Not So Tiny Task №13 (1 point)

Implement generic function

```
template <typename Checker, typename... Args>
int getIndexOfFirstMatch(Checker check, Args... args);
```

that takes a function (check) and variadic list of arguments and returns index of the first argument on which checker returns true.

Avoid unnecessary copying inside and try to use folding for that.

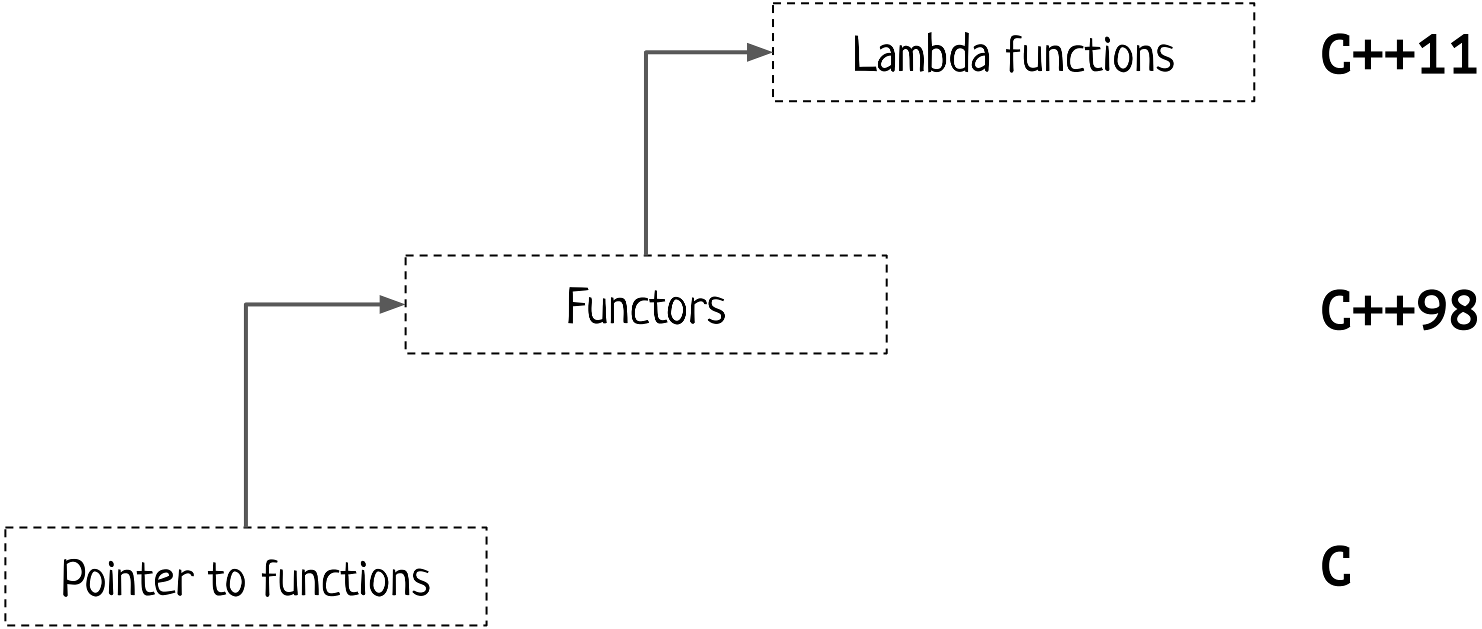
System Programming with C++

Lambdas, perfect forwarding, variadic templates

Lambdas

Lambdas

Functions as first-class citizens in C/C++



Lambdas

Functions as first-class citizens in C/C++

Lambda functions — **C++11**

↑

Functor objects — **C++98**

↑

Pointers to functions — **C**

Lambdas

```
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

Lambdas

```
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

lambda's body

Lambdas

```cpp
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

arguments and return value
(deducted in this example)

lambda's
body

Lambdas

```
int main() {

    auto powerFunction =

            [](int base, int pow) -> int {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

arguments and return value

lambda's body

Lambdas

```cpp
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

arguments and return value (deducted in this example)

lambda's body

Lambdas

```
int main() {

    auto powerFunction =

            [](auto base, auto pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

arguments and return value
(all deducted in this example)

lambda's
body

Lambdas

```cpp
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

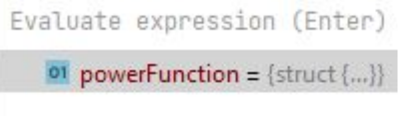
Lambdas

But which type was deducted here?

```
int main() {

    auto powerFunction =

          [](int base, int pow) {
              auto result = 1;
              for (size_t i = 0; i < pow; ++i) {
                  result *= base;
              }
              return result;

          };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

Lambdas

But which type was deducted here?




```
int main() {

    auto powerFunction =

              [](int base, int pow) {
                  auto result = 1;
                  for (size_t i = 0; i < pow; ++i) {
                      result *= base;
                  }
                  return result;

              };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

Lambdas

But which type was deducted here?

Evaluate expression (Enter)

powerFunction = {struct {...}}

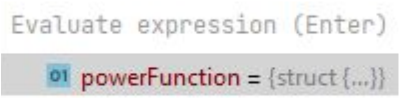
```
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

https://cppinsights.io/s/b5be8c8e

Lambdas

Special class with overloaded operator(int, int) will be generated by the compiler and used here.

But which type was deducted here?

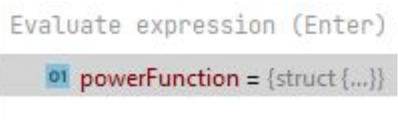



```
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

https://cppinsights.io/s/b5be8c8e

Lambdas

But which type was deducted here?




```
int main() {

    auto powerFunction =

            [](int base, int pow) {
                auto result = 1;
                for (size_t i = 0; i < pow; ++i) {
                    result *= base;
                }
                return result;

            };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

Special class with overloaded `operator(int, int)` will be generated by the compiler and used here.

That's why `auto` is very important here (you literally can't name the right type)

Lambdas

There are situations, when you actually want to name such types (to store lambdas for example).

Lambdas

There are situations, when you actually want to name such types (to store lambdas for example). In such situations just use `std::function` (they have all needed converters).

Lambdas

```
#include <functional>

int main() {

    std::map<const char, std::function<double(double, double)>> tab;






    return 0;
}
```

Lambdas

```
#include <functional>

int main() {

    std::map<const char, std::function<double(double, double)>> tab;
    tab['+'] = [](double a, double b) { return a + b; };
    tab['-'] = [](double a, double b) { return a - b; };
    tab['*'] = [](double a, double b) { return a * b; };
    tab['/'] = [](double a, double b) { return a / b; };




    return 0;
}
```

Lambdas

```cpp
#include <functional>

int main() {

    std::map<const char, std::function<double(double, double)>> tab;
    tab['+'] = [](double a, double b) { return a + b; };
    tab['-'] = [](double a, double b) { return a - b; };
    tab['*'] = [](double a, double b) { return a * b; };
    tab['/'] = [](double a, double b) { return a / b; };

    std::cout << "3.5 + 4.5 = " << tab['+'](3.5, 4.5) << std::endl; // 8
    std::cout << "3.5 * 4.5 = " << tab['*'](3.5, 4.5) << std::endl; // 15.75

    return 0;
}
```

Lambdas: capture

Lambdas: capture

```
int main() {

    auto powerFunction =

        [](int base, int pow) {
            auto result = 1;
            for (size_t i = 0; i < pow; ++i) {
                result *= base;
            }
            return result;

        };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

arguments and return value
(deducted in this example)

lambda's
body

Lambdas: capture

```
int main() {

    auto powerFunction =

          [](int base, int pow) {
              auto result = 1;
              for (size_t i = 0; i < pow; ++i) {
                  result *= base;
              }
              return result;

          };

    std::cout << powerFunction(10, 2);
    return 0;
}
```

arguments and return value (deducted in this example)

capture

lambda's body

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() { return a + b; };







    return 0;
}
```

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() { return a + b; };









    return 0;
}
```

```
a captured by value
b captured by ref
```

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() { return a + b; };

    std::cout << my_lambda() << std::endl; // 43




    return 0;
}
```

a captured by value
b captured by ref

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() { return a + b; };

    std::cout << my_lambda() << std::endl; // 43
    a = 42;
    std::cout << my_lambda() << std::endl; // 43



    return 0;
}
```

```
a captured by value
b captured by ref
```

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() { return a + b; };

    std::cout << my_lambda() << std::endl; // 43
    a = 42;
    std::cout << my_lambda() << std::endl; // 43
    b = 1;
    std::cout << my_lambda() << std::endl; // 11

    return 0;
}
```

a captured by value
b captured by ref

actually a and b are just fields in that generated by the compiler class for this lambda

https://cppinsights.io/s/65aaff4f

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() { return a + b; };

    std::cout << my_lambda() << std::endl; // 43
    a = 42;
    std::cout << my_lambda() << std::endl; // 43
    b = 1;
    std::cout << my_lambda() << std::endl; // 11

    return 0;
}
```

By default captured by value things are not mutable.

a captured by value
b captured by ref

actually a and b are just fields in that generated by the compiler class for this lambda

https://cppinsights.io/s/65aaff4f

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() {
       a = 13;
       return a + b;
    };
    return 0;
}
```

error: assignment of read-only variable 'a'

By default captured by value things are not mutable.

a captured by value
b captured by ref

actually a and b are just fields in that generated by the compiler class for this lambda

https://cppinsights.io/s/65aaff4f

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() mutable {
       a = 13;
       return a + b;
    };
    return 0;
}
```

By default captured by value things are not mutable.

Use mutable if you really want that.

a captured by value
b captured by ref

actually a and b are just fields in that generated by the compiler class for this lambda

https://cppinsights.io/s/65aaff4f

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [a, &b]() mutable {
       a = 13;
       return a + b;
    };
    return 0;
}
```

Special forms of capture:

[] - nothing,

Lambdas: capture

```cpp
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [=]() mutable {
       a = 13;
       return a + b;
    };
    return 0;
}
```

Special forms of capture:

```
[]  - nothing,
[=] - everything by
      value
```

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [=, &a]() {
       a = 13;
       return a + b;
    };
    return 0;
}
```

Special forms of capture:

[] - nothing,
[=] - everything by value

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [&]() {
       a = 13;
       return a + b;
    };
    return 0;
}
```

Special forms of capture:

```
[]  - nothing,
[=] - everything by
      value

[&] - everything by
      reference
```

Lambdas: capture

```
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [&]() {
       a = 13;
       return a + b;
    };
    return 0;
}
```

Special forms of capture:

[] - nothing,
[=] - everything by value

[&] - everything by reference

"Everything" here means everything with automatic storage duration that was visible during declaration of lambda.

