Филипп Хандельянц

Лекция 3/12

Вывод типов в C++

PVS-Studio

Докладчик

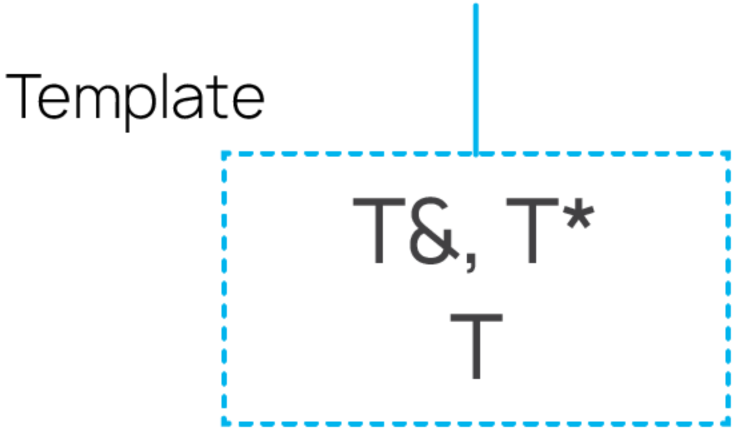
Хандельянц
Филипп Александрович

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

Зачем это нужно

- До C++11 вывод типов применялся в шаблонах
- Начиная с C++11 все заверте...
- C++11: rvalue/forwarding reference, auto, decltype, lambda capture, return type deduction for lambda
- C++14: function return type deduction, lambda capture with initialization
- Как все это работает – не всегда понятно

