/
https://radioprog.ru/post/1419
https://itnan.ru/post.php?c=1&p=707948
https://telegra.ph/Algoritmy-diapazonov-C20--7-nemodificiruyushchih-operacij-12-20-2
https://mariusbancila.ro/blog/2019/01/20/cpp-code-samples-before-and-after-ranges

C++20. Coroutines

Благодаря **сопрограммам** (**coroutines**) асинхронное программирование может стать мейнстримом в C++. Сопрограммы являются основой для современных задач, циклов событий, бесконечных потоков данных или конвейеров.

https://habr.com/ru/companies/piter/articles/491996/
https://gamedev.ru/code/articles/cpp_coroutines_1
https://habr.com/ru/articles/519464/

Сопрограммы (**корутины, coroutine**) - это потоки исполнения кода, которые организуются «поверх» аппаратных (системных) потоков.

Поток исполнения кода - это последовательность операций, которые выполняются друг за другом. В нужные моменты эта последовательность может быть приостановлена, и вместо нее может начать выполняться часть другой последовательности операций.

Системные потоки состоят из инструкций процессора, и на одном ядре процессора могут по очереди работать несколько системных потоков.
Сопрограммы работают на более высоком уровне - несколько сопрограмм могут по очереди выполнять свой код на одном системном потоке. (В зависимости от реализации, сопрограмма может быть не привязана к конкретному системному потоку, а например выполнять свой код на пуле потоков).

В отличие от системных потоков, которые переключаются системой в произвольные моменты времени (*вытесняющая многозадачность*), сопрограммы переключаются вручную, в местах, указанных программистом (*кооперативная многозадачность*).

C++20. Function vs Coroutine

Стандартный способ работы с функциями – вызвать и дожидаться, пока она завершится:

```
void foo(){
    return; // здесь мы выходим из функции
}
…
foo();    // здесь мы вызываем/запускаем функцию
```

После вызова функцию уже невозможно приостановить, или возобновить ее работу. Над функциями можно производить всего две операции: *start* и *finish*. Когда функция запущена, необходимо дожидаться, пока она завершится. Если функция будет вызвана повторно, то ее выполнение пойдет с самого начала.

Coroutines можно не только запускать и останавливать, но также приостанавливать и возобновлять. Они все равно отличаются от потоков ядра, поскольку сами по себе **coroutines** не являются вытесняющими (с другой стороны, **coroutines** обычно относятся к потоку, а поток является вытесняющим).

Функция является сопрограммой (**coroutine**), если ее определение выполняет одно из следующих действий:

- использует оператор **co_await** для приостановки выполнения до возобновления;
- использует ключевое слово **co_yield**, чтобы приостановить выполнение, возвращающее значение;
- использует ключевое слово **co_return** для завершения выполнения, возвращая значение;

C++(modules)

Модули (modules) — механизм, являющийся новым способом разделения исходного кода, призванный преодолеть ограничения, возникающие из-за использования заголовочных файлов, и, в конечном счете, заменить всю систему препроцессоров. В результате мы должны получить более быстрый и простой способ сборки пакетов.

Modules — это, по сути, новый способ разделить код, заменив *#include* на *import*, а также позволяя нам убрать разделение между интерфейсами и реализациями, тем самым потенциально вдвое сократив количество файлов. Теперь мы можем поместить все в один файл и открыть внешнему миру только то, что мы явно помечаем как экспортируемое (**export**). Это может быть полезно в некоторых случаях, когда разделение файлов необходимо только для ускорения компиляции и не способствует пониманию кода. Стоит отметить, что разделение файла интерфейса / реализации по-прежнему возможно с помощью модулей.

https://habr.com/ru/companies/otus/articles/742818/
https://habr.com/ru/companies/otus/articles/575954/
https://pcnews.ru/blogs/standart_c20_obzor_novyh_vozmoznostej_c_cast_1_moduli_i_kratkaa_istoria_c-1078273.html#gsc.tab=0

А что кроме этого в C++ 20

```
auto res = a <=> b
if a<b
  return -1
else if a>b
  return 1
else //a==b
  return 0
end
```

- **оператор трехстороннего сравнения <=>;**
- календарь и расширения часовых поясов библиотеки chrono;
- **std::span как представление массива;**
- два новых ключевых слова: **consteval** и **constinit**;
- **designated initializers (назначенные инициализаторы);**
- constexpr-контейнеры;
- строковые литералы как параметры шаблона;
- тип char8_t — стандартный тип для представления строк в формате UTF-8.