Lambdas: capture

```cpp
int main() {

    int a = 10;
    int b = 33;
    auto my_lambda = [&]() {
       a = 13;
       return a + b;
    };
    return 0;
}
```

Special forms of capture:

[] - nothing,

[=] - everything by value

[&] - everything by reference

[this] - well, capture this pointer.

```cpp
class Clazz {
    int a, b;
public:
    Clazz(int a, int b) : a(a), b(b) {}

    void setValues(int a, int b) {
        this->a = a;
        this->b = b;
    }

    auto getSummator() {
        return [this]() { return this->a + this->b; };
    }
};
```

```cpp
class Clazz {
    int a, b;
public:
    Clazz(int a, int b) : a(a), b(b) {}

    void setValues(int a, int b) {
        this->a = a;
        this->b = b;
    }

    auto getSummator() {
        return [this]() { return this->a + this->b; };
    }
};

int main() {
   Clazz example(10, 20);
   auto summator = example.getSummator();

   std::cout << summator() << std::endl; // output: 30



}
```

```cpp
class Clazz {
    int a, b;
public:
    Clazz(int a, int b) : a(a), b(b) {}

    void setValues(int a, int b) {
        this->a = a;
        this->b = b;
    }

    auto getSummator() {
        return [this]() { return this->a + this->b; };
    }
};

int main() {
   Clazz* example = new Clazz(10, 20);
   auto summator = example->getSummator();

   std::cout << summator() << std::endl; // output: 30
   delete example;

   std::cout << summator() << std::endl; // output: KACHUMBA
}
```

```cpp
class Clazz {
    int a, b;
public:
    Clazz(int a, int b) : a(a), b(b) {}

    void setValues(int a, int b) {
        this->a = a;
        this->b = b;
    }

    auto getSummator() {
        return [this]() { return this->a + this->b; };
    }
};

int main() {
   Clazz* example = new Clazz(10, 20);
   auto summator = example->getSummator();

   std::cout << summator() << std::endl; // output: 30
   delete example;

   std::cout << summator() << std::endl; // output: KACHUMBA
}
```

Capturing of pointers or references will not prolong life of corresponding objects.

Lambdas: applications

1. Comparators

Lambdas: applications

1. Comparators

```
#include <algorithm>

bool comp(int i, int j) { return (i < j); }

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5, 4, 3, 2, 1};

    std::sort(v.begin(), v.end());
    return 0;
}
```

Lambdas: applications

1. Comparators

```
#include <algorithm>

bool comp(int i, int j) { return (i < j); }

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5, 4, 3, 2, 1};

    std::sort(v.begin(), v.end(), [](int i, int j) { return i < j; });
    return 0;
}
```

Lambdas: applications

```
int main() {

    auto next_integer = [n = 0]() mutable { return n++; };



    return 0;
}
```

Lambdas: applications

Lambda-specific var, reused across calls

```
int main() {
    auto next_integer = [n = 0]() mutable { return n++; };



    return 0;
}
```

Lambdas: applications

Lambda-specific var, reused across calls

```
int main() {

    auto next_integer = [n = 0]() mutable { return n++; };
    std::cout << next_integer() << std::endl; // 0
    std::cout << next_integer() << std::endl; // 1
    std::cout << next_integer() << std::endl; // 2
    return 0;
}
```

Lambdas: applications

1. Comparators
2. Generators!

Lambdas: applications

1. Comparators

2. Generators!

```cpp
std::vector<int> v(5);
std::generate(v.begin(), v.end(), [n = 0] () mutable {return n++;});

// v: 0 1 2 3 4
```

Lambdas: applications

1. Comparators
2. Generators!
3. Function programming style transformations:

Lambdas: applications

1. Comparators

2. Generators!

3. Function programming style transformations:

```
std::vector<int> v = {1, 2, 3, 4, 5, 4, 3, 2, 1};

// i -> i + 1
std::transform(v.begin(), v.end(), v.begin(), [](int i) {return ++i;});
```

Lambdas: applications

1. Comparators

2. Generators!

3. Function programming style transformations:
   std::transform, std::remove_if, std::accumulate

4. ...

Lambdas: call a lambda?

Task: write a generic function, that takes a lambda, its arguments and call lambda with these arguments!

Lambdas: call a lambda?

Task: write a generic function, that takes a lambda, its arguments and call lambda with these arguments!

Let’s start from a lambda of 2 arguments.

Lambdas: call a lambda?

```
template<typename F, typename T, typename U>
```

Lambdas: call a lambda?

```
template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
   return func(arg1, arg2);
}
```

Lambdas: call a lambda?

```cpp
template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
   return func(arg1, arg2);
}


int main() {
   std::cout << forwarder([](int a, int b){ return a + b;}, 13, 42);
}
```

Lambdas: call a lambda?

See any problems here?

```
template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
   return func(arg1, arg2);
}


int main() {
   std::cout << forwarder([](int a, int b){ return a + b;}, 13, 42);
}
```

Call a function?

```
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   return v1 += v2;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{16};
   std::cout << forwarder(concat<int>, lv1, lv2);
}
```

Call a function?

```
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    return v1 += v2;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

How many copies will be created?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

How many copies will be created?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

How many copies will be created?

3 copies.

2 for arguments, one for return value.

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

How many copies will be created?

3 copies.

2 for arguments, one for return value.

Why?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

How many copies will be created?

3 copies.

2 for arguments, one for return value.

Why? Because auto (and template args deduction) cut of refs!

How to fix?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

Let's start from return value.

auto cut of refs, but who doesn't?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

Let's start from return value.

auto cut of refs, but who doesn't?

decltype!

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
auto forwarder(F func, T arg1, U arg2) -> decltype(func(arg1, arg2)) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

auto cut of refs, but who doesn’t?

decltype!

```
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

auto cut of refs, but who doesn't?

decltype!

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T arg1, U arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

auto cut of refs, but who doesn't?

decltype!

decltype(auto) is just "use decltype rules for type deduction"

lvalue => lvalue ref; prvalue => no ref; xvalue => rvalue ref.

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T arg1, U arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{16};
   std::cout << forwarder(concat<int>, lv1, lv2);
}
```

2 copies left.

How to fix that?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T& arg1, U& arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{16};
    std::cout << forwarder(concat<int>, lv1, lv2);
}
```

2 copies left.

How to fix that?
References!

Problems?

```
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T& arg1, U& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

```
error: cannot bind non-const lvalue reference of type
'Vector<int>&' to an rvalue of type 'Vector<int>'
```

2 copies left.

How to fix that?
References!

Problems?

Will not work
with rvalues.

How to fix?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

~~2 copies left.~~

How to fix that?
References!

Problems?

Will not work
with rvalues.

How to fix?

Universal
references!

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

~~2 copies left.~~

How to fix that?
References!

Problems?

Will not work
with rvalues.

How to fix?

Universal
references!

Any more
problems here?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

~~2 copies left.~~

How to fix that?
References!

Problems?

Will not work
with rvalues.

How to fix?

Universal
references!

Any more
problems here?

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << concat<int>(lv1, lv2 + lv3);
}
```

Suppose Vector class has move constructor.

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << concat<int>(lv1, lv2 + lv3);
}
```

rvalue

Suppose Vector class has move constructor.

How many copies should be created here?

1 new vector will be created inside of operator+.

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << concat<int>(lv1, lv2 + lv3);
}
```

no copying
for this arg!

rvalue

Suppose Vector class has move constructor.

How many copies should be created here?

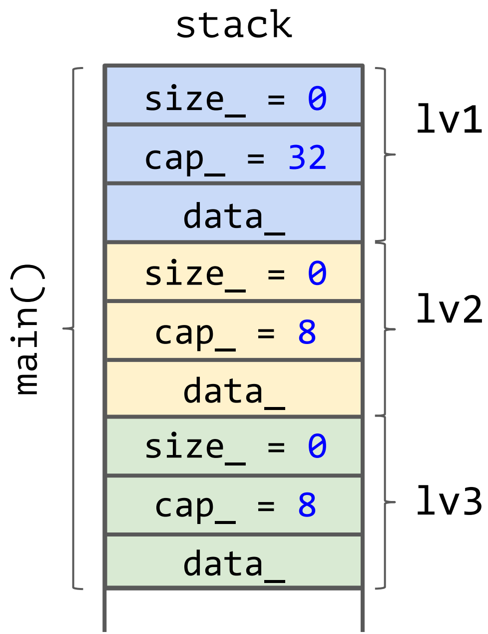
1 new vector will be created inside of operator+.

