Removing references when storing values

Passing template function arguments by value/reference

- How the arguments are passed in a call to a generic function?

```
std::string a1; std::string & a2 = a1; const std::string & a3 = a1;
f(a1); f(a2); f(a3); f(std::move(a1));
```

  - It depends on the declaration of the function
    - Presence of reference type in the declaration of a2/a3 does NOT matter

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

→ vždycky to bude hodnotou...

  - In this case, x is always passed by value

  - If you want pass by reference, always use a “forwarding reference”

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

→ 4. příklad by byl r-value, ostatní jen l-value

    - “Reference collapsing rules” ensure adaptation to both lvalues and rvalues

- většinou to všechno chci, pokud to jde, předávat jako reference

  - Rationale: plain references cannot adapt to some actual arguments:

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

    - In these cases, the function cannot be called with an rvalue argument:

```
f(a1+".txt"); f(std::move(a1));
```

→ tyhle nebudou fungovat

```
std::vector<bool> k(100); f(k[0]);   // k[0] is an object simulating bool&
```

r-value je většinou výsledek výpočtu, občas tedy opravdu chci předat jenom l-value

tohle je špek, vrací to jako r-value.

Example – storing values of any type

- Goal: Hand-made functor corresponding to the following lambda

```
[p](T & x){ x += p; }
```

- Naive approach

```
template< typename T> class ftor {
public:
```

tohle není forwarding reference, protože neumím ovlivnit, co to bude, za překladu

```
  ftor( T && p) : p_( std::forward< T>( p)) {}
  void operator()( T & x) { x += p_; }
private:
  T p_;
};
```

tohle je ale const char*, r-value

  - Does not work well

```
auto f1 = ftor< std::string>( "Hello");        // works, passed by moving
std::string s = "Hello";
```

→ tohle je l-value

```
auto f2 = ftor< std::string>( s);     // does not work: can't bind rvalue reference p
```

Example – storing values of any type

- A better implementation

```
template< typename T> class ftor {
public:
```

vlastní templatovaný konstruktor, aby se to mohlo přizpůsobit proměnné constantou

```
  template< typename T2> ftor( T2 && p) : p_( std::forward< T2>( p)) {}
  void operator()( T & x) { x += p_; }
private:
  T p_;
};
```

  - Everything works

```
auto f1 = ftor< std::string>( "Hello");        // passed as const char * && to conversion
std::string s = "Hello";
auto f2 = ftor< std::string>( s);      // passed as std::string & to copy-ctor
auto f3 = ftor< std::string>( std::move( s)); // passed as std::string && to move-ctor
```

  - But why the user needs to specify std::string explicitly?

```
auto f2 = make_ftor( s);
```

Example – storing values of any type

- A class

```
template< typename T> class ftor { /*...*/ T p_; };
```

- and its wrapper function (naive attempt)

```
template< typename T>
inline ftor<T> make_ftor( T && p)     // must use forwarding reference
{ return ftor<T>( std::forward< T>(p));
}
```

  - This implementation is wrong and dangerous!

```
std::string s = "Hello";
std::for_each( b, e, make_ftor( s)); // stores std::string & - works faster, but...
```

-> tohle je l-value, takže všechny T jsou l-value string

```
std::vector< std::string> v = { "Hello" };
auto f = make_ftor( v.back());                   // stores std::string &
v.pop_back();
std::for_each( b, e, f);                         // crash!!!
```

—> tady jsem zahodil referenci, která je v tom f-ku.

  - Such implementation may be useful, but users must know that it may store by reference

Konvence: pokud chci schovat referenci, mám dávat pointer, nikoliv referenci.
<-> s tou můžu data jen prohlížet.

Example – storing values of any type

- A class

```
template< typename T> class ftor { /*...*/ T p_; };
```

- and its correct wrapper function

```
template< typename T>
inline ftor<std::remove_cvref_t<T>> make_ftor( T && p)
{ return ftor<std::remove_cvref_t<T>>( std::forward< T>(p));
}
```

=> kdybych předal const ref, tak tohle mi to const a ref vezme.

<=> je mi jedno, že to bylo const, protože my si udělali kopii.