Новый тип **std::span<T>,** который позволяет ссылаться на любую последовательность значений типа T - это может быть и std::vector<T>, и std::array<T>, и стандартный массив, и ряд других последовательностей.

```
struct Date {
    int year;
    int month;
    int day;
};
Date inFuture { .year = 2050, .month = 4, .day = 10 };
```

VS `Date inFuture { 2050, 4, 10 };`

Ключевое слово **consteval** объявляет функцию, результат которой вычисляется на этапе компиляции, а ключевое слово **constinit** гарантирует, что переменная будет инициализирована на этапе компиляции.

https://github.com/AnthonyCalandra/modern-cpp-features/blob/master/CPP20.md

C++17

список основных улучшений, добавленных в C++17

- идентификатор препроцессора **__has_include** для проверки доступности дополнительных заголовочных файлов;
- if-стейтменты, которые обрабатываются во время компиляции (**Constexpr if**);
- инициализаторы в стейтментах if и switch;
- **inline - переменные**;
- **fold-выражения** (свертка параметров шаблона);
- вложенные пространства имен теперь можно определять как
- пространство имен **X::Y**;
- удаление **std::auto_ptr** и других устаревших типов;
- **static_assert** больше не требует параметра в виде текстового сообщения;
- **std::any**;
- **std::byte**;
- **std::filesystem**;
- **std::optional**;
- **std::shared_ptr** теперь может управлять массивами C-style (но через
- std::make_shared() их по-прежнему нельзя создавать);
- **std::size**;
- триграфы были удалены;
- **UTF-8** (u8) **символьные литералы**.

https://ps-group.github.io/cxx/cxx17
https://habr.com/ru/companies/pvs-studio/articles/340014/

Constexpr if:
Появилась возможность выполнять условные конструкции на этапе компиляции. Это очень мощный инструмент, особенно полезный в метапрограммировании.

Инициализатор в if и switch:
Появились операторы if и switch с инициализатором:

if (init; condition)
switch(init; condition)

Пример использования:

```
if (auto it = m.find(key); it != m.end())
{  .... }
```

Filesystem:
C++17 предоставляет возможности для кроссплатформенной работы с файловой системой. Эта библиотека фактически является **boost::filesystem**, которую перенесли в стандарт.

has_include:
Предикат препроцессора *__has_include* позволяет проверить, доступен ли заголовочный файл для подключения.

```
#if __has_include(<optional>)
#include <optional>
#define have_optional 1
#endif
```

inline переменные:
В C++17 в дополнение к inline функциям появились также *inline переменные*. Переменная или функция, объявленная inline, может быть определена (обязательно одинаково) в нескольких единицах трансляции.
До C++17 пришлось бы объявлять переменную *XXVar* как *extern* и в одном из **.cpp* файлов присваивать ей значение.

std::optional:

Это шаблонный класс, который хранит опциональное значение. Его удобно использовать, чтобы, например, возвращать значение из функции, в которой может произойти какая-то ошибка:

std::any:

Объект класса std::any может хранить информацию любого типа. Так, одна и та же переменная типа std::any может сначала хранить int, затем float, а затем строку.
Чтобы получить доступ к информации, хранящейся в объекте *std::any*, нужно воспользоваться *std::any_cast*

std::variant:

std::variant — это шаблонный класс, который представляет собой union, который помнит, какой тип он хранит. Для получения значений из *std::variant* используется функция *std::get*. Она выбросит исключение *std::bad_variant_access,* если попытаться взять не тот тип.