But then it will be moved to initialize v2!

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << concat<int>(lv1, lv2 + lv3);
}
```

no copying
for this arg!

rvalue

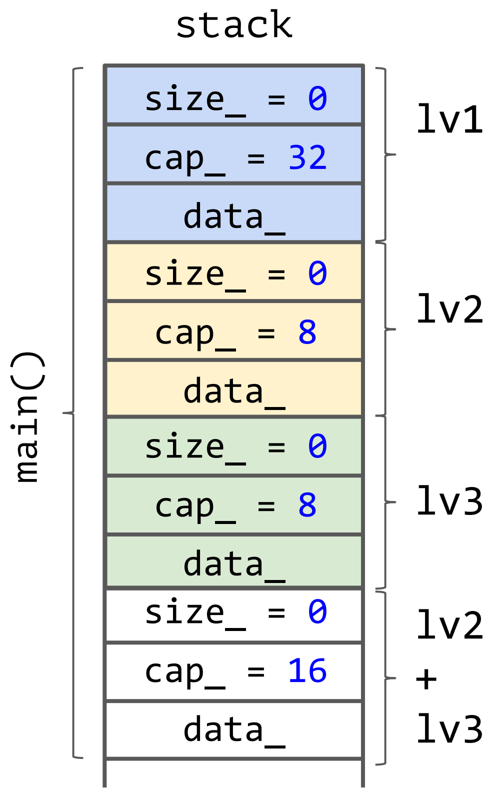


the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << concat<int>(lv1, lv2 + lv3);
}
```

no copying
for this arg!

rvalue



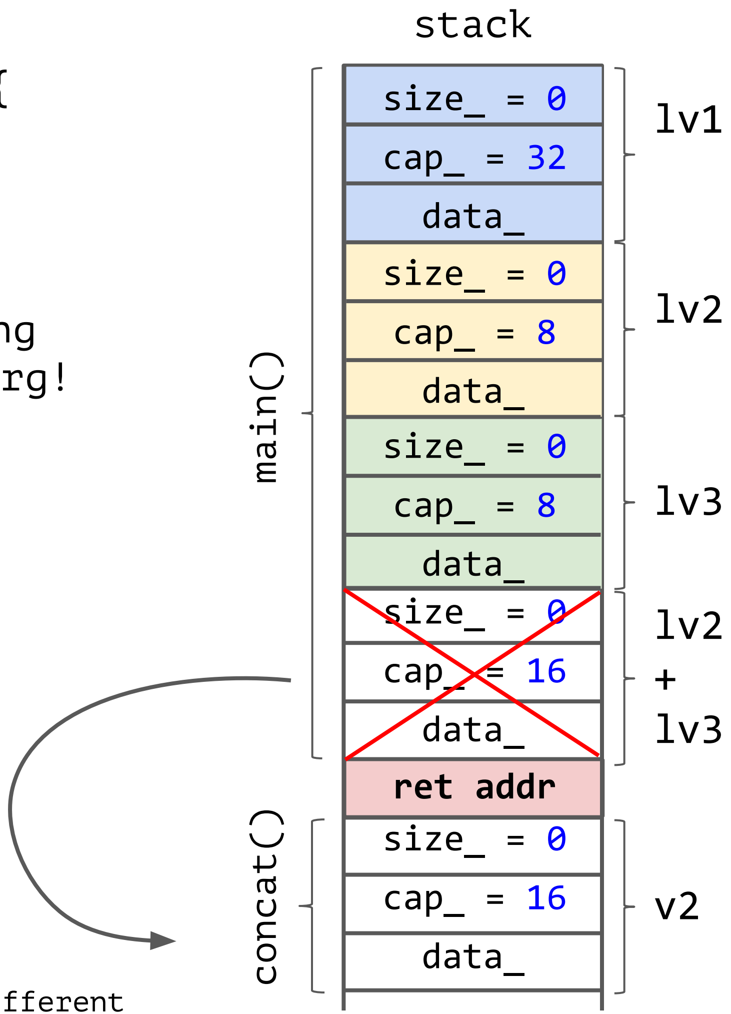
the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << concat<int>(lv1, lv2 + lv3);
}
```

no copying
for this arg!

rvalue

moved-to



the real stack layout can be different

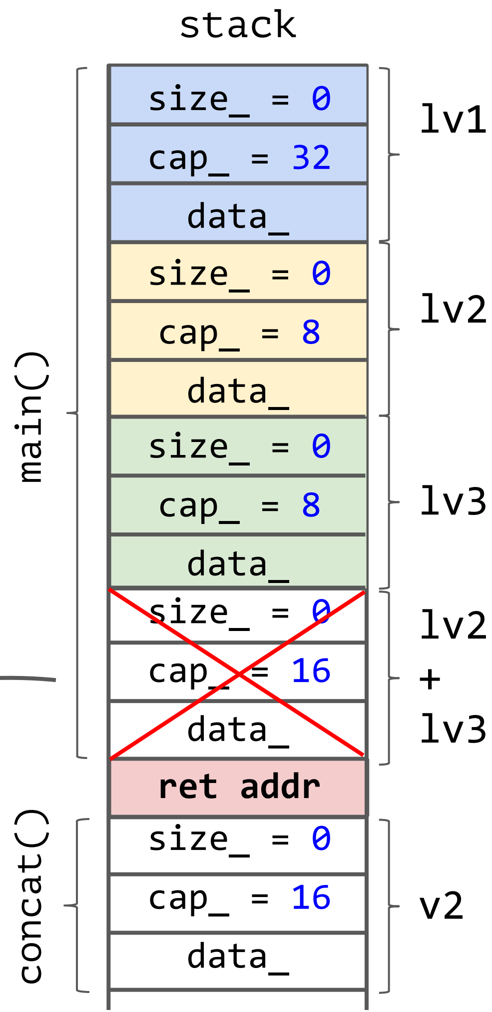
```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << concat<int>(lv1, lv2 + lv3);
}
```

no copying
for this arg!

rvalue

Why move constructor was used?
Because "lv2 + lv3" is rvalue
expression

moved-to



the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

Here we expect the same: no copy, only call of move constructor.

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

Here we expect the same: no copy, only call of move constructor.

However…

https://godbolt.org/z/GM4MxY43K

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

U&& will be deducted to Vector<T>&&

Here we expect the same: no copy, only call of move constructor.

However…

https://godbolt.org/z/GM4MxY43K

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

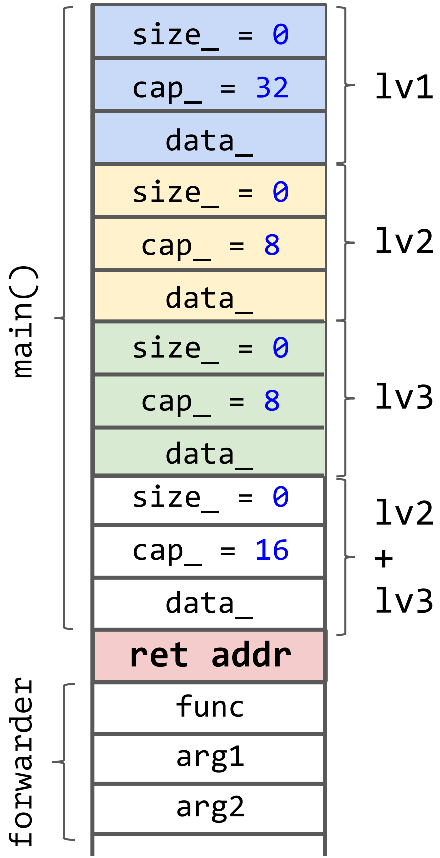
U&& will be deducted to Vector<T>&&, so, temporary object lv2 + lv3 will be materialized (via move ctr)

Here we expect the same: no copy, only call of move constructor.

However…

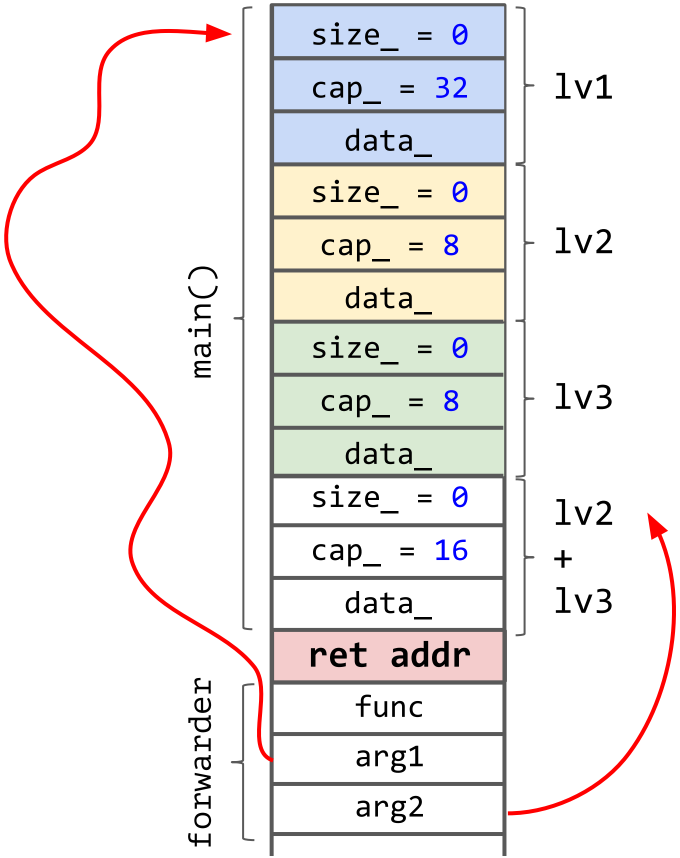
https://godbolt.org/z/GM4MxY43K

