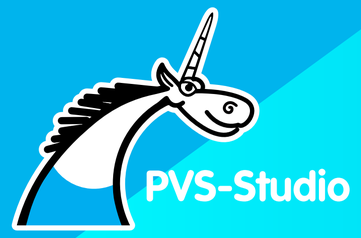
Филипп Хандельянц

Лекция 1/12

Нововведения стандарта C++11



Докладчик

Хандельянц
Филипп Александрович

- Ведущий разработчик в команде PVS-Studio (C++/C#)
- 3 года участвую в разработке ядра C++ анализатора
- Автор статей о проверке open source-проектов

Внесенные изменения в C++11

Расширения ядра языка:

- ~~Multithreading~~
- ~~Uniform initialization~~
- ~~std::initializer_list~~
- ~~Move semantics~~
- ~~Variadic templates~~
- ~~noexcept~~
- ~~constexpr~~
- ~~POD-type~~
- ~~Range-based for loop~~
- ~~Auto/decltype~~
- ~~Lambda functions~~
- ~~Alternative function syntax~~
- ~~Default value for non-static class member~~
- ~~Delegate constructors~~
- ~~'default' / 'delete' specifiers~~
- ~~'override' / 'final' specifiers~~
- ~~nullptr~~
- ~~enum class~~
- ~~enum underlying type~~
- ~~Explicit cast operators~~
- 'using' for types
- Relaxed rules for unions
- extern templates
- New string literals types
- User-defined literals
- static_assert
- alignof/alignas
- Attributes

Расширения стандартной библиотеки:

- Updates to the existing components
- std::tuple
- Associative unordered containers
- Smart pointers
- std::function
- std::reference_wrapper
- Regex library
- Random library
- Chrono library

'using' for types

```
#include <vector>

typedef std::vector<int>::iterator vec_int_iterator; // ok

template <typename T>
typedef std::vector<T>::iterator vec_t_iterator; // error
```

```cpp
#include <vector>

using vec_int_iterator = std::vector<int>::iterator; // C++11

template <typename T>
using vec_t_iterator = std::vector<T>::iterator;   // C++11

vec_t_iterator<int> it; // ok
```

Relaxed rules for unions

```
#include <new>

struct Point
{
  Point() {}
  Point(int x, int y): m_x {x}, m_y {y} {}
  int m_x, m_y;
};

union U
{
  int z;
  double w;
  Point p;                          // ok since C++11
  U() { new( &p ) Point(); } // must be implemented explicitly
};
```

Extern templates

```
// SomeHeader.h

template <typename T>
void foo(T arg) { .... }

template <typename T>
class SomeClass { .... };
```

```
// A.cpp

#include “SomeHeader.h”

void bar1()
{
  foo(0);
  SomeClass<int> obj1;
}
```

```
// B.cpp

#include “SomeHeader.h”

void bar2()
{
  foo(255);
  SomeClass<int> obj2;
}
```

A.o

```
void bar1()
void foo<int>(int)
SomeClass<int>
```

B.o

```
void bar2()
void foo<int>(int)
SomeClass<int>
```

```
// SomeHeader.h

template <typename T>
void foo(T arg) { .... }

template <typename T>
class SomeClass { .... };
```

```
// A.cpp

#include “SomeHeader.h”

void bar1()
{
  foo(0);
  SomeClass<int> obj1;
}
```

```
// B.cpp

#include “SomeHeader.h”

extern template void foo<int>(int);
extern template class SomeClass<int>;

void bar2()
{
  foo(255);
  SomeClass<int> obj2;
}
```

A.o

```
void bar1()
void foo<int>(int)
SomeClass<int>
```

B.o

```
void bar2()
~~void foo<int>(int)~~
~~SomeClass<int>~~
```

New string literals types

```
 "Hello, world" // const char[28]
L"Hello, world" // const wchar_t[28]
```

```
 "Hello, world"  // const char[28]
L"Hello, world" // const wchar_t[28]

u8"Hello, мир"  // const char[47]
 u"Hello, мир"  // const char16_t[31]
 U"Hello, мир"  // const char32_t[31]
```

```
 "Hello, world"  // const char[28]
L"Hello, world" // const wchar_t[28]

u8"Hello, мир"  // const char[47]
 u"Hello, мир"  // const char16_t[31]
 U"Hello, мир"  // const char32_t[31]

  R"delimiter(Hello, мир)delimiter" // const char[11]
 LR"delimiter(Hello, мир)delimiter" // const wchar_t[11]

u8R"delimiter(Hello, мир)delimiter" // const char[14]
 uR"delimiter(Hello, мир)delimiter" // const char16_t[11]
 UR"delimiter(Hello, мир)delimiter" // const char32_t[11]
```

```
R"regex((?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&
'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x
23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:
(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9
-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9
]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9
-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-
\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])regex"
```

User-defined literals

```
uint64_t weight = 2 * 1000;

double rad = 45.0 * M_PI / 180.0;

uint64_t seconds = 2 * 3600 + 30 * 60;
```

```
uint64_t weight = 2 * 1000;

double rad = 45.0 * M_PI / 180.0;

uint64_t seconds = 2 * 3600 + 30 * 60;


uint64_t weight = 2_ton;

double rad = 45.0_degrees;

uint64_t seconds = 2_hours + 30_minutes;
```

```
long double operator""_degrees(long double value)
{
  return value * M_PI / 180.0;
}

template <typename CharT>
std::basic_string<CharT> operator""_str(const CharT *str,
                                        size_t size)
{
  return { str, size };
}
```

```
unsigned long long operator "" _b(const char *str)
{
  unsigned long long result = 0;
  size_t size = strlen(str);

  for (size_t i = 0; i < size; ++i)
  {
    assert(str[i] == '1' || str[i] == '0');
    result |= (str[i] - '0') << (size - i - 1);
  }

  return result;
}
```

static_assert

```
template <typename T>
class MyVectorForPODs
{
  ....
};

struct POD { .... };
struct NonPOD { .... };

MyVectorForPODs<POD>    vec1; // ok
MyVectorForPODs<NonPod> vec2; // want compile-time error
```

```
template <typename T>
class MyVectorForPODs
{
  static_assert(std::is_pod<T>::value, "Vector can be used "
                                       "only with POD-classes");
  ....
};

struct POD { .... };
struct NonPOD { .... };

MyVectorForPODs<POD>    vec1; // ok
MyVectorForPODs<NonPod> vec2; // compile-time error
```