Type deduction

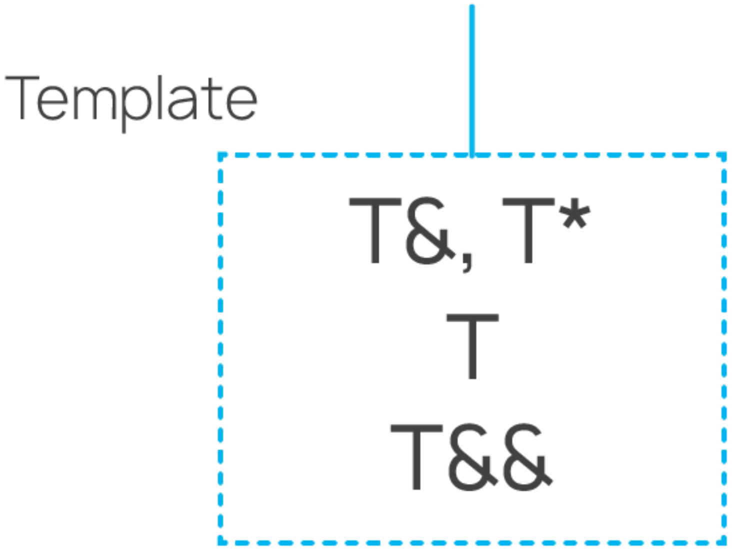


Template

T&, T*

T

Type deduction



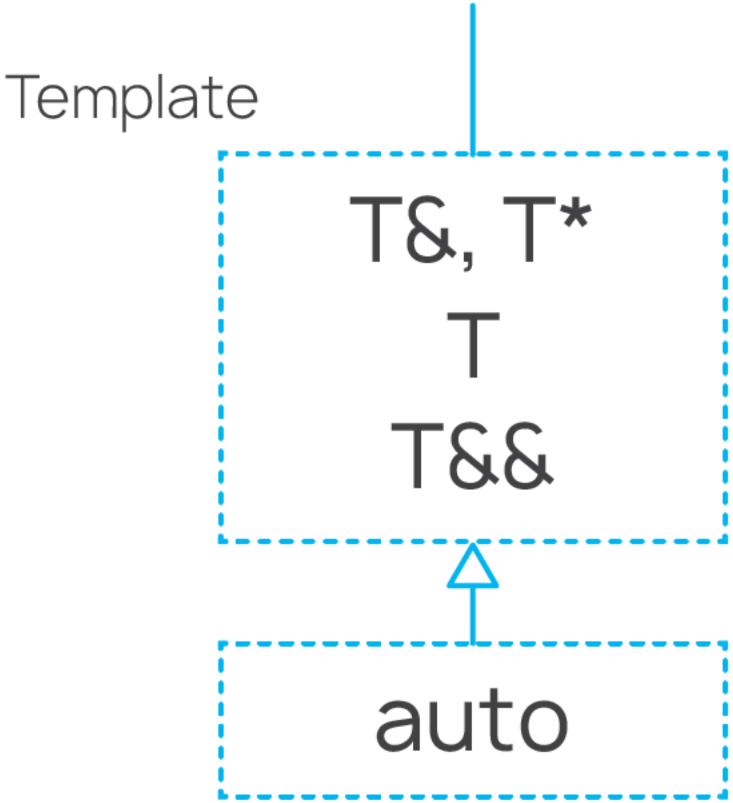
Template

T&, T*

T

T&&

Type deduction



Template

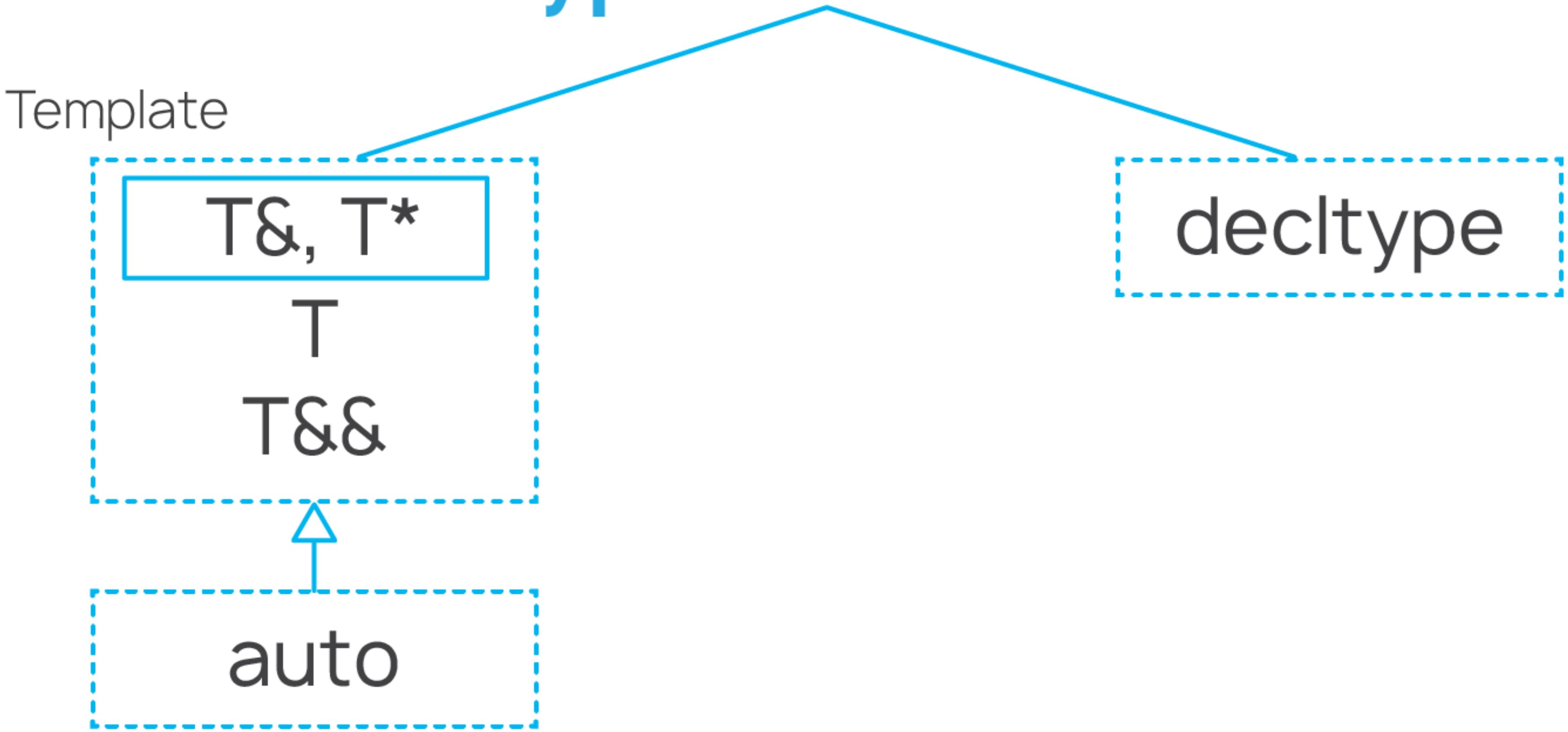
T&, T*

T

T&&

auto

Type deduction



Template

T&, T*

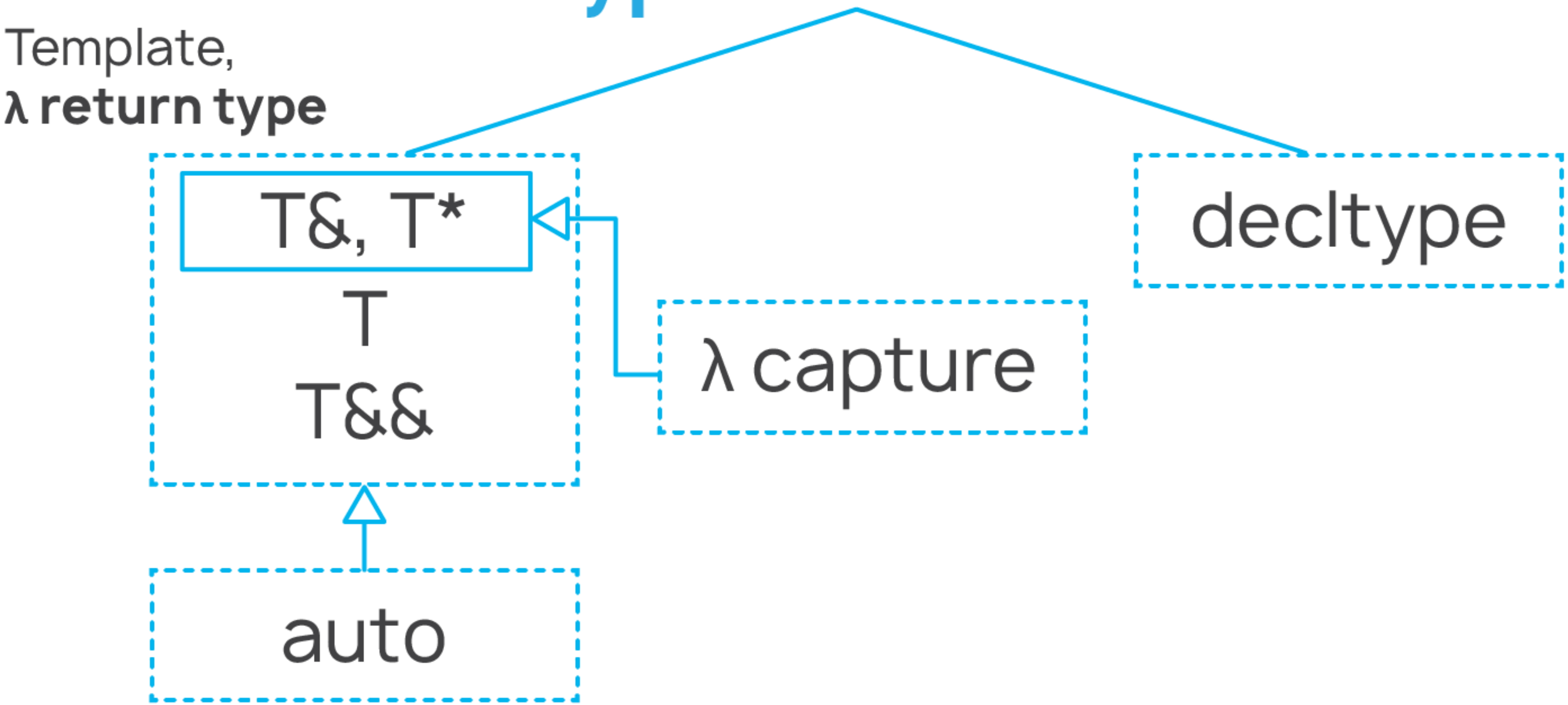
T

T&&

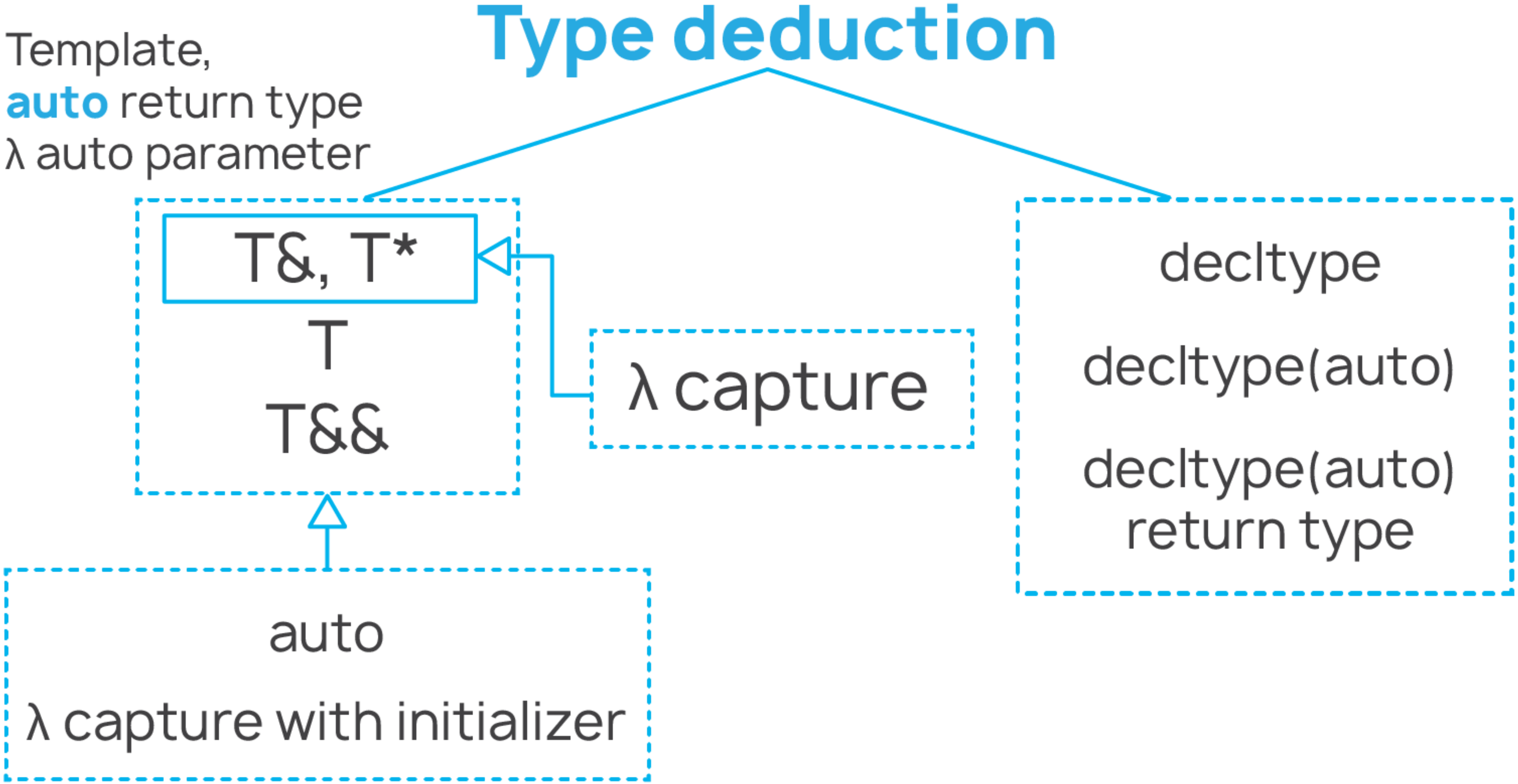
auto

decltype

Type deduction



Type deduction



Template,
auto return type
λ auto parameter

T&, T*

T

T&&

λ capture

decltype

decltype(auto)

decltype(auto)
return type

auto

λ capture with initializer

Template type deduction

```
template <typename T>
void foo(ParamType param) { .... }

foo(expr);
```

- *`T`* – выводимый тип, шаблонный аргумент
- *`ParamType`* – аргумент шаблонной функции, может быть отличен от *`T`* (`const T&`)

Template type deduction for "by-value" parameters

```
template <typename T>
