СВЕРТКА ССЫЛОК

Правило «свертки» ссылок (reference collapsing)

Замечание: в общем случае конструкции «ссылка на ссылку» в C++ запрещены:

int& & r = …; //ошибка

но иногда это может происходить => правило:

A& & -> A&

A&& & -> A&

A& && -> A&

A&& && -> A&&

Когда компилятор применяет свертку ссылок:

- при инстанцировании шаблонов
- вывод типа для auto-переменных
- задание псевдонимов – typedef

Демонстрация правила «свертки» ссылок на примере auto

```
int n = 1;
auto x = n; //int
auto& y = n; //int&
//auto &  & = n; //ошибка – reference to reference are not allowed
```

auto && z = n; //(n - lvalue) => int& && -> **int&**

auto&& w = 1; // (1- rvalue) => int&& && -> **int&&**

«Снятие» ссылочности

Если параметр шаблона – ссылочный тип, то предоставляет псевдоним указываемого типа, иначе тип

template< class **T** > struct **remove_reference** {typedef T type;};

template< class T > struct **remove_reference<T&>** {typedef T type;};

template< class T > struct **remove_reference<T&&>** {typedef T type;};

Как работает **remove_reference**

```
std::remove_reference<A>::type x; //A
std::remove_reference<A&>::type y; //A
std::remove_reference<A&&>::type z; //A

A a(1);
std::remove_reference<A>::type& rx=a; //A&
std::remove_reference<A&>::type& ry = a; //A&
std::remove_reference<A&&>::type& rz = a; //A&

std::remove_reference<A>::type&& rrx = A(2); //A&&
std::remove_reference<A&>::type&& rry = A(3); //A&&
std::remove_reference<A&&>::type&& rrz = A(4); //A&&
```

Также <type_traits>

- std::is_reference
- std::is_lvalue_reference
- std::is_rvalue_reference

Перегружены:

- operator bool
- operator ()

Пример:

bool b=std::is_rvalue_reference<A>::value;

b=std::is_rvalue_reference<A&>::value ;

b=std::is_rvalue_reference<A&&>::value ;

FORWARD

Perfect forwarding

В первую очередь предназначен для уменьшения дублирующего текста

Экспериментальный класс:

```
class A {
   int m_a;
public:
   explicit A(int a=0):m_a(a){}
   A(const A& other) ;
   A(A&& other) ;
   A& operator=(const A& other) ;
   A& operator=(A&& other) ;
   ~A() ;
};
```

Независимо от типа действия требуется сделать одни и те же. Проблемы?

```
class A{…};
void f(A& a) { … }

int main(){
    A a1;
    f(a1); //???

    const A a2;
    f(a2); //???

    f(A()); //???
}
```

Решение до C++11
Универсальное, но неэффективное

```
class A{…};
void f(const A& a) { … }

int main(){
    A a1;
    f(a1); //???

    const A a2;
    f(a2); //???

    f(A()); //???
}
```

Решение до C++11
Эффективное, но текст дублируется

```
class A{…};
void f(const A& a) { … <использование a>; …}
void f(A&& a) { … <использование std::move(a);>… }

int main(){
    A a1;
    f(a1); //???

    const A a2;
    f(a2); //???

    f(A()); //???
}
```

А если требуется для объектов разного типа выполнить разные действия?

```
class A{…};
void f(const A& a) { std::cout<<a; }
void f(A& a) {++a;}
void f(A&& a) {…  std::move(a));…}

int main(){
      A a1;
      f(a1); //???

      const A a2;
      f(a2); //???

      f(A()); //???
}
```

Пояснение проблемы

void wrap(<**тип**> t) // тип параметра – любой из требуемых – A&, const A&, A&&

{

//действия, не зависящие от типа “t”

f(t); **//здесь должна быть вызвана соответствующая функция f()!!!**

// действия, не зависящие от типа “t”

} => сколько раз нужно перегрузить функцию **wrap()**?

Проблемы множатся **нелинейно**…

А если функция f() принимает не один параметр?

- два ,
- …

Нужно предусмотреть **все** сочетания разных типов => при росте числа параметров (N) количество перегруженных функций будет расти **нелинейно** ???

=> нужно обеспечить:

универсальную **перенаправляющую** функцию независимо от типа параметров

Идеальная перенаправляющая ф-ция (perfect fowarding function) **wrap(a1,a2,...,aN)**, которая вызывает **f(a1,a2,...,aN)**

должна удовлетворять следующим критериям:

- Для всех наборов a1,a2,...,aN, для которых запись f(a1,a2,...,aN) корректна(well-formed), запись wrap(a1,a2,...,aN) должна быть так же корректна.
- Для всех наборов a1,a2,...,aN, для которых запись f(a1,a2,...,aN) некорректна(ill-formed), запись wrap(a1,a2,...,aN) должна быть так же некорректна.
- Количество работы, которую придётся проделать для реализации такой идеально-перенаправляющей ф-ции f должно не более чем **линейно** зависеть от N.