Alignof / alignas

```
__declspec(align(16)) float a[4] = { 300.0, 4.0, 4.0, 12.0 };
__declspec(align(16)) float b[4] = {   1.5, 2.5, 3.5,  4.5 };

__asm {
    movups xmm0, a     ; // поместить a в xmm0
    movups xmm1, b     ; // поместить b в xmm1
    mulps xmm0, xmm1   ; // перемножить xmm0 и xmm1

    movups a, xmm0     ; // выгрузить результаты из xmm0 в a
};
```

```
alignas(16) float a[4] = { 300.0, 4.0, 4.0, 12.0 };
alignas(16) float b[4] = {   1.5, 2.5, 3.5,  4.5 };

__asm {
    movups xmm0, a     ; // поместить a в xmm0
    movups xmm1, b     ; // поместить b в xmm1
    mulps xmm0, xmm1   ; // перемножить xmm0 и xmm1

    movups a, xmm0     ; // выгрузить результаты из xmm0 в a
};
```

Attributes

MSVC:

```
__declspec(align(#))
__declspec(noreturn)
__declspec(novtable)
__declspec(nothrow)
__declspec(restrict)
__declspec(deprecated)
__declspec(noalias)
....
```

GCC/Clang:

```
__attribute__((noreturn))
__attribute__((pure))
__attribute__((deprecated))
__attribute__((unused))
__attribute__((hot))
__attribute__((cold))
__attribute__((fall-
through))
__attribute__((aligned(#)))
  ....
```

```
[[attribute-list]]

attribute-list:

  identifier : [[noreturn]],
               [[carries_dependencies]]
  attribute-namespace::identifier : [[gnu::unused]]
  identifier(argument-list)
  attribute-namespace::identifier(argument-list)
```

```
[[noreturn]] void foo();

void bar(size_t N)
{
  char *p = static_cast<char*>(malloc(N));
  if (p == nullptr)
  {
    foo();
  }

  p[0]; // Dereferencing of the correct pointer
}
```

Updates to the standard library

New language features in the standard library

- std::container<T>::emplace()
- std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end
- std::associative_container<T>::emplace_hint()
- std::sequence_container<T>::shrink_to_fit()
- std::vector<T>::data()
- std::list<T> complexity constraints
- No COW in std::string

- New language features in the standard library
- **std::container<T>::emplace()**
- std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end
- std::associative_container<T>::emplace_hint()
- std::sequence_container<T>::shrink_to_fit()
- std::vector<T>::data()
- std::list<T> complexity constraints
- No COW in std::string

```cpp
#include <cstring>

class SomeClass
{
public:
  SomeClass(std::string name, std::string surname)
    : m_name { std::move(name) }
    , m_surname { std::move(surname) } { .... }
private:
  std::string m_name, m_surname;
};

void foo
{
  std::vector<SomeClass> v;
  v.push_back({ "foo", "bar" }); // std::vector<T>::push_back(T &&)
  v.emplace_back("foo", "bar");  // std::vector<T>::emplace_back(T &&...)
}
```

- New language features in the standard library
- std::container<T>::emplace()
- **std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end**
- std::associative_container<T>::emplace_hint()
- std::sequence_container<T>::shrink_to_fit()
- std::vector<T>::data()
- std::list<T> complexity constraints
- No COW in std::string

- New language features in the standard library
- std::container<T>::emplace()
- std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end
- **std::associative_container<T>::emplace_hint()**
- std::sequence_container<T>::shrink_to_fit()
- std::vector<T>::data()
- std::list<T> complexity constraints
- No COW in std::string

- New language features in the standard library
- std::container<T>::emplace()
- std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end
- std::associative_container<T>::emplace_hint()
- **std::sequence_container<T>::shrink_to_fit()**
- std::vector<T>::data()
- std::list<T> complexity constraints
- No COW in std::string

- New language features in the standard library
- std::container<T>::emplace()
- std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end
- std::associative_container<T>::emplace_hint()
- std::sequence_container<T>::shrink_to_fit()
- **std::vector<T>::data()**
- std::list<T> complexity constraints
- No COW in std::string

- New language features in the standard library
- std::container<T>::emplace()
- std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end
- std::associative_container<T>::emplace_hint()
- std::sequence_container<T>::shrink_to_fit()
- std::vector<T>::data()
- **std::list<T> complexity constraints**
- No COW in std::string

- New language features in the standard library
- std::container<T>::emplace()
- std::container<T>::cbegin, std::container<T>::cend, std::begin, std::end
- std::associative_container<T>::emplace_hint()
- std::sequence_container<T>::shrink_to_fit()
- std::vector<T>::data()
- std::list<T> complexity constraints
- **No COW in std::string**

std::tuple

```
#include <tuple>

void foo()
{
  std::tuple<size_t, double, std::string> t { 24, 4.9, "Phillip" };
  auto t = make_tuple(24ull, 4.9, std::string { "Phillip" });

  auto  age = std::get<0>(t); // size_t
  auto rank = std::get<1>(t); // double
  auto name = std::get<2>(t); // std::string
}
```

```
void get_person(size_t id,
                std::string &name,
                std::string &surname, ....) { .... }
```

```
void get_person(size_t id,
                std::string &name,
                std::string &surname, ....) { .... }

std::tuple<std::string, std::string, ....> get_person(size_t id)
{
  ....
}
```

```
std::tuple<std::string, std::string, ....> get_person(size_t id)
{
  ....
}

void foo()
{
  std::string name, surname;
  std::tie(name, surname, ....) = get_person(1);
  // std::tuple<std::string &, std::string &, ....>
}
```

```
struct SomeStruct
{
  int n; std::string s; float d;
};

bool operator<(const SomeStruct &l, const SomeStruct &r)
{
  return l.n < r.n
      && l.s < r.s
      && l.d < r.d;
}
```

```
struct SomeStruct
{
  int n; std::string s; float d;
};

bool operator<(const SomeStruct &l, const SomeStruct &r)
{
  return std::tie(l.n, l.s, l.d) < std::tie(r.n, r.s, r.d);
}
```

Associative unordered containers

```
template <class Key,
          class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Allocator = std::allocator<Key>>
class unordered_(multi)set;

template <class Key,
          class T,
          class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Allocator = std::allocator<std::pair<const Key,
                                                     T>>>
class unordered_(multi)map;
```