```
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```



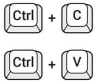
the real stack layout can be different

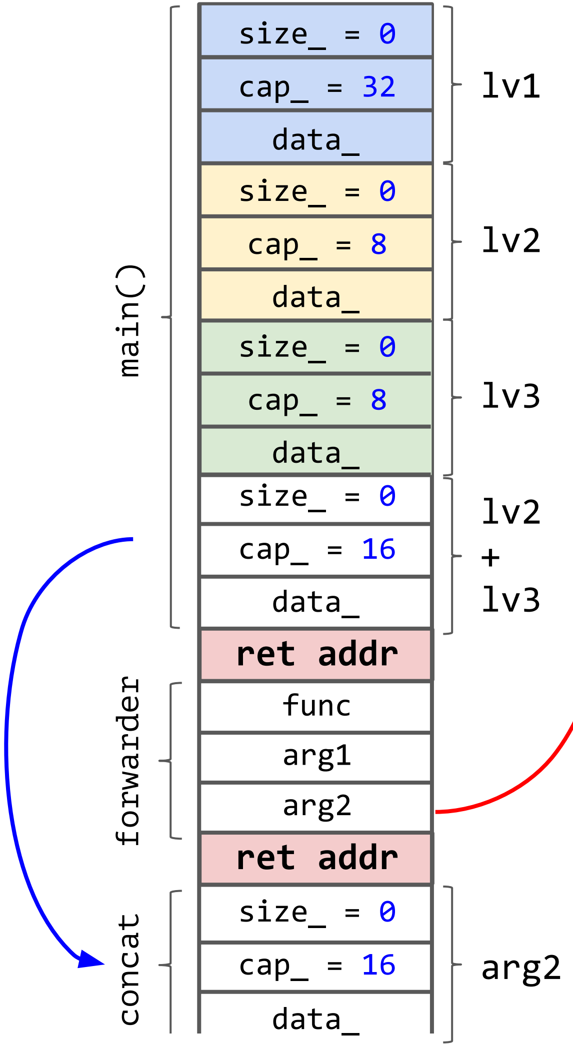
```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```



the real stack layout can be different

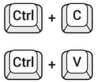
```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```





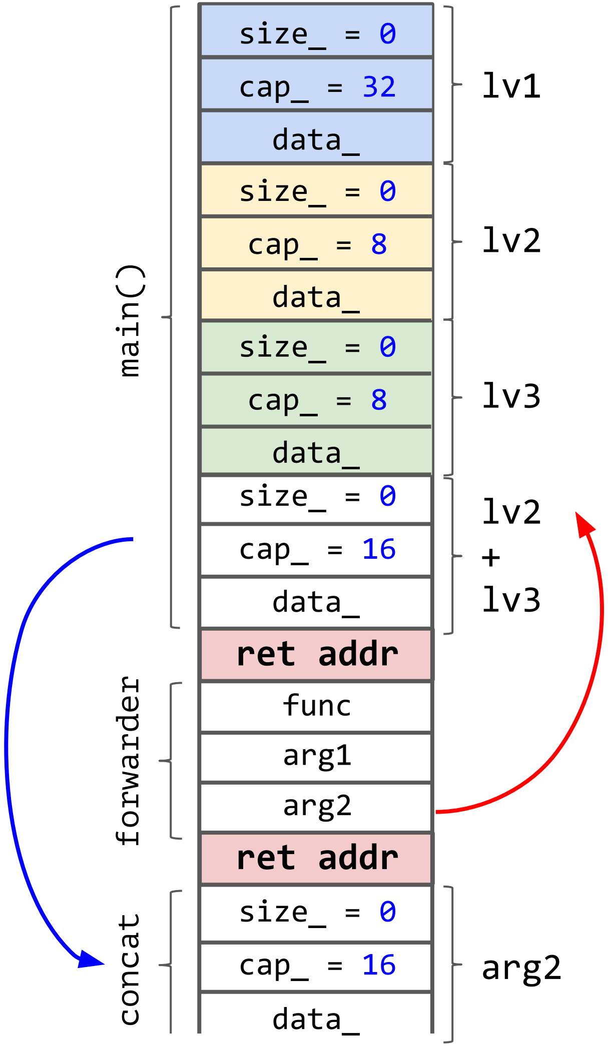
the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

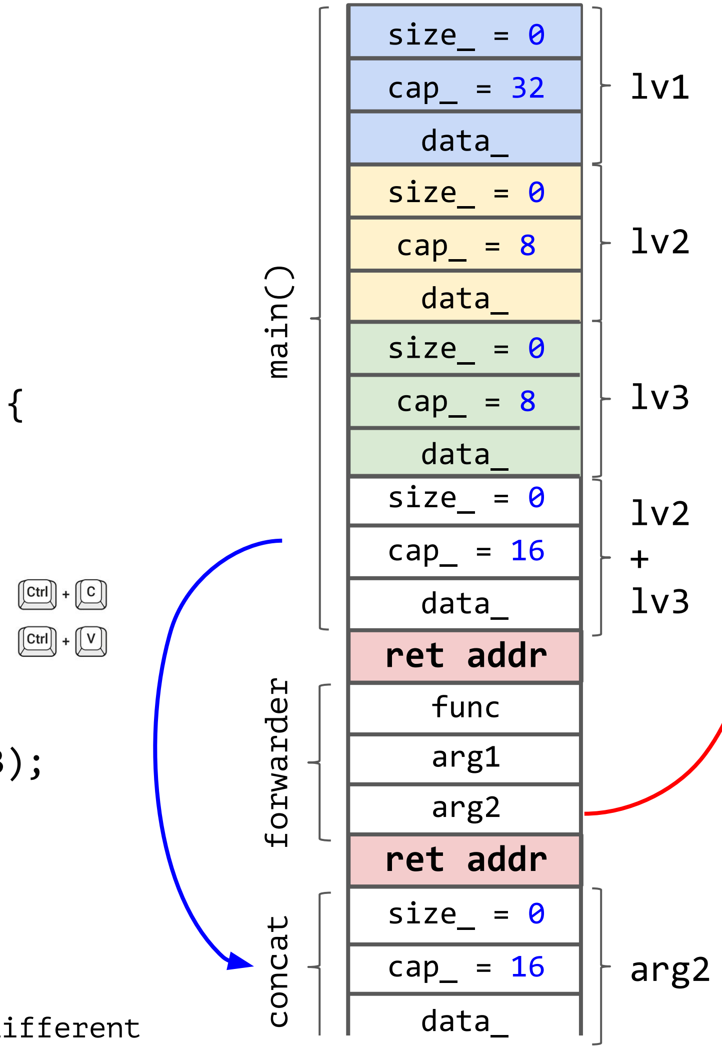


Why copy constructor was used? Because "arg2" is lvalue expression (of type Vector<T>&&).

the real stack layout can be different



```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, arg2);
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```



Why copy constructor was used? Because "arg2" is lvalue expression (of type Vector<T>&&).
How to fix?

the real stack layout can be different

Flashback from lecture #4

```
struct PairOfVectors {
    Vector vec1;
    Vector vec2;

    PairOfVectors(const Vector& v1, const Vector& v2): vec1(v1), vec2(v2) {}

    PairOfVectors (PairOfVectors&& other) {
        this->vec1 = other.vec1;
        this->vec2 = other.vec2;
    }
};
```

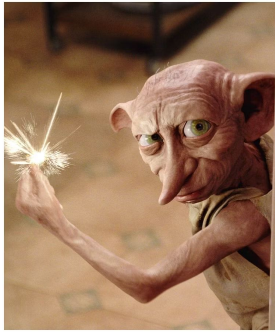
Ok, we are stealing from PairOfVectors, but what about fields? Looks like we are still copying them! How to fix?

Flashback from lecture #4

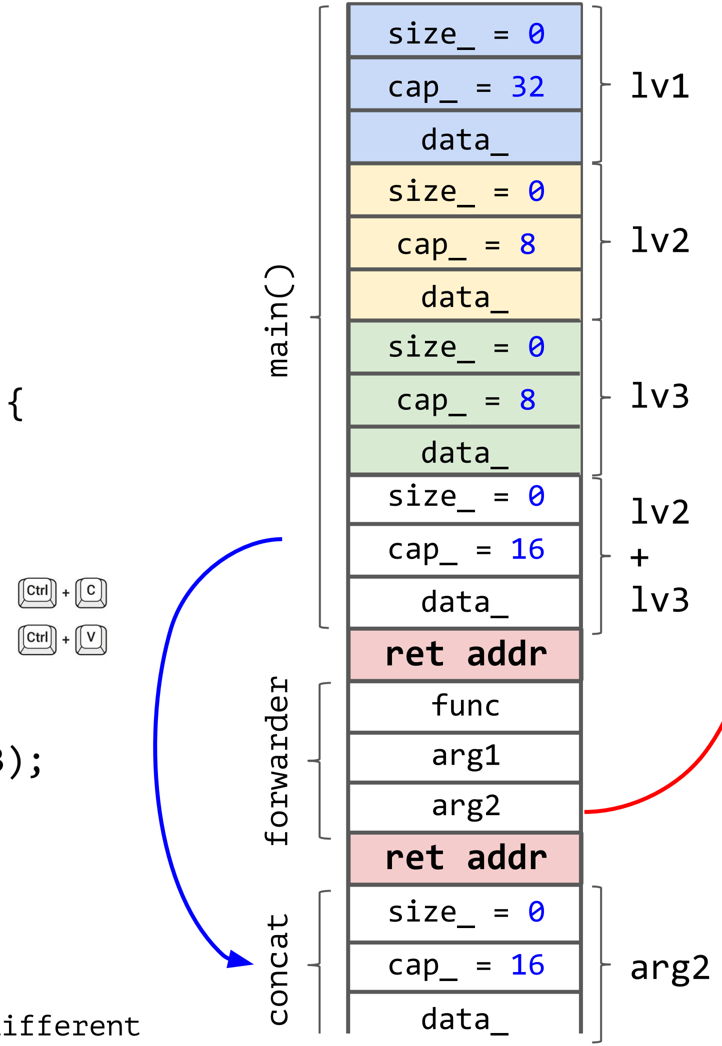
```
struct PairOfVectors {
    Vector vec1;
    Vector vec2;

