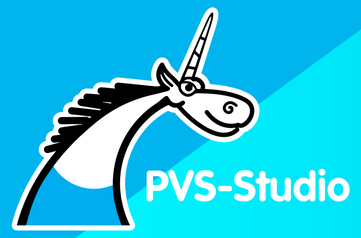
Филипп Хандельянц

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

“Есть всего два типа языков программирования:
те, на которые люди всё время ругаются,
и те, которые никто не использует.”

© Бьерн Страуструп

Где C++ живет

и по-прежнему жжёт

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)
- Высоконагруженные сервера (Яндекс.Такси)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)
- Высоконагруженные сервера (Яндекс.Такси)
- Поисковые движки (Google, Яндекс, ...)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)
- Высоконагруженные сервера (Яндекс.Такси)
- Поисковые движки (Google, Яндекс, ...)
- Компиляторы, виртуальные машины (GCC, Clang, JVM, CLR, ...)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)
- Высоконагруженные сервера (Яндекс.Такси)
- Поисковые движки (Google, Яндекс, ...)
- Компиляторы, виртуальные машины (GCC, Clang, JVM, CLR, ...)
- Научные программы (CERN)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)
- Высоконагруженные сервера (Яндекс.Такси)
- Поисковые движки (Google, Яндекс, ...)
- Компиляторы, виртуальные машины (GCC, Clang, JVM, CLR, ...)
- Научные программы (CERN)
- Операционные системы (драйвера, userspace)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)
- Высоконагруженные сервера (Яндекс.Такси)
- Поисковые движки (Google, Яндекс, ...)
- Компиляторы, виртуальные машины (GCC, Clang, JVM, CLR, ...)
- Научные программы (CERN)
- Операционные системы (драйвера, userspace)
- Встаиваемые системы (BMW, Tesla, Toyota, ...)

- Игровые движки (Unreal Engine, CryEngine, idTech, Serious Engine, ...)
- Спецэффекты и анимация (Ps, Disney, Pixar, ...)
- Браузеры (Chromium, Яндекс.Браузер, Mozilla Firefox, ...)
- Высоконагруженные сервера (Яндекс.Такси)
- Поисковые движки (Google, Яндекс, ...)
- Компиляторы, виртуальные машины (GCC, Clang, JVM, CLR, ...)
- Научные программы (CERN)
- Операционные системы (драйвера, userspace)
- Встаиваемые системы (BMW, Tesla, Toyota, ...)

....

ТЫСЯЧИ IT-СФЕР!!!

Популярность C++

- Нулевые накладные расходы (zero-overhead, «don’t pay for what you don’t use»)
- Неограниченные возможности
- Работает на большом числе платформ
- Безопасность
- Небольшой рантайм

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:

- 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

Multithreading

- До C++11 формально не существовало многопоточных приложений
- Все городили свои малые архитектурные решения на основе WinAPI, pthread, Cocoa, ...
- C++11 добавил:

Потоки выполнения	std::thread
Механизмы синхронизации	std::mutex std::recursive_mutex std::timed_mutex std::recursive_timed_mutex std::conditional_variable
Модели и барьеры памяти	std::memory_order std::atomic_thread_fence
Атомарные переменные	std::atomic
Асинхронные вычисления	std::future std::packaged_task std::promise

```
#include <thread>

void threadFunction(int i, double d, const std::string &s)
{
  std::cout << i << ", " << d << ", " << s << std::endl;
}

int main()
{
  std::thread thr { threadFunction, 1, 2.34, "example" };
  thr.join();
  return 0;
}
```

Uniform initialization

Как мы можем инициализировать объект значением в C++?

```
int i1;                     // (1) default initialization
int i2(42);                 // (2) direct initialization
int i3 = 42;                // (3) copy initialization
int i4 = int();             // (4) value initialization
int arr[] = { 1, 2, 3 };    // (5) aggregate initialization

struct Point { double x, y; } point { 0.0, 0.0 };                  // (5)
std::complex<double> cmpl(0.0, 0.0);                               // (2)
std::complex<double> cmpl = std::complex<double>(0.0, 0.0); // (3)
```

С новым синтаксисом C++:

```
int i1;                     // (1) default initialization
int i2 { 42 };              // (2) direct initialization
int i3 = { 42 };            // (3) copy initialization
int i4 {};                  // (4) value initialization
int arr[] = { 1, 2, 3 };    // (5) aggregate initialization

struct Point { double x, y; } point { 0.0, 0.0 };              // (5)
std::complex<double> cmpl { 0.0, 0.0 };                        // (2)
std::complex<double> cmpl = std::complex<double> { 0.0, 0.0 }; // (3)
```

Компилируется ли этот код?

```
struct Timer
{
  Timer();
};

struct TimeKeeper
{
  TimeKeeper(const Timer &t);
  int get_time();
};
....

TimeKeeper time_keeper(Timer());
int time = time_keeper.get_time();
```