void foo(T param); // param is a type of T

               int  i    = 0; // int
               int &ri   = i; // int&
const          int &rci  = i; // const int&
      volatile int &rvi  = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

foo(ri);   // T ≡ int, param's type ≡ int
foo(rci);  // T ≡ int, param's type ≡ int
foo(rvi);  // T ≡ int, param's type ≡ int
foo(rcvi); // T ≡ int, param's type ≡ int
```

```
template <typename T>
void foo(const T param); // param is a type of const T

               int  i    = 0; // int
               int &ri   = i; // int&
const          int &rci  = i; // const int&
      volatile int &rvi  = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

foo(ri);   // T ≡ int, param's type ≡ const int
foo(rci);  // T ≡ int, param's type ≡ const int
foo(rvi);  // T ≡ int, param's type ≡ const int
foo(rcvi); // T ≡ int, param's type ≡ const int
```

```cpp
template <typename T>
void foo(T param); // param is a type of T

               int   i   = 0;  // int
const          int * pci = &i; // const int*
      volatile int * pvi = &i; // volatile int*

const          int * const             cpci = &i; // const int * const
      volatile int *       volatile   vpvi = &i; // volatile int * volatile
const volatile int * const volatile cvpcvi = &i; // cv int * cv

foo(pci);    // T ≡ const int*, param's type ≡ const int*
foo(pvi);    // T ≡ volatile int*, param's type ≡ volatile int*
foo(cpci);   // T ≡ const int*, param's type ≡ const int*
foo(vpvi);   // T ≡ volatile int*, param's type ≡ volatile int*
foo(cvpcvi); // T ≡ cv int*, param's type ≡ cv int*
```

```
template <typename T>
void foo(T param); // param is a type of T

void bar();
int arr[10]; // int[10]

foo(arr);    // T ≡ int*, param's type ≡ int*
foo(bar);    // T ≡ void (*)(), param's type ≡ void (*)()

foo({ 1, 2, 3 }); // fails to deduce type!
```