- Shorter syntax

```
template< typename T>
inline ftor<std::remove_cvref_t<T>> make_ftor( T && p)
{ return { std::forward< T>(p)};                    // if ctor is not explicit, {} may be omitted
}
```

- C++14: auto with return values

```
template< typename T>
inline auto make_ftor( T && p)
{ return ftor<std::remove_cvref_t<T>>( std::forward< T>(p));
}
```

Useful standard library type traits

```
#include <type_traits>
```

- Type properties
  - is_void_v, is_enum_v, is_pointer_v, is_const_v, is_abstract_v, is_copy_assignable_v, …
  - Logically: compile-time functions returning bool parametrized by a type
  - Technically: constexpr bool variable templates parametrized by a type
    - xxx_v<T> is a shortcut for xxx<T>::value
  - Usage:

```
template< typename T> struct example {
  static constexpr bool v = std::is_reference_v<T>;
};
```

- Type transformations
  - remove_reference_t, remove_cv_t, remove_cvref_t [C++20]
  - Logically: compile-time functions returning type parametrized by a type
  - Technically: type alias (using) templates parametrized by a type
    - xxx_t<T> is a shortcut for typename xxx<T>::type
  - Usage:

```
template< typename T> struct example {
  using U = std::remove_reference_t<T>;
};
```

- More complex functionality
  - is_same_v, is_convertible_v, make_signed_t, conditional_t …

Type traits – possible implementation

- Type traits – master definition

```
template< typename T> struct is_reference {
  static constexpr bool value = false;
};
```

- Type traits – partial specializations

```
template< typename U> struct is_reference< U &> {
  static constexpr bool value = true;
};
template< typename U> struct is_reference< U &&> {
  static constexpr bool value = true;
};
```

- Global constexpr variable template

```
template< typename T>
inline constexpr bool is_reference_v = is_reference<T>::value;
```

Type traits – possible implementation

- Type traits – master definition

```
template< typename T> struct remove_reference {
  using type = T;
};
```

- Type traits – partial specializations

```
template< typename U> struct remove_reference< U &> {
  using type = U;
};
template< typename U> struct remove_reference< U &&> {
  using type = U;
};
```

- Global type alias template

```
template< typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

Type traits – possible implementation

- Frequently used type alias templates in <type_traits>

```
template< typename T>
using remove_reference_t = typename remove_reference<T>::type;

template< typename T>
using remove_cv_t = typename remove_cv<T>::type;

template< typename T>
using remove_cvref_t = remove_cv_t<remove_reference_t<T>>;
```

  - Usage:

```
template< typename T> inline void example(T && v) {
   std::remove_cvref_t<T> a_copy_of_v = std::forward<T>(v);
   ++a_copy_of_v; std::cout << a_copy_of_v;
}
void test_example(const std::string & x) {
   example(x[0]);  // T = const char &
}
```

Example – storing values of any type

```
template< typename T> class ftor { public:
  template< typename T2> ftor( T2 && p);
};
```

- C++17: deduction guides

```
template< typename T2>
ftor( T2 && p) -> ftor< std::remove_cvref_t< T2>>;
```

  - Allows use of this syntax:

```
std::string s = “hello”;
ftor x( s);
auto y = ftor( s);
```

  - Wrapper functions no longer needed

```
std::pair p(i,d);
```

    - instead of

```
std::pair<int,double> p(i,d);
auto p = std::make_pair(i,d);
```

Example – storing values of any type

- A class

```
template< typename T> class ftor {
public:
  template< typename T2> ftor( T2 && p) : p_( std::forward< T2>( p)) {}
  void operator()( T & x) { x += p_; }
private:
  T p_;
};
```

- and its wrapper function

```
template< typename T> auto make_ftor( T && p)
{ return ftor<std::remove_cvref_t<T>>( std::forward< T>(p));
}
```

- still does not work

```
std::vector< std::string> v;
std::for_each( v.begin(), v.end(), make_ftor( “Hello”));
```

  - ftor<const char *>::operator() requires const char * &

Example – storing values of any type

- The correct implementation is

```
template< typename T1> class ftor {
public:
  template< typename T2> ftor( T2 && p) : p_( std::forward< T2>( p)) {}
  template< typename T3> void operator()( T3 && x) { x += p_; }
private:
  T1 p_;
};
template< typename T4>
ftor( T4 && p) -> ftor< std::remove_cvref_t< T4>>;
```

→ tady to nechci ořezávat, protože bych jinak mohl mít více implementací ftoru.

- Note: always use forwarding reference T3 && instead of lvalue reference T3 &
  - this allows functionality on containers producing fake references (like vector< bool>)
- Note: why we did not hide the std::remove_cvref_t inside ftor?

```
template< typename T> class ftor { /*...*/ std::remove_cvref_t<T> p_; };
```

  - Because ftor(x) and ftor(x+1) would produce different instantiations of ftor