Компилируется ли этот код? Нет :)

```
MSVC
<source>(13): error C2228: left of '.get_time' must have class/struct/union

Compiler returned: 2


GCC
<source>:13:24: error: request for member 'get_time' in 'time_keeper', which is of non-class type
'TimeKeeper(Timer (*)())'

   13 | int time = time_keeper.get_time();

      |                        ^~~~~~~~

Compiler returned: 1


Clang
<source>:13:23: error: member reference base type 'TimeKeeper (Timer (*)())' is not a structure or
union

int time = time_keeper.get_time();

           ~~~~~~~~~~~^~~~~~~~~

1 error generated.

Compiler returned: 1
```

```
struct Timer
{
  Timer();
};

struct TimeKeeper
{
  TimeKeeper(const Timer &t);
  int get_time();
};
....

TimeKeeper time_keeper { Timer() };
int time = time_keeper.get_time();
```

std::initializer_list<T>

```
std::vector<std::string> musicGenres;

musicGenres.push_back("pop");
musicGenres.push_back("metal");
musicGenres.push_back("rap");
musicGenres.push_back("jazz");
musicGenres.push_back("reggae");
musicGenres.push_back("folk");
musicGenres.push_back("blues");
....
```

```
std::vector<std::string> musicGenres;
musicGenres.reserve(...);

musicGenres.push_back("pop");
musicGenres.push_back("metal");
musicGenres.push_back("rap");
musicGenres.push_back("jazz");
musicGenres.push_back("reggae");
musicGenres.push_back("folk");
musicGenres.push_back("blues");
....
```

```
std::vector<std::string> musicGenres {
   "pop",
   "metal",
   "rap",
   "jazz",
   "reggae",
   "folk",
   "blues"
};
```

```
#include <vector>
#include <initializer_list>

template <typename T>
class MyStack
{
  ....
  MyStack(std::initializer_list<T> list)
  {
    m_vec.reserve(list.size());
    typedef std::initializer_list<T>::const_iterator list_iter;
    for (list_iter it = list.begin(); it != list.end(); ++it)
      m_vec.push_back(*it);
  }

  std::vector<T> m_vec;
};
```

Move semantics

```
class MyString
{
  char *m_data;
  size_t m_size;
  size_t m_capacity;
  ....
  MyString(const MyString &str)
    : m_data(strdup(str.m_data)) { .... }
  MyString(const char *str)
    : m_data(strdup(str)) { .... }
};
```

```
std::vector<std::pair<MyString, int>> data;

data.reserve(1000000);
for (size_t i = 0; i < 1000000; ++i)
  myMap.push_back(std::pair<MyString, int> { "Hello", i });
```

```
"Hello" -> MyString (new + memcpy)
MyString -> std::pair<MyString, int> (new + memcpy)
std::pair<MyString, int> -> std::vector<...> (new + memcpy)
```

```
class MyString
{
  MyString(MyString &&str) { std::swap(m_data, str.m_data); .... }
  MyString& operator=(MyString &&str) { .... }
  ....
  char *m_data;
  size_t m_size;
  size_t m_capacity;
};
```

```
class MyString
{
  MyString(MyString &&str) { std::swap(m_data, str.m_data); .... }
  MyString& operator=(MyString &&str) { .... }
  ....
  char *m_data;
  size_t m_size;
  size_t m_capacity;
};

template <class T>
typename std::remove_reference<T>::type&& move(T &&t) noexcept
{
  return static_cast<T&&>(t);
}
```

```
"Hello" -> MyString (new + memcpy)
MyString -> std::pair<MyString, int> (move)
std::pair<MyString, int> -> std::vector<...> (move)
```

noexcept

```
void foobar();

void foo() throw(A, B)
{
  foobar();
}
```

```
void foobar();

void foo() throw(A, B)
{
  try
  {
    foobar();
  }
  catch (A &) { throw; }
  catch (B &) { throw; }
  catch (...) { std::unexpected(); }
}
```

```
void foobar();

void foo() throw()
{
  try
  {
    foobar();
  }
  catch (...) { std::unexpected(); }
}
```

```
void foobar();

void foo() noexcept
{
   ....
}

void foo() noexcept(compile-time-expr)
{
   ....
}
```

```
class FastMove_SlowCopy
{
  ....
public:
  FastMove_SlowCopy(FastMove_SlowCopy &&);
  FastMove_SlowCopy(const FastMove_SlowCopy &);

  FastMove_SlowCopy& operator=(FastMove_SlowCopy &&);
  FastMove_SlowCopy& operator=(const FastMove_SlowCopy &);
};
```

```
class FastMove_SlowCopy
{
   ....
public:
   FastMove_SlowCopy(FastMove_SlowCopy &&) noexcept;
   FastMove_SlowCopy(const FastMove_SlowCopy &);

   FastMove_SlowCopy& operator=(FastMove_SlowCopy &&) noexcept;
   FastMove_SlowCopy& operator=(const FastMove_SlowCopy &);
};
```

Variadic templates

```
int printf(const char * const format, ...);

int x = 10;
double y = 0.0;
printf("My beautiful string, x = %i, y = %i\n", x, y);
```

```
int printf(const char * const format, ...);

int x = 10;
double y = 0.0;
printf("My beautiful string, x = %i, y = %i\n", x, y);
    // incorrect specifier, expected “%f” ^
```

```
template <typename ...Args>
void printf(const char * const format, const Args &...args);

int x = 10;
double y = 0.0;
printf("My beautiful string, x = {}, y = {}\n", x, y);
```

```
template <typename ...Args>
void printf(const char * const format, const Args &...args);

int x = 10;
double y = 0.0;
printf("My beautiful string, x = {}, y = {}\n", x, y);

printf<int, double>(const char * const format,
                    int arg1,
                    double arg2);
```

```
template <typename ...T>
struct tuple;

template <typename Head, typename ...Tail>
struct tuple<Head, Tail...> : tuple<Tail...> { .... };

template <>
struct tuple<> {};
```

