Don’t sacrifice the API to speed

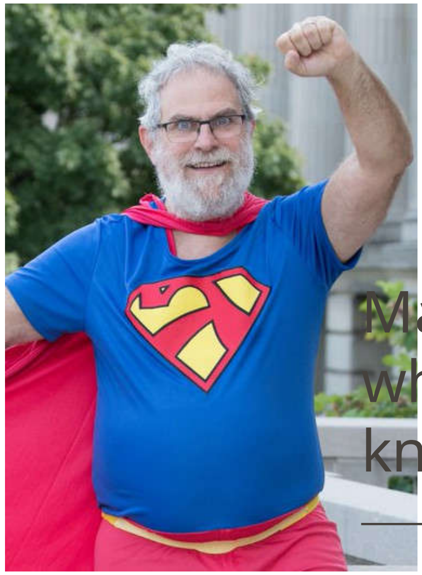
C++ Summit 2020, China

dr Ivan Čukić

About me

- KDAB senior software engineer
  Software Experts in Qt, C++ and 3D / OpenGL
- Author of the ”Functional Programming in C++” book
  available in English, Chinese, Korean, Russian, Polish
- Trainer / consultant
- KDE developer
- University lecturer

Disclaimer

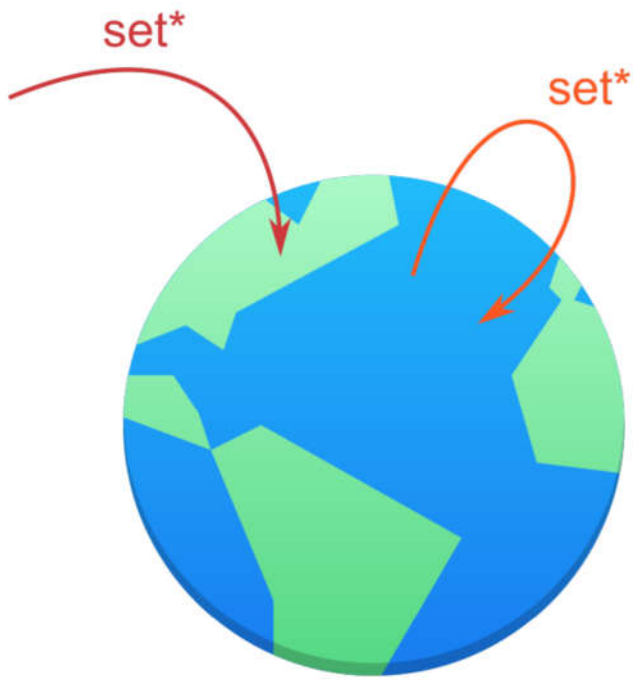


Make your code readable. Pretend the next person who looks at your code is a psychopath and they know where you live.

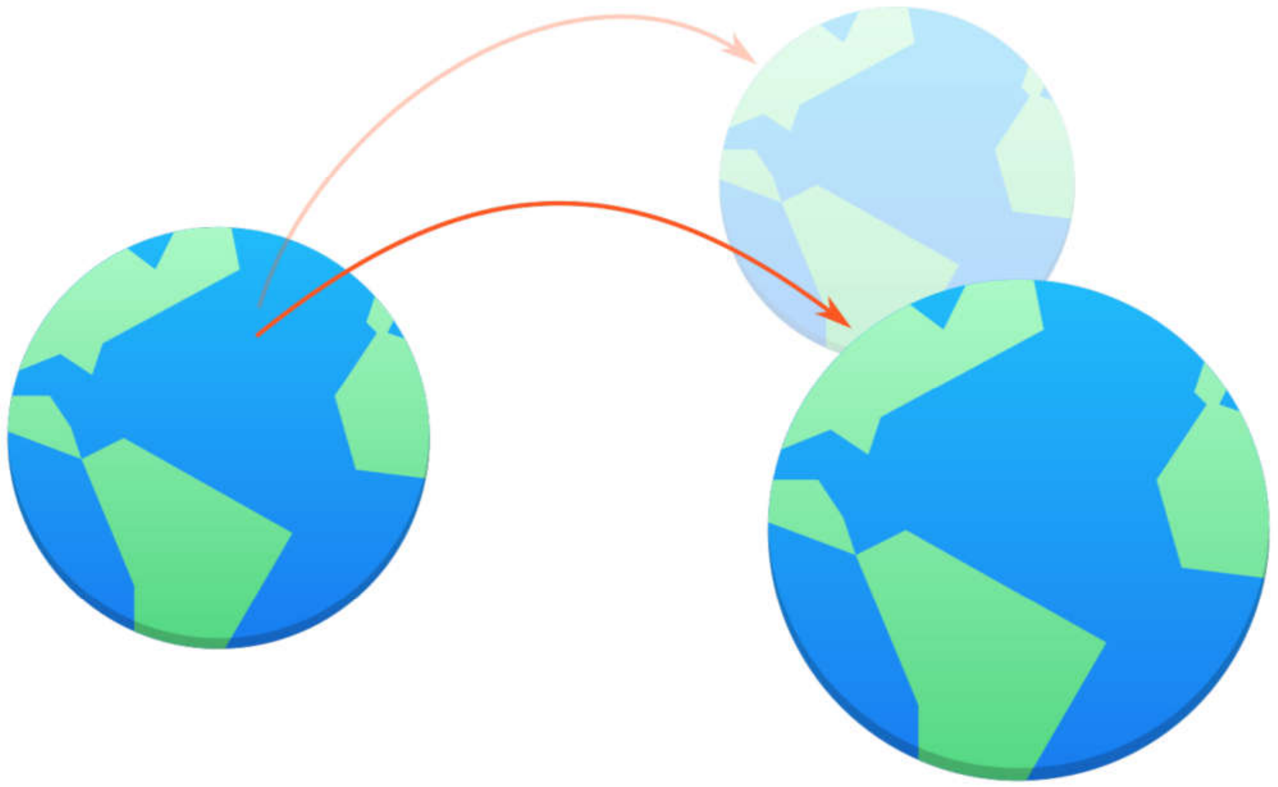
Philip Wadler

FAR AWAY WORLDS

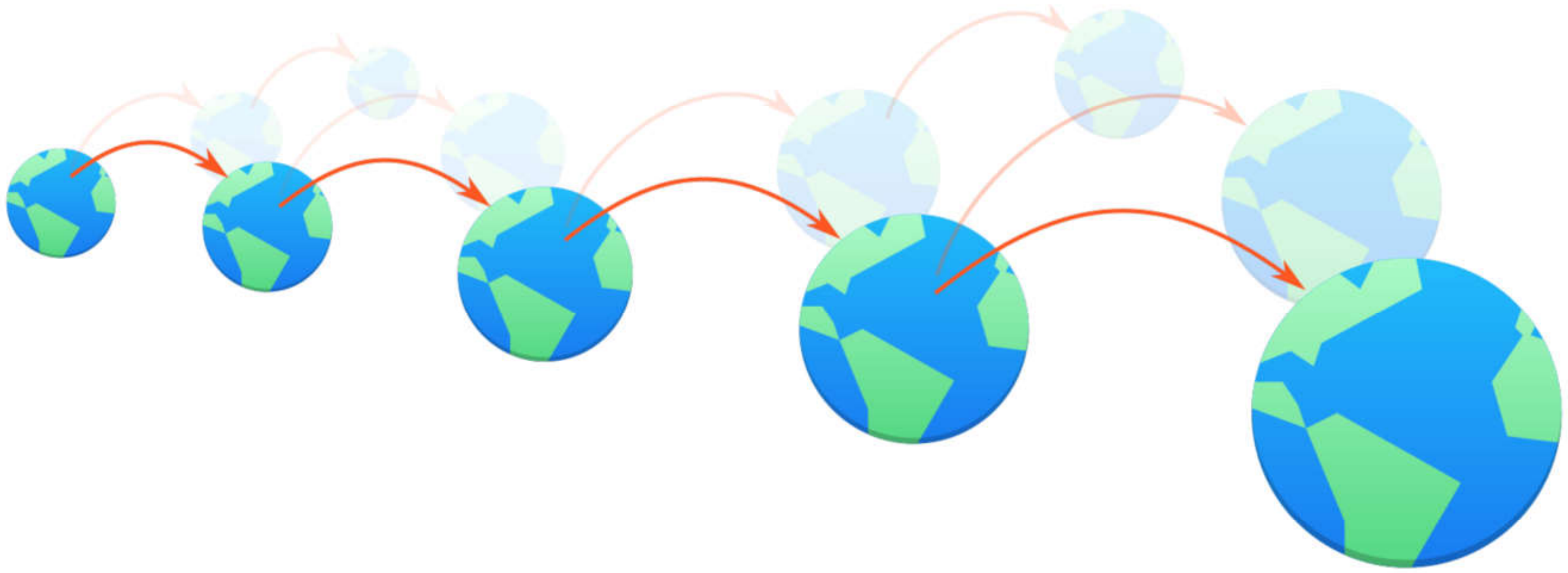
Far away worlds



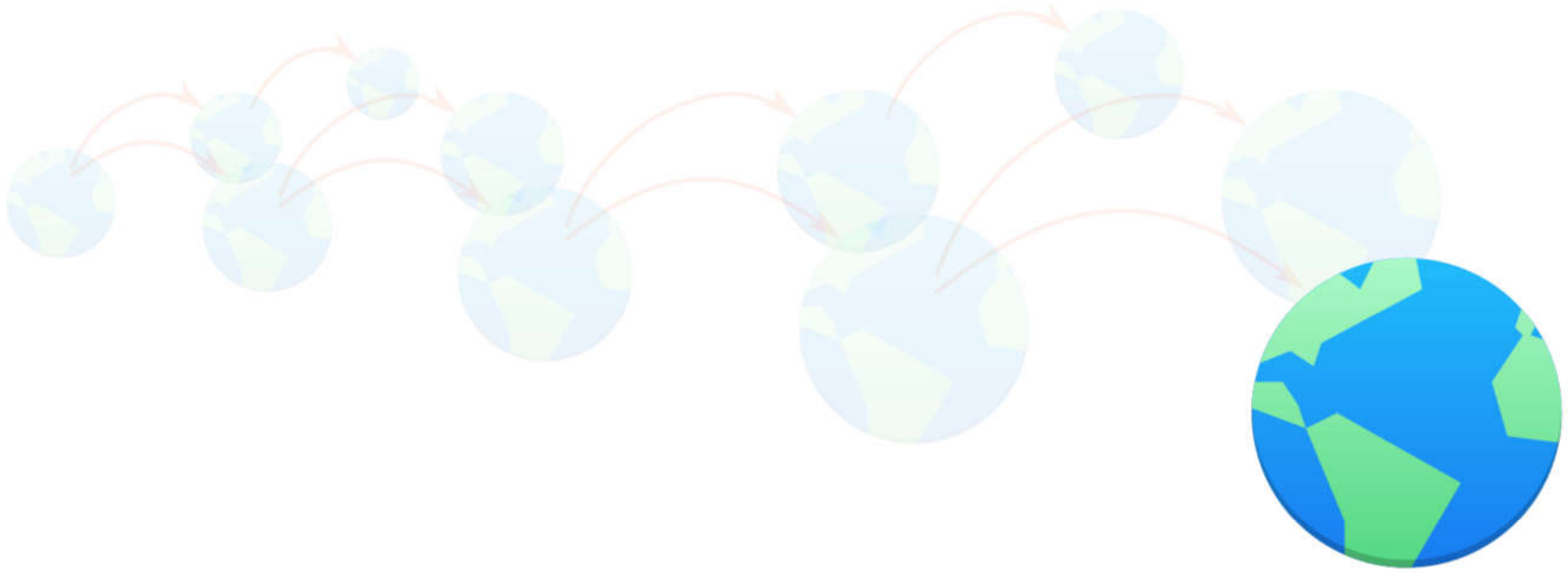
Far away worlds



Far away worlds



Far away worlds



Far away worlds

Values belonging to a linear type must be **used exactly once**: like the world, they can not be duplicated or destroyed. Such values require no reference counting or garbage collection...