T&& применительно к шаблонам имеет смысл «**универсальной ссылки**»

- принимает «все, что угодно» с сохранением категории (lvalue/rvalue) и cv-квалификаторов
- Обрабатывается согласно правилам:
  - вывода аргумента шаблона T&&
  - свертывания ссылок

Смысл forward<> - **условный** cast:

для

template<typename T> void wrap(T&& t){…}

- Если функция вызывается для lvalue, то результирующий тип будет **A&**
- Если функция вызывается для rvalue, то результирующий тип будет **A&&**

Демонстрация правила

template<typename T> void f(T**&&** t){...} //!!! здесь t – не является rvalue-reference, а обозначает универсальный тип параметра!!!

При инстанцировании:

f(4); // 4 - это rvalue: T становится int => f(int&&)

```
int n = 3;
f(n); // n это lvalue; T становится int& => f(int& &&)->f(int&)

int f1() {...}
f(f1()); // f() это rvalue; T становится int => f(int&&)

int f2(int i) {
  f(i); // i это lvalue; T становится int& => f(int& &&)->f(int&)
}
```

Возможная имплементация forward: <utility>

```
template<typename T> T&& forward(
typename remove_reference<T> :: type& param)
                                                noexcept
{
        return static_cast<T&&> ( param) ;
}
```

Замечание: в стандарте C++14 добавлена версия для constexpr

Пример – пояснение:

template<typename T1, typename T2> void
f(T1&& t1, T2&& t2) {…}

template <typename T1, typename T2> void
wrap(T1&& e1, T2&& e2)
{… **f**(**forward**<T1>(e1), **forward**<T2>(e2)); …}

Продолжение примера:

```
int x=1;
float y = 2.2;
wrap(x, y); //x – lvalue, y – lvalue => T==int&
                                  wrap<int&, float&>(int&, float&)
//инстанцируется:
int& && forward(int& t) noexcept
{ return static_cast<int& &&>(t); }

//по правилу свертки ссылок ->
int&  forward(int& t) noexcept
{ return static_cast<int& >(t); }
```

=> f(int&, float&)

Продолжение примера:

wrap(1, 2.2); //x – rvalue, y – rvalue => **T==int**

wrap<int, double>(int&&, double&&)

//по правилу свертки ссылок:

int&& forward(int& t) noexcept

{ return static_cast<int&&>(t); }

=> f(int&&, double&&)

Полезный пример использования

forward

Перегрузка конструкторов (без использования forward)

```
class MyString{
        char* m_pStr;
public:
    …
        MyString(const MyString& );
        MyString(MyString&& );
};

class A{
        MyString m_str;
public:
        A(MyString && s) : m_str( std::move(s) ){}
        A(const MyString & s) : m_str(s) {}
        A(const char* p) : m_str(p){}
        …
};
```

Перегрузка конструкторов (без использования forward)

```
{
    MyString s(“ABC”);
    A a1(s); //???
    A a2(MyString(“QWERTY”)); //???
    A a3(“Hello”); //???
}
```

Пытаемся объединить три перегруженных конструктора посредством шаблона **(пока без forward)**

```
class MyString{
        char* m_pStr;
public:
        …
        MyString(const MyString& );
        MyString(MyString&& );
};

class A{
        MyString m_str;
public:
        template<typename T> A(T&& t) : m_str(t){}
};
```

Пытаемся объединить два перегруженных конструктора посредством шаблона (пока без forward)

```
{
  MyString s("ABC");
  A a1(s); //MyString(const MyString& )

  A a2(MyString("QWERTY")); //MyString(const MyString& );
                                                    !!!
  A a3("Hello"); //MyString(const char* );
}
```

Полезный пример использования **forward**

```
class MyString{
        char* m_pStr;
public:
    …
        MyString(const MyString& );
        MyString(MyString&& );
        MyString(const char*);
};

class A{
        MyString m_str;
public:
        template<typename T> A(T&& t) : m_str(std::forward<T>(t) ){}
};
```

Полезный пример использования **forward**

```
{
  MyString s(“ABC”);
  A a1(s); //MyString(const MyString& )

  A a2(MyString(“QWERTY”)); //MyString(MyString&& ) !!!

  A a3(“www”); ???
}
```

Еще один полезный пример использования forward

```
class MyString{…
        char* m_pStr;
public:
    …
        MyString& operator=(const MyString& );
        MyString& operator=(MyString&& );
        MyString& operator=(const char*);
};

class A{
        MyString m_str;
public:
        …
        template<typename T> void setNewString(T&& newStr)
        { m_str = std::forward<T>(newStr) ;}
};
```