Smart pointers

```
int foo()
{
  FILE *f = fopen("/path/to/file", "r");
  ....
  if (!some_expression)
    return -1;
  ....
  fclose(f);
  return 0;
}
```

```
int foo()
{
  FILE *f = fopen("/path/to/file", "r");
  ....
  if (!some_expression)
    return -1;
  ....
  fclose(f);
  return 0;
}
```

```
#include <memory>

int foo()
{
  const auto deleter = [](FILE *fp) { fclose(fp); };
  using SmartFile = std::unique_ptr<FILE, decltype(deleter)>;

  SmartFile f { fopen("/path/to/file", "r"), deleter };
  ....
  if (!some_expression)
    return -1;
  ....
  return 0;
}
```

```
template <class T, class Deleter = std::default_delete<T>>
class unique_ptr;

template <class T>
class shared_ptr;

template <class T>
class weak_ptr;
```

std::function

```
#include <string>
#include <algorithm>

using LineChecker = bool (*)(const std::string &);

void foo(const std::string &line,
         const std::vector<LineChecker> &checks)
{
  for (auto check : checks)
  {
    if (check(line)) { .... }
  }
}
```

```
#include <string>
#include <algorithm>

using LineChecker = bool (*)(const std::string &);

bool ContainsBadWords(const std::string &line) { .... }
....

foo("bar", { &ContainsBadWords });
```

```
class SomeParser
{
  bool operator()(const std::string &) { .... }
};
....

SomeParser parser;
foo("bar", { ....,
             [](const std::string &) -> bool { .... },
             std::bind(&SomeParser::operator(),
                       parser,
                       std::placeholders::_1) });
```

```
template <class>
class function;

template <class R, class ...Args>
class function<R(Args...)>;

template <class M, class T>
/*unspecified*/ mem_fn(M T::*pm);

template <class F, class ...Args>
/* unspecified */ bind(F &&f, Args &&...args);

template <class R, class F, class ...Args>
/* unspecified */ bind(F &&f, Args &&...args);
```

```
using LineChecker = std::function<bool(const std::string &)>;

bool ContainsBadWords(const std::string &line) { .... }

Class SomeParser
{
  bool operator()(const std::string &) { .... }
};
....

SomeParser parser;
foo("bar", { &ContainsBadWords,
             [](const std::string &) -> bool { .... },
             std::bind(&SomeParser::operator(),
                       parser,
                       std::placeholders::_1) });
```

std::reference_wrapper

```
#include <vector>

class SlowCopyable { .... };

void foo(const std::vector<SlowCopyable> &vec)
{
  auto tmp = vec;
  std::sort(tmp.begin(), tmp.end());
  ....
}
```

```
#include <vector>

class SlowCopyable { .... };

void foo(const std::vector<SlowCopyable> &vec)
{
  using vector_type = typename std::decay<decltype(vec)>::type;
  using value_type = typename vector_type::value_type;
  using const_reference = std::reference_wrapper<const value_type>;

  std::vector<const_reference> tmp { vec.begin(), vec.end() };
  std::sort(tmp.begin(), tmp.end());
  ....
}
```

```
template <class T>
class reference_wrapper;

template <class T>
std::reference_wrapper<T> ref(T &t) noexcept;

template <class T>
void ref(const T&&) = delete;

template <class T>
std::reference_wrapper<const T> cref(const T &t) noexcept;

template <class T>
void cref(const T&&) = delete;
```

Regex library

```cpp
#include <regex>

void foo(const std::string &str) // "I was born 01.11.1994"
{
  std::regex pattern { R"((\d{2}).(\d{2}).(\d{2,4}))" };
  for (auto it = std::sregex_iterator { str.begin(), str.end(), pattern },
            end = std::sregex_iterator {};
       it != end;
       ++it)
  {
    auto &&match = *it;
    std::string day     = match[1]; // "01"
    std::string month   = match[2]; // "11"
    std::string year    = match[3]; // "1994"
    std::string unknown = match[4]; // empty string
  }
}
```

```cpp
#include <regex>
#include <cstring>

std::string hideDates(const std::string &str)
{
  std::regex pattern { R"((\d{2}).(\d{2}).(\d{2,4}))" };
  return std::regex_replace(str, pattern, "xx.xx.xxxx");
}
```

Random library

```
#include <cstdlib>

void foo()
{
  srand(time(0));
  int random_variable = rand();
}
```

```
template <class UIntType, UIntType a, UIntType c, UIntType m>
class linear_congruential_engine;

template <class UIntType, size_t w, size_t s, size_t r>
class subtract_with_carry_engine;

template <class UIntType, size_t w, size_t n, size_t m, size_t r,
          UIntType a, size_t u, UIntType d, size_t s,
          UIntType b, size_t t, UIntType c, size_t l,
          UIntType f>
class mersenne_twister_engine;

class random_device;
```

Uniform distributions

- uniform_int_distribution
- uniform_real_distribution

Bernoulli distributions

- bernoulli_distribution
- binomial_distribution
- negative_binomial_distribution
- geometric_distribution

Poisson distributions

- poisson_distribution
- exponential_distribution
- gamma_distribution
- weibull_distribution
- extreme_value_distribution

Normal distributions

- normal_distribution
- lognormal_distribution
- chi_squared_distribution
- cauchy_distribution
- fisher_f_distribution
- student_t_distribution

Sampling distributions

- discrete_distribution
- piecewise_constant_distribution
- piecewise_linear_distribution

```
#include <random>
#include <functional>
#include <iostream>

void foo()
{
  std::mt19937_64 engine { std::random_device{}() };
  std::uniform_int_distribution<> distr { 0, 100 };

  std::cout << distr(engine);

  auto generator = std::bind(distr, engine);
  std::cout << generator();
}
```

```
#include <random>
#include <functional>
#include <iostream>

void foo()
{
  std::mt19937_64 engine { std::random_device{}() };
  std::uniform_real_distribution<> distr { 0.0, 1.0 };

  std::cout << distr(engine);

  auto generator = std::bind(distr, engine);
  std::cout << generator();
}
```

Chrono library

```c
#include <time.h>

typedef /* unspecified */ clock_t; // Arithmetic type (until C11)
                                   // Real type (since C11)

void SomeBigFunction(....) { .... }

void foo()
{
  const clock_t start = clock();
  SomeBigFunction(....);
  const clock_t end = clock();

  const double duration = ((double) end - start) / CLOCKS_PER_SEC;
}
```

```
template <intmax_t Num, intmax_t Denom = 1>
class ratio;

template <class Rep, class Period = std::ratio<1>>
class duration; // (nano / micro / milli) seconds, minutes, ...

template <class Clock,
          class Duration = typename Clock::duration>
class time_point; // time since epoch

class system_clock;
class steady_clock;
Class high_resolution_clock;
```

```
#include <chrono>

void SomeBigFunction(....) { .... }

void foo()
{
  using namespace std::chrono;

  const auto start = steady_clock::now();
  SomeBigFunction(....);
  const auto end = steady_clock::now();

  const auto ns_duration   = end – start;
}
```

```
void foo()
{
  ....

  const auto ns_duration   = end - start;
  const auto us_duration   = duration_cast<microseconds>(end - start);
  const auto ms_duration   = duration_cast<milliseconds>(end - start);
  const auto s_duration    = duration_cast<seconds>(end - start);
  const auto min_durtation = duration_cast<minutes>(end - start);
  const auto h_duration    = duration_cast<hours>(end - start);
}
```

```
    using days = duration<int, ratio<86400>>;
// using days = duration<int, ratio_multiply<ratio<3600>, ratio<24>>;

using       udays = duration<uint32_t, ratio<86400>>;
using usafe_days = duration<safeint<int>, ratio<86400>>;

using fdays = duration<float, ratio<86400>>;
using ddays = duration<double, ratio<86400>>;
```

```
void foo()
{
   ....

   const auto d_duration     = duration_cast<days>(end - start);
   const auto ud_duration    = duration_cast<udays>(end - start);
   const auto usafe_duration = duration_cast<usafe_days>(end - start);

   const auto fd_duration = end - start;
   const auto dd_duration = end - start;
}
```