std::string_view

В C++17 появился особый класс – std::string_view, который хранит указатель на начало существующей строки и ее размер. Таким образом, std::string_view представляет собой не владеющую память строку.

```
// C++14
void Func(const char* str);
void Func(const char str[10]);
void Func(const std::string &str);
 // C++17 void Func(std::string_view str);
```

Новый тип std::byte:

Тип std::byte предлагается использовать при работе с 'сырой' памятью. Обычно для этого используется char, unsigned char или uint8_t. Тип std::byte является более типобезопасным, так как к нему можно применить только побитовые операции, а арифметические операции и неявные преобразования недоступны.

C++14

В отличие от версии C++11, в которой добавилось относительно много нового функционала, в C++14 произошло лишь небольшое обновление — в основном исправления ошибок и небольшие улучшения.

- бинарные литералы;
- атрибут **deprecated**;
- цифровые разделители;
- автоматическое определение возвращаемого типа функции — вывод типов;
- **relaxed constexpr** функции;
- шаблоны переменных;
- стандартные пользовательские литералы;
- **std::make_unique**().

В C++14 мы можем использовать **бинарные (двоичные) литералы**, добавляя **префикс 0b**:

```
int bin(0);
   bin = 0b1; // присваиваем переменной бинарный литерал 0000 0001
   bin = 0b11; // присваиваем переменной бинарный литерал 0000 0011
```

в C++14 добавили возможность использовать одинарную кавычку ' в качестве **разделителя цифр**:

```
   int bin = 0b1011'0010; // присваиваем переменной бинарный литерал 1011 0010
   long value = 2'532'673'462; // намного проще читать, нежели 2532673462
```

В C++14 функционал ключевого слова **auto** был расширен **до автоматического определения типа возвращаемого значения функции**. Например:

```
auto subtract(int a, int b)
{
   return a - b;
}
```

Пользовательские литералы для **std::complex**: Добавлены следующие литералы для быстрого создания комплексных чисел, состоящих только из мнимой части:

```
using namespace std;
complex<double> a = 1.5 + 0.3i;
auto b = 2.3 - 0.2i;
```

C++11. То что должен знать каждый.

- **long long int**;
- конструктор перем-я и оператор присваивания перем-ем;
- спецификатор **noexcept**;
- **nullptr**;
- модификаторы **override** и **final**;
- цикл foreach;
- ссылки r-value;
- **static_assert**;
- **std::initializer_list**;
- псевдонимы типов;
- **uniform**-инициализация;
- пользовательские литералы;
- вариативные шаблоны.

- **auto**;
- **char16_t**, char_32t и новые литералы для их поддержки;
- **constexpr**;
- **decltype**;
- спецификатор **default**;
- делегирующие конструкторы;
- ключевое слово **delete**;
- классы **enum**;
- внешние шаблоны;
- **лямбда-выражения**;

- лучшая поддержка многопоточности и локальное хранилище потоков;
- хеш-таблицы;
- улучшенная генерация случайных чисел;
- std::reference_wrapper;
- регулярные выражения;
- **std::tuple**;
- **std::unique_ptr**.

C++11. Декларация типа с помощью auto

```
// c++03 решение
for (std::vector<std::map<int, std::string>>::const_iterator it = container.begin();  it != container.end();  ++it)
{
   // do something
}
```

```
// c++11 решение
for (auto it = container.begin(); it != container.end(); ++it)
{
// do something
}
```

Оператор **auto** обеспечивает автоматическое определение типа во время компиляции

Особенности auto

```
void foo()
{
    auto x = 5;          // тип переменной x будет int
    x = "foo";          // ошибка! не соответствие типов
   auto y = 4, z = 3.14; // ошибка! нельзя объявлять  переменные разных типов
}
```