Linear types can change the world!
Philip Wadler

ATTACK OF THE CLONES

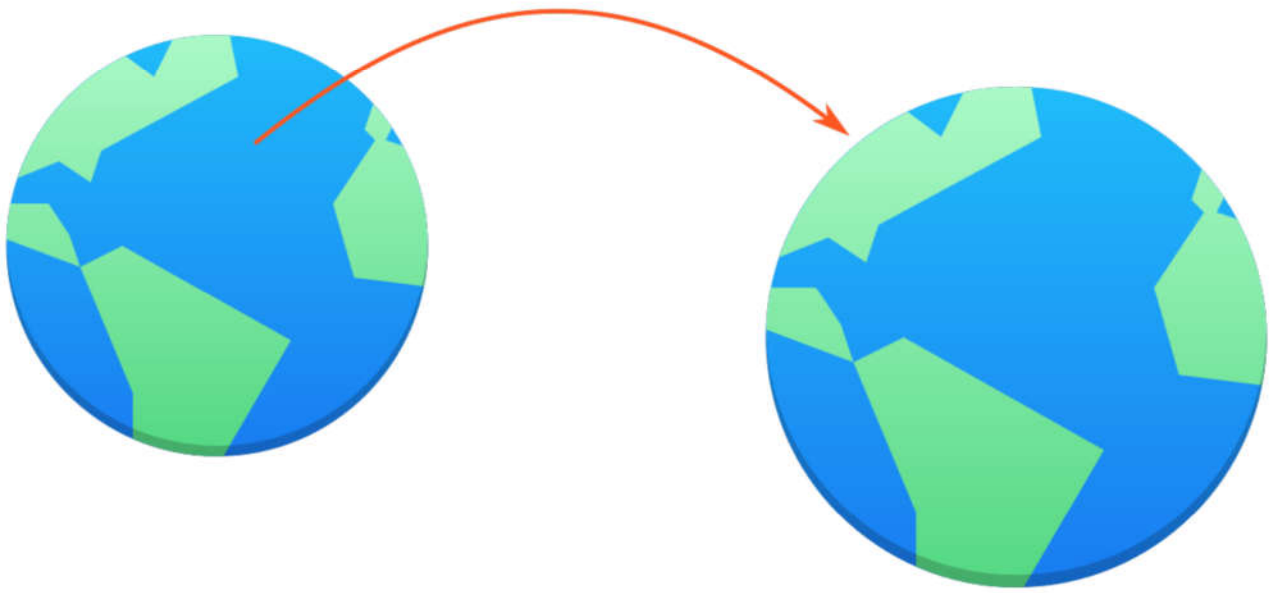
RAII

```
}
```

3
4
5

Clones

Clones



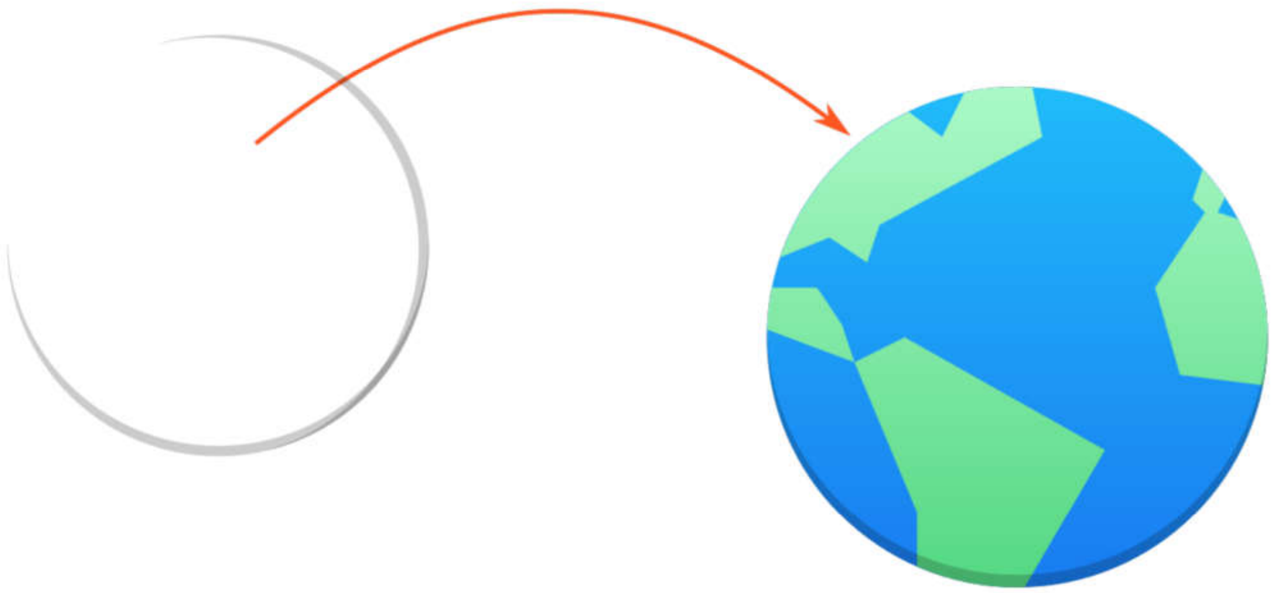
Clones

Clones

&&

Clones

Clones



Clones

Move semantics:

- Resource ownership transfer
- Optimization
- API documentation / usage restriction

Clones

Move semantics:

- Resource ownership transfer
- Optimization
- API documentation / usage restriction

Clones

```
void foo(type&& v)
{
    ...
}
```

Clones

```
class type {
    void foo() &&
    {
        ...
    }
}
```

*this is a temporary

Clones

```
type&& foo()
{
    ...
}
```

Clones

```
type&& foo(type&& v)
{
      ...
}
```

Clones

```
std::getline(std::cin, s);
```

Clones

```
std::string&& getline(std::istream& in, std::string

s = getline(std::cin, std::move(s));
```

GENERIC

Concepts and constraints

How to enforce moves with generic programming?

```
template <typename T>
void foo(T&& val)
{
    ...
}
```

Concepts and constraints

```
template <typename T>
constexpr bool is_int_v = std::is_same_v<T, int>;
```

Concepts and constraints

```
template <typename T>
concept IsInt = std::is_same_v<T, int>;
```

> not a proper concept,
> demonstration purposes only!

Concepts and constraints

```
template <typename T>
    requires (IsInt<T>)
void foo(T&& v)
{
    ...
}
```

Concepts and constraints

```
template <typename T>
    requires (is_int_v<T>)
void foo(T&& v)
{
    ...
}
```

Clones

```
template <typename T>
    requires (???)
void foo(T&& v)
{
    ...
}
```

Clones

```
typedef T&  lref;
typedef T&& rref;

T value;

lref&  r1 = value; // type of r1 is T&
lref&& r2 = value; // type of r2 is T&
rref&  r3 = value; // type of r3 is T&
rref&& r4 = T();   // type of r4 is T&&
```

Clones

```
template <typename T>
    requires (!std::is_lvalue_reference_v<T>)
void foo(T&& v)
{
    ...
}
```

Attack of the clones

```
istream_sequence<std::string> in{std::cin};

std::string result;
for (const auto& token: in) {
    result.append(token);
}
```

Attack of the clones

```
istream_sequence<std::string> in{std::cin};

std::string result;
for (const auto& token: in) {
    result.append(token);
}
```

Remember what Sean said?

Attack of the clones

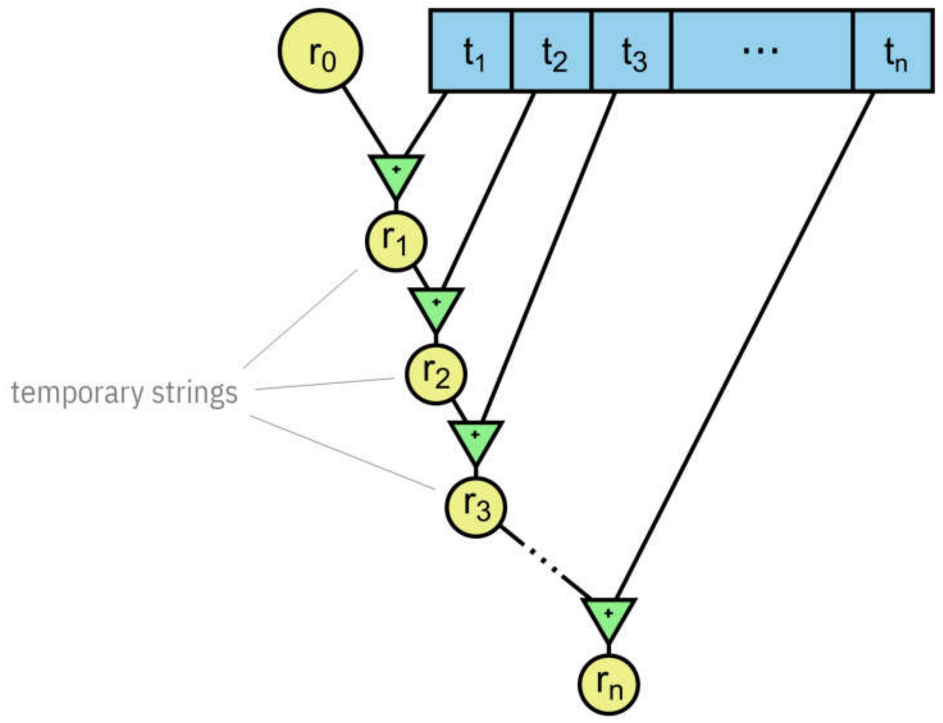
```
istream_sequence<std::string> in{std::cin};

const auto result =
          accumulate(in, string{});
```

Remember what Sean said?