    PairOfVectors(const Vector& v1, const Vector& v2): vec1(v1), vec2(v2) {}

    PairOfVectors (PairOfVectors&& other) {
        this->vec1 = std::move(other.vec1);
        this->vec2 = std::move(other.vec2);
    }
};
```

Ok, we are stealing from PairOfVectors, but what about fields? Looks like we are still copying them! How to fix?

Ok, now we are stealing fields as well!



```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   return func(arg1, arg2);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

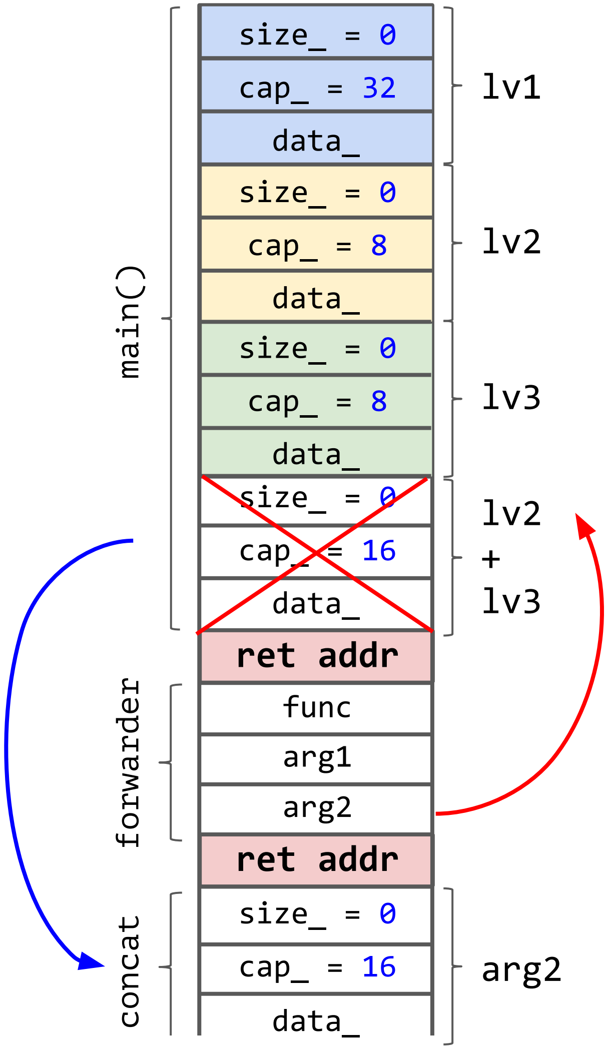


Why copy constructor was used? Because "arg2" is lvalue expression (of type Vector<T>&&). How to fix?

the real stack layout can be different

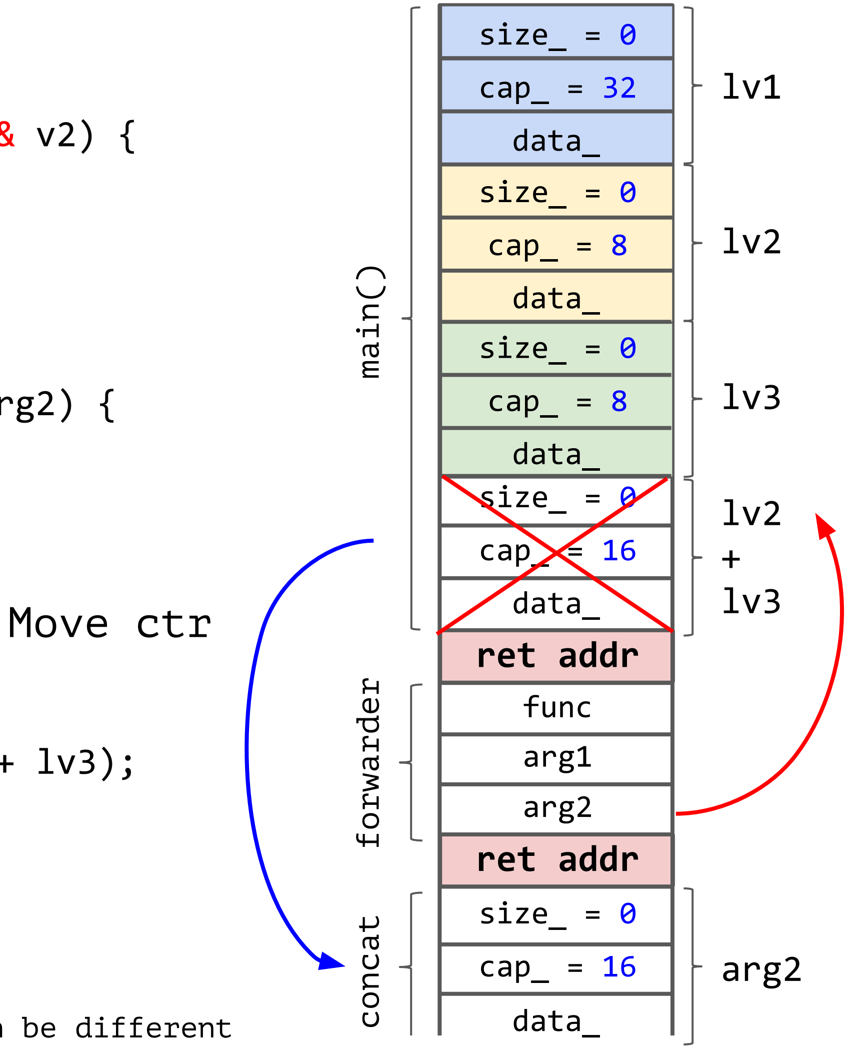
```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, Vector<T> v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, std::move(arg2));
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

Move ctr



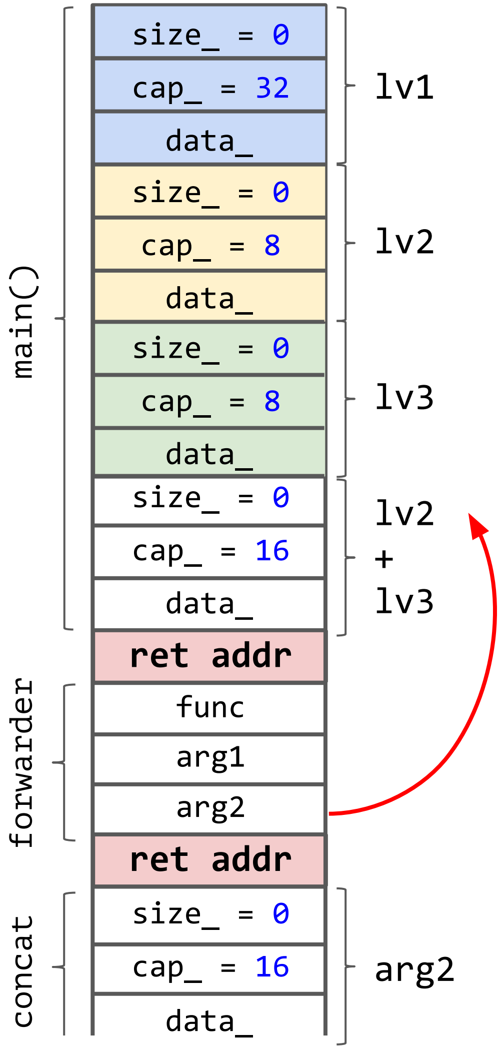
the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
    return func(arg1, std::move(arg2));
}

int main() {
    Vector<int> lv1{32};
    Vector<int> lv2{8};
    Vector<int> lv3{8};
    std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```



the real stack layout can be different

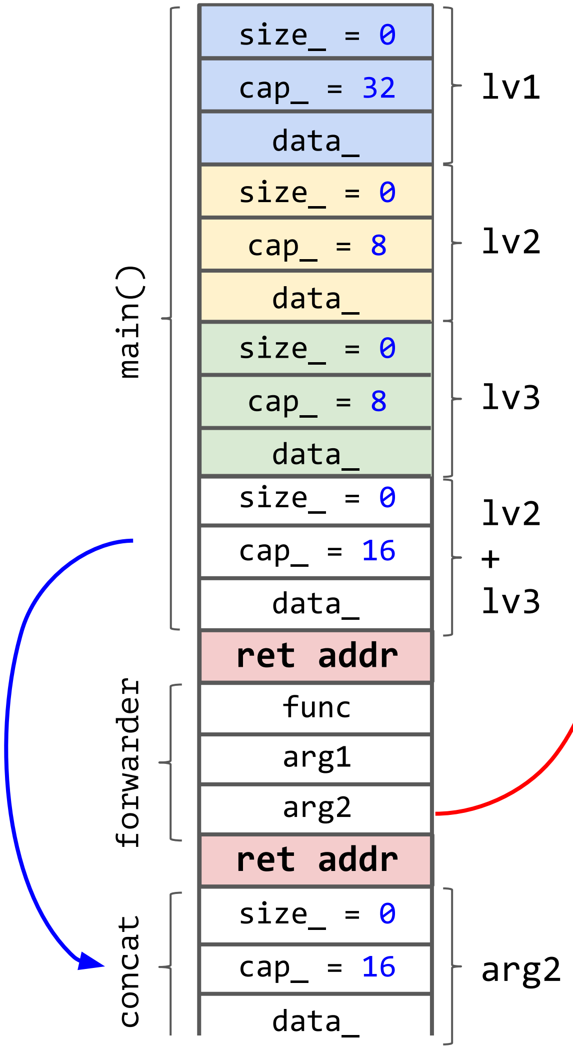
```
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
   if (arg2 is rvalue)
     return func(arg1, arg2);
   else
     return func(arg1, std::move(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```



