Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 4 – Type Deduction

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 14.03.2019
FS2019

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR SOFTWARE

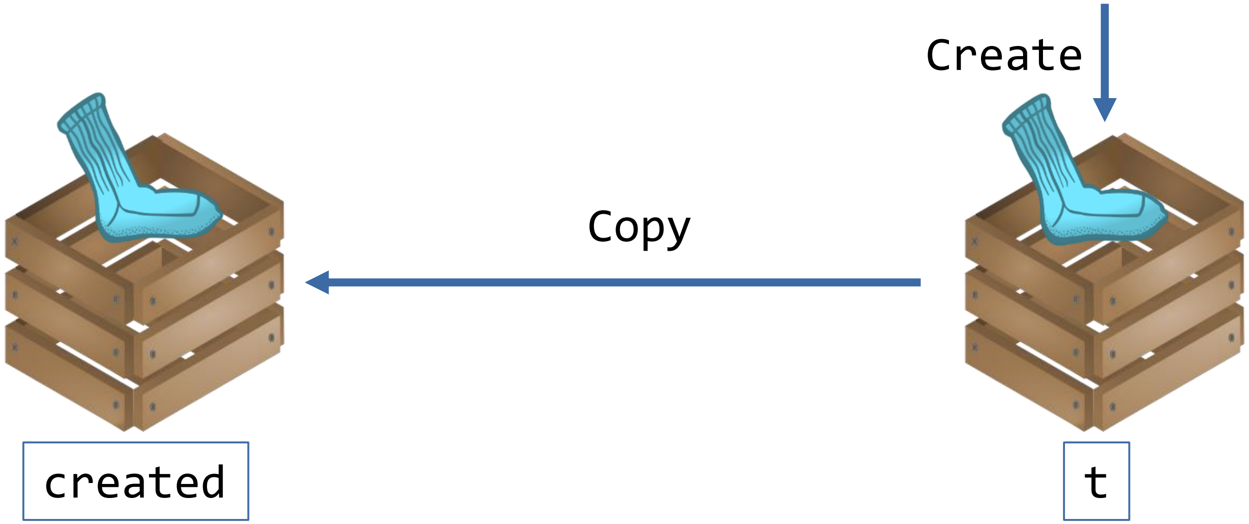
- **Topics:**
  - Recap: Move Semantics
  - Overload Resolution
  - Type Deduction

Recap: Move Semantics

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

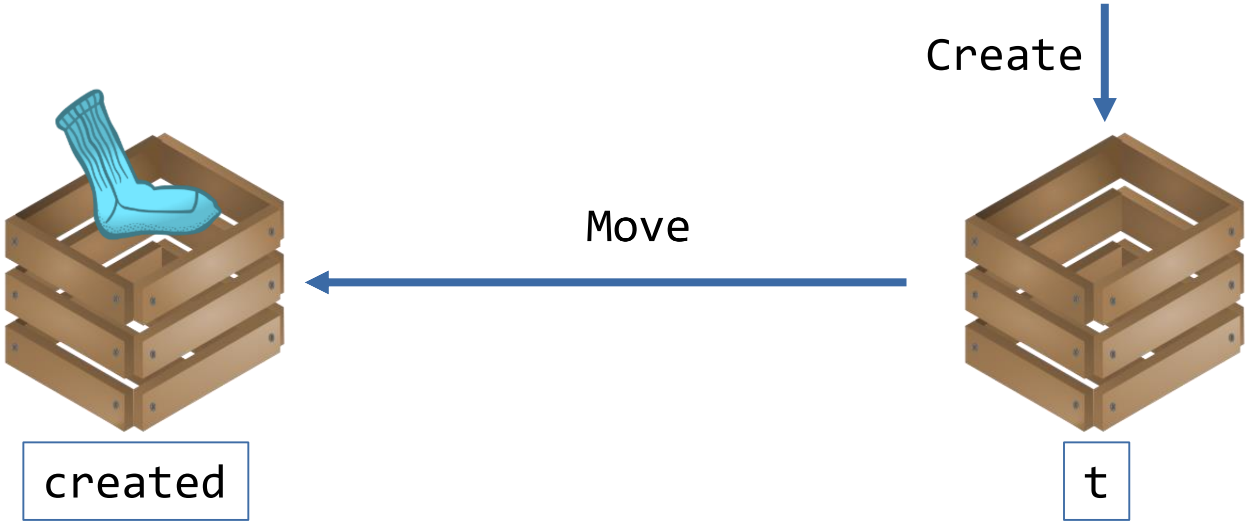



```
Create Thing
Copy Thing
```

```cpp
struct CopyableThing {
  CopyableThing() {
    std::cout << "Create Thing\n";
  }
  CopyableThing(CopyableThing const &) {
    std::cout << "Copy Thing\n";
  }
};

CopyableThing create() {
  CopyableThing t{};
  return t;
}

int main() {
  CopyableThing created = create();
}
```