Attack of the clones

```
template <typename InputIt, typename T>
T accumulate(InputIt first, InputIt last, T init)
{
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

Attack of the clones



Attack of the clones

```
template <typename InputIt, typename T>
T accumulate(InputIt first, InputIt last, T init)
{
    while (first != last) {
        init = std::move(init) + *first;
        ++first;
    }
    return init;
}
```

Attack of the clones

Copying is the silent (performance) killer

Move-only types

Can we enforce linearity?

Move-only types

- For unit testing generic code
- For message passing, ranges, reactive streams
- For compile-time type tagging

Testing generic code

```
std::accumulate(
        std::cbegin(items), std::cend(items),
        std::make_unique<std::string>("Hello, Itali
        ...
    );
```

Ranges and reactive streams

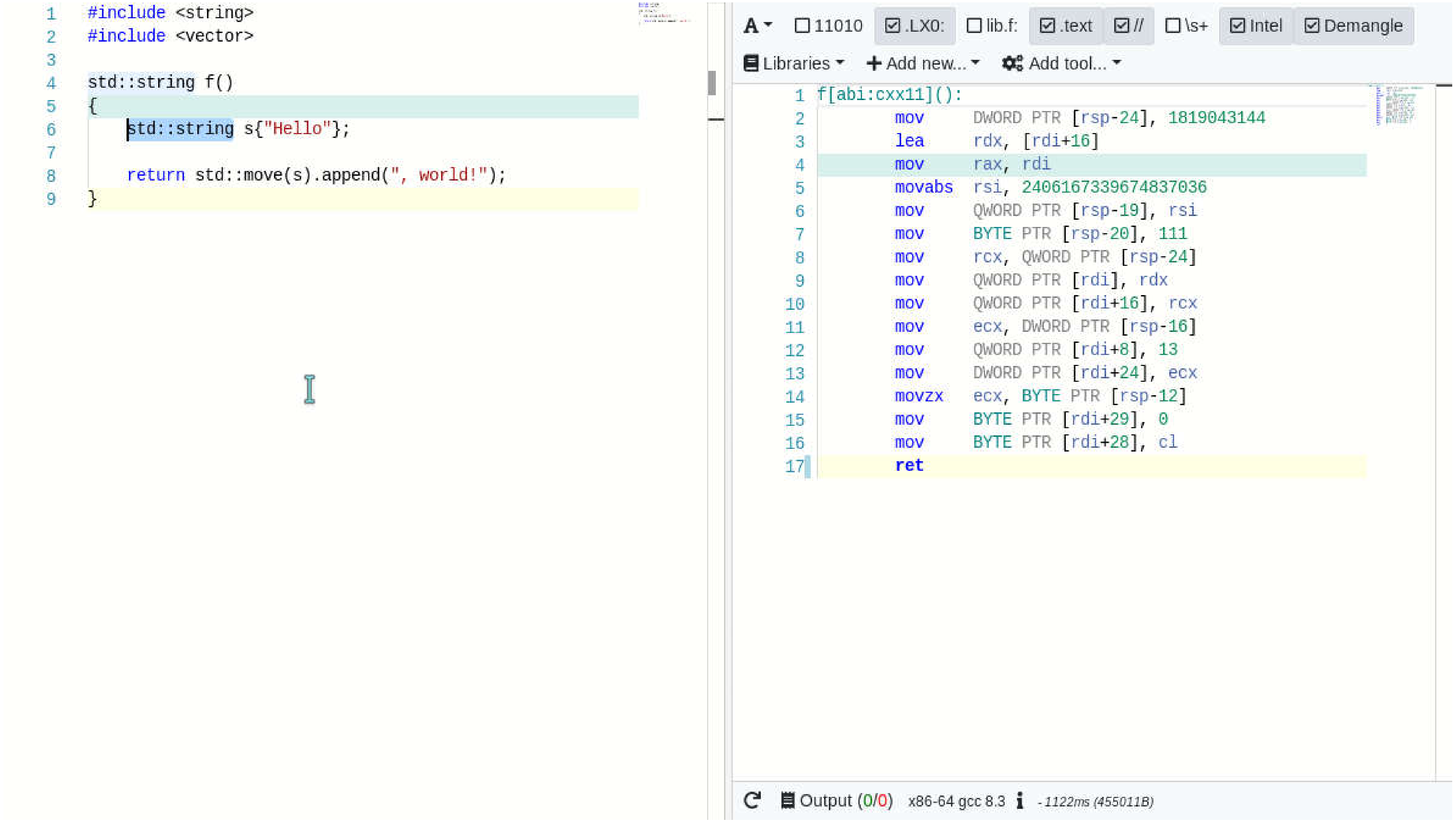
```
auto pipeline =
    voy::system_cmd(”ping”s, ”localhost”s)
    | voy::transform([] (lstring value) {
          std::transform(value.begin(), value.end(), value.begin(), toupper);
          return value;
      })
    | append_pid

    | voy::transform([] (lstring value) {
          const auto pos = value.find_last_of('=');
          return std::make_pair(std::move(value), pos);
      })
    | voy::transform([] (std::pair<lstring, size_t>&& pair) {
          auto [ value, pos ] = pair;
          return pos == std::string::npos
                      ? std::move(value)
                      : std::string(value.cbegin() + pos + 1, value.cend());
      })
    | append_pid

    | voy::filter([] (lstring value) {
          return value < ”0.145”s;
      })
    | append_pid

    | voy::sink{cout};
```

CTTT

```
template <typename... NodeMeta>
class node {

    template <typename Meta>
    auto with_meta() &&
    {
        return node<Meta, NodeMeta...>(std::move(*t
    }

};
```

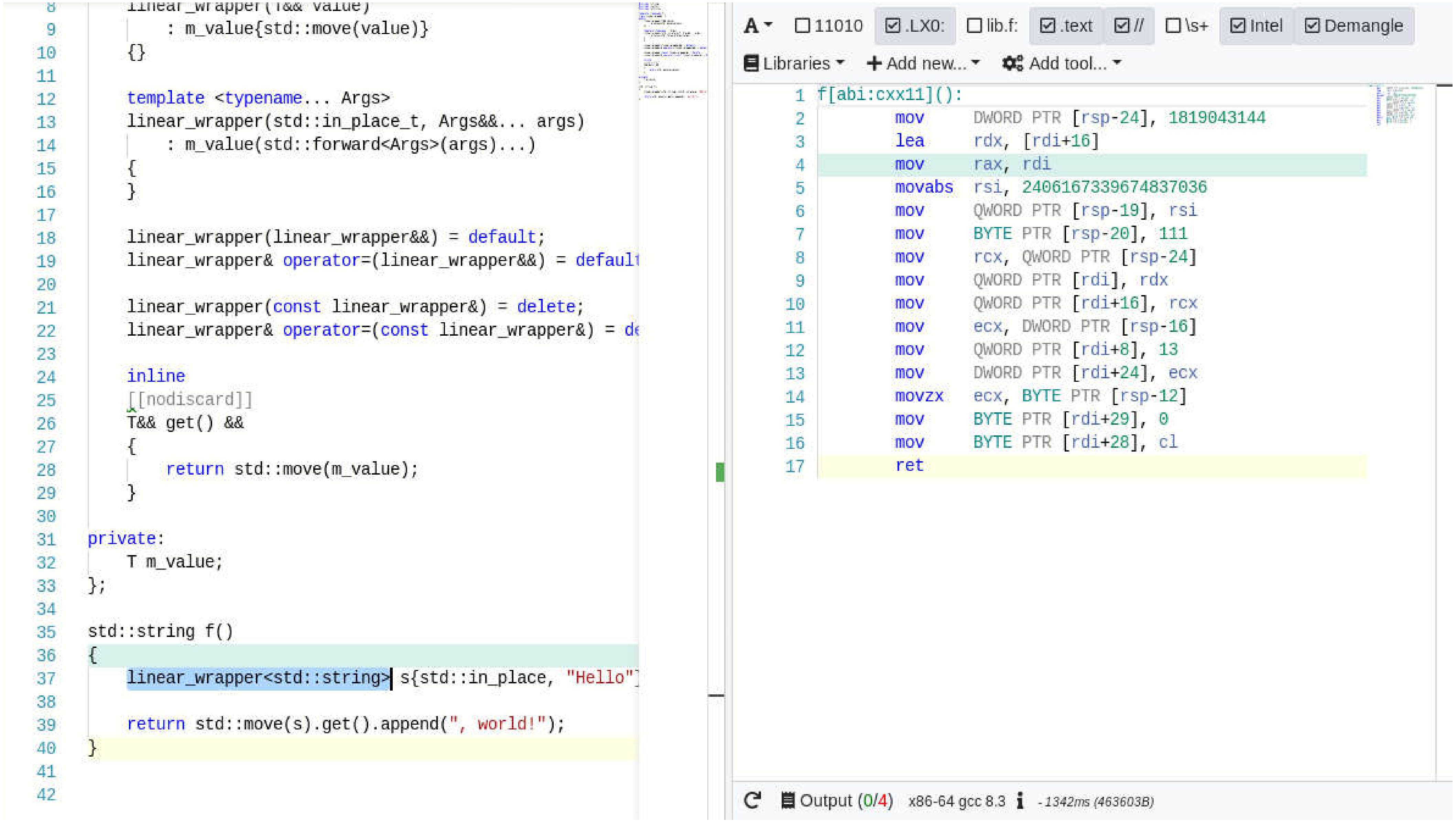
need to move from *this

PERFORMANCE



```cpp
#include <string>
#include <vector>

std::string f()
{
    std::string s{"Hello"};

    return std::move(s).append(", world!");
}
```

```
f[abi:cxx11]():
        mov     DWORD PTR [rsp-24], 1819043144
        lea     rdx, [rdi+16]
        mov     rax, rdi
        movabs  rsi, 2406167339674837036
        mov     QWORD PTR [rsp-19], rsi
        mov     BYTE PTR [rsp-20], 111
        mov     rcx, QWORD PTR [rsp-24]
        mov     QWORD PTR [rdi], rdx
        mov     QWORD PTR [rdi+16], rcx
        mov     ecx, DWORD PTR [rsp-16]
        mov     QWORD PTR [rdi+8], 13
        mov     DWORD PTR [rdi+24], ecx
        movzx   ecx, BYTE PTR [rsp-12]
        mov     BYTE PTR [rdi+29], 0
        mov     BYTE PTR [rdi+28], cl
        ret
```



```cpp
    linear_wrapper(T&& value)
        : m_value{std::move(value)}
    {}

    template <typename... Args>
    linear_wrapper(std::in_place_t, Args&&... args)
        : m_value(std::forward<Args>(args)...)
    {
    }

    linear_wrapper(linear_wrapper&&) = default;
    linear_wrapper& operator=(linear_wrapper&&) = default
    
    linear_wrapper(const linear_wrapper&) = delete;
    linear_wrapper& operator=(const linear_wrapper&) = de

    inline
    [[nodiscard]]
    T&& get() &&
    {
        return std::move(m_value);
    }

private:
    T m_value;
};