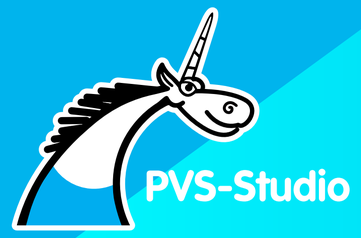
END

Q&A

Филипп Хандельянц

Лекция 2/12

Нововведения стандарта C++14



First X3J16 meeting

Somerset, NJ, USA (1990)

Completed C++11

Madrid, Spain (2011)

Completed C++14

Issaquah, WA, USA (2014)

Photo: Chandler Carruth and Olivier Giroux. License: tinyurl.com/9wn439f

Completed C++17

Kona, HI, USA (2017)

First X3J16 meeting

Somerset, NJ, USA (1990)

Completed C++11

Madrid, Spain (2011)

Completed C++14

Issaquah, WA, USA (2014)

Photo: Chandler Carruth and Olivier Giroux. License: tinyurl.com/9wn439f

Completed C++17

Kona, HI, USA (2017)

Внесенные изменения

Расширения ядра языка:

- Function return type deduction
- decltype(auto)
- Relaxed constexpr rules
- Variable templates
- Aggregate member initialization
- Binary literals
- Digit separators
- Generic lambdas
- Lambda capture expressions
- [[deprecated]]
- Memory allocation elision / combining

Расширения стандартной библиотеки:

- Heterogeneous lookup in associative
- containers
- Standard user-defined literals
- Tuple addressing via type
- Small new features

Function return type deduction

```
template <typename T1, typename T2>
auto sum(const T1 &lhs, const T2 &rhs) -> decltype(lhs + rhs)
{
  return lhs + rhs;
}
```

```
template <typename T1, typename T2>
auto sum(const T1 &lhs, const T2 &rhs) -> decltype(lhs + rhs)
{
  return lhs + rhs;
}
```

```
auto Factorial(size_t n);

auto Factorial(size_t n)
{
  if (n == 0)
  {
    return size_t {1};
  }

  return n * Factorial(n – 1);
}
```

```
auto Factorial(size_t n);

auto Factorial(size_t n)
{
  if (n != 0)
  {
    return n * Factorial(n – 1); // compile-time error
  }

  return size_t {1};
}
```

```
std::vector<size_t> foo(size_t n)
{
  static size_t called = 2;

  std::vector<size_t> vec;
  vec.reserve(n);

  std::generate_n(std::back_inserter(vec), n,
                  [&called]()
                  {
                    ++called;
                    if (called % 2 == 1)
                      return (called - 1) * called;
                    else
                      return called * called;
                  });
  return vec;
}
```

decltype(auto)

```
double   foo();
double&& bar();

      double  v1 = 0.0; // double
const double &v2 = v1;  // const double &

decltype(v1)   v3 = v1;   // double
decltype((v1)) v4 = (v1); // double &
decltype(v2)   v5 = v2;   // const double &

decltype(foo()) v6 = foo(); // double
decltype(bar()) v7 = bar(); // double &&
```

```
double   foo();
double&& bar();

      double  v1 = 0.0; // double
const double &v2 = v1;  // const double &

decltype(auto) v3 = v1;   // double
decltype(auto) v4 = (v1); // double &
decltype(auto) v5 = v2;   // const double &

decltype(auto) v6 = foo(); // double
decltype(auto) v7 = bar(); // double &&
```

Relaxed constexpr rules

Что запрещено:

- Ассемблерные вставки
- 'goto'
- Любые метки, кроме 'case' и 'default' в 'switch'
- 'try'-блок
- Переменные нелитерального типа
- 'static' / 'thread_local' переменные
- Переменные без инициализации

```
constexpr size_t Factorial(size_t n)
{
  return n == 0 ? 1 : n * Factorial(n - 1);
}

constexpr size_t Factorial(size_t n) // since C++14
{
  size_t acc = 1;
  for (size_t i = 1; i <= n; ++i)
    acc *= i;

  return acc;
}
```

Variable templates

```
template <typename Ret, typename T1, typename T2>
Ret sum(const T1 &lhs, const T2 &rhs)
{
  return Ret { lhs + rhs };
}

void foo()
{
  std::string hello { "Hello, " };
  std::string world { "world" };

  using ConstStringRef = const std::string &;
  auto helloWorld = sum<ConstStringRef>(hello, world);
}
```

```
template <class T>
struct is_reference
{
  static constexpr bool value = false;
};

template <class T>
struct is_reference<T&>
{
  static constexpr bool value = true;
};

template <class T>
struct is_reference<T&&>
{
  static constexpr bool value = true;
};
```

```
template <typename Ret, typename T1, typename T2>
Ret sum(const T1 &lhs, const T2 &rhs)
{
  static_assert(!is_reference<Ret>::value, "Local non-static object "
                                           "can't be returned as "
                                           "reference, it's UB.");
  return Ret { lhs + rhs };
}
```

```
template <typename T>
constexpr bool is_reference_v = is_reference<T>::value; // since C++14

template <typename Ret, typename T1, typename T2>
Ret sum(const T1 &lhs, const T2 &rhs)
{
  static_assert(!is_reference_v<Ret>, "Temporary object can't "
                                      "be returned as reference, "
                                      "it's UB.");
  return Ret { lhs + rhs };
}
```

Aggregate initialization

with default member initialiazer

```
struct Person
{
  std::string m_firstName, m_secondName;
  uint8_t m_age;
};

void foo()
{
  Person p { "Phillip", "Khandeliants", 24 }; // ok

  // m_firstName == "Phillip"
  // m_secondName == "Khandeliants“
  // m_age == 24
}
```

```
struct Person
{
  std::string m_firstName, m_secondName;
  uint8_t m_age;
};

void foo()
{
  Person p { "Phillip", "Khandeliants" }; // ok

  // m_firstName == "Phillip"
  // m_secondName == "Khandeliants“
  // m_age == 0
}
```

```
struct Person
{
  std::string m_firstName, m_secondName;
  uint8_t m_age = std::numeric_limits<decltype(m_age)>::max();
};

void foo()
{
  Person p { "Phillip", "Khandeliants" }; // ok since C++14

  // m_firstName == "Phillip"
  // m_secondName == "Khandeliants“
  // m_age == 255
}
```