- **Compile in GCC with –`fno-elide-constructors`**
- **Pre C++17: One additional copy would happen without optimization**

Moving Elements




```
Create Thing
Move Thing
```

```cpp
struct MoveableThing {
  MoveableThing() {
    std::cout << "Create Thing\n";
  }

  MoveableThing(MoveableThing &&) {
    std::cout << "Move Thing\n";
  }
};

MoveableThing create() {
  MoveableThing t{};
  return t;
}

int main() {
  MoveableThing created = create();
}
```

- **Compile in GCC with `-fno_elide_constructors`**
- **Pre C++17: One additional move would happen without optimization**

Eliding Copies and Moves

Create

created/t

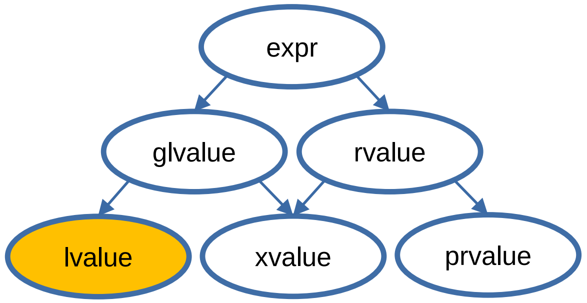
```
Create Thing
```

```
struct Thing {
  Thing() {
    std::cout << "Create Thing\n";
  }

  Thing(Thing const &) {
    std::cout << "Copy Thing\n";
  }

  Thing(Thing &&) {
    std::cout << "Move Thing\n";
  }
};

Thing create() {
  Thing t{};
  return t;
}

int main() {
  Thing created = create();
}
```

Example Move Constructor


```
struct ContainerForBigObject {
  ContainerForBigObject()
      : resource{std::make_unique<BigObject>()} {}

  ContainerForBigObject(ContainerForBigObject const & other)
      : resource{std::make_unique<BigObject>(*other.resource)} {}

  ContainerForBigObject(ContainerForBigObject && other)
      : resource{std::move(other.resource)} {}

  ContainerForBigObject & operator=(ContainerForBigObject const & other) {
    resource = std::make_unique<BigObject>(*other.resource);
    return *this;
  }

  ContainerForBigObject & operator=(ContainerForBigObject && other) {
    std::swap(resource, other.resource);
    //resource = std::move(other.resource) is possible too
    return *this;
  }

private:
  std::unique_ptr<BigObject> resource;
};
```

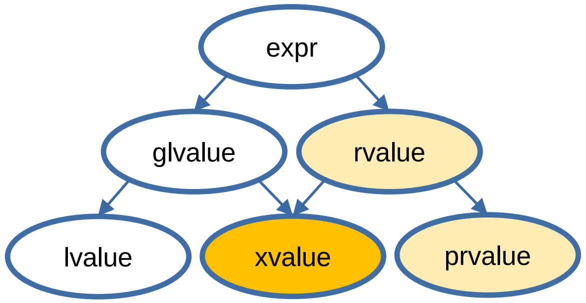
- **An lvalue Reference is an alias for a variable**
  - Syntax: T&
  - The original must exist as long as it is referred to!