std::string f()
{
    linear_wrapper<std::string> s{std::in_place, "Hello"}

    return std::move(s).get().append(", world!");
}
```

☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+ ☑ Intel ☑ Demangle

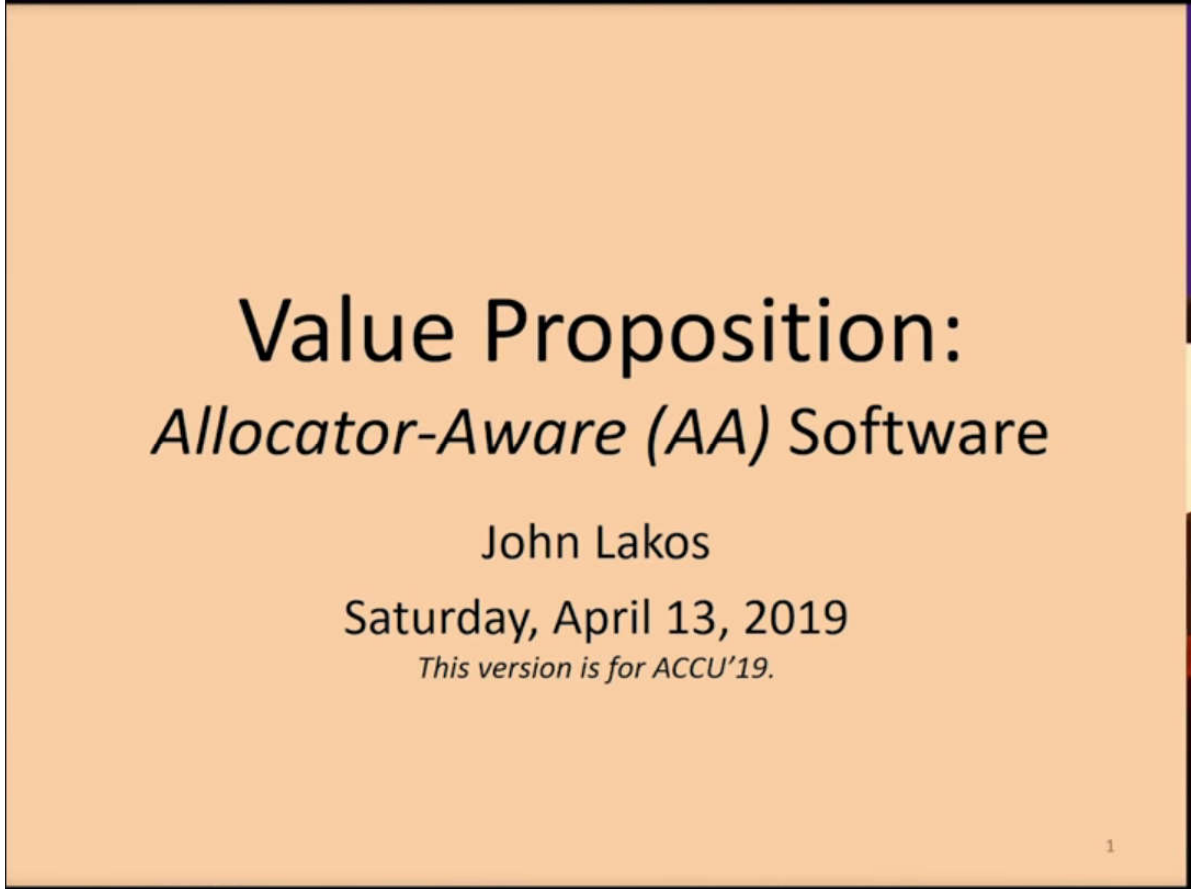
```asm
f[abi:cxx11]():
        mov     DWORD PTR [rsp-24], 1819043144
        lea     rdx, [rdi+16]
        mov     rax, rdi
        movabs  rsi, 2406167339674837036
        mov     QWORD PTR [rsp-19], rsi
        mov     BYTE PTR [rsp-20], 111
        mov     rcx, QWORD PTR [rsp-24]
        mov     QWORD PTR [rdi], rdx
        mov     QWORD PTR [rdi+16], rcx
        mov     ecx, DWORD PTR [rsp-16]
        mov     QWORD PTR [rdi+8], 13
        mov     DWORD PTR [rdi+24], ecx
        movzx   ecx, BYTE PTR [rsp-12]
        mov     BYTE PTR [rdi+29], 0
        mov     BYTE PTR [rdi+28], cl
        ret
```

Output (0/4) x86-64 gcc 8.3 - 1342ms (463603B)

Testing strings

Better than RVO?

/tongue-in-cheek/



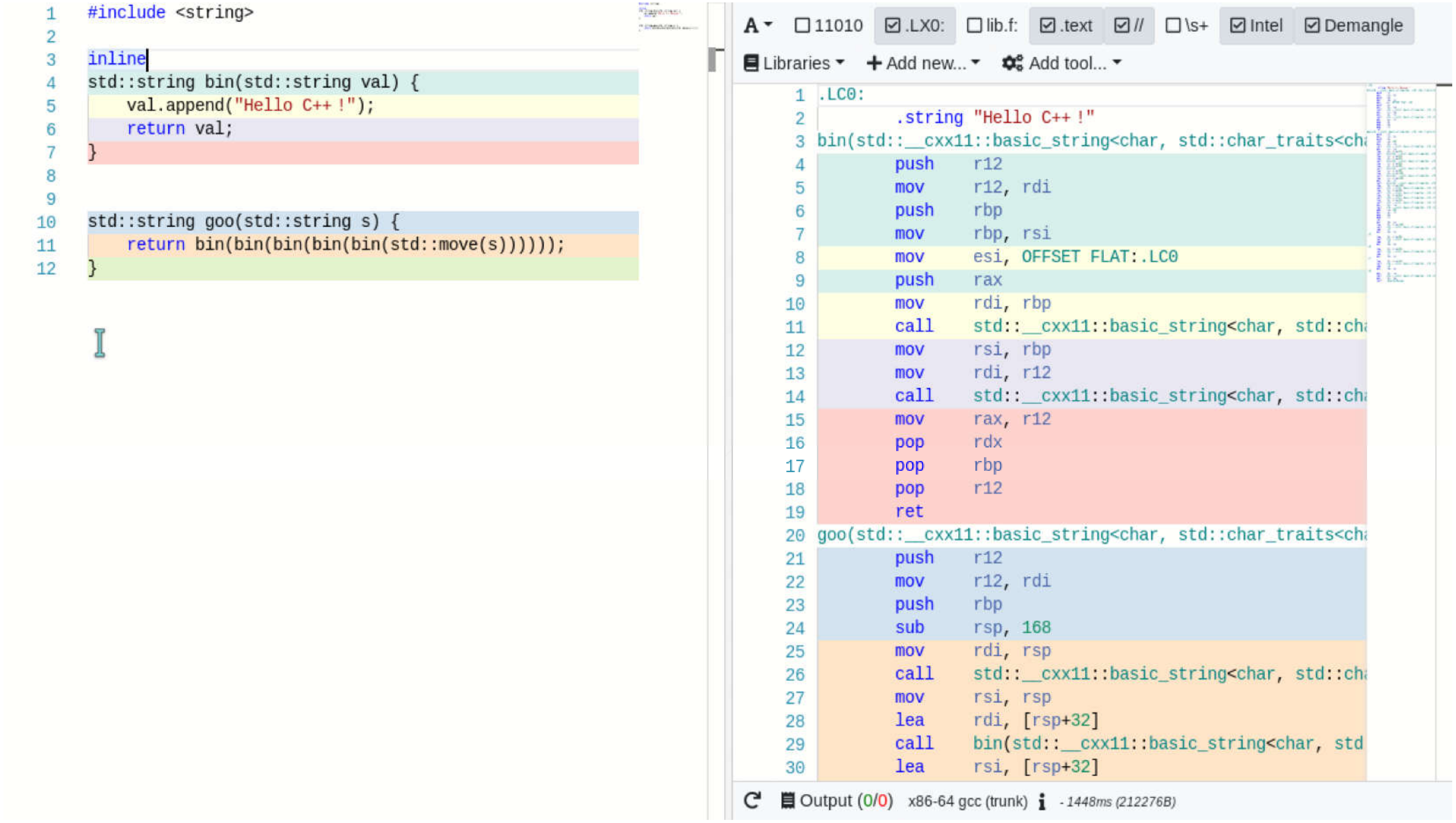
Value Proposition:

Allocator-Aware (AA) Software

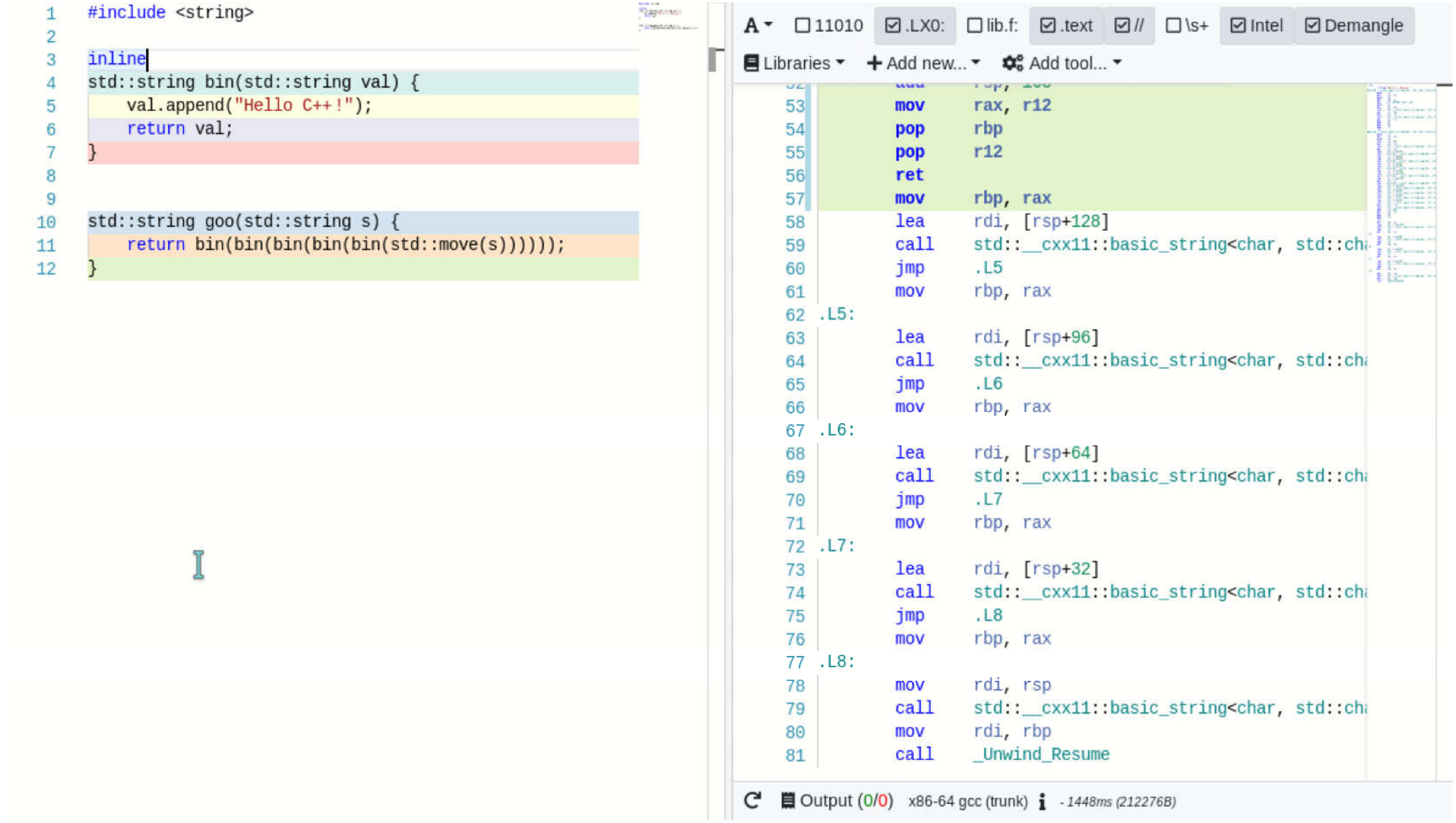
John Lakos

Saturday, April 13, 2019

This version is for ACCU'19.