Binary literals

```
void foo(uint32_t bits)
{
  constexpr uint32_t mask = 15; // 1111₂
  uint8_t significant_bits = static_cast<uint8_t>(bits & mask);
  ....
}
```

```
void foo(uint32_t bits)
{
  constexpr uint32_t mask = 0x0F; // 1111₂
  uint8_t significant_bits = static_cast<uint8_t>(bits & mask);
  ....
}
```

```
void foo(uint32_t bits)
{
  constexpr uint32_t mask = 0x0B; // 1011₂
  uint8_t significant_bits = static_cast<uint8_t>(bits & mask);
  ....
}
```

```
uint64_t operator""_b(const char *p)
{
  const size_t len = strlen(p);

  uint64_t res = 0;
  for (size_t i = 0; i < len; ++i)
  {
    assert(p[i] == '0' || p[i] == '1');
    result |= (p[i] - '0') << (size - i - 1);
  }

  return res;
}

void foo(uint32_t bits)
{
  constexpr uint32_t mask = 1011_b; // 1011₂
  uint8_t significant_bits = static_cast<uint8_t>(bits & mask);
  ....
}
```

```
template <char ...bits>
struct to_binary;

template <char high_bit, char ...bits>
struct to_binary<high_bit, bits...>
{
  static_assert(high_bit == '0' || high_bit == '1', "Not a binary value!");

  constexpr size_t bitsCount = sizeof...(bits);
  static const unsigned long long value = (high_bit - '0') << bitsCount
                                        | to_binary<bits...>::value;
};

template <char high_bit>
struct to_binary<high_bit>
{
  static_assert(high_bit == '0' || high_bit == '1', "Not a binary value!");
  static const unsigned long long value = (high_bit - '0');
};
```

```
template <char ...bits>
constexpr unsigned long long operator""_b()
{
  return to_binary<bits...>::value;
}

void foo(uint32_t bits)
{
  constexpr uint32_t mask = 1111_b; // 1111₂
  uint8_t significant_bits = static_cast<uint8_t>(bits & mask);
  ....
}
```

```
void foo(uint32_t bits)
{
  constexpr uint32_t mask = 0b1011; // since C++14, 1011₂
  uint8_t significant_bits = static_cast<uint8_t>(bits & mask);
  ....
}
```

Digit separator

```
void foo(uint32_t bits)
{
  constexpr uint64_t max = 18446744073709551615;
  constexpr double pi    = 3.14159265358979323846264338327;
  ....
}
```

```
void foo(uint32_t bits)
{
  constexpr uint64_t max = 18'446'744'073'709'551'615;
  constexpr double pi    = 3.141'592'653'589'793'238'462'643'383;
  ....
}
```

```
void foo(uint32_t bits)
{
  constexpr uint64_t max = 18'446'744'073'709'551'615;
  constexpr double pi    = 3.141'592'653'589'793'238'462'643'383;
  ....

  constexpr uint64_t max = 1'8'4'4'6'7'4'4'0'7'3'7'0'9'5'5'1'6'1'5;
}
```

Generic lambdas

```
void foo()
{
  std::vector<size_t> vec;
  ....

  // Descending order
  std::sort(vec.begin(), vec.end(),
            [](size_t lhs, size_t rhs)
            {
              return lhs > rhs;
            });
}
```

```
void foo()
{
  std::vector<double> vec;
  ....

  // Descending order
  std::sort(vec.begin(), vec.end(),
            [](size_t lhs, size_t rhs) // probable data-loss
            {
              return lhs > rhs;
            });
}
```

```
void foo()
{
  std::vector<double> vec;
  ....

  // Descending order
  std::sort(vec.begin(), vec.end(),
            [](auto lhs, auto rhs) // since C++14
            {
              return lhs > rhs;
            });
}
```

```
[lambda-capture](params) mutable -> ret_type
{
  lambda_body
}

class LambdaInCompiler
{
private:
  lambda-capture-members...
public:
  template <typename ...params>
  ret_type operator()(params...) const { .... }
}
```

```
auto variadic_lambda = [](auto ...params)
{
  ....
};

auto variadic_lambda = [](auto &...params)
{
  ....
};

auto variadic_lambda = [](const auto &...params)
{
  ....
};

auto variadic_lambda = [](auto &&...params)
{
  ....
};
```

Lambda capture expressions

```
std::function<std::vector()> bar()
{
  auto deleter = [](FILE *handler) { fclose(handler); };

  std::unique_ptr<FILE, decltype(deleter)> f {
    fopen("/path/to/file", "r"),
    deleter
  };

  auto lambda = [=]{ .... }; // any manipulations with file
  return lambda;
}
```

```
std::function<std::vector()> bar()
{
  auto deleter = [](FILE *handler) { fclose(handler); };

  std::unique_ptr<FILE, decltype(deleter)> f {
    fopen("/path/to/file", "r"),
    deleter
  };

  auto lambda = [f]{ .... }; // any manipulations with file
  return lambda;
}
```

```
<source>:33:18: error: call to deleted constructor of 'std::unique_ptr<FILE,
decltype(deleter)>' (aka 'unique_ptr<_IO_FILE, (lambda at <source>:31:18)>')

  auto lambda = [f]{ .... };

                     ^

/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-li-
nux-gnu/8.3.0/../../../../include/c++/8.3.0/bits/unique_ptr.h:
394:7: note: 'unique_ptr' has been explicitly marked deleted here

      unique_ptr(const unique_ptr&) = delete;

      ^

1 error generated.

Compiler returned: 1
```

```
std::function<std::vector()> bar()
{
  auto deleter = [](FILE *handler) { fclose(handler); };

  std::unique_ptr<FILE, decltype(deleter)> f {
    fopen("/path/to/file", "r"),
    deleter
  };

  auto lambda = [file = std::move(f)]{ .... }; // any manipulations
                                               // with file
  return lambda;
}
```

```
void foo()
{
  int x = 4;
  auto lambda = [&r = x, x = x + 1]
  {
    r += 2;
    return x * x;
  };

  auto y = lambda();
  // x == 6, y == 25
  ....
}
```

[[deprecated]]

```
void foo() { .... }
```

```
void foo() { .... } // obsolete and should not be used

void bar() { .... }
```

```
[[deprecated]] void foo() { .... } // obsolete and should not be used

void bar() { .... }
```

```
[[deprecated]] void foo() { .... } // obsolete and should not be used

void bar() { .... }

void foobar()
{
  foo();
  bar();
}
```

```
<source>:35:5: warning: 'foo' is deprecated [-Wdeprecated-declarations]
    foo();
    ^
<source>:29:3: note: 'foo' has been explicitly marked deprecated here
[[deprecated]] void foo() { }
  ^
1 warning generated.
Compiler returned: 0
```