- **Can be used as**
  - Function parameter type (most useful: no copy and side-effect on argument possible)
  - Member or local variable (barely useful)
  - Return type (Must survive!)

```
void increment(int & i) {
  ++i; // side-effect on argument
}
```

- Beware of dangling references: undefined behavior!



- **References for rvalues**
  - Syntax: T&&
  - Binds to an rvalue (xvalue or prvalue)
- **Argument is either a literal or a temporary object**

```
std::string createGlass();

void fancy_name_for_function() {
  std::string mug{"cup of coffee"};
  std::string && glass_ref = createGlass(); //life-extension of temporary
  std::string && mug_ref = std::move(mug);  //explicit conversion lvalue to xvalue
  int && i_ref = 5;                          //binding rvalue reference to prvalue
}
```

- **Beware: Parameters and variables declared as rvalue references are lvalues in the context of function bodies! (Everything with a name is an lvalue)**
- **Beware 2.0: `T&&/auto&&` is not always an rvalue reference! (We'll come to that later)**

Where to Put the Ampersand?

Opinion


```
int&ref    //1
int& ref   //2
int &ref   //3
int & ref  //4
```

1. **No spaces at all**
   - Makes visual distinction of the individual parts a bit difficult
   - Possible reason: You need to stay within 80 characters wide lines (valid pre 2000)
2. **Attached to the type**
   - If you consider the reference as belonging to the type
   - Seems logical as it groups the all parts of the type and separates the declarator (name of the variable)
   - Might be misleading when declaring multiple variables in a single declaration!

```
int& ref, is_this_a_ref_too;
```

Misleading Structure

```
int&ref   //1
int& ref  //2
int &ref  //3
int & ref //4
```

3. **Attached to the declarator**
   - Is consistent with the way the C++ grammar is structured. The & is part of the declarator (`ref`) not the decl specifier (`int`)
   - Avoids confusion when declaring multiple variables in a single declaration
4. **Space before and after the ampersand**
   - If you cannot make up your mind

- **Eventually, you should adhere the coding guidelines of your project!**

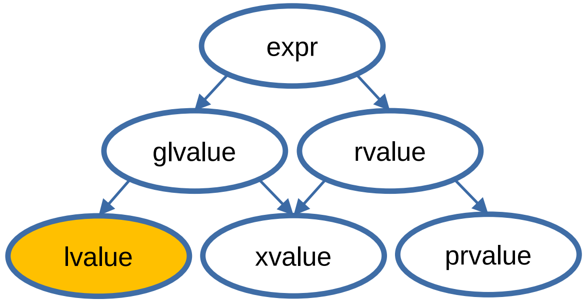
Overload Resolution

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

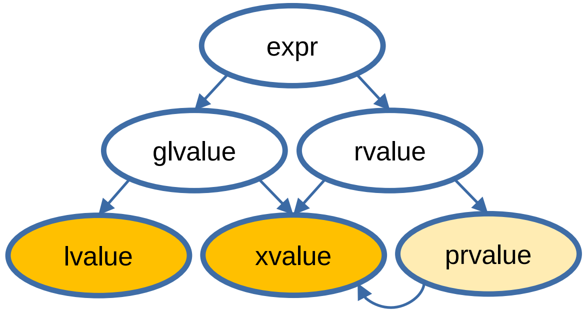
IFS INSTITUTE FOR SOFTWARE

- **lvalue Reference**
  - binds



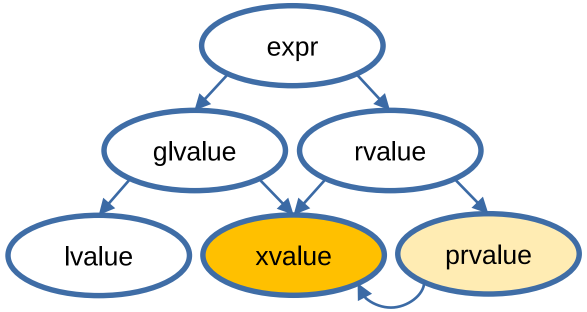

```
Type t{};
Type & tRef = t;
```

- **const lvalue Reference**
  - binds




```
Type t{};
Type const & tRef = t;
Type const & tMoved = std::move(t);
Type const & tTemp = Type{};
```