```cpp
#include <string>

inline
std::string bin(std::string val) {
    val.append("Hello C++ !");
    return val;
}


std::string goo(std::string s) {
    return bin(bin(bin(bin(bin(std::move(s))))));
}
```

```asm
.LC0:
        .string "Hello C++ !"
bin(std::__cxx11::basic_string<char, std::char_traits<cha
        push    r12
        mov     r12, rdi
        push    rbp
        mov     rbp, rsi
        mov     esi, OFFSET FLAT:.LC0
        push    rax
        mov     rdi, rbp
        call    std::__cxx11::basic_string<char, std::cha
        mov     rsi, rbp
        mov     rdi, r12
        call    std::__cxx11::basic_string<char, std::cha
        mov     rax, r12
        pop     rdx
        pop     rbp
        pop     r12
        ret
goo(std::__cxx11::basic_string<char, std::char_traits<cha
        push    r12
        mov     r12, rdi
        push    rbp
        sub     rsp, 168
        mov     rdi, rsp
        call    std::__cxx11::basic_string<char, std::cha
        mov     rsi, rsp
        lea     rdi, [rsp+32]
        call    bin(std::__cxx11::basic_string<char, std
        lea     rsi, [rsp+32]
```



```cpp
#include <string>

inline
std::string bin(std::string val) {
    val.append("Hello C++!");
    return val;
}


std::string goo(std::string s) {
    return bin(bin(bin(bin(bin(std::move(s))))));
}
```

☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+ ☑ Intel ☑ Demangle

```asm
        mov     rax, r12
        pop     rbp
        pop     r12
        ret
        mov     rbp, rax
        lea     rdi, [rsp+128]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L5
        mov     rbp, rax
.L5:
        lea     rdi, [rsp+96]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L6
        mov     rbp, rax
.L6:
        lea     rdi, [rsp+64]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L7
        mov     rbp, rax
.L7:
        lea     rdi, [rsp+32]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L8
        mov     rbp, rax
.L8:
        mov     rdi, rsp
        call    std::__cxx11::basic_string<char, std::ch
        mov     rdi, rbp
        call    _Unwind_Resume
```

Output (0/0) x86-64 gcc (trunk) - 1448ms (212276B)

Returning values

- (N)RVO – result is constructed in the caller
- Moved to the caller (CWG 1579)
- Copied into the caller

CWG 1579

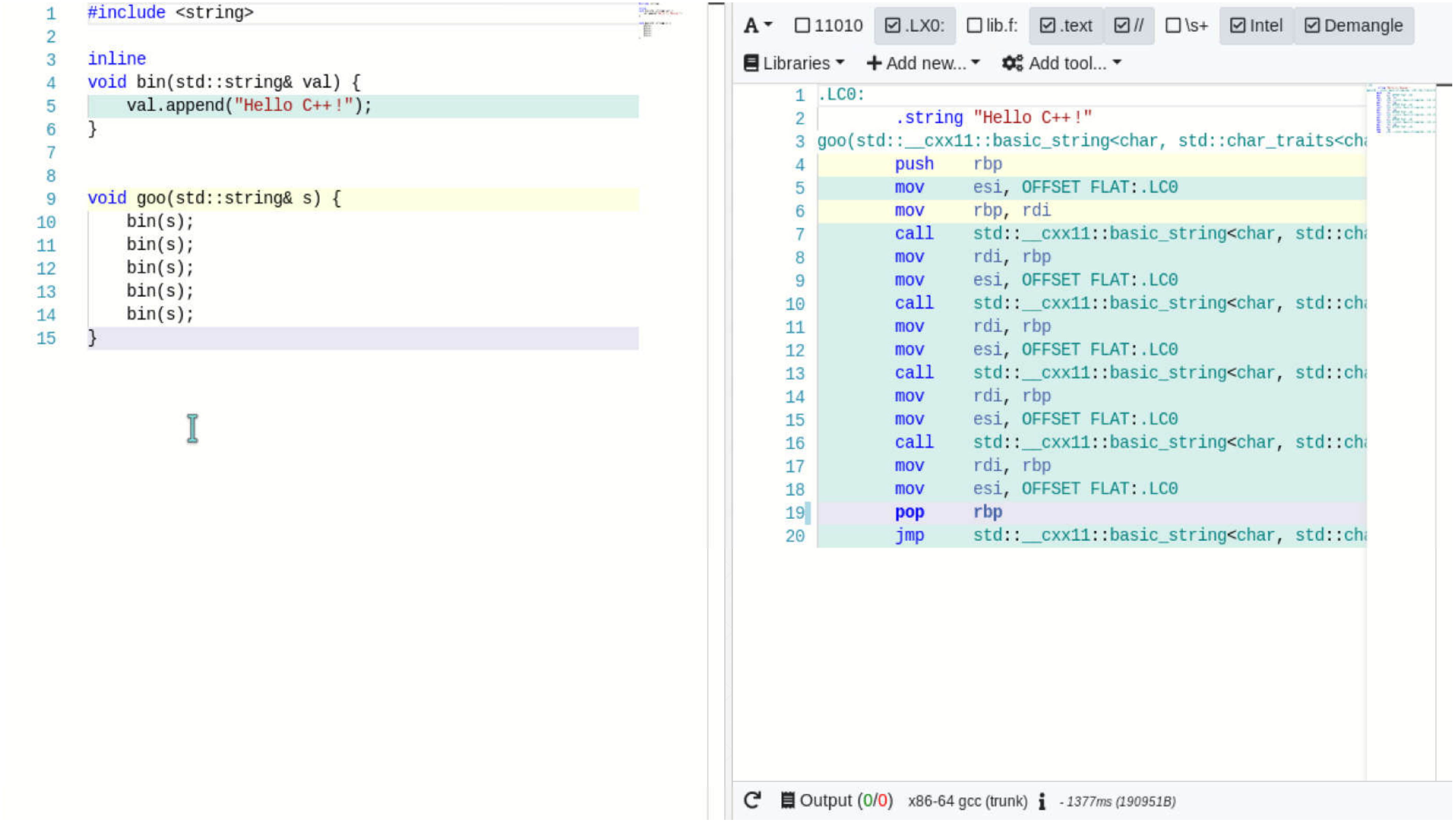
Currently the conditions for moving from an object returned from a function are tied closely to the criteria for copy elision, which requires that the type of the object being returned be the same as the return type of the function. Another possibility that should be considered is to allow something like

```
optional<T> foo() {
    T t;
    ...
    return t;
}
```

and allow `optional<T>::optional(T&&)` to be used for the initialization of the return type. **Currently this can be achieved explicitly by use of std::move, but it would be nice not to have to remember to do so.**

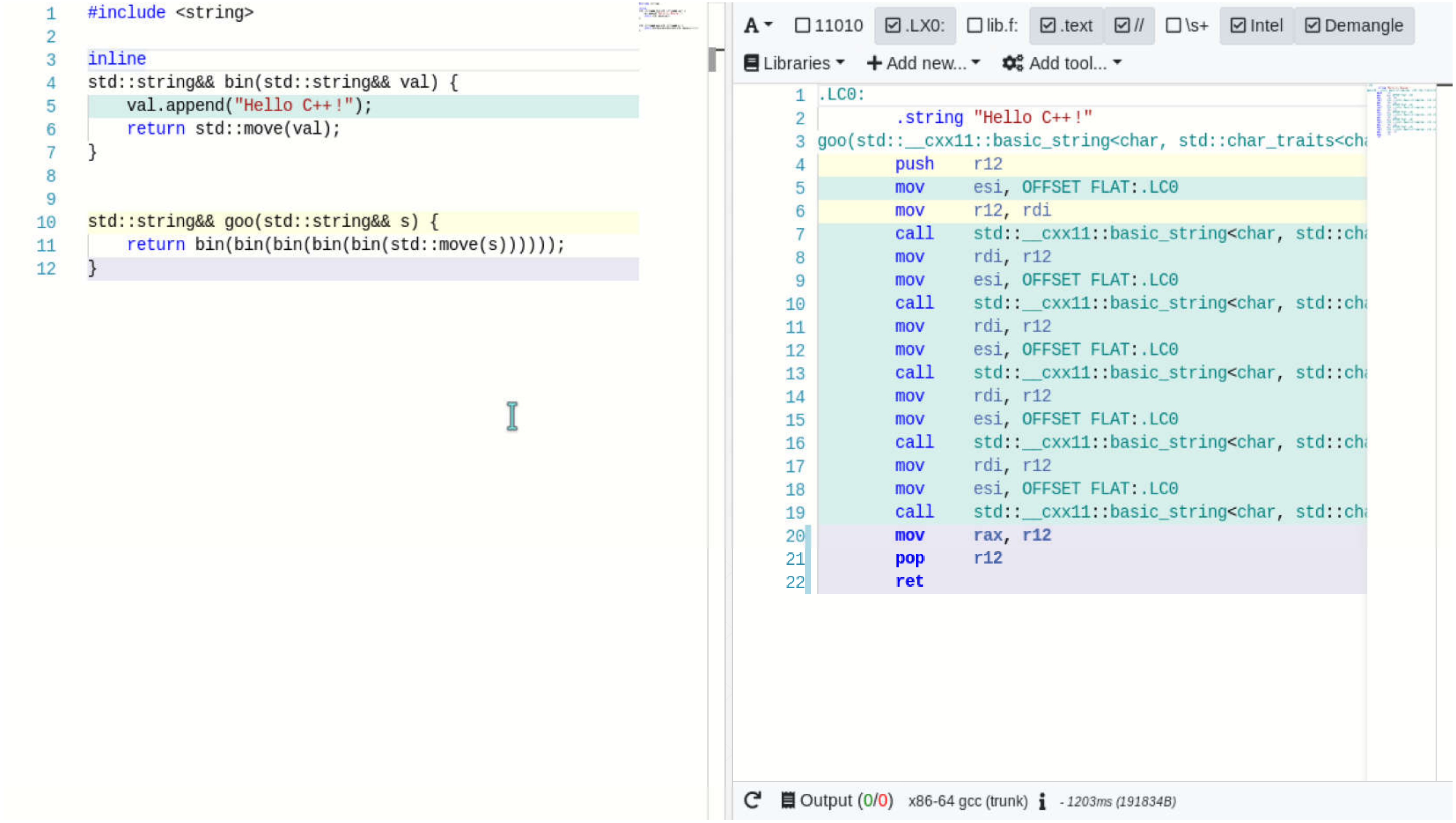
Returning values

```
U fun( )
{
    T value;
    ...
    return value; // move constructed
}
```



```cpp
#include <string>

inline
void bin(std::string& val) {
    val.append("Hello C++!");
}


void goo(std::string& s) {
    bin(s);
    bin(s);
    bin(s);
    bin(s);
    bin(s);
}
```

