Doskonałe przekazywanie argumentów

dr inż. Ireneusz Szcześniak

jesień 2017

1 Złe rozwiązania

Programista ma napisać funkcję `f`, która wywołuje funkcję `g`. Nie wiemy jaki jest typ parametru funkcji `g`, ale chcielibyśmy wywoływać funkcję `f` tak, jak wykonujemy funkcję `g`.

Rozwiązanie z taką definicją szablonu (typem parametru funkcji szablonowej jest niestała l-referencja) nie kompiluje się:

```
void g(const int &)
{
}

template <typename T>
void f(T &p)
{
  g(p);
}

int main()
{
  // We can call "g" all right with an rvalue.
  g(1);
  // But we cannot call "f" with an rvalue.
  f(1);
}
```

Rozwiązanie z taką inną definicją szablonu (typem parametru funkcji szablonowej jest stała l-referencja