- **rvalue Reference**
  - binds




```
Type t{};
Type && tTemp = Type{};
Type && tMoved = std::move(t);
```

Overload Resolution for Free Functions

	`f(S)`	`f(S &)`	`f(S const &)`	`f(S &&)`
`S s{};` `f(s);`	✓	✓ (preferred over const &)	✓	✗
`S const s{};` `f(s);`	✓	✗	✓	✗
`f(S{});`	✓	✗	✓	✓ (preferred over const &)
`S s{};` `f(std::move(s));`	✓	✗	✓	✓ (preferred over const &)

- **The overload for value parameters imposes ambiguities.**
- **For deciding between two lvalue reference overloads (const and non-const) the constness of the argument is considered.**

Overload Resolution for Member Functions

	`S::m()`	`S::m() const`	`S::m() &`	`S::m() const &`	`S::m() &&`
`S s{}; s.m();`	✓	✓	✓ (preferred over const &)	✓	✗
`S const s{}; s.m();`	✗	✓	✗	✓	✗
`S{}.m();`	✓	✓	✗	✓	✓ (preferred over const &)
`S s{}; std::move(s).m();`	✓	✓	✗	✓	✓ (preferred over const &)

- **Reference and non-reference overloads cannot be mixed!**
- **The reference qualifier affects the `this` object and the overload resolution**
- **`const &&` would theoretically be possible, but it is an artificial case**

Forwarding References

- **In some contexts `T&&` does not necessarily mean rvalue reference**
- **Exceptions**
  - `auto &&`
  - `T &&` when template type deduction applies for type T
- **In these cases the reference can bind to rvalues and lvalues depending on the context**

```
template<typename T>
void f(T && param);
```

```
int x = 23;
f(x);          //lvalue
```

↓

```
void f(int & param);
```

```
f(23); //rvalue
```

↓

```
void f(int && param);
```

Type deduction

Based on Modern Effective C++ by Scott Meyers

Cevelop

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

```
template<typename T>
void f(ParamType param);
```

- **`T` and `ParamType` are not necessarily exactly the same type**

```
template<typename T>
void f(T const & param);
```

- **Now what is `T` and `ParamType` for the following call?**

```
int x = 0;
f(x);
```

- **T: `int`**
- **ParamType: `int const &`**

Rules for Type Deduction

- **Context:**

```
template<typename T>
void f(ParamType param);
```

- **Deduction of type `T` depends on the structure of `ParamType`**
- **Cases:**

1. ParamType is a value type (e.g. `void f(T param)`)
2. ParamType is pointer type (e.g. `void f(T * param)`)
3. ParamType is a reference (e.g. `void f(T & param)`)
4. ParamType is a forwarding reference (exactly: `void f(T && param)`)

Note: `ParamType` might be a nested composition of templates (e.g. `void f(std::vector<T> param)`)

- **ParamType is a value type**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Ignore `const` of `<expr>` (outermost)
  3. Pattern match `<expr>`'s type against ParamType to figure out T

```
template<typename T>
void f(T param);
```

```
f(<expr>);
```

Declarations:

```
int           x   = 23;
int const     cx  = x;
int const &   crx = x;
```

Calls:	Instances:	Deduced Ts:
`f(x);`	`void f(int param);`	`T = int`
`f(cx);`	`void f(int param);`	`T = int`
`f(crx);`	`void f(int param);`	`T = int`

- **ParamType is a value type**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Ignore `const` of `<expr>` (outermost)
  3. Pattern match `<expr>`'s type against ParamType to figure out T
- **Example const pointer to const int**

```
template<typename T>
void f(T param);
```

```
f(<expr>);
```

Call:

```
char const * const ptr = "...";
f(ptr);
```

Instance:

```
void f(char const * param);
```

Deduced `T`:

```
T = char const *
```

- **ParamType is a pointer type**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Ignore `const` of `<expr>` (outermost)
  3. Pattern match `<expr>`'s type against ParamType to figure out T
- **Examples for Pointers:**

```
template<typename T>
void f(T * param);
```

```
f(<expr>);
```