Constant expressions (constexpr)

```
size_t ArraySize() { return 42; }
int some_value[ArraySize()];
```

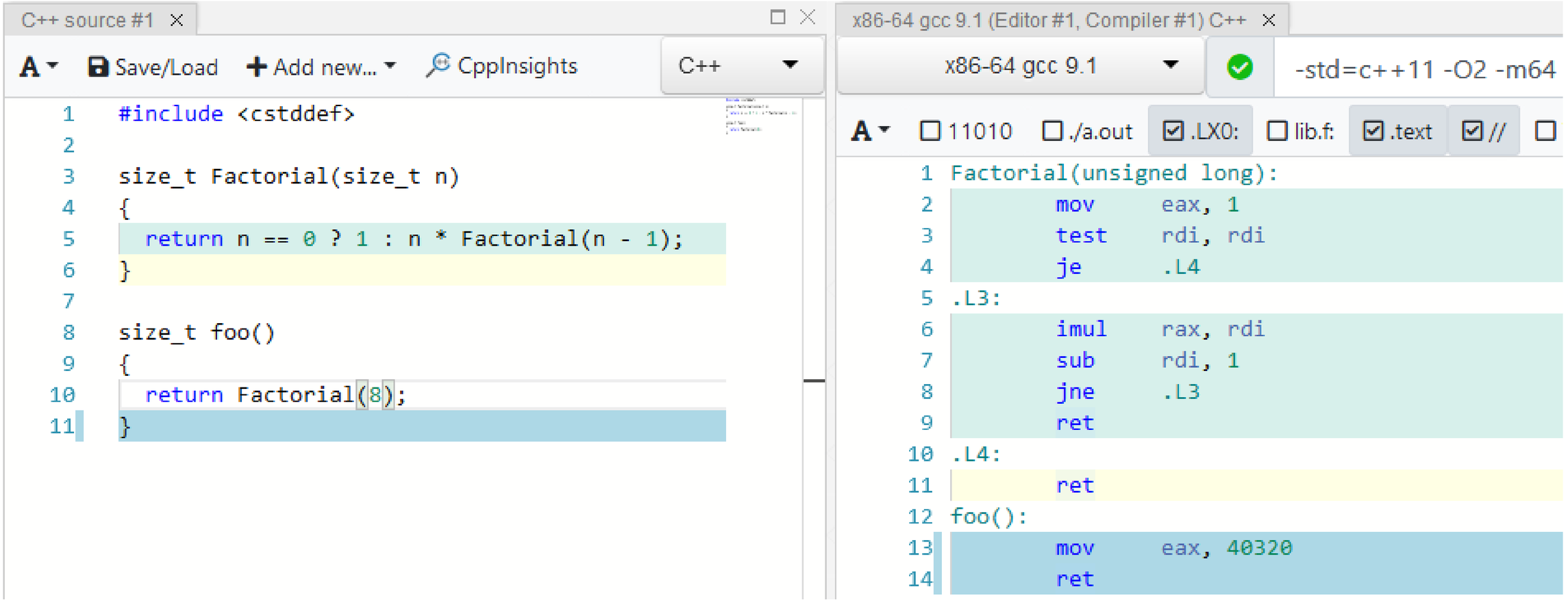
```
size_t ArraySize() { return 42; }
int some_value[ArraySize()];

<source>:4:5: error: variable length array declaration not
allowed at file scope

int some_value[ArraySize()];

    ^           ~~~~~~~~~~

1 error generated.

Compiler returned: 1
```