the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
     return func(arg1, std::forward<U>(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

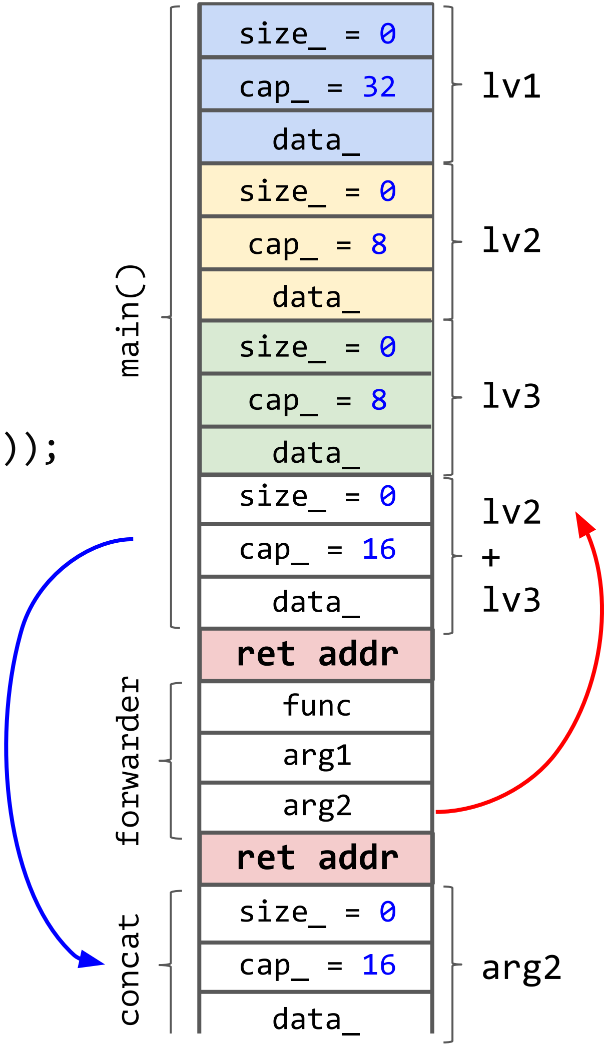
Move ctr



the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
     return func(std::forward<T>(arg1), std::forward<U>(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

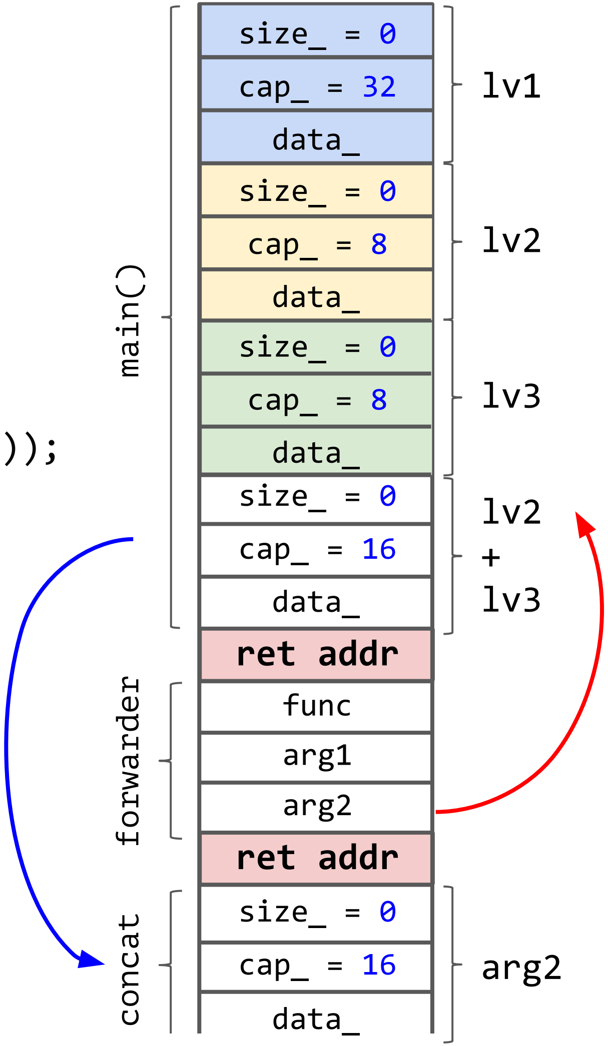
Move ctr



the real stack layout can be different

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
     return func(std::forward<T>(arg1), std::forward<U>(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

Move ctr



This is called perfect forwarding and usually used to pass properties of universal references forward.

the real stack layout can be different

Reference collapsing

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);          // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;   // ---> deducted to int&

}
```

Naive approach: do the same as with lvalue ref, try to add to ampersands.

We can't have 3 ampersands, so, let's collapse them! How?

Universal references: impl

```
template <typename T> void foo1(T& t);
template <typename T> void foo2(T&& t);

int main() {
    int a = 10;
    int& lra = a;
    int&& rra = a + 1;

    foo2(lra);                  // ---> deducted to foo2<int&>(int&)
    auto&& ura = lra;           // ---> deducted to int&

    foo2(lra + 1);              // ---> deducted to foo2<int>(int&&)
    auto&& ura2 = lra + 1;      // ---> deducted to int&&
}
```

Deducting of T into lvalue ref when universal reference is binded to lvalue, is artificial exception in the language to implement std::forward!

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
     return func(std::forward<T>(arg1), std::forward<U>(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

Lambdas: call a lambda?

Task: write a generic function, that takes a lambda, its arguments and call lambda with these arguments!

Let’s start from a lambda of 2 arguments.

Lambdas: call a lambda?

Task: write a generic function, that takes a lambda, its arguments and call lambda with these arguments!

~~Let’s start from a lambda of 2 arguments.~~

Variadic templates

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

Variadic templates

foo is parameterized by
variadic number of types

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

Variadic templates

foo is parameterized by
variadic number of types

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

it also has variadic number of
arguments of these variadic
number of types

Variadic templates

foo is parameterized by variadic number of types

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

it also has variadic number of arguments of these variadic number of types (they are also called "parameter pack")

Variadic templates

foo is parameterized by variadic number of types

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

it also has variadic number of arguments of these variadic number of types (they are also called "parameter pack")

```
foo();                      // the pack is empty
```

Variadic templates

foo is parameterized by
variadic number of types

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

it also has variadic number of arguments of these variadic number of types (they are also called "parameter pack")

```
foo();                          // the pack is empty
foo(1, 2.7, 42, 3.14);  // the pack int, double, int, double
```

Variadic templates

foo is parameterized by
variadic number of types

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

it also has variadic number of arguments of these variadic number of types (they are also called "parameter pack")

```
foo();                    // the pack is empty
foo(1, 2.7, 42, 3.14);    // the pack int, double, int, double
foo(1, 2.7, 42, 3.14, Vector<int>{});
                          // the pack int, double, int, double, Vector<int>
```

Variadic templates

foo is parameterized by
variadic number of types

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

it also has variadic number of arguments of these variadic number of types (they are also called "parameter pack")

But what can we do inside?

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

But what can we do inside?

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

First of all, you can ask: how many ~~wolfs~~ args in pack do I have?

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

First of all, you can ask: how many ~~wolfs~~ args in pack do I have?

For this use:
`sizeof...(args)`

It returns number of args, not size in bytes.

Variadic templates

```
template<typename Head, typename... Tail>
void foo(Head head, Tail... tail) {
   if (sizeof...(tail) == 0) {
       std::cout << "Well, looks like " << head << " is the last one." << std::endl;
   } else {
       std::cout << "Looking at element " << head << ", " << std::endl;
       foo(tail...);
   }
}
```

Variadic templates

```
template<typename None = void>
void foo() {}


template<typename Head, typename... Tail>
void foo(Head head, Tail... tail) {
    if (sizeof...(tail) == 0) {
        std::cout << "Well, looks like " << head << " is the last one." << std::endl;
    } else {
        std::cout << "Looking at element " << head << ", " << std::endl;
        foo(tail...);
    }
}
```

Variadic templates

Well, of course sizeof… is usually used for other things, e.g. allocating an array to store arguments (but we need constexpr for that, will discuss later).