Declarations:

```
int           x  = 23;
int const *   px = &x;
```

Calls:		Instances:		Deduced Ts:
`f(&x);`	➡	`void f(int * param);`	➡	`T = int`
`f(px);`	➡	`void f(int const * param);`	➡	`T = int const`

Rules for Type Deduction: Reference Type

- **ParamType is a reference type, but not a forwarding reference**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Pattern match `<expr>`'s type against ParamType to figure out T
- **Examples for References:**

```
template<typename T>
void f(T & param);
```

```
f(<expr>);
```

Declarations:

```
int           x   = 23;
int const     cx  = x;
int const &   crx = x;
```

Calls:		Instances:		Deduced Ts:
`f(x);`	➡	`void f(int & param);`	➡	`T = int`
`f(cx);`	➡	`void f(int const & param);`	➡	`T = int const`
`f(crx);`	➡	`void f(int const & param);`	➡	`T = int const`

Rules for Type Deduction: Reference Type (const)

- **ParamType is a reference type, but not a forwarding reference**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Pattern match `<expr>`'s type against ParamType to figure out T

```
template<typename T>
void f(T const & param);
```

```
f(<expr>);
```

- **Examples for Const References:**

Declarations:

```
int           x   = 23;
int const     cx  = x;
int const &   crx = x;
```

Calls:		Instances:		Deduced Ts:
`f(x);`	➡	`void f(int const & param);`	➡	`T = int`
`f(cx);`	➡	`void f(int const & param);`	➡	`T = int`
`f(crx);`	➡	`void f(int const & param);`	➡	`T = int`

Rules for Type Deduction: Forwarding Reference Type

- **ParamType is a forwarding reference**
- **Cases:**
  1. `<expr>` is an lvalue: T and ParamType become lvalue references!
  2. Otherwise (if `<expr>` is an rvalue): Rules for references apply

```
template<typename T>
void f(T && param);
```

```
f(<expr>);
```

Declarations:

```
int            x   = 23;
int const      cx  = x;
int const &    crx = x;
```

Calls:		Instances:		Deduced Ts:
`f(x);`	➡	`void f(int & param);`	➡	`T = int &`
`f(cx);`	➡	`void f(int const & param);`	➡	`T = int const &`
`f(crx);`	➡	`void f(int const & param);`	➡	`T = int const &`
`f(27);`	➡	`void f(int && param);`	➡	`T = int`

- **What happens if `initializer_lists` are used for template type deduction?**
  - It does not work!

```
template<typename T>
void f(T param);
```

```
f({23});        //error
```

- **Correct way:**

```
template<typename T>
void f(std::initializer_list<T> param);
```

```
f({23});        //T = int
                //ParamType = std::initializer_list<T>
```

Keywords auto and decltype

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

- **Essentially type deduction for auto is the same as we have seen before**
- **auto takes the place of ParamType**

```
auto x = 23;                //auto is a value type
auto const cx = x;          //auto is a value type
auto & rx = x;              //auto is a reference type

auto && uref1 = x;          //x is an lvalue, uref1 is int &
auto && uref2 = cx;         //cx is an lvalue, uref2 is int const &
auto && uref3 = 23;         //23 is an rvalue, uref3 is int &&
```

- **Special case**

```
auto init_list1 = {23};     //std::initializer_list<int>
auto init_list2{23};        //int, was std::initializer_list<int>1
auto init_list3{23, 23};    //Error, requires one single argument
```

[1]Fixed in C++17 (N3922) – Some compiler vendors have retroactively applied this fix to earlier C++ versions

- **Since C++14 it is possible to use `auto` as return type and `auto` for parameter declarations in lambdas and functions**
  - Body must be available to deduce the type
- **Rules of these uses of `auto` follow the rules of template type deduction**

```
auto createInitList() {
  return {1, 2, 3};
}
```

```
[](auto p) {
  ...
}
```

```
auto createInt() {
  return 23;
}
```

```
void f(auto p) {
  ...
}
```