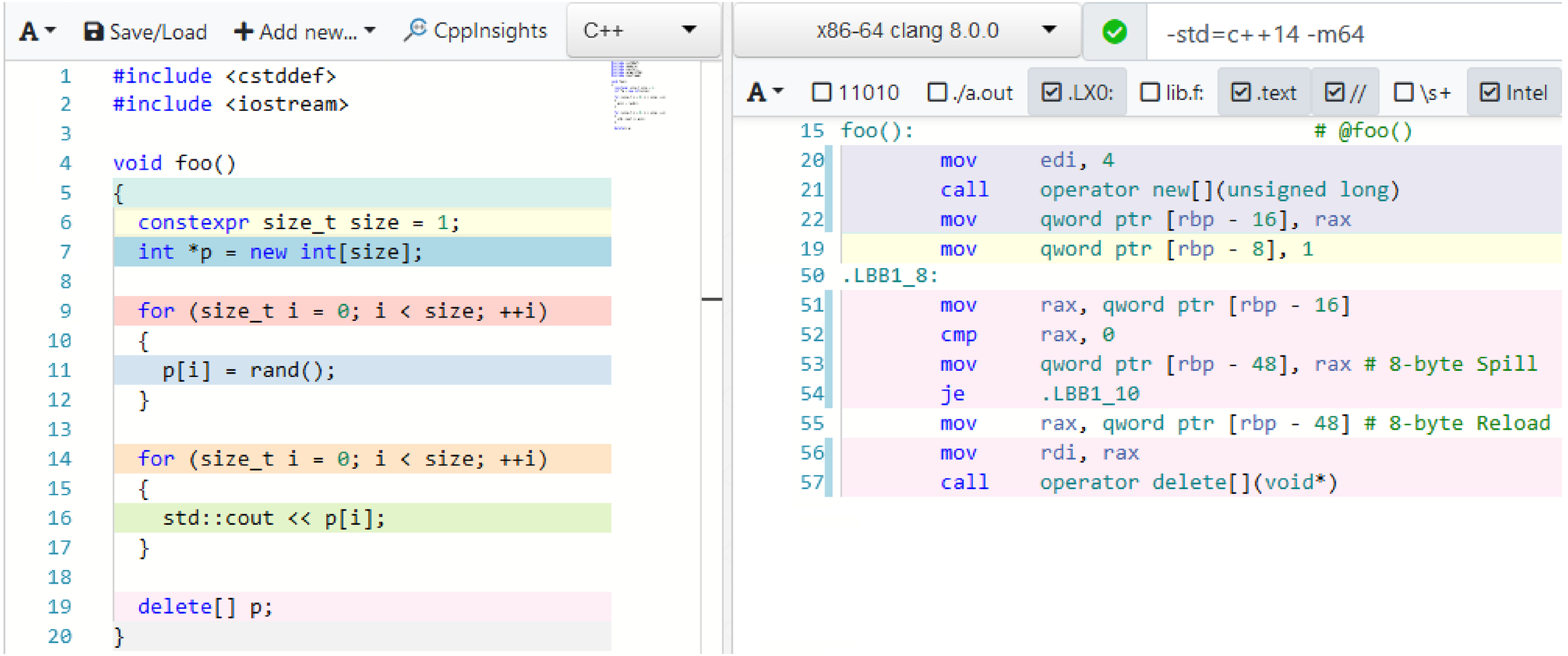
```
[[deprecated("don't use this dangerous function")]]
void foo() { .... } // obsolete and should not be used

void bar() { .... }

void foobar()
{
  foo();
  bar();
}
```

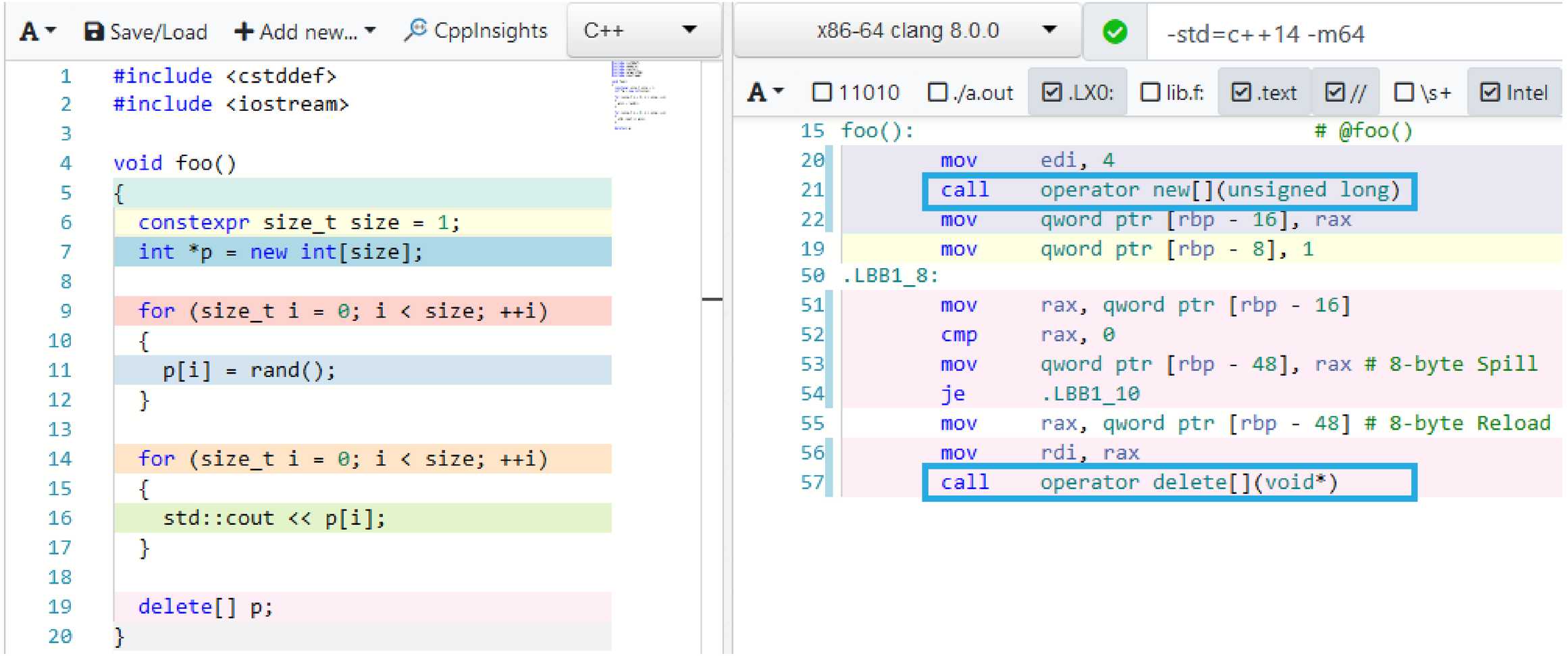
```
<source>:36:3: warning: 'foo' is deprecated: don't use this
dangerous function [-Wdeprecated-declarations]
  foo();
  ^
<source>:29:3: note: 'foo' has been explicitly marked deprecated here
[[deprecated("don't use this dangerous function")]]
  ^
1 warning generated.
Compiler returned: 0
```