Template type deduction for non-forwarding reference and pointer parameters

```
template <typename T>
void foo(T &param); // param is a reference to T

               int i   = 0; // int
const          int ci  = i; // const int
      volatile int vi  = i; // volatile int
const volatile int cvi = i; // const volatile int

foo(i);   // T ≡ int, param's type ≡ int&
foo(ci);  // T ≡ const int, param's type ≡ const int&
foo(vi);  // T ≡ volatile int, param's type ≡ volatile int&
foo(cvi); // T ≡ cv int, param's type ≡ cv int&
```

```
template <typename T>
void foo(T &param); // param is a reference to T

               int  i     = 0; // int
               int &ri    = i; // int&
const          int &rci   = i; // const int&
      volatile int &rvi   = i; // volatile int&
const volatile int &rcvi  = i; // const volatile int&

foo(ri);   // T ≡ int, param's type ≡ int&
foo(rci);  // T ≡ const int, param's type ≡ const int&
foo(rvi);  // T ≡ volatile int, param's type ≡ volatile int&
foo(rcvi); // T ≡ cv int, param's type ≡ cv int&
```

```
template <typename T>
void foo(const T &param); // param is a reference to const T

               int  i     = 0; // int
               int &ri    = i; // int&
const          int &rci   = i; // const int&
      volatile int &rvi   = i; // volatile int&
const volatile int &rcvi  = i; // const volatile int&

foo(ri);   // T ≡ int, param's type ≡ const int&
foo(rci);  // T ≡ int, param's type ≡ const int&
foo(rvi);  // T ≡ volatile int, param's type ≡ cv int&
foo(rcvi); // T ≡ cv int, param's type ≡ cv int&
```

```
template <typename T>
void foo(T *param); // param is a pointer to T

               int  i     = 0;  // int
               int *pi    = &i; // int*
const          int *pci   = &i; // const int*
      volatile int *pvi   = &i; // volatile int*
const volatile int *pcvi  = &i; // const volatile int*

foo(pi);   // T ≡ int, param's type ≡ int*
foo(pci);  // T ≡ const int, param's type ≡ const int*
foo(pvi);  // T ≡ volatile int, param's type ≡ volatile int*
foo(pcvi); // T ≡ cv int, param's type ≡ cv int*
```

```
template <typename T>
void foo(const T *param); // param is a pointer to const T

               int  i     = 0;  // int
               int *pi    = &i; // int*
const          int *pci   = &i; // const int*
      volatile int *pvi   = &i; // volatile int*
const volatile int *pcvi  = &i; // const volatile int*

foo(pi);   // T ≡ int, param's type ≡ const int*
foo(pci);  // T ≡ int, param's type ≡ const int*
foo(pvi);  // T ≡ volatile int, param's type ≡ const volatile int*
foo(pcvi); // T ≡ volatile int, param's type ≡ cv int*
```

```
template <typename T>
void foo(T &param); // param is a reference to T

void bar();
int arr[10]; // int[10]

foo(arr);    // T ≡ int [10], param's type ≡ int (&)[10]
foo(bar);    // T ≡ void (), param's type ≡ void (&)()

foo({ 1, 2, 3 }); // fail to deduce type!
```