```
template<typename None = void>
void foo() {}


template<typename Head, typename... Tail>
void foo(Head head, Tail... tail) {
    if (sizeof...(tail) == 0) {
        std::cout << "Well, looks like " << head << " is the last one." << std::endl;
    } else {
        std::cout << "Looking at element " << head << ", " << std::endl;
        foo(tail...);
    }
}
```

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

But what can we do inside?

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

But what can we do inside?
We can fold the pack.

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

```
But what can we do inside?
We can fold the pack.

(… op pack)

is the same as:

(…(p1 op p2) op p3) … op pN)
```

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    std::cout << (... + args) << std::endl;
}
```

```
But what can we do inside?
We can fold the pack.

(… op pack)

is the same as:

(…(p1 op p2) op p3) … op pN)
```

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    std::cout << (... + args) << std::endl;
}

foo(1, 3.14);              // 4.14
foo(1, 2.7, 42, 3.14);   // 48.84
```

```
But what can we do inside?
We can fold the pack.

(… op pack)

is the same as:

(…(p1 op p2) op p3) … op pN)
```

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    std::cout << (... + args) << std::endl;
}

foo(1, 3.14);               // 4.14
foo(1, 2.7, 42, 3.14);      // 48.84

foo();
error: fold of empty expansion over operator+
```

But what can we do inside?
We can fold the pack.

(… op pack)

is the same as:

(…(p1 op p2) op p3) … op pN)

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    std::cout << (0 + ... + args) << std::endl;
}

foo(1, 3.14);              // 4.14
foo(1, 2.7, 42, 3.14);    // 48.84

foo();
```

```
But what can we do inside?
We can fold the pack.

(init op … op pack)

is the same as:

(…(init op p1) op p2) … op pN)
```

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

```
But what can we do inside?
We can fold the pack.

(… op pack)

is the same as:

(…(p1 op p2) op p3) … op pN)
```

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

```
But what can we do inside?
We can fold the pack.

(pack op ...)

is the same as:

(p1 op (p2 op (...(pN-1 op pN)...)
```

Variadic templates

But what can we do inside?
We can fold the pack.

(pack op ...)

is the same as:

(p1 op (p2 op (...(pN-1 op pN)...)

```
template<typename... Args>
void foo(Args... args) {
    std::cout << (args / ...) << std::endl;
}

template<typename... Args>
void baz(Args... args) {
    std::cout << (... / args) << std::endl;
}

foo(1, 2, 4, 8.0);   // ???
baz(1, 2, 4, 8.0);   // ???
```

Variadic templates

But what can we do inside?
We can fold the pack.

(pack op …)

is the same as:

(p1 op (p2 op (…(pN-1 op pN)…)

```
template<typename... Args>
void foo(Args... args) {
   std::cout << (args / ...) << std::endl;
}

template<typename... Args>
void baz(Args... args) {
   std::cout << (... / args) << std::endl;
}

foo(1, 2, 4, 8.0);   // 0.25
baz(1, 2, 4, 8.0);   // 0
```

Variadic templates

But what can we do inside?
We can fold the pack.

(pack op … op fini)

is the same as:

(p1 op (p2 op (…(pN op fini)…)

```
template<typename... Args>
void foo(Args... args) {
   std::cout << (args / ... / 1) << std::endl;
}

template<typename... Args>
void baz(Args... args) {
   std::cout << (1.0 / ... / args) << std::endl;
}

foo(1, 2, 4, 8.0);   // 0.25
baz(1, 2, 4, 8.0);   // 0.015625
```

Variadic templates

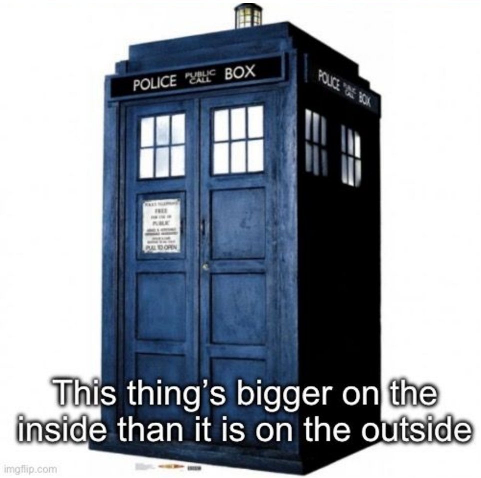
```
(... op pack)            <=> (...(p1 op p2) op p3) ... op pN)
(init op ... op pack) <=> (...(init op p1) op p2) ... op pN)

(pack op ...)            <=> (p1 op (p2 op (...(pN-1 op pN)...)
(pack op ... op fini) <=> (p1 op (p2 op (...(pN op fini)...)
```

Works well for any binary operations (that is supported for every pair of adjacent types).

Variadic templates

```
template<typename ... Args>
void foo(Args... args) {
    ...
}
```

But what else can we do inside?



You can call another functions!

Variadic templates

With variadic with templates args or not.

```
template<typename T, typename U>
void baz(T t, U u) {
   std::cout << t << ", " << u << std::endl;
}

template<typename... Args>
void foo(Args... args) {
   baz(args...);
}

template<typename ... Args>
void bar(Args... args) {
   foo(args...);
}
```

Variadic templates

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

```
template<typename T, typename U>
void baz(T t, U u) {
   std::cout << t << ", " << u << std::endl;
}

template<typename... Args>
void foo(Args... args) {
   baz(args...);
}

template<typename ... Args>
void bar(Args... args) {
   foo(args...);
}
```

Variadic templates

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

```
template<typename T, typename U>
void baz(T t, U u) {
   std::cout << t << ", " << u << std::endl;
}

template<typename... Args>
void foo(Args... args) {
   baz(args...);
}

foo(42, 3.14); // 42, 3.14
```

During the expansion you can actually modify types and values!

Variadic templates

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

```
template<typename T, typename U>
void baz(T t, U u) {
   std::cout << t << ", " << u << std::endl;
}

template<typename... Args>
void foo(Args... args) {
   baz((int) args...);
}

foo(42, 3.14); // 42, 3
```

Variadic templates

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

```
template<typename T, typename U>
void baz(T t, U u) {
   std::cout << t << ", " << u << std::endl;
}

template<typename T>
T transform(T&& t) { return t; }

template<typename... Args>
void foo(Args... args) {
   baz(transform(args)...);
}
Vector<int> v; v.push(13);
foo(42, v); // 42, {13, ...} <--- copy ctr called on return for Vector
```

Variadic templates

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

```
template<typename T, typename U>
void baz(T t, U u) {
   std::cout << t << ", " << u << std::endl;
}

template<typename... Args>
void foo(Args... args) {
   baz(const_cast<const Args*>(&args)...);
}

Vector<int> v; v.push(13);
foo(42, v);
```

Variadic templates

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

```
template<typename T, typename U>
void baz(T t, U u) {
   std::cout << t << ", " << u << std::endl;
}

template<typename... Args>
void foo(Args... args) {
   baz(const_cast<const Args*>(&args)...);
   // --> baz(const_cast<const int*>(&arg1),
              const_cast<const Vector<int>*>(&arg2))
}

Vector<int> v; v.push(13);
foo(42, v);
```

Variadic templates

```
template<typename... Args>
void bar(Args... args) {
   ...
}

template<typename... Args>
void foo(Args... args) {
   bar(args...);
   // ---> bar(1, 2, 3);
}

foo(1, 2, 3);
```

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

General rule: start from ... and go to the left until you have valid expression. Then stop and expand.

Variadic templates

```
template<typename... Args>
void bar(Args... args) {
    ...
}

template<typename... Args>
void foo(Args... args) {
    bar(args...);
    // ---> bar(1, 2, 3);
}

foo(1, 2, 3);
```

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

General rule: start from ... and go to the left until you have valid expression. Then stop and expand.

Variadic templates

```
template<typename... Args>
void bar(Args... args) {
    ...
}

template<typename... Args>
void foo(Args... args) {
    bar(h(args)...);
    // ---> bar(h(1), h(2), h(3));
}

foo(1, 2, 3);
```

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

General rule: start from ... and go to the left until you have valid expression. Then stop and expand.

Variadic templates

```
template<typename... Args>
void bar(Args... args) {
    ...
}

template<typename... Args>
void foo(Args... args) {
    bar(h(args...));
    // ---> bar(h(1, 2, 3));
}

foo(1, 2, 3);
```

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

General rule: start from ... and go to the left until you have valid expression. Then stop and expand.

Variadic templates

```
template<typename... Args>
void bar(Args... args) {
   ...
}

template<typename... Args>
void foo(Args... args) {
   bar(h(args...) + args...);
   --->h(1, 2, 3)
 }

foo(1, 2, 3);
```

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

General rule: start from the most nested ... and go to the left until you have valid expression. Then stop and expand.

Variadic templates

```
template<typename... Args>
void bar(Args... args) {
   ...
}

template<typename... Args>
void foo(Args... args) {
   bar(h(args...) + args...);
   --->bar(h(1, 2, 3) + 1,
           h(1, 2, 3) + 2,
           h(1, 2, 3) + 3)
 }

foo(1, 2, 3);
```

You can call another functions!

With variadic with templates args or not.

For this, we expand the pack.

During the expansion you can actually modify types and values!

General rule: start from the most nested ... and go to the left until you have valid expression. Then stop and expand.

Not So Tiny Task №13 (1 point)

Implement generic function