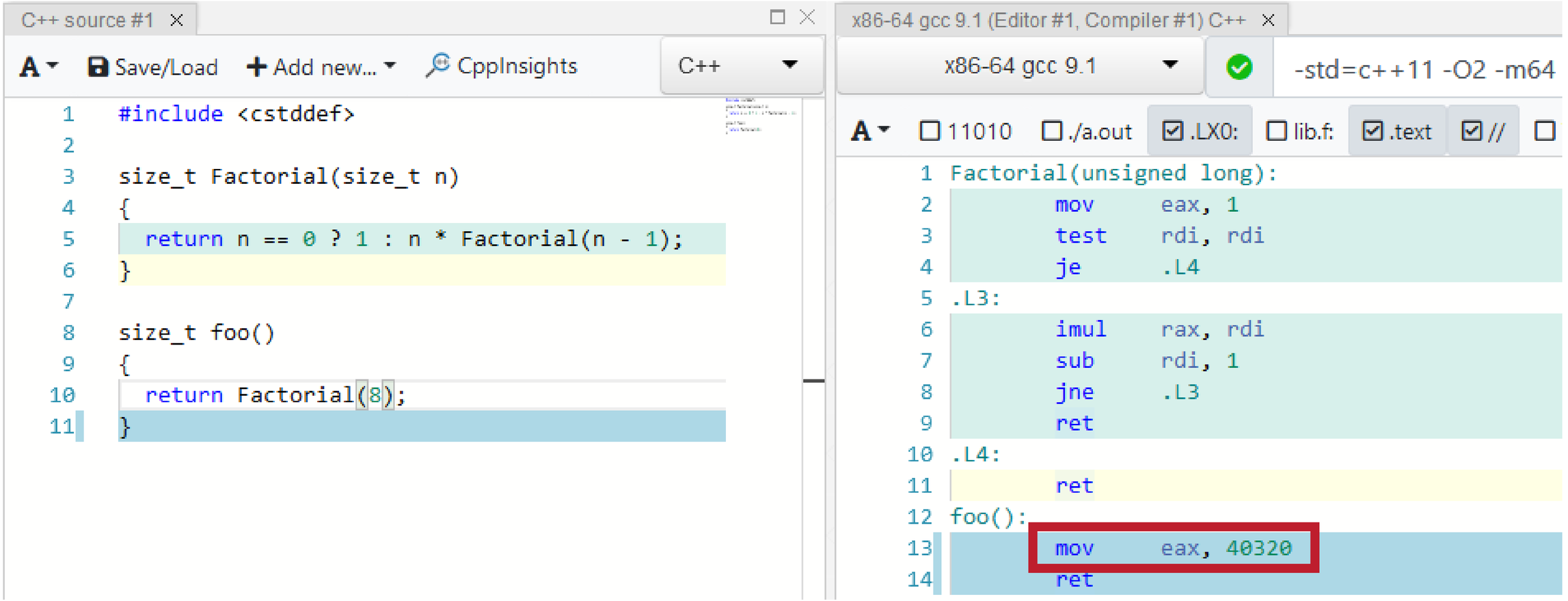
```
constexpr size_t ArraySize() { return 42; }
int some_value[ArraySize()];
```

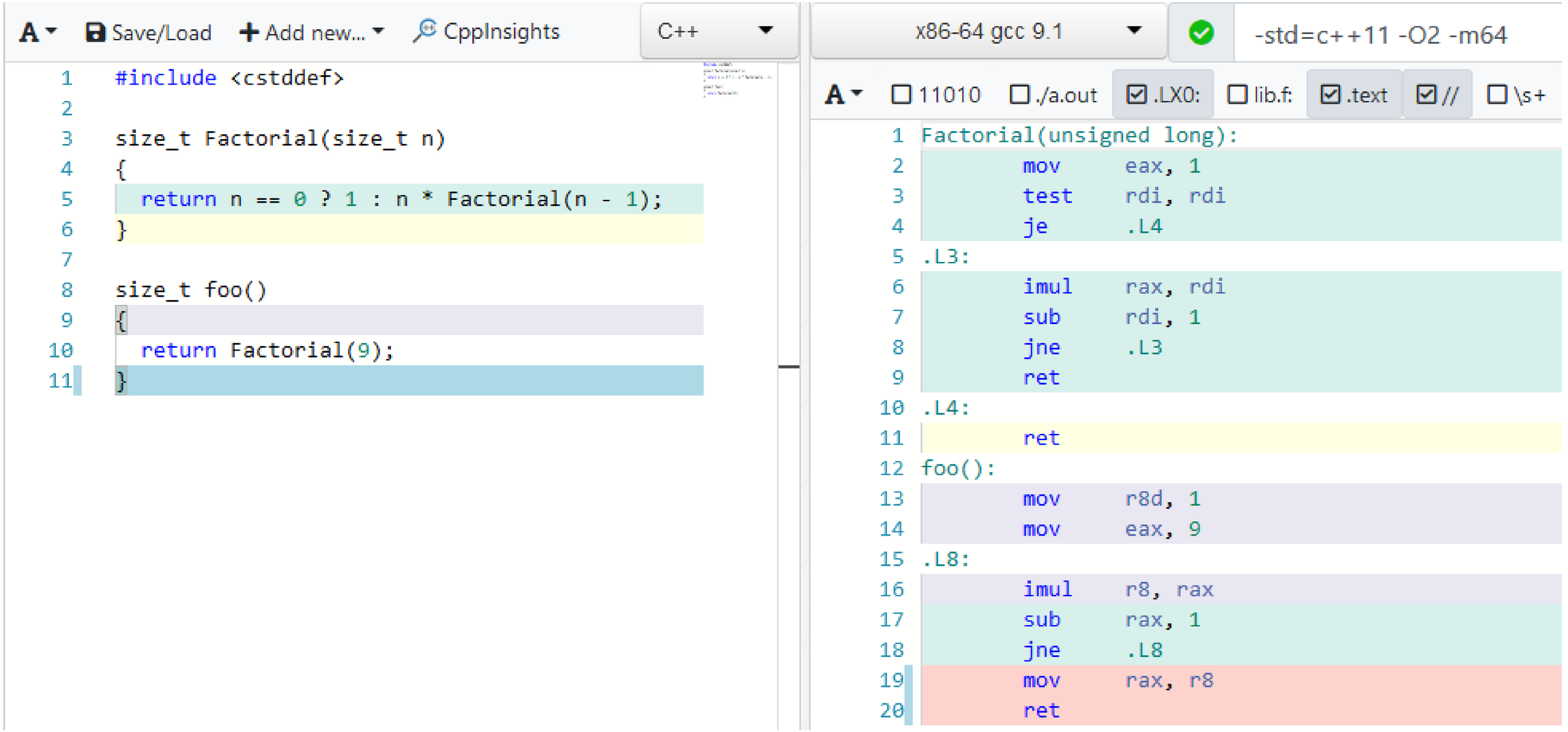
Ограничения:

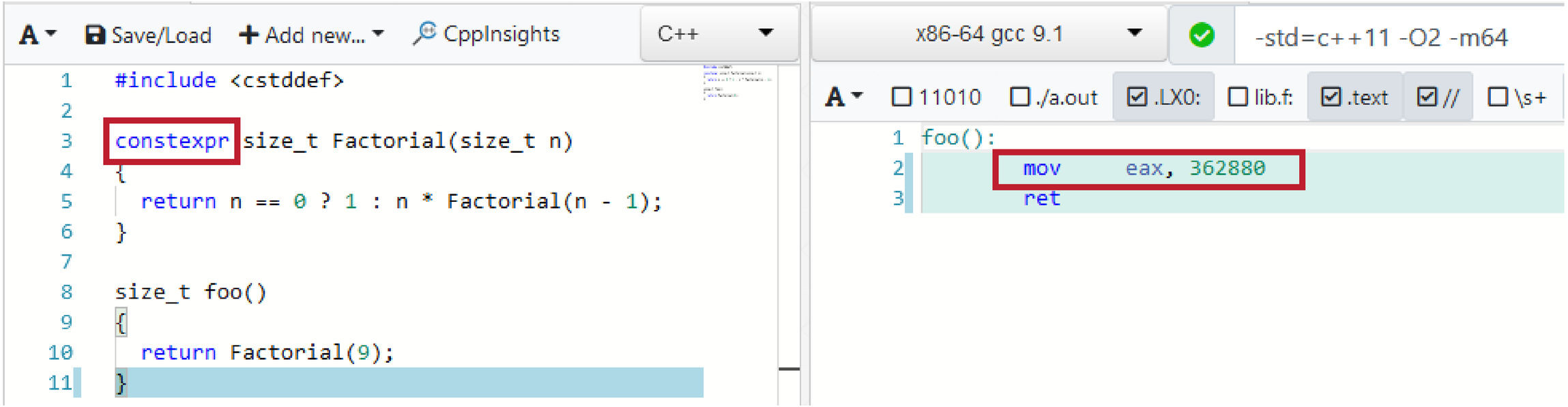
- Non-void функции
- Тело должно состоять из return expr
- expr должен состоять из констант или вызовов constexpr-функций
- Не может использоваться до определения (опережающее объявление)

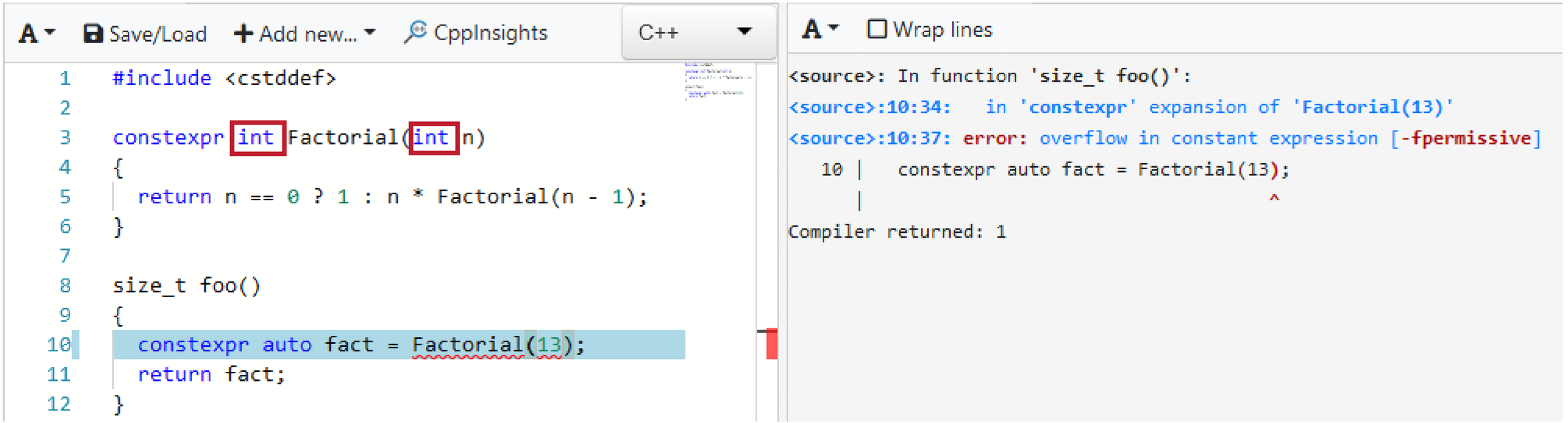
```
size_t Factorial(size_t n)
{
  return n == 0 ? 1 : n * Factorial(n – 1);
}

size_t foo { return Factorial(8); }
```