Template type deduction for forwarding reference

```
template <typename T>
void foo(T &&param); // param is a forwarding reference to T

               int  i    = 0; // int
               int &ri   = i; // int&
const          int &rci  = i; // const int&
      volatile int &rvi  = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

foo(ri);   // T ≡ int&, param's type ≡ int&
foo(rci);  // T ≡ const int&, param's type ≡ const int&
foo(rvi);  // T ≡ volatile int&, param's type ≡ volatile int&
foo(rcvi); // T ≡ cv int&, param's type ≡ cv int&

foo(42);   // T ≡ int, param's type ≡ int&&
```

```cpp
struct SomeClass { .... };

std::vector<SomeClass> vec;

// 1 redundant move constructor is called
vec.push_back(SomeClass { arg1, arg2, ... });

// construct object in-place using perfect forwarding
vec.emplace_back(arg1, arg2, ...);
```

```
template <class ...Args>
void emplace_back(Args ...args); // 1) very bad
```

```
template <class ...Args>
void emplace_back(Args ...args); // 1) very bad

template <class ...Args>
void emplace_back(Args &...args); // 2) good, but not perfect
```

```
template <class ...Args>
void emplace_back(Args ...args); // 1) very bad

template <class ...Args>
void emplace_back(Args &...args); // 2) good, but not perfect

template <class ...Args>
void emplace_back(Args &&...args) // 3) perfect
{
  T *ptr = ....; // memory region from allocator
  new (ptr) T { std::forward<Args>(args)... };
}
```

```
template <typename T>
constexpr T&& forward(remove_reference_t<T> &arg) noexcept
{
  return static_cast<T&&>(arg);
}

std:string arg2 = "foobar";
vec.emplace_back(std::vector<int> { 0, 1, 2 },
                 arg2,
                 ...);
```

```
template <typename T>
constexpr T&& forward(remove_reference_t<T> &arg) noexcept
{
  return static_cast<T&&>(arg);
}

// void emplace_back<std::vector<int>, std::string &>(
          std::vector<int> &&arg1,
          std::string &arg2,
          ...)

// std::forward<std::vector<int>>(arg1) ≡ std::vector<int> &&
// std::forward<std::string &>(arg2)    ≡ std::string &
```

'auto' type deduction

```
               int  i     = 0; // int
               int &ri    = i; // int&
const          int &rci   = i; // const int&
      volatile int &rvi   = i; // volatile int&
const volatile int &rcvi  = i; // const volatile int&

auto    a_i = i;     // auto ≡ int, a_i's type ≡ int
auto   a_ri = ri;    // auto ≡ int, a_ri's type ≡ int
auto  a_rci = rci;   // auto ≡ int, a_rci's type ≡ int
auto  a_rvi = rvi;   // auto ≡ int, a_rvi's type ≡ int
auto a_rcvi = rcvi;  // auto ≡ int, a_rcvi's type ≡ int
```

```
               int  i    = 0; // int
               int &ri   = i; // int&
const          int &rci  = i; // const int&
      volatile int &rvi  = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

               auto a_i      = i;    // auto ≡ int, a_i's type ≡ int
const          auto ca_ri    = ri;   // auto ≡ int, ca_ri's type ≡ const int
      volatile auto va_rvi   = rvi;  // auto ≡ int, va_rvi's type ≡ volatile int
const volatile auto cva_rcvi = rcvi; // auto ≡ int, cva_rcvi's type ≡ cv int
```

```
               int  i    = 0; // int
               int &ri   = i; // int&
const          int &rci  = i; // const int&
      volatile int &rvi  = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

auto &a_i    = i;    // auto ≡ int, a_i's type ≡ int&
auto &a_ri   = ri;   // auto ≡ int, a_ri's type ≡ int&
auto &a_rci  = rci;  // auto ≡ const int, a_rci's type ≡ const int&
auto &a_rvi  = rvi;  // auto ≡ volatile int, a_rvi's type ≡ volatile int&
auto &a_rcvi = rcvi; // auto ≡ cv int, a_rcvi's type ≡ cv int&
```

```
               int  i    = 0; // int
               int &ri   = i; // int&
const          int &rci  = i; // const int&
      volatile int &rvi  = i; // volatile int&
const volatile int &rcvi = i; // const volatile int&

               auto &a_i      = i;    // auto ≡ int, a_i's type ≡ int&
const          auto &ca_rci   = rci;  // auto ≡ int, ca_rci's type ≡ const int&
      volatile auto &va_rvi   = rvi;  // auto ≡ int, va_rvi's type ≡ volatile int&
const volatile auto &cva_rcvi = rcvi; // auto ≡ int, cva_rcvi's type ≡ cv int&
```

```
int foo();
int&& bar();

int  i    = 0;  // int
int &ri   = i;  // int&
int &&rri = 42; // int&&

auto &&a_i   = i;    // auto ≡ int&, a_i's type ≡ int&
auto &&a_ri  = ri;   // auto ≡ int&, a_ri's type ≡ int&

auto &&a_foo = foo(); // auto ≡ int, a_foo's type ≡ int&&
auto &&a_bar = bar(); // auto ≡ int, a_bar's type ≡ int&&
void bar();
int arr[10]; // int[10]

auto &rarr = arr; // auto ≡ int [10], rarr's type ≡ int (&)[10]
auto &rbar = bar; // auto ≡ void (), rbar's type ≡ void (&)()

auto parr = arr; // auto ≡ int*, parr's type ≡ int*
auto pbar = bar; // auto ≡ void (*)(), abar's type ≡ void (*)()

auto init_list1 { 1, 2, 3 };   // auto ≡ std::initializer_list<int>
auto init_list2 = { 1, 2, 3 }; // auto ≡ std::initializer_list<int>

auto err_init_list = { 1, 2.0, 3 }; // fails to deduce type
```