```asm
.LC0:
        .string "Hello C++!"
goo(std::__cxx11::basic_string<char, std::char_traits<ch
        push    rbp
        mov     esi, OFFSET FLAT:.LC0
        mov     rbp, rdi
        call    std::__cxx11::basic_string<char, std::ch
        mov     rdi, rbp
        mov     esi, OFFSET FLAT:.LC0
        call    std::__cxx11::basic_string<char, std::ch
        mov     rdi, rbp
        mov     esi, OFFSET FLAT:.LC0
        call    std::__cxx11::basic_string<char, std::ch
        mov     rdi, rbp
        mov     esi, OFFSET FLAT:.LC0
        call    std::__cxx11::basic_string<char, std::ch
        mov     rdi, rbp
        mov     esi, OFFSET FLAT:.LC0
        pop     rbp
        jmp     std::__cxx11::basic_string<char, std::ch
```

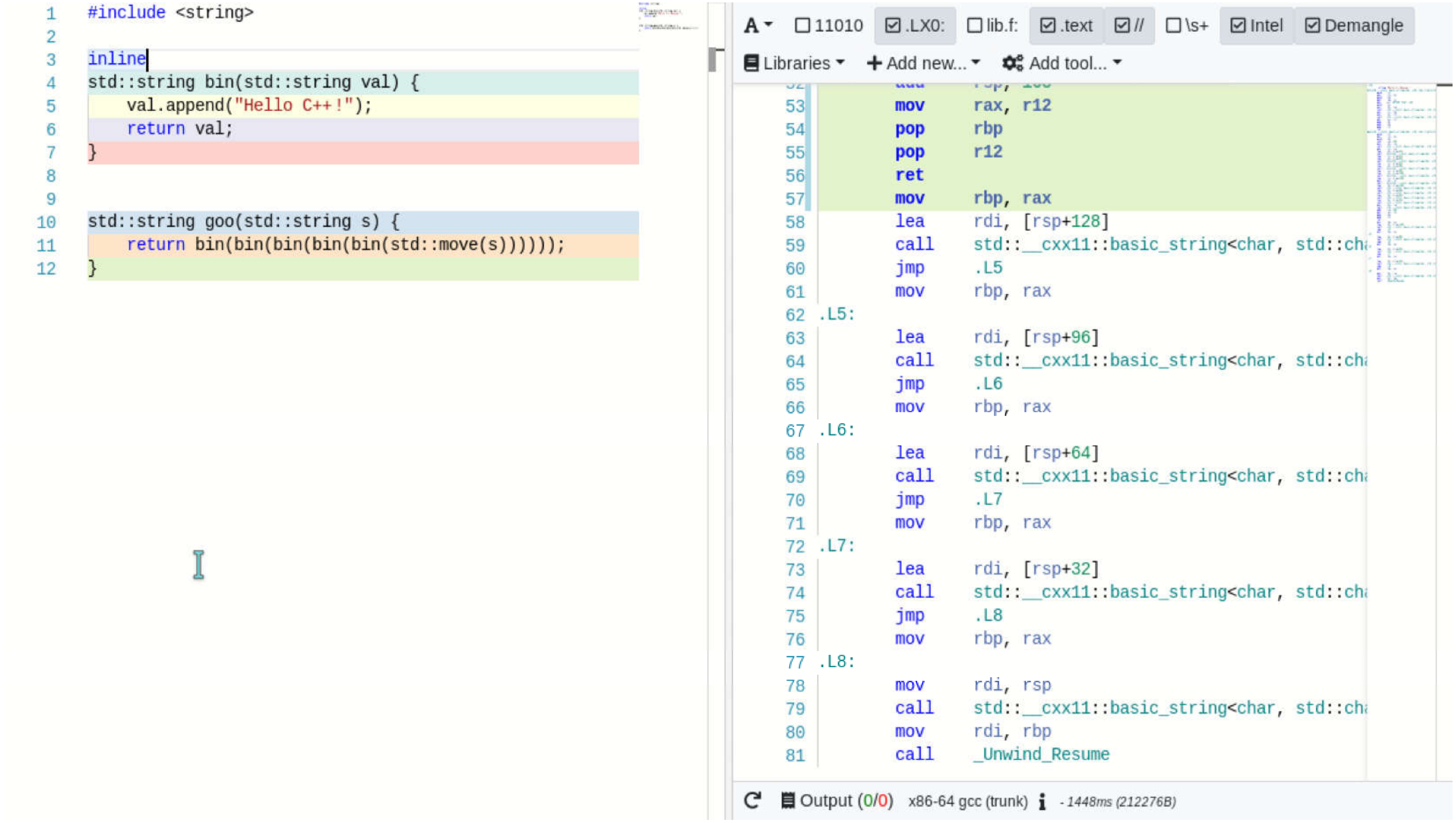


```cpp
#include <string>

inline
std::string&& bin(std::string&& val) {
    val.append("Hello C++!");
    return std::move(val);
}


std::string&& goo(std::string&& s) {
    return bin(bin(bin(bin(bin(std::move(s))))));
}
```

☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+ ☑ Intel ☑ Demangle

```asm
.LC0:
        .string "Hello C++!"
goo(std::__cxx11::basic_string<char, std::char_traits<cha
        push    r12
        mov     esi, OFFSET FLAT:.LC0
        mov     r12, rdi
        call    std::__cxx11::basic_string<char, std::cha
        mov     rdi, r12
        mov     esi, OFFSET FLAT:.LC0
        call    std::__cxx11::basic_string<char, std::cha
        mov     rdi, r12
        mov     esi, OFFSET FLAT:.LC0
        call    std::__cxx11::basic_string<char, std::cha
        mov     rdi, r12
        mov     esi, OFFSET FLAT:.LC0
        call    std::__cxx11::basic_string<char, std::cha
        mov     rdi, r12
        mov     esi, OFFSET FLAT:.LC0
        call    std::__cxx11::basic_string<char, std::cha
        mov     rax, r12
        pop     r12
        ret
```

Output (0/0) x86-64 gcc (trunk) - 1203ms (191834B)



```cpp
#include <string>

inline
std::string bin(std::string val) {
    val.append("Hello C++!");
    return val;
}


std::string goo(std::string s) {
    return bin(bin(bin(bin(bin(std::move(s))))));
}
```

A ☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+ ☑ Intel ☑ Demangle

Libraries ▾ + Add new... ▾ Add tool... ▾

```asm
        mov     rax, r12
        pop     rbp
        pop     r12
        ret
        mov     rbp, rax
        lea     rdi, [rsp+128]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L5
        mov     rbp, rax
.L5:
        lea     rdi, [rsp+96]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L6
        mov     rbp, rax
.L6:
        lea     rdi, [rsp+64]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L7
        mov     rbp, rax
.L7:
        lea     rdi, [rsp+32]
        call    std::__cxx11::basic_string<char, std::ch
        jmp     .L8
        mov     rbp, rax
.L8:
        mov     rdi, rsp
        call    std::__cxx11::basic_string<char, std::ch
        mov     rdi, rbp
        call    _Unwind_Resume
```

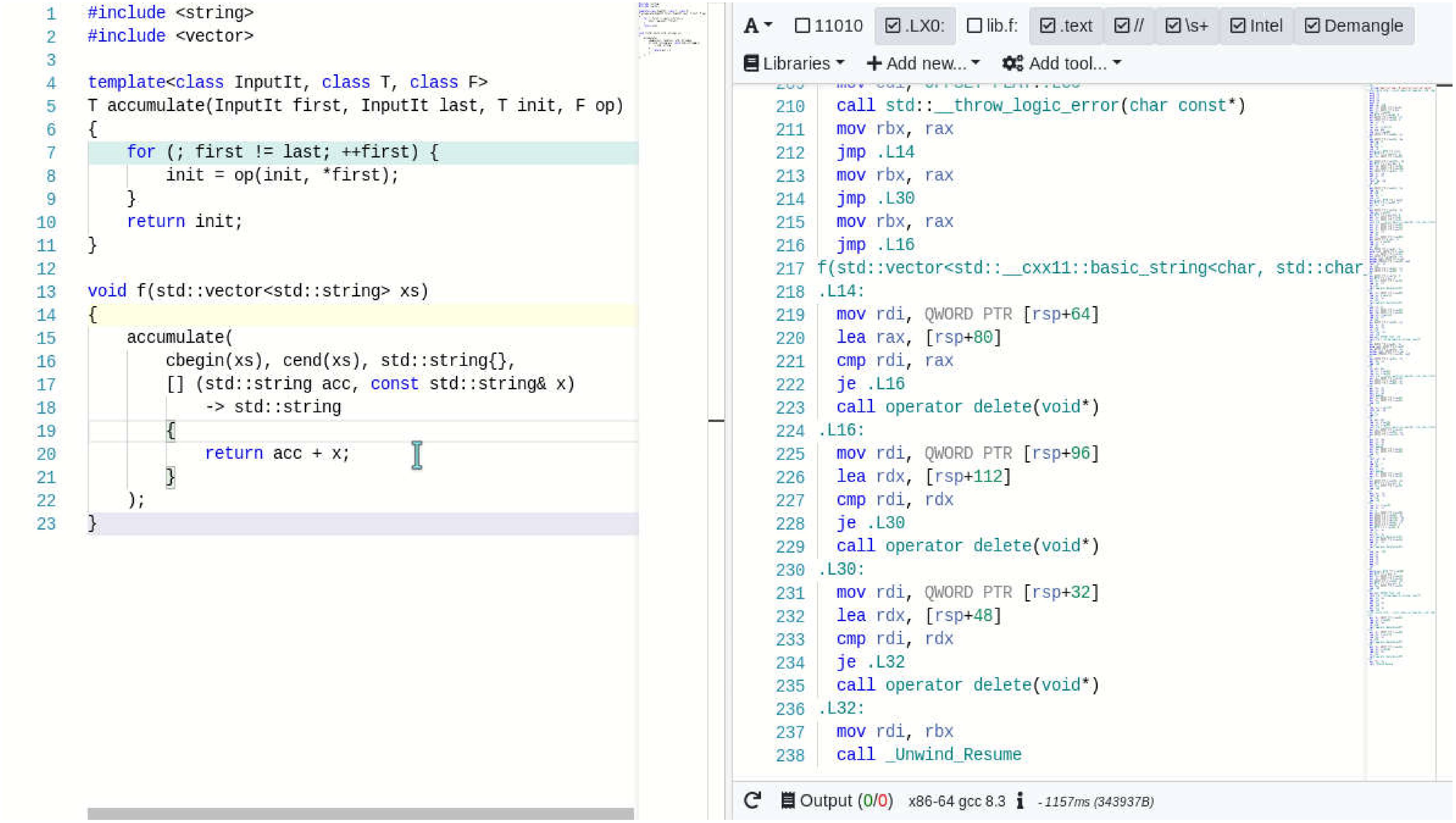
Output (0/0) x86-64 gcc (trunk) - 1448ms (212276B)

Returning values

All temporary objects are destroyed as the last step in evaluating the full-expression that (lexically) contains the point where they were created, and if multiple temporary objects were created, they are destroyed in the order opposite to the order of creation.

Testing strings

```
template <typename InputIt, typename T>
T accumulate(InputIt first, InputIt last, T init)
{
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