Что такое decltype и с чем его едят?

```
int x = 5;
double y = 5.1;

decltype(x) foo; // int

decltype(y) bar; // double

decltype(x + y) baz; // double
```

Decltype позволяет выбирать такой же тип, как у обьекта в скобках

Финты ушами с auto, decltype и шаблонами

```
template <typename T, typename E>
auto compose(T a, E b) -> decltype(a + b) {
        return a + b;
}
auto c = compose(2, 3.14);    // c - double
```

“>>” как закрытие вложенных шаблонов

```
//Как было раньше
std::vector<std::map<int, int>> foo; // ошибка компиляции

std::vector<std::map<int, int> > foo; // вполне корректный код
```

Range-based for

```
std::vector<int> foo;
    // заполняем вектор
for (int x : foo)
std::cout <<  x  << std::endl;
```

```
std::vector<std::pair<int, std::string>> container;
      // ...
for (const auto& i : container)
std::cout << i.second << std::endl;
```

NULLPTR – полноценный указатель на пустой объект

```
//Раньше NULL был макросом языка C, который означал 0
Foo* foo = 0;      // можно было писать так
void func(int x);
void func(const Foo* ptr);
// ...
func(0);        //вызовется func(int x), хотя мы могли
                // подразумевать func(const Foo* ptr) с знач. NULL
```

```
//Тут описана более серьезная проблема
std::vector<Foo*> foos;
// ...
std::fill(foos.begin(), foos.end(), 0);//страшная ошибка компиляции на полтора листа
//Решение C++ 11
std::vector<Foo*> foos;
// ...
std::fill(foos.begin(), foos.end(), nullptr);   //все отлично!
```

Списки инициализации (initializer_list<T>)

```
//C++ 03
struct Struct
{
    int x;
    std::string str;
};
// инициализируем атрибуты структуры.
Struct s = { 4, "four" };
// инициализируем массив
int arr[] = { 1, 8, 9, 2,4 };
```

```
//C++ 11
std::vector<int> v = { 1, 5, 6, 0, 9 };
v.insert(v.end(), { 0, 1, 2, 3, 4});

class Foo
{
  public:
  // ...
     Foo(std::initializer_list<int>list);
};

Foo::Foo(std::initializer_list<int> list)
{
  // do something
}
```

Универсальная инициализация

```
class Foo
{
  public:
    // ...
  Foo(int x, double y, std::string z);
};
```

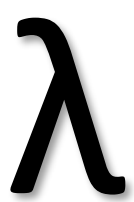
```
// ...
Foo::Foo(int x, double y, std::string z)
{
    // do something
}
// ...
  Foo one = { 1, 2.5, "one" };
  Foo two{ 5, 3.14, "two" };
//эквивалентно вызову конструктора
  Foo foo(1, 2.5, "one");
```

Не все только классам, не забудем и структуры

```
struct Foo
{
  std::string str;
  double x;
  int y;
};

  Foo foo{ "C++11", 4.0, 42 }; // {str, x, y}
  Foo bar{ "C++11", 4.0 };      // {str, x}, y = 0
```

Lambdas



λ. Общее определение

Лямбда-выражение (или просто «**лямбда**») в программировании позволяет определить анонимную функцию внутри другой функции. Возможность сделать функцию вложенной является очень важным преимуществом, так как позволяет избегать как захламления пространства имен лишними объектами, так и определить функцию как можно ближе к месту её первого использования.

```
[ captureClause ] ( параметры ) -> возвращаемыйТип
{
    Стейтменты(действия);
}
```

Поля *captureClause* и параметры могут быть пустыми, если они не требуются программисту.
Поле *возвращаемыйТип* является опциональным, и, если его нет, то будет использоваться вывод типа с помощью ключевого слова *auto*. Также обратите внимание, что *лямбда-выражения* могут не иметь имени, поэтому нам и не нужно будет их предоставлять.

```
int main()
{
  []() {};     // без captureClause, пар-ов и типа возв-та
  return 0;
}
```