λ capture type deduction

```
[lambda-capture](params...) -> return_type
{
  lambda-body
}
```

```
lambda-capture ::= '=', '&', 'this',
                   identifier,
                   identifier initializer, // since C++14
                  &identifier,
                  &identifier initializer, // since C++14
```

```
const int cx = 42;

auto lambda = [cx] { .... }; // [=] { .... }
```

```
class LambdaCompilerRepresentation
{
  const int cx;

public:
  auto operator()() const { .... };
};
```

```
int x = 42;

auto lambda = [x] { x = 0; }; // fails to compile
```

```
class LambdaCompilerRepresentation
{
  int x;

public:
  auto operator()() const { x = 0; };
};
```

```
int x = 42;

auto lambda = [x]() mutable { x = 0; }; // ok
```

```
class LambdaCompilerRepresentation
{
  int x;

public:
  auto operator()() { x = 0; };
};
```

```
const int x = 42;

auto lambda = [x]() mutable { x = 0; }; // fails to compile
```

```
class LambdaCompilerRepresentation
{
  const int x;

public:
  auto operator()() { x = 0 };
};
```

```
int x = 42;

auto lambda = [&x]() { x = 0; }; // ok
```

```
class LambdaCompilerRepresentation
{
  int &x;
public:
  auto operator()() const { x = 0; };
};
```

```
const int x = 42;

auto lambda = [&x]() { x = 0; }; // fails to compile
```

```
class LambdaCompilerRepresentation
{
  const int &x;
public:
  auto operator()() const { x = 0; };
};
```

```
#include <memory>

class SomeClass { .... };
auto p = std::make_unique<SomeClass>(....); // std::unique_ptr<SomeClass>

auto lambda = [p = std::move(p)]() { p->....; }; // ok


class LambdaCompilerRepresentation
{
  std::unique_ptr<SomeClass> p;
public:
  auto operator()() const { p->....; };
};
```

```
int x = 42;

auto lambda = [&rx = x]() { rx = 0; }; // ok
```

```
class LambdaCompilerRepresentation
{
  int &rx;
public:
  auto operator()() { rx = 0; };
};
```

'decltype' type deduction

```
int   foo();
int&& bar();

int arr[10];

      int   v1  = 0.0; // int
const int  &v2  = v1;  // const int&
      int &&v3  = 0;   // int&&

decltype(v1)   v4 = v1;   // int
decltype((v1)) v5 = (v1); // int&
decltype(v2)   v6 = v2;   // const int&

decltype(foo()) v7 = foo(); // int
decltype(bar()) v8 = bar(); // int&&

decltype(foo)  v9 = foo; // int   ()(), compile-time error
decltype(bar) v10 = bar; // int&& ()(), compile-time error

decltype(arr[0]) v11 = arr[0]; // int&
```

```
int   foo();
int&& bar();

int arr[10];

      int   v1  = 0.0; // int
const int  &v2  = v1;  // const int&
      int &&v3  = 0;   // int&&

decltype(auto) v4 = v1;   // int
decltype(auto) v5 = (v1); // int&
decltype(auto) v6 = v2;   // const int &

decltype(auto) v7 = foo(); // int
decltype(auto) v8 = bar(); // int&&

decltype(auto)  v9 = foo(); // int   ()(), compile-time error
decltype(auto) v10 = bar(); // int&& ()(), compile-time error

decltype(auto) v11 = arr[0]; // int&
```