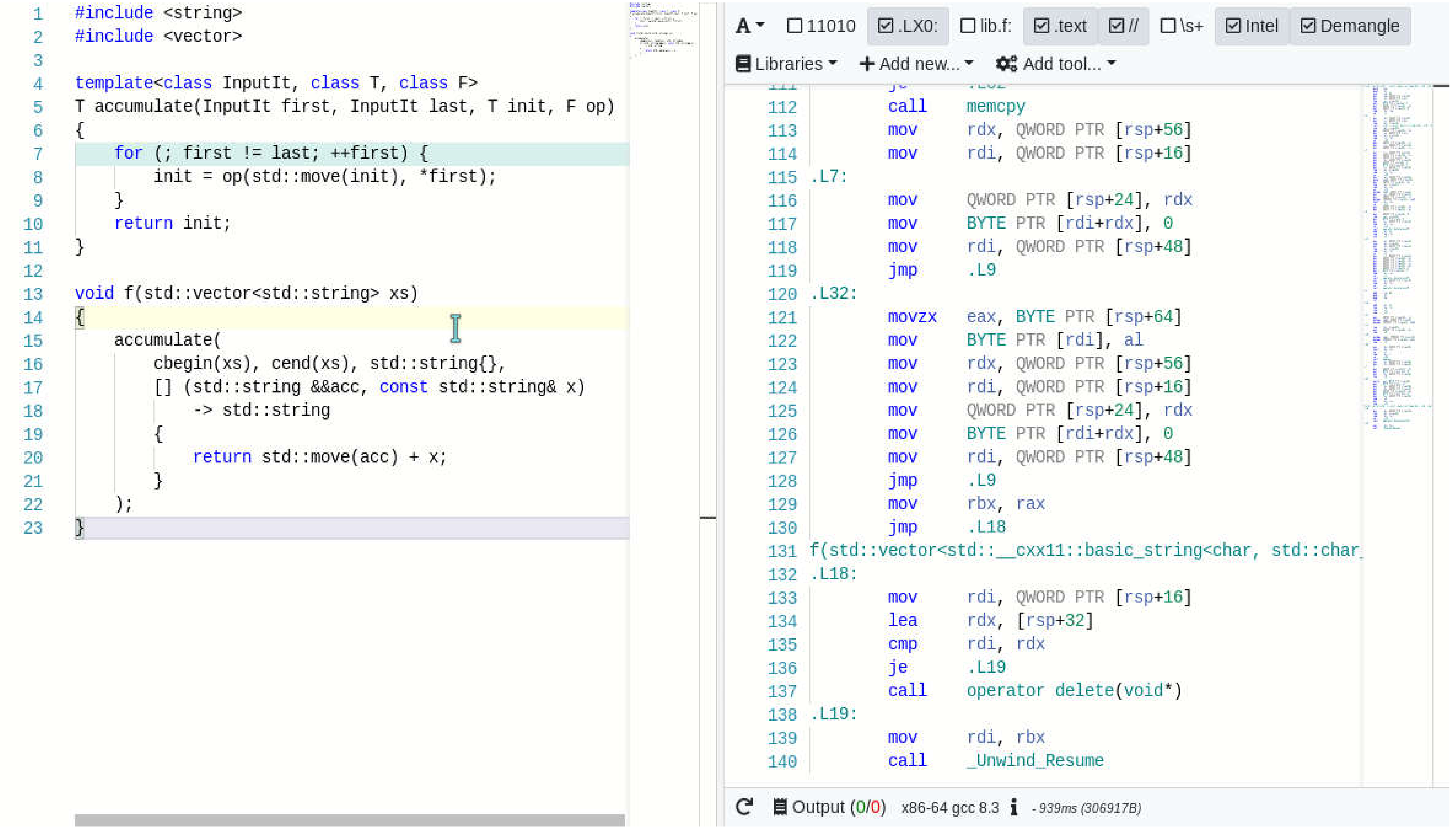


```cpp
#include <string>
#include <vector>

template<class InputIt, class T, class F>
T accumulate(InputIt first, InputIt last, T init, F op)
{
    for (; first != last; ++first) {
        init = op(init, *first);
    }
    return init;
}

void f(std::vector<std::string> xs)
{
    accumulate(
        cbegin(xs), cend(xs), std::string{},
        [] (std::string acc, const std::string& x)
            -> std::string
        {
            return acc + x;
        }
    );
}
```

☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☑ \s+ ☑ Intel ☑ Demangle

```asm
  call std::__throw_logic_error(char const*)
  mov rbx, rax
  jmp .L14
  mov rbx, rax
  jmp .L30
  mov rbx, rax
  jmp .L16
f(std::vector<std::__cxx11::basic_string<char, std::char_
.L14:
  mov rdi, QWORD PTR [rsp+64]
  lea rax, [rsp+80]
  cmp rdi, rax
  je .L16
  call operator delete(void*)
.L16:
  mov rdi, QWORD PTR [rsp+96]
  lea rdx, [rsp+112]
  cmp rdi, rdx
  je .L30
  call operator delete(void*)
.L30:
  mov rdi, QWORD PTR [rsp+32]
  lea rdx, [rsp+48]
  cmp rdi, rdx
  je .L32
  call operator delete(void*)
.L32:
  mov rdi, rbx
  call _Unwind_Resume
```

Output (0/0) x86-64 gcc 8.3 - 1157ms (343937B)



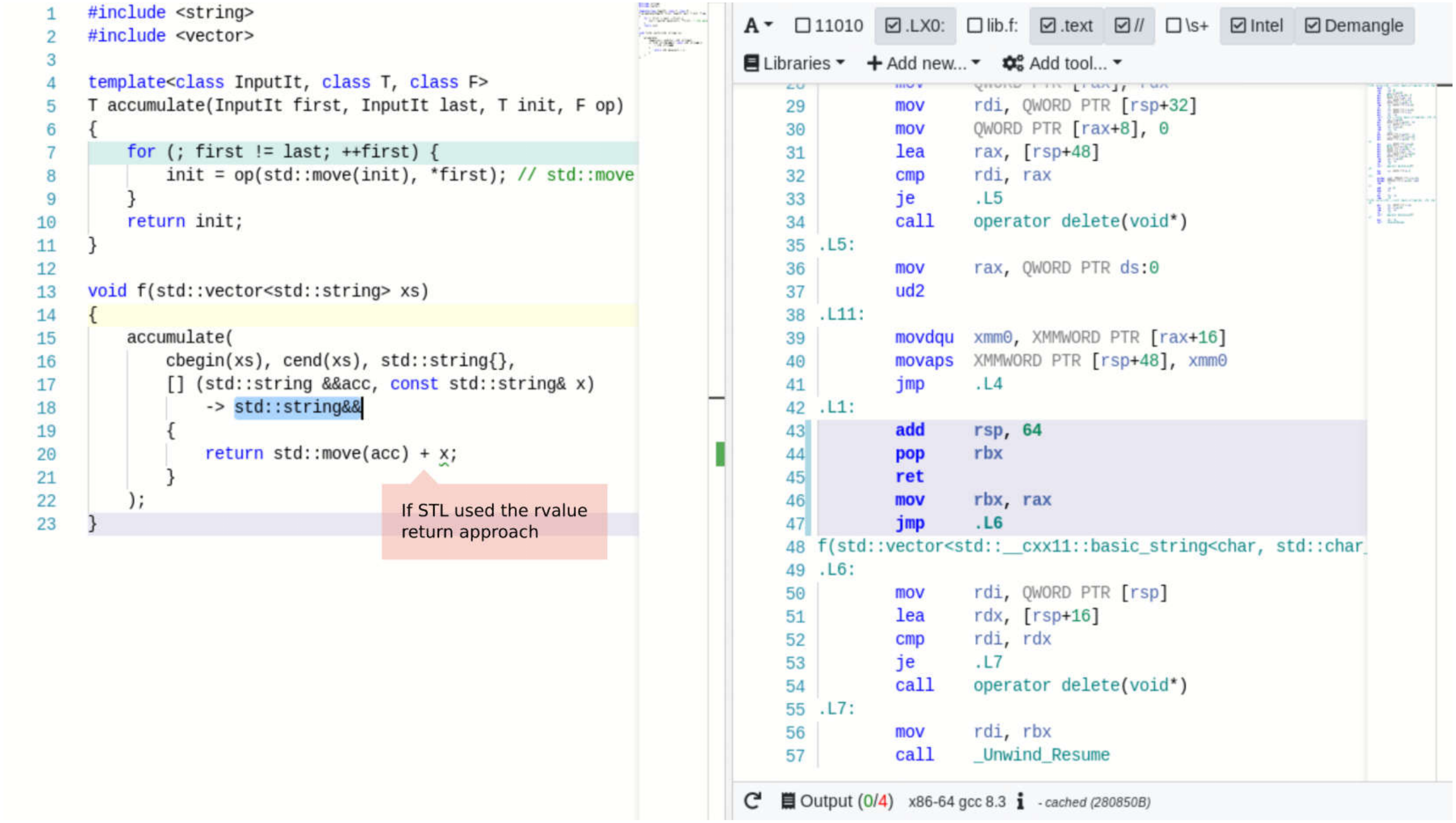
```cpp
#include <string>
#include <vector>

template<class InputIt, class T, class F>
T accumulate(InputIt first, InputIt last, T init, F op)
{
    for (; first != last; ++first) {
        init = op(std::move(init), *first);
    }
    return init;
}

void f(std::vector<std::string> xs)
{
    accumulate(
        cbegin(xs), cend(xs), std::string{},
        [] (std::string &&acc, const std::string& x)
            -> std::string
        {
            return std::move(acc) + x;
        }
    );
}
```

A ☐ 11010 ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+ ☑ Intel ☑ Demangle

Libraries ▾ ✚ Add new... ▾ Add tool... ▾

```asm
        call    memcpy
        mov     rdx, QWORD PTR [rsp+56]
        mov     rdi, QWORD PTR [rsp+16]
.L7:
        mov     QWORD PTR [rsp+24], rdx
        mov     BYTE PTR [rdi+rdx], 0
        mov     rdi, QWORD PTR [rsp+48]
        jmp     .L9
.L32:
        movzx   eax, BYTE PTR [rsp+64]
        mov     BYTE PTR [rdi], al
        mov     rdx, QWORD PTR [rsp+56]
        mov     rdi, QWORD PTR [rsp+16]
        mov     QWORD PTR [rsp+24], rdx
        mov     BYTE PTR [rdi+rdx], 0
        mov     rdi, QWORD PTR [rsp+48]
        jmp     .L9
        mov     rbx, rax
        jmp     .L18
f(std::vector<std::__cxx11::basic_string<char, std::char_
.L18:
        mov     rdi, QWORD PTR [rsp+16]
        lea     rdx, [rsp+32]
        cmp     rdi, rdx
        je      .L19
        call    operator delete(void*)
.L19:
        mov     rdi, rbx
        call    _Unwind_Resume
```