λ. Вызов лямбда-выражения

```
// простейшее лямбда-выражение

[]() { std::cout << "Hello" << std::endl; }
```

Каждый раз, когда компилятор встречает лямбда-выражение, он генерирует новый тип класса, который представляет объект-функцию. В примере выше сгенерированный класс упрощенно может выглядеть

```
class __Lambda1234
{
public:
    auto operator()() const {
        std::cout << "Hello" << std::endl;
    }
};
```

```
int main()
{
    [](){std::cout << "Hello" << std::endl;} ();
    // или так
    []{std::cout << "Hello" << std::endl;} ();
}
```

Мы можем непосредственно при определении сразу же вызвать лямбда-выражения, указав после тела выражения круглые скобки со значениями для параметров лямбды:

Лямбда-выражение можно определить как переменную:

```
//Именованные лямбда-выражения
int main(){
    // переменная hello представляет лямбда-выражение
    auto hello { [](){std::cout << "Hello" << std::endl;} };
    // через переменную вызываем лямбда-выражение
    hello();    // Hello
}
```

λ. Параметры

```
int main()
{
    auto print { [](const std::string& text){std::cout << text << std::endl;} };

    // вызываем лямбда-выражение
    print("Hello World!");              // Hello World!
    print("Good bye, World...");    // Good bye, World...
}
```

```
 //можно сразу же при определении вызвать лямбда-
//выражение, передав в него строку

[](const std::string& text){std::cout << text << std::endl;} ("Hell");
```

λ. Возвращение значения

Лямбда-выражение может возвращать произвольное значение. В этом случае, как и в обычной функции, применяется оператор **return**:

```
int main()
{
    auto sum { [](int a, int b){return a + b;} };
    // вызываем лямбда-выражение
    std::cout << sum(10, 23) << std::endl;  // 33

    // присваиваем его результат переменной
    int result { sum(1, 4)};
    std::cout << result << std::endl;       // 5
}
```

По умолчанию компилятор сам определяет, значение какого именно типа будет возвращаться из лямбды. Однако мы также можем явным образом указать возвращаемый тип.

```
int main()
{
    auto sum { [](int a, int b) -> double {return a + b;} };
    // вызываем лямбда-выражение
    std::cout << sum(10, 23) << std::endl;  // 33
}
```

λ как параметры функций

Лямбда-выражение может передаваться в качестве значения параметру функции, который представляет указатель на функцию.

```
void do_operation(int a, int b, int (*op)(int, int))
{
    std::cout << op(a, b) << std::endl;
}

int main()
{
    auto sum { [](int a, int b) {return a + b;} };
    auto subtract { [](int a, int b) {return a - b;} };

    do_operation(10, 4, sum);          // 14
    do_operation(10, 4, subtract);     // 6

    ……
do_operation(10, 4, [](int a, int b) {return a * b;});
// 40

}
```

Также можно определить лямбда-выражение непосредственно при использовании, что может быть полезно, если лямбду больше нигде не планируется использовать:

λ. Универсальные лямбда-выражения

Универсальное лямбда-выражение (generic lambda) — это лямбда-выражение, в котором как минимум для одного параметра в качестве типа указано слово **auto** или выражения **auto&** или **const auto&.** Это позволяет уйти от жесткой привязки параметров к определенному типу.

```
int main()
{
    auto add = [](auto a, auto b) {return a + b;};
    //auto print = [](const auto& value) {std::cout << value << std::endl; };
    std::cout << add(2, 3) << std::endl;        // 5 - складываем числа int
    std::cout << add(2.2, 3.4) << std::endl;    // 5.6 - складываем числа double
    std::string hello{"hello "};
    std::string world{"world"};
    std::cout << add(hello, world) << std::endl;    // hello world - складываем
строки
}
```

На момент написания мы не знаем, какие типы будут представлять параметры. Конкретные типы будет выводить компилятор при вызове лямбда-выражения исходя из переданных в него значений:

std::cout << add(2, 3) << std::endl;