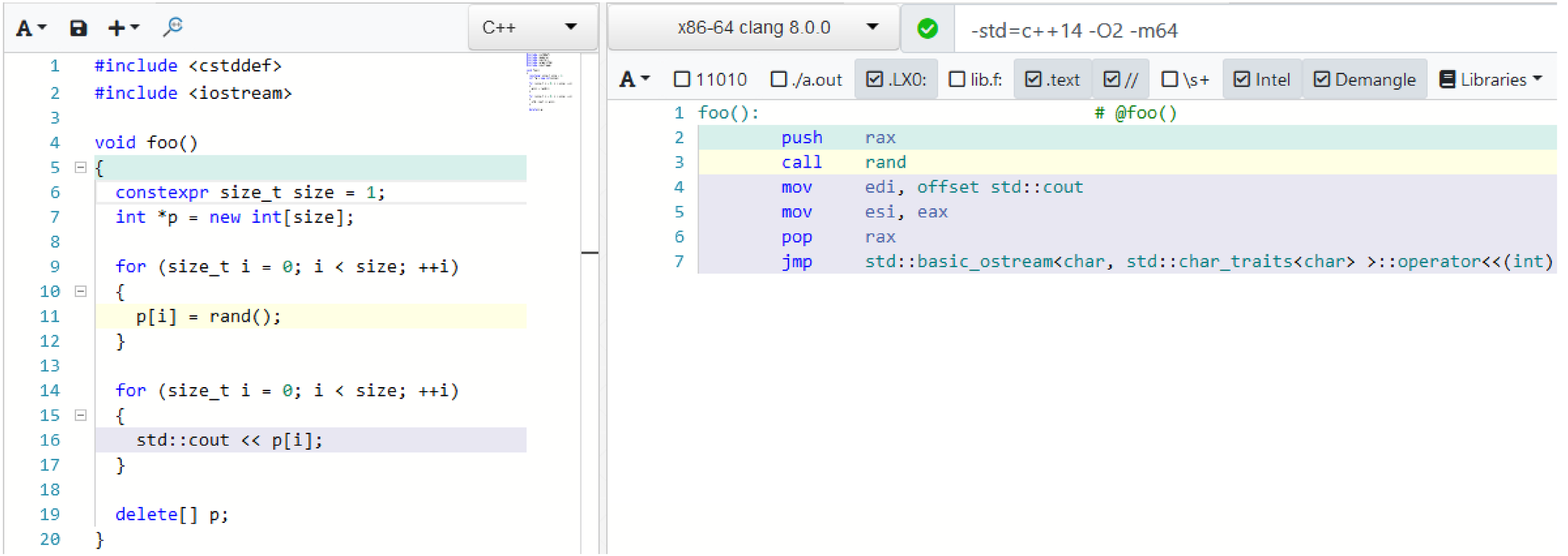
- struct [[deprecated]] SomeStruct;
- [[deprecated]] typedef S *PS;
- using PS [[deprecated]] = S*;
- [[deprecated]] int x;
- union U { [[deprecated]] int n; };
- [[deprecated]] void foo();
- namespace [[deprecated]] NS { int x; }
- enum [[deprecated]] E { .... };
- enum E { A [[deprecated]], B [[deprecated]] = 42 };
- template < > struct [[deprecated]] X<int> {};

Memory allocation elision / combining

```
void foo()
{
  constexpr size_t size = 3;
  int *p = new int[size];

  for (size_t i = 0; i < size; ++i)
  {
    p[i] = rand();
  }

  for (size_t i = 0; i < size; ++i)
  {
    std::cout << p[i];
  }

  delete[] p;
}
```


A
Save/Load
Add new...
CppInsights
C++
x86-64 clang 8.0.0
-std=c++14 -m64
A
11010
./a.out
.LX0:
lib.f:
.text
//
\s+
Intel
#include <cstddef>
#include <iostream>
void foo()
{
constexpr size_t size = 1;
int *p = new int[size];
for (size_t i = 0; i < size; ++i)
{
p[i] = rand();
}
for (size_t i = 0; i < size; ++i)
{
std::cout << p[i];
}
delete[] p;
}
foo(): # @foo()
mov edi, 4
call operator new[](unsigned long)
mov qword ptr [rbp - 16], rax
mov qword ptr [rbp - 8], 1
.LBB1_8:
mov rax, qword ptr [rbp - 16]
cmp rax, 0
mov qword ptr [rbp - 48], rax # 8-byte Spill
je .LBB1_10
mov rax, qword ptr [rbp - 48] # 8-byte Reload
mov rdi, rax
call operator delete[](void*)


Save/Load
Add new...
CppInsights
C++
x86-64 clang 8.0.0
-std=c++14 -m64
11010
./a.out
.LX0:
lib.f:
.text
//
\s+
Intel
#include <cstddef>
#include <iostream>
void foo()
{
constexpr size_t size = 1;
int *p = new int[size];
for (size_t i = 0; i < size; ++i)
{
p[i] = rand();
}
for (size_t i = 0; i < size; ++i)
{
std::cout << p[i];
}
delete[] p;
}
foo(): # @foo()
mov edi, 4
call operator new[](unsigned long)
mov qword ptr [rbp - 16], rax
mov qword ptr [rbp - 8], 1
.LBB1_8:
mov rax, qword ptr [rbp - 16]
cmp rax, 0
mov qword ptr [rbp - 48], rax # 8-byte Spill
je .LBB1_10
mov rax, qword ptr [rbp - 48] # 8-byte Reload
mov rdi, rax
call operator delete[](void*)


C++
x86-64 clang 8.0.0
-std=c++14 -O2 -m64
11010
./a.out
.LX0:
lib.f:
.text
//
\s+
Intel
Demangle
Libraries
#include <cstddef>
#include <iostream>
void foo()
{
constexpr size_t size = 1;
int *p = new int[size];
for (size_t i = 0; i < size; ++i)
{
p[i] = rand();
}
for (size_t i = 0; i < size; ++i)
{
std::cout << p[i];
}
delete[] p;
}
foo(): # @foo()
push rax
call rand
mov edi, offset std::cout
mov esi, eax
pop rax
jmp std::basic_ostream<char, std::char_traits<char> >::operator<<(int)

```
#include <vector>

void foo()
{
  constexpr size_t size = 42;
  std::vector<int> v(size, 0);

  for(const auto val : v)
  {
    std::cout << val << ' ';
  }
}
```