C++ source #1
Save/Load
Add new...
CppInsights
C++
#include <cstddef>
size_t Factorial(size_t n)
{
return n == 0 ? 1 : n * Factorial(n - 1);
}
size_t foo()
{
return Factorial(8);
}
x86-64 gcc 9.1 (Editor #1, Compiler #1) C++
x86-64 gcc 9.1
-std=c++11 -O2 -m64
11010
./a.out
.LX0:
lib.f:
.text
//
Factorial(unsigned long):
mov eax, 1
test rdi, rdi
je .L4
.L3:
imul rax, rdi
sub rdi, 1
jne .L3
ret
.L4:
ret
foo():
mov eax, 40320
ret


C++ source #1
Save/Load
Add new...
CppInsights
C++
1 #include <cstddef>
2
3 size_t Factorial(size_t n)
4 {
5 return n == 0 ? 1 : n * Factorial(n - 1);
6 }
7
8 size_t foo()
9 {
10 return Factorial(8);
11 }
x86-64 gcc 9.1 (Editor #1, Compiler #1) C++
x86-64 gcc 9.1
-std=c++11 -O2 -m64
11010
./a.out
.LX0:
lib.f:
.text
//
1 Factorial(unsigned long):
2 mov eax, 1
3 test rdi, rdi
4 je .L4
5 .L3:
6 imul rax, rdi
7 sub rdi, 1
8 jne .L3
9 ret
10 .L4:
11 ret
12 foo():
13 mov eax, 40320
14 ret


Save/Load
Add new...
CppInsights
C++
x86-64 gcc 9.1
-std=c++11 -O2 -m64
11010
./a.out
.LX0:
lib.f:
.text
//
\s+
#include <cstddef>
size_t Factorial(size_t n)
{
return n == 0 ? 1 : n * Factorial(n - 1);
}
size_t foo()
{
return Factorial(9);
}
Factorial(unsigned long):
mov eax, 1
test rdi, rdi
je .L4
.L3:
imul rax, rdi
sub rdi, 1
jne .L3
ret
.L4:
ret
foo():
mov r8d, 1
mov eax, 9
.L8:
imul r8, rax
sub rax, 1
jne .L8
mov rax, r8
ret


A
Save/Load
Add new...
CppInsights
C++
#include <cstddef>
constexpr size_t Factorial(size_t n)
{
return n == 0 ? 1 : n * Factorial(n - 1);
}
size_t foo()
{
return Factorial(9);
}
x86-64 gcc 9.1
-std=c++11 -O2 -m64
11010
./a.out
.LX0:
lib.f:
.text
//
\s+
foo():
mov eax, 362880
ret


A
Save/Load
Add new...
CppInsights
C++
1 #include <cstddef>
2
3 constexpr int Factorial(int n)
4 {
5 return n == 0 ? 1 : n * Factorial(n - 1);
6 }
7
8 size_t foo()
9 {
10 constexpr auto fact = Factorial(13);
11 return fact;
12 }
A
Wrap lines
<source>: In function 'size_t foo()':
<source>:10:34: in 'constexpr' expansion of 'Factorial(13)'
<source>:10:37: error: overflow in constant expression [-fpermissive]
10 | constexpr auto fact = Factorial(13);
| ^
Compiler returned: 1

POD-type

Тривиальный класс:

- `T() = default;`
- `T(const T&) = default;`
- `T& operator=(const T&) = default;`
- `T(T&&) = default;`
- `T& operator=(T&&) = default;`
- `~T() = default;`
- Нет виртуальных методов и виртуального наследования
- Все нестатические поля тривиальны
- Все базовые классы тривиальны (при наличии)

Класс со стандартным размещением:

- Все нестатические поля имеют одинаковый доступ (private, public, protected)
- Нет виртуальных методов и виртуального наследования
- Нет нестатических полей-ссылок
- Все нестатические поля и базовые классы со стандартным размещением
- Все нестатические поля объявлены в одном классе в иерархии наследования
- Нет базовых классов того же типа, что и первое нестатическое поле

auto

```
void foo(const std::vector<int> &nums)
{
  ....

  for (int i = 0; i < nums.size(); ++i) // Only for 32-bit
    // some computations with nums[i]

  ....
}
```

```
void foo(const std::vector<int> &nums)
{
  ....

  for (int i = 0; i < nums.size(); ++i) // Only for 32-bit
    // some computations with nums[i]

  for (std::vector<int>::const_iterator it = nums.begin();
       it != nums.end();
       ++it)
    // some computations with *it

  ....
}
```

```
void foo(const std::vector<int> &nums)
{
  ....

  typedef std::vector<int>::const_iterator const_iterator;
  for (const_iterator it = nums.begin(); it != nums.end(); ++it)
    // some computations with *it

  ....
}
```

```
void foo(const std::vector<double> &nums)
{
   ....

   typedef std::vector<int>::const_iterator const_iterator;
   for (const_iterator it = nums.begin(); it != nums.end(); ++it)
      // some computations with *it

   ....
}
```

```
void foo(const std::vector<double> &nums)
{
   ....

  for (auto it = container.begin(); it != container.end(); ++it)
     // some computations with *it

   ....
}
```

```
int bar();

               auto i   = 0;   // int
               auto ui  = 0u;  // unsigned int
      volatile auto ci  = i;   // volatile int
const volatile auto cvi = i;   // const volatile int
               auto j   = cvi; // int

      auto &ri  = i; // int &
const auto &cri = i; // const int &

auto &&fri  = i;   // int &
auto &&fcri = cri; // const int &

auto &&frv  = 0;      // int &&
auto &&frvf = bar(); // int &&
```