This is a GCC Extension, not a Standard C++ Feature
Will be available with Concepts (C++20)

- **Since C++11 there is the `decltype` keyword**
- **`decltype` can be applied to an expression: `decltype(x)`**
  - Represents the declared type of a name expression
  - Since C++14: `decltype(auto)` deduces the type, but does not strip references like `auto`

```
int            x       = 23;
int const      cx      = x;
decltype(cx)   cx_too  = cx; //type of cx_too is int const
int &          rx      = x;
decltype(rx)   rx_too  = rx; //type of rx_too is int &

auto           just_x  = rx; //type of just_x is int
decltype(auto) more_rx = rx; //type of more_rx is int &
```

- **This works the same for return types `auto` and `decltype(auto)`**

Rules for decltype

- **Unparenthesized variable name or data member**
  - T - Type of the expression (retains reference)
- **Expression of value category xvalue**
  - T&& - Rvalue reference to type of the expression
- **Expression of value category lvalue**
  - T& - Lvalue reference to type of the expression
- **Expression of value category prvalue**
  - T – Value type of the expression

```
decltype(auto) funcName() {
  int local = 42;
  return local; //decltype(local) => int
}
decltype(auto) funcNameRef() {
  int local = 42;
  int & lref = local;
  return lref; //decltype(lref) => int &
}
decltype(auto) funcXvalue() {
  int local = 42;
  return std::move(local); //int && -> bad
}
decltype(auto) funcLvalue() {
  int local = 42;
  return (local); //int & -> bad
}
decltype(auto) funcPrvalue() {
  return 5;  //int
}
```

- **`decltype(auto)` allows deduction of inline function return types**

```
template<typename Container, typename Index>
decltype(auto) access(Container & c, Index i) {
  return c[i];
}
```

- **`decltype` can take an expression depending on parameters for specifying trailing return types**

```
template<typename Container, typename Index>
auto access(Container & c, Index i) -> decltype(c[i]) {
  return c[i];
}
```

Type Deduction for decltype: Beware!

- **decltype(expr) for non-name lvalue expressions returns an lvalue reference**
  - Check the difference:

```cpp
decltype(auto) f1() {
  int x = 23;
  return x;
}
```

→

```cpp
int f1() {
  int x = 23;
  return x;
}
```

```cpp
decltype(auto) f1() {
  int x = 23;
  return (x);
}
```

→

```cpp
int & f1() {
  int x = 23;
  return (x);
}
```

Deduction in Lambdas

Cevelop

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

- **What do you think about this code snippet?**

```
int i0 = 42;
auto missingMutable = [i0] {return i0++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() const {
    return i0++;
  }
  int i0;
};
```

- **The code won't compile as the generated operator is `const`**

- **How about now?**

```
int i1 = 42;
auto everyThingIsOk = [i1] () mutable {return i1++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() {
    return i1++;
  }
  int i1;
};
```

- **The code will compile as the generated operator is not `const`**

- **How about now?**

```
int const i2 = 42;
auto surprise = [i2] () mutable {return i2++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() {
    return i2++;
  }
  int const i2;
};
```

- **The code won't compile since `i2` is `const`**

- **How about now?**

```
int const i3 = 42;
auto srslyWhy = [i3 = i3] () mutable {return i3++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() {
    return i3++;
  }
  int i3;
};
```

- **The init capture is deduced as if it was `auto`**

```
template<typename T>
void f(ParamType param);
```

- **ParamType is a value/pointer type**
  - `<expr>` is a reference type: ignore the reference
  - Ignore `const` of `<expr>` type (outermost)
  - Pattern match `<expr>`'s type against ParamType to figure out T
- **ParamType is a reference/pointer type**
  - `<expr>` is a reference type: ignore the reference
  - Pattern match `<expr>`'s type against ParamType to figure out T
- **ParamType is a forwarding reference (T&&/auto&&)**
  - <expr> is an lvalue: T and ParamType become lvalue references!
  - Otherwise (if `<expr>` is an rvalue): Rules for pointer/references apply