Function return type deduction

```
[capture-list](params) -> T // use template argument deduction
{
  ....;
  return ....;
}

auto foo() -> T // use template argument deduction
{
  ....
  return ....;
}

decltype(auto) bar() // use decltype(auto) type deduction
{
  ....
  return ....;
}
```

```
template <typename T1, typename T2>
auto operator+(T1 &&lhs, T2 &&rhs)
{
  return std::forward<T1>(lhs) + std::forward<T2>(rhs);
}
```

```
template <typename Callable, typename ...Args>
auto call(Callable &&op, Args &&...args) // can't return reference
{
  return std::forward<Callable>(op)(std::forward<Args>(args)...);
}
```

```
template <typename Callable, typename ...Args>
auto call(Callable &&op, Args &&...args) // can't return reference
{
  return std::forward<Callable>(op)(std::forward<Args>(args)...);
}
```

```
template <typename Callable, typename ...Args>
auto&& call(Callable &&op, Args &&...args) // possible dangling reference!!!
{
  return std::forward<Callable>(op)(std::forward<Args>(args)...);
}
```

```cpp
template <typename Callable, typename ...Args>
auto call(Callable &&op, Args &&...args) // can't return reference
{
  return std::forward<Callable>(op)(std::forward<Args>(args)...);
}

template <typename Callable, typename ...Args>
auto&& call(Callable &&op, Args &&...args) // possible dangling reference!!!
{
  return std::forward<Callable>(op)(std::forward<Args>(args)...);
}

template <typename Callable, typename ...Args>
decltype(auto) call(Callable &&op, Args &&...args) // perfectly return
{
  return std::forward<Callable>(op)(std::forward<Args>(args)...);
}
```

```
template <typename T>
decltype(auto) lookup(T value)
{
  static const std::vector<SomeClass> values = ....;

  size_t idx = ....; // calculate index based on the value

  auto ret = values[idx]; // SomeClass object
  return ret; // return type is SomeClass
}
```

```
template <typename T>
decltype(auto) lookup(T value)
{
  static const std::vector<SomeClass> values = ....;

  size_t idx = ....; // calculate index based on the value

  auto ret = values[idx]; // SomeClass object
  return (ret); // return type is SomeClass&!!!
}
```

How to find out deduced type?

```
template <typename T, typename ...Types>
class TP; // type printer

template <typename T>
void foo(const T &t)
{
  TP<T, decltype(t)> _;
}

class SomeClass { .... };

SomeClass obj;
foo(obj);
```

Clang:

```
<source>:10:24: error: implicit instantiation of undefined template 'TP<SomeClass,
const SomeClass &>'
  TP<T, decltype(arg)> _;
```

GCC:

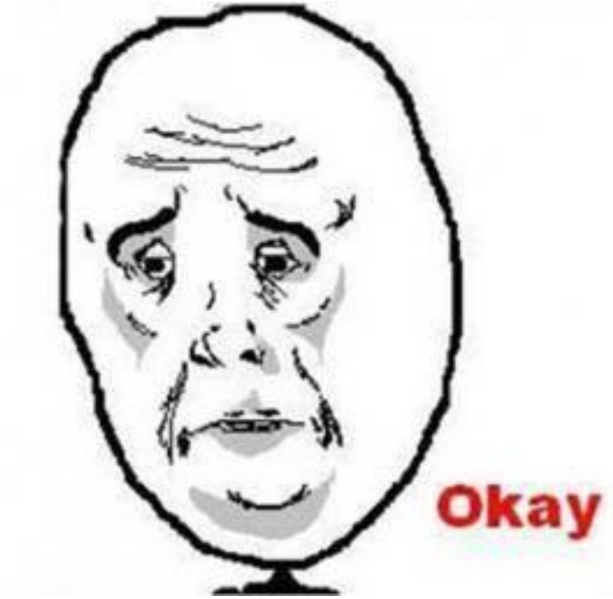
```
<source>:10:24: error: 'TP<SomeClass, const SomeClass&> _' has incomplete type
   10 |   TP<T, decltype(arg)> _;
```

MSVC:

```
<source>(10): error C2079: '_' uses undefined class 'TP<T, const T &>'
        with
        [
            T=SomeClass
        ]
```

```
template <typename T>
void print_type_to_cout(const T &arg)
{
  std::cout << "T = " << typeid(T).name() << '\n';
  std::cout << "arg = " << typeid(arg).name() << '\n';
}

class SomeClass { .... };

void foo()
{
  std::vector<SomeClass> vec { .... };
  print_type_to_cout(vec.data());
}
```

```
// Expectation:
//     T = SomeClass *
//   arg = SomeClass * const &

// Real life:
//     T = P9SomeClass, demangle – SomeClass*
//   arg = P9SomeClass, demangle - SomeClass*
// Expectation:
//      T = SomeClass *
//    arg = SomeClass * const &

// Real life:
//      T = P9SomeClass, demangle – SomeClass*
//    arg = P9SomeClass, demangle - SomeClass*
```



```
template <typename T>
void foo(const T &arg)
{
  using namespace boost::typeindex;
  std::cout << "T = "
            << type_id_with_cvr<T>().pretty_name()
            << '\n';

  std::cout << "arg = "
            << type_id_with_cvr<decltype(arg)>().pretty_name()
            << '\n';
}
```

```
// Expectation:
//     T = SomeClass *
//   arg = SomeClass * const &

// Real life:
//     T = SomeClass *
//   arg = SomeClass * const &
// Expectation:
//      T = SomeClass *
//    arg = SomeClass * const &

// Real life:
//      T = SomeClass *
//    arg = SomeClass * const &
```

END

Q&A