decltype

```
using namespace std;

void foo(const vector<int> &nums)
{
  typedef vector<int>::value_type     v_type;
  typedef vector<int>::const_iterator c_iter;

  for (c_iter it = nums.begin(); it != nums.end(); ++it)
  {
    v_type tmp = *it;
    // some calculations with tmp
  }
}
```

```
using namespace std;

void foo(const vector<double> &nums)
{
  typedef vector<int>::value_type     v_type;
  typedef vector<int>::const_iterator c_iter;

  for (c_iter it = nums.begin(); it != nums.end(); ++it)
  {
    v_type tmp = *it;
    // some calculations with tmp
  }
}
```

```
using namespace std;

void foo(const vector<double> &nums)
{
  typedef decay<decltype(nums)>::type::value_type     v_type;
  typedef decay<decltype(nums)>::type::const_iterator c_iter;

  for (c_iter it = nums.begin(); it != nums.end(); ++it)
  {
    v_type tmp = *it;
    // some calculations with tmp
  }
}
```

```
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 &&
```

Range-based for loop

```
std::vector<SomeClass> container;
....

for (const auto &obj : container)
  // some computations with obj
```

```
std::vector<SomeClass> container;
....

auto &&tmp = container;
for (auto it = std::begin(tmp); it != std::end(tmp); ++it)
{
  const MyClass &obj = *it;
  // some computations with obj
}
```

```
struct SomeClass { std::vector<int> m_vec; .... };
SomeClass foo();
....

for (const auto &obj : foo().m_vec)
  // some computations with obj
```

```
struct SomeClass { std::vector<int> m_vec; .... };
SomeClass foo();
....

auto &&tmp = foo().m_vec;
for (auto it = tmp.begin(); it != tmp.end(); ++it)
{
  const auto &obj = *it;
  // some computations with obj
}
```

Lambda functions

```
#include <vector>
#include <algorithm>

void foo()
{
  std::vector<size_t> vec;
  ....
  // Ascending order
  std::sort(vec.begin(), vec.end());

  // Descending order
  std::sort(vec.begin(), vec.end(), ???);
}
```

```
bool IsLhsMoreThanRhs(size_t lhs, size_t rhs)
{
  return lhs > rhs;
}

struct DescComp
{
  bool operator()(size_t lhs, size_t rhs) const noexcept
  {
    return lhs > rhs;
  }
};


std::sort(vec.begin(), vec.end(), IsLhsMoreThanRhs);
std::sort(vec.begin(), vec.end(), DescComp{});
```

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

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

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

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

```
void foo()
{
  static size_t i = 0;
  auto f = [&i](size_t lhs, size_t rhs) -> bool
           {
             ++i;
             return lhs > rhs;
           };

  decltype(f) g = f;

  bool Is2_MoreThan_5 = f(2, 5); // false
}
```

Alternative function syntax

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

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

Delegate constructors

```
class SomeClass
{
  int *m_p, *m_q;
public:
  SomeClass()
  {
    this->SomeClass::SomeClass(NULL, NULL);
  }

  SomeClass(int *p, int *q) : m_p{p}, m_q{q} {}
};
```

```
class SomeClass
{
  int *m_p, *m_q;
public:
  SomeClass()
  {
    new (this) SomeClass(NULL, NULL);
  }

  SomeClass(int *p, int *q) : m_p{p}, m_q{q} {}
};
```

```
class SomeClass
{
  int *m_p, *m_q;
public:
  SomeClass() : SomeClass(NULL, NULL) {}

  SomeClass(int p, int q) : m_p{p}, m_q{q} {}
};
```

Default value for non-static class member

```
class SomeClass
{
  int *p = NULL, *q = NULL;
public:
  SomeClass(int *p, int *q) : m_p{p}, m_q{q} {}
};
```

'default' specifier

```
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
public:
  SomeClass() {}
  SomeClass(const SomeClass &other)
    : m_name { other.m_name }, m_surname { other.m_surname } {}
  SomeClass& operator=(const SomeClass &other)
    : m_name { other.m_name }, m_surname { other.m_surname } {}
};
```

```
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
public:
  SomeClass() = default;
  SomeClass(const SomeClass &other) = default;
  SomeClass& operator=(const SomeClass &other) = default;
};
```

'delete' specifier

```
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
  SomeClass(const SomeClass &other);
  SomeClass& operator=(const SomeClass &other);
public:
  SomeClass() = default;
};

SomeClass a;
SomeClass b { a }; // compile-time error
a = b;             // compile-time error
```

```cpp
#include <string>

class SomeClass
{
  std::string m_name, m_surname;
public:
  SomeClass() = default;
  SomeClass(const SomeClass &other) = delete;
  SomeClass& operator=(const SomeClass &other) = delete;
};

SomeClass a;
SomeClass b { a }; // compile-time error
a = b;             // compile-time error
```