```
template <typename Checker, typename... Args>
int getIndexOfFirstMatch(Checker check, Args... args);
```

that takes a function (check) and variadic list of arguments and returns index of the first argument on which checker returns true.

Avoid unnecessary copying inside and try to use folding for that.

Lambdas: call a lambda?

Task: write a generic function, that takes a lambda, its arguments and call lambda with these arguments!

~~Let's start from a lambda of 2 arguments.~~

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename T, typename U>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
     return func(std::forward<T>(arg1), std::forward<U>(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename... Args>
decltype(auto) forwarder(F func, T&& arg1, U&& arg2) {
     return func(std::forward<T>(arg1), std::forward<U>(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename... Args>
decltype(auto) forwarder(F func, Args&&... args) {
     return func(std::forward<T>(arg1), std::forward<U>(arg2));
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename... Args>
decltype(auto) forwarder(F func, Args&&... args) {
     return func(std::forward<Args>(args)...);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
   for (auto&& e: v2) v1.push(e);
   return v1;
}

template<typename F, typename... Args>
decltype(auto) forwarder(F func, Args&&... args) {
     return func(std::forward<Args>(args)...);
}

int main() {
   Vector<int> lv1{32};
   Vector<int> lv2{8};
   Vector<int> lv3{8};
   std::cout << forwarder(concat<int>, lv1, lv2 + lv3);
}
```

```cpp
template<typename T>
Vector<T>& concat(Vector<T>& v1, const Vector<T>& v2) {
    for (auto&& e: v2) v1.push(e);
    return v1;
}

template<typename F, typename... Args>
decltype(auto) forwarder(F func, Args&&... args) {
     return func(std::forward<Args>(args)...);
}

int main() {
    Vector<int> lv1{32};
    lv1.push(5);
    std::cout << forwarder(
                    [](int i, double d, Vector<int>& v) {
                        v.push(i + d);
                        return v.size();
                    }, 42, 13.0, v);
}
```

Why to use such forwarders?

Why to use such forwarders?

1. If you have a factory of objects of different types, implemented as number of lambdas (with different number of arguments!)

   Real scenarios from your projects.

Why to use such forwarders?

1. If you have a factory of objects of different types, implemented as number of lambdas (with different number of arguments!)

   Real scenarios from your projects.

2. emplace(_back) functions in standard containers.

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   Node(Node* next, const T& val):
   next(next), value(val)  {}
};
```

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   Node(Node* next, const T& val):
   next(next), value(val)  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   void addToHead(const U& value) {
       auto new_node = new Node(head, value);
       head = new_node;
   }
};
```

```cpp
template<typename T>
struct Node {
    T value;
    Node* next;

    Node(Node* next, const T& val):
    next(next), value(val)  {}
};


template<typename T>
class LinkedList {
    Node<T>* head;
public:

    void addToHead(const U& value) {
        auto new_node = new Node(head, value);
        head = new_node;
    }
};
```

```cpp
template<typename T>
struct Buffer {
    ...
    Buffer(size_t s, T def_val) {
        std::cout << "created";
        // ... heavy operations
    }

    Buffer(const Buffer& other) {
        std::cout << "copied";
        // ... heavy operations
    }


    Buffer(Buffer&& other) {
        std::cout << "moved";
        // ... not so heavy operations
    }

};
```

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   Node(Node* next, const T& val):
   next(next), value(val)  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   void addToHead(const U& value) {
       auto new_node = new Node(head, value);
       head = new_node;
   }
};
```

```cpp
LinkedList<Buffer<int>> l;
l.addToHead(Buffer<int>{1000, 1});
```

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   Node(Node* next, const T& val):
   next(next), value(val)  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   void addToHead(const U& value) {
       auto new_node = new Node(head, value);
       head = new_node;
   }
};
```

```cpp
LinkedList<Buffer<int>> l;
l.addToHead(Buffer<int>{1000, 1});

// created
// copied

// Heavy temporary object
// just copied. How to fix?
```

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   Node(Node* next, const T& val):
   next(next), value(val)  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   void addToHead(const U& value) {
       auto new_node = new Node(head, value);
       head = new_node;
   }
};
```

```cpp
LinkedList<Buffer<int>> l;
l.addToHead(Buffer<int>{1000, 1});

// created
// copied

// Heavy temporary object
// just copied. How to fix?
```

Via perfect forwarding!

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   Node(Node* next, const T& val):
   next(next), value(val)  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   template<typename U>
   void addToHead(U&& value) {
       auto new_node = new Node(head,
                               std::forward<U>(value));
       head = new_node;
   }
};
```

```cpp
LinkedList<Buffer<int>> l;
l.addToHead(Buffer<int>{1000, 1});

// created
// copied

// Heavy temporary object
// just copied. How to fix?
```

Via perfect forwarding!

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   template<typename U>
   Node(Node* next, U&& val):
   next(next), value(std::forward<U>(val))  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   template<typename U>
   void addToHead(U&& value) {
       auto new_node = new Node(head,
                           std::forward<U>(value));
       head = new_node;
   }
};
```

```cpp
LinkedList<Buffer<int>> l;
l.addToHead(Buffer<int>{1000, 1});

// created
// moved
```

Via perfect forwarding!

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   template<typename U>
   Node(Node* next, U&& val):
   next(next), value(std::forward<U>(val))  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   template<typename U>
   void addToHead(U&& value) {
       auto new_node = new Node(head,
                            std::forward<U>(value));
       head = new_node;
   }
};
```

```cpp
LinkedList<Buffer<int>> l;
l.addToHead(Buffer<int>{1000, 1});

// created
// moved
```

Via perfect forwarding!
Better, no copying.

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   template<typename U>
   Node(Node* next, U&& val):
   next(next), value(std::forward<U>(val))  {}
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   template<typename U>
   void addToHead(U&& value) {
       auto new_node = new Node(head,
                                std::forward<U>(value));
       head = new_node;
   }
};
```

```cpp
LinkedList<Buffer<int>> l;
l.addToHead(Buffer<int>{1000, 1});

// created
// moved
```

Via perfect forwarding!
Better, no copying.

But why we even create
this temporary object?

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   template<typename... Args>
   Node(Node* next, Args&&... args): next(next), value(args...)  { }
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   template<typename... Args>
   void emplaceToHead(Args&&... args) {
      auto new_node = new Node(head, std::forward<Args>(args)...);
      head = new_node;
   }

};
```

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   template<typename... Args>
   Node(Node* next, Args&&... args): next(next), value(args...)  { }
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   template<typename... Args>
   void emplaceToHead(Args&&... args) {
      auto new_node = new Node(head, std::forward<Args>(args)...);
      head = new_node;
   }

};
```

Creates new element emplace!
Just inside newly allocated Node

```cpp
template<typename T>
struct Node {
   T value;
   Node* next;

   template<typename... Args>
   Node(Node* next, Args&&... args): next(next), value(args...)  { }
};


template<typename T>
class LinkedList {
   Node<T>* head;
public:

   template<typename... Args>
   void emplaceToHead(Args&&... args) {
      auto new_node = new Node(head, std::forward<Args>(args)...);
      head = new_node;
   }

};
```

Creates new element emplace!
Just inside newly allocated Node

Perfect forwarding for all arguments

```cpp
template<typename T>
struct Node {
    T value;
    Node* next;

    template<typename... Args>
    Node(Node* next, Args&&... args): next(next), value(args...)  { }
};


template<typename T>
class LinkedList {
    Node<T>* head;
public:

    template<typename... Args>
    void emplaceToHead(Args&&... args) {
        auto new_node = new Node(head, std::forward<Args>(args)...);
        head = new_node;
    }

};
```

Expand arguments to invoke constructor!

Creates new element emplace!
Just inside newly allocated Node

Perfect forwarding for all arguments

```cpp
template<typename T>
struct Node {
    T value;
    Node* next;

    template<typename... Args>
    Node(Node* next, Args&&... args): next(next), value(args...)  { }
};


template<typename T>
class LinkedList {
    Node<T>* head;
public:

    template<typename... Args>
    void emplaceToHead(Args&&... args) {
        auto new_node = new Node(head, std::forward<Args>(args)...);
        head = new_node;
    }

};
```

So, there will be only initial ctr call, no copy ctr, no move ctr 🥳

Expand arguments to invoke constructor!

Creates new element emplace!
Just inside newly allocated Node

Perfect forwarding for all arguments

https://godbolt.org/z/ze1cc5df9

Takeaways

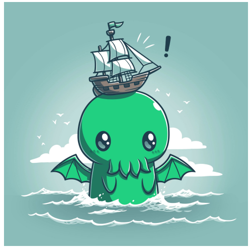
- lambdas are wonderful

- Perfect forwarding is perfect

Takeaways

- lambdas are wonderful

- Perfect forwarding is perfect



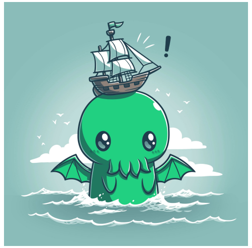
- Variadic templates are so functional (wow)

Takeaways

- lambdas are wonderful

- Perfect forwarding is perfect

- Variadic templates are so functional (wow)



- emplace_back is efficient (use it right)