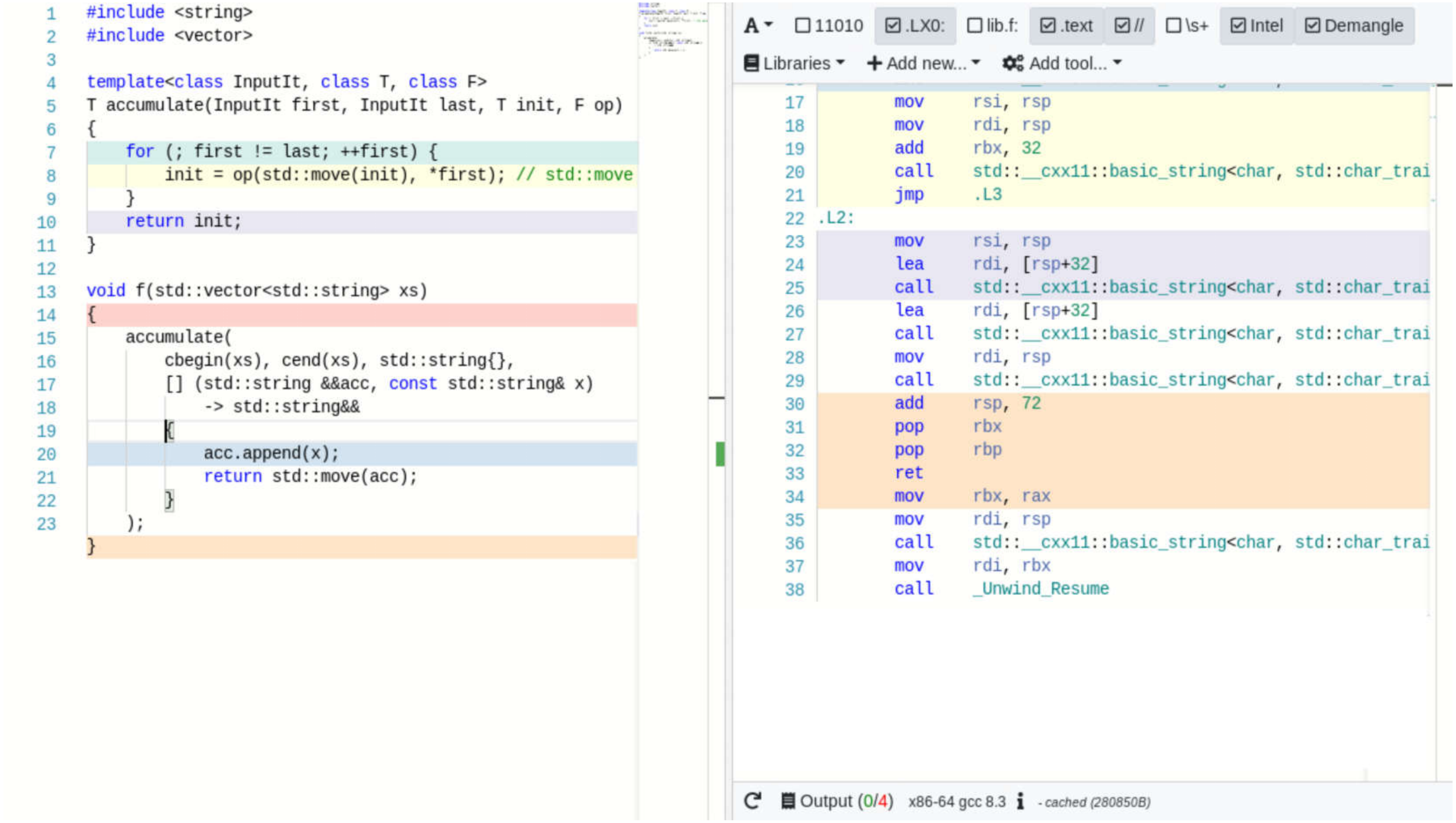
Output (0/0) x86-64 gcc 8.3 - 939ms (306917B)



```cpp
#include <string>
#include <vector>

template<class InputIt, class T, class F>
T accumulate(InputIt first, InputIt last, T init, F op)
{
    for (; first != last; ++first) {
        init = op(std::move(init), *first); // std::move
    }
    return init;
}

void f(std::vector<std::string> xs)
{
    accumulate(
        cbegin(xs), cend(xs), std::string{},
        [] (std::string &&acc, const std::string& x)
            -> std::string&&
        {
            return std::move(acc) + x;
        }
    );
}
```

If STL used the rvalue return approach

```asm
        mov     rdi, QWORD PTR [rsp+32]
        mov     QWORD PTR [rax+8], 0
        lea     rax, [rsp+48]
        cmp     rdi, rax
        je      .L5
        call    operator delete(void*)
.L5:
        mov     rax, QWORD PTR ds:0
        ud2
.L11:
        movdqu  xmm0, XMMWORD PTR [rax+16]
        movaps  XMMWORD PTR [rsp+48], xmm0
        jmp     .L4
.L1:
        add     rsp, 64
        pop     rbx
        ret
        mov     rbx, rax
        jmp     .L6
f(std::vector<std::__cxx11::basic_string<char, std::char_
.L6:
        mov     rdi, QWORD PTR [rsp]
        lea     rdx, [rsp+16]
        cmp     rdi, rdx
        je      .L7
        call    operator delete(void*)
.L7:
        mov     rdi, rbx
        call    _Unwind_Resume
```



```cpp
#include <string>
#include <vector>

template<class InputIt, class T, class F>
T accumulate(InputIt first, InputIt last, T init, F op)
{
    for (; first != last; ++first) {
        init = op(std::move(init), *first); // std::move
    }
    return init;
}

void f(std::vector<std::string> xs)
{
    accumulate(
        cbegin(xs), cend(xs), std::string{},
        [] (std::string &&acc, const std::string& x)
            -> std::string&&
        {
            acc.append(x);
            return std::move(acc);
        }
    );
}
```

```asm
        mov     rsi, rsp
        mov     rdi, rsp
        add     rbx, 32
        call    std::__cxx11::basic_string<char, std::char_trai
        jmp     .L3
.L2:
        mov     rsi, rsp
        lea     rdi, [rsp+32]
        call    std::__cxx11::basic_string<char, std::char_trai
        lea     rdi, [rsp+32]
        call    std::__cxx11::basic_string<char, std::char_trai
        mov     rdi, rsp
        call    std::__cxx11::basic_string<char, std::char_trai
        add     rsp, 72
        pop     rbx
        pop     rbp
        ret
        mov     rbx, rax
        mov     rdi, rsp
        call    std::__cxx11::basic_string<char, std::char_trai
        mov     rdi, rbx
        call    _Unwind_Resume
```

Testing strings

- Consider returning &&
- But be cautious of dangling references
- Store result by-value

Testing strings

```
for (auto x: foo().value()) {
}
```

Testing strings

```
for (auto f = foo(); auto x: f.value()) {
}
```

LINEAR IN C++

Linear in C++

- Moving is required
- Copies should be disallowed
- Moves should be efficient (*)

Moving

- T can be *seen* as T
- T&& can be *seen* as T

Moving

```
detail::linear_usable_as_v<T, T> and
detail::linear_usable_as_v<T, T&&>
```

Moving

```
namespace detail {

template <typename T, typename U>
constexpr bool linear_usable_as_v =

    std::is_nothrow_constructible_v<T, U> and
    std::is_nothrow_assignable_v<T&, U> and
    std::is_nothrow_convertible_v<U, T>;

}
```

No copies allowed

- T& is not T
- `const` T& is not T
- `const` T is not T

Gray place

There’s a thin line between love and hate
Wider divide that you can see between good and bad
There’s a grey place between black and white

Dave Murray, Steve Harris

No copies allowed

```
detail::linear_unusable_as_v<T, T&> and
detail::linear_unusable_as_v<T, const T&> and
detail::linear_unusable_as_v<T, const T>
```

No copies allowed

```
namespace detail {

template <typename T, typename U>
constexpr bool linear_unusable_as_v =

    not std::is_constructible_v<T, U> and
    not std::is_assignable_v<T&, U> and
    not std::is_convertible_v<U, T>;

}
```

Linear in C++

```
template <typename T>
concept Linear =
    std::is_nothrow_destructible_v<T> and

    detail::linear_usable_as<T, T> and
    detail::linear_usable_as<T, T&&> and

    detail::linear_unusable_as<T, T&> and
    detail::linear_unusable_as<T, const T&> and
    detail::linear_unusable_as<T, const T>;
```

Linear in C++

```
auto ptr = std::make_unique<person>();

auto str = ”Hello, Italian C++!”s;
```

Linear in C++

```
Linear ptr = std::make_unique<person>(); // OK

Linear str = ”Hello, Italian C++!”s; // ERROR
```

Linear in C++

```
template <typename T>
    requires(Linear<T>)
auto accumulate(auto xs, T init)
{
    ...
}
```

Linear in C++

```
template <Linear T>
auto accumulate(auto xs, T init)
{
    ...
}
```

Linear in C++

```
auto accumulate(auto xs, Linear auto init)
{
    ...
}
```

Wrapper

What to do with non-linear types?

Linear wrapper

```
template <typename T>
class linear_wrapper {
public:
    linear(const linear&) = delete;
    linear(linear&&) = default; // noexcept

    linear& operator=(const linear&) = delete;
    linear& operator=(linear&&) = default; // noexcept

    …

private:
    T m_value;
};
```

Linear wrapper

```
template <typename T>
class linear_wrapper {
public:
    linear_wrapper(T&& value)
        : m_value{std::move(value)}
    {
    }


    …


private:
    T m_value;
};
```

rvalue ref. -- so we use move on it

Linear wrapper

```
template <typename T>
class linear_wrapper {
public:
    template <typename... Args>
    linear_wrapper(std::in_place_t, Args&&... args)
        : m_value(std::forward<Args>(args)...)
    {
    }

    …

private:
    T m_value;
};
```

Linear wrapper

```
template <typename T>
class linear_wrapper {
public:
    [[nodiscard]] T&& get() && noexcept
    {
        return std::move(value);
    }



    …


private:
    T m_value;
};
```

Linear wrapper

```
template <typename T>
class linear_wrapper {
public:
    [[nodiscard]] T&& operator*() && noexcept
    {
        return std::move(value);
    }


    …

private:
    T m_value;
};
```

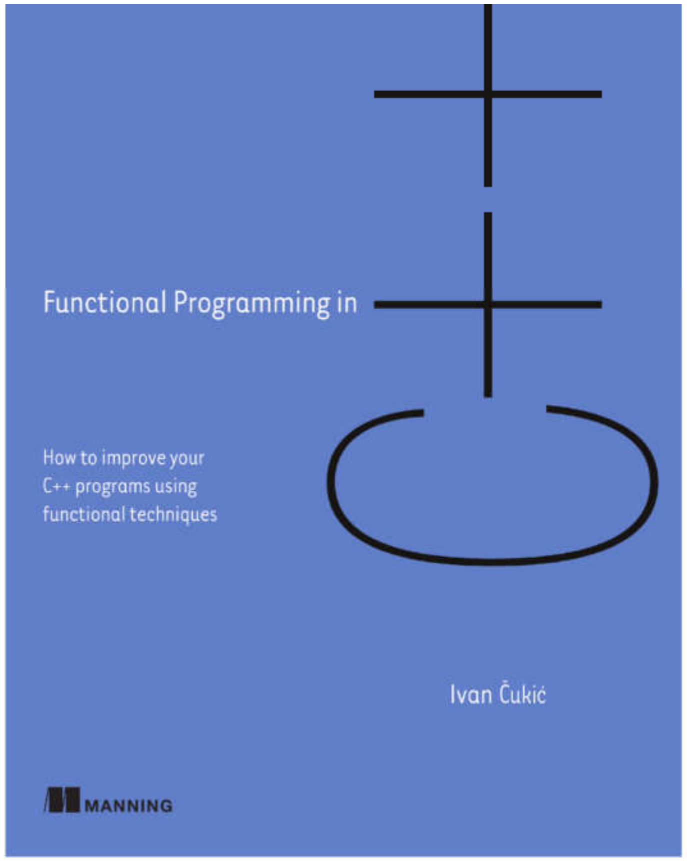
Linear wrapper

```
auto operator"" _ls(const char* data,
                    std::size_t len)
{
    return linear_wrapper<std::string>(std::in_place, data);
}

accumulate(in, "Concatenated:"_ls); // ERROR before C++20
```

Additional

- Use after move (`clang-tidy:bugprone-use-after-move`)
- Unused variable error (`-Werror=unused-variable`)
- Error handling (`optional<T>, expected<T,E>`)

Answers? Questions! Questions? Answers!



cukic.co/to/fp-in-cpp

Functional Programming in C++