'override' specifier

```
struct A
{
  virtual void foo(int) { /* something special for A */ }
};

struct B : A
{
  void foo(long) { /* something special for B */ }
};

A *p = new B;
p->foo();
delete p;
```

```
struct A
{
  virtual void foo(int) { /* something special for A */ }
};

struct B : A
{
  // compile-time error
  virtual void foo(long) override { /* something special for B */ }
};

A *p = new B;
p->foo();
delete p;
```

'final' specifier

```
struct A
{
  virtual void foo(int) const
  { /* something special for A */ }
};

struct B : A
{
  virtual void foo(int) const override
  { /* something special for B */ };
```

```
struct A
{
  virtual void foo(int) const final
  { /* something special for A */ }
};

struct B : A
{
  // compile-time error
  virtual void foo(int) const override
  { /* something special for B */ }
};
```

```
struct A
{
  virtual void foo(int) const
  { /* something special for A */ }
};

struct B : A
{
  virtual void foo(int) const override
  { /* something special for B */ }
};
```

```
struct A
{
  virtual void foo(int) const
  { /* something special for A */ }
};

struct B final : A
{
  virtual void foo(int) const override
  { /* something special for B */ }
};

// compile-time error
struct C : B { };
```

nullptr

```
#include <cstddef>

class SomeClass
{
public:
  void foo(int *) const;
  void foo(int) const;
};

SomeClass bar;
bar.foo(NULL); // SomeClass::foo(int *) or
               // SomeClass::foo(int)?
```

```
#define NULL (0)
#define NULL (1 - 1)
#define NULL ((void*)0)
#define NULL (static_cast<void*>(100500 - 100500))

....
```

Тысячи способов написать NULL!

```
nullptr; // decltype(nullptr) ≡ std::nullptr_t

char       *p1 = nullptr;
int        *p2 = nullptr;
SomeClass *p3 = nullptr;
bool         b = nullptr; // b == false


....
p3->foo(0);       // SomeClass::foo(int);
p3->foo(nullptr); // SomeClass::foo(int*);
```

```
#include <cstddef>

class SomeClass
{
public:
  void foo(int *) const;
  void foo(int) const;
  void foo(std::nullptr_t) const;
};

SomeClass bar;
bar.foo(nullptr); // SomeClass::foo(std::nullptr_t)
```

enum class

```
enum EHAlign
{
  Left,
  Middle,
  Right
};
```

```
enum EVAlign
{
  Top,
  Center,
  Bottom
};
```

```
void foo(EHAlign enuHAlign)
{

  ....
  if (enuHAlign == Center)
  ....
}
```

```
enum class EHAlign
{
  Left,
  Middle,
  Right
};
```

```
enum class EVAlign
{
  Top,
  Center,
  Bottom
};
```

```
void foo(EHAlign enuHAlign)
{
  if (enuHAlign == Center) // compile-time error
  {
    ....
  }
}
```

```
enum class EHAlign
{
  Left,
  Middle,
  Right
};
```

```
enum class EVAlign
{
  Top,
  Center,
  Bottom
};
```

```
void foo(EHAlign enuHAlign)
{
  if (enuHAlign == EHAlign::Middle) // ok
  {
    ....
  }
}
```

enum underlying type

```
enum EHAlign
{
  Left,
  Middle,
  Right
};

void foo(size_t num)
{
 ....

 // Unspecified behavior
 EHAlign align = static_cast<EHAlign>(num);
}
```

```
enum EHAlign : size_t
{
   Left,
   Middle,
   Right
};

void foo(size_t num)
{
  ....

  // ok
  EHAlign align = static_cast<EHAlign>(num);
}
```

```
enum EHAlign
{
  Left,
  Middle,
  Right
};

struct SomeClass
{
  uint32_t rgba;
  bool flag;
  EHAlign align;
};
```

```
enum EHAlign
{
  Left,
  Middle,
  Right
};

struct SomeClass    // size: 12 bytes; align: 4 bytes
{
  uint32_t rgba;    // size: 4 bytes; align: 4 bytes
  bool flag;        // size: 5 bytes; align: 1 byte
  EHAlign align;    // size: 5 -> 8 -> 12 bytes; align: 4 bytes
};
```

```
enum EHAlign : uint8_t
{
  Left,
  Middle,
  Right
};

struct SomeClass    // size: 8 bytes; align: 4 bytes
{
  uint32_t rgba;    // size: 4 bytes; align: 4 bytes
  bool flag;        // size: 5 bytes; align: 1 byte
  EHAlign align;    // size:  6 -> 8 bytes; align: 4 bytes
};
```

Explicit cast operators

```
template <typename T>
class unique_ptr
{
  T *m_ptr;
public:
  explicit unique_ptr(T *ptr) : m_ptr(ptr) {}
  ~unique_ptr() { delete m_ptr; }
  ....

  operator bool() { return m_ptr != nullptr; }
};


unique_ptr<int> ptr { new int };
int flag = ptr; // ok
```

```
template <typename T>
class unique_ptr
{
  T *m_ptr;
public:
  explicit unique_ptr(T *ptr) : m_ptr(ptr) {}
  ~unique_ptr() { delete m_ptr; }
  ....

  explicit operator bool() { return m_ptr != nullptr; }
};

unique_ptr<int> ptr { new int };

int flag = ptr;    // compile-time error
if (ptr) { .... } // ok, bool implicit cast
```

To be continued...