Heterogeneous lookup in associative containers

```
#include <set>
#include <string>

const char* getDaenerysTitle() { .... }

void foo()
{
  std::set<std::string> songOfIceAndFireTitles {
    "Queen Daenerys Stormborn of the House Targaryen, ....",
    "Robert Baratheon, King of the Andals, ....",
    "Tywin Lannister, Lord of Casterly Rock Shield of Lannisport, ....",
    ....
  };

  auto it = songOfIceAndFireTitles.find(getDaenerysTitle());
}
```

```
#include <set>
#include <string>
#include <functional>

const char* getDaenerysTitle() { .... }

void foo()
{
  std::set<std::string, std::less<>> songOfIceAndFireTitles {
    "Queen Daenerys Stormborn of the House Targaryen, ....",
    "Robert Baratheon, King of the Andals, ....",
    "Tywin Lannister, Lord of Casterly Rock Shield of Lannisport, ....",
    ....
  };

  auto it = songOfIceAndFireTitles.find(getDaenerysTitle());
}
```

```
template <class T = void>
struct less
{
  bool operator()(const T &lhs, const T &rhs)
  {
    return lhs < rhs;
  }
}
```

```
template <>
struct less<void>
{
  using is_transparent = ....; // unspecified

  template <class T, class U>
  bool operator()(T &&lhs, U &&rhs)
  {
    return std::forward<decltype(lhs)>(lhs) <
           std::forward<decltype(rhs)>(rhs);
  }
};
```

```
#include <set>
#include <string>
#include <functional>

const char* getDaenerysTitle() { .... }

void foo()
{
  std::set<std::string, std::less<>> songOfIceAndFireTitles {
    "Queen Daenerys Stormborn of the House Targaryen, ....",
    "Robert Baratheon, King of the Andals, ....",
    "Tywin Lannister, Lord of Casterly Rock Shield of Lannisport, ....",
   ....
  };

  auto it = songOfIceAndFireTitles.find(getDaenerysTitle());
}
```

Standard user-defined literals

```
template <typename CharT,
          typename Traits = std::char_traits<CharT>
          class Allocator = std::allocator<CharT>>
std::basic_string<CharT, Traits, Allocator>
operator""s(const CharT *str, size_t len)
{
  return { str, len };
}

using std::literals;
auto str = "True story, bro"s; // std::string
```

```
constexpr std::chrono::seconds operator""s(uint64_t secs)
{
  return std::chrono::seconds { secs };
}

constexpr std::chrono::duration<long double>
operator""s(long double secs)
{
  return std::chrono::duration<long double> { secs };
}

using std::literals;
auto half_min  = 30s;   // std::chrono::seconds
auto fhalf_min = 30.0s; // std::chrono::duration<long double>
```

```
constexpr std::complex<double> operator""i(uint64_t imag)
{
  return { 0.0, static_cast<double>(imag) };
}

constexpr std::complex<double> operator""i(long double
imag)
{
  return { 0.0, static_cast<double>(imag) };
}

using std::literals;
auto one_one = 1.0 + 1i; // std::complex<double>
```

Tuple addressing via type

```
using SomeTuple = std::tuple<int, std::string, float>;

SomeTuple foo(....) { .... }

void bar()
{
  SomeTuple t = foo();
  int         &x = std::get<0>();
  std::string &y = std::get<1>();
  float       &z = std::get<2>();
}
```

```
using SomeTuple = std::tuple<std::string, int, float>;

SomeTuple foo(....) { .... }

void bar()
{
  SomeTuple t = foo();
  int         &x = std::get<0>(); // compile-time error
  std::string &y = std::get<1>();
  float       &z = std::get<2>();
}
```

```
using SomeTuple = std::tuple<std::string, int, float>;

SomeTuple foo(....) { .... }

void bar()
{
  SomeTuple t = foo();
  auto &x = std::get<float>();        // float
  auto &y = std::get<std::string>(); // std::string
  auto &z = std::get<int>();          // int
}
```

Small new features

Small new features

- std::make_unique
- std::exchange
- std::cbegin / std::cend / std::rbegin / std::rend / std::crbegin / std::crend

```
#include <memory>
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
public:
  SomeClass(std::string name, std::string surname)
    : m_name { std::move(name) }, m_surname { std::move(surname) }
  { .... }
}

void foo()
{
  std::shared_ptr<SomeClass> p { new SomeClass { "Foo", "Bar" } };
}
```

```
#include <memory>
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
public:
  SomeClass(std::string name, std::string surname)
    : m_name { std::move(name) }, m_surname { std::move(surname) }
  { .... }
}

void foo()
{
  std::shared_ptr<SomeClass> p { new SomeClass { "Foo", "Bar" } };
  auto q = std::make_shared<SomeClass>("Foo", "Bar");
}
```

```
#include <memory>
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
public:
  SomeClass(std::string name, std::string surname)
    : m_name { std::move(name) }, m_surname { std::move(surname) }
  { .... }
}

void foo()
{
  std::unique_ptr<SomeClass> p { new SomeClass { "Foo", "Bar" } };
  auto q = std::make_unique<SomeClass>("Foo", "Bar");
}
```

Small new features

- std::make_unique
- std::exchange
- std::cbegin / std::cend / std::rbegin / std::rend / std::crbegin / std::crend

```
template <class T, class U = T>
constexpr T exchange(T &obj, U &&new_value) noexcept(noexcept(....))
{
  T old_value = std::move(obj);
  obj = std::forward<U>(new_value);
  return old_value;
}
```

```
#include <vector>

template <typename T>
void perform_actions_and_clear(std::vector<T> &vec)
{
  for (const auto &elem : vec)
    // some manipulations with elements

  vec.clear();
}
```

```cpp
#include <vector>

template <typename T>
void perform_actions_and_clear(std::vector<T> &vec)
{
  for (const auto &elem : std::exchange(vec, {}))
    // some manipulations with elements
}
```

```
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
public:
  SomeClass(SomeClass &&other)
    : m_name { std::exchange(other.m_name, {}) }
    , m_surname { std::exchange(other.m_surname, {}) } { .... }

  SomeClass& operator=(SomeClass &&other)
  {
    m_name    = std::exchange(other.m_name, {});
    m_surname = std::exchange(other.m_surname, {});
  }
}
```

- std::make_unique
- std::exchange
- std::cbegin / std::cend / std::rbegin / std::rend / std::crbegin / std::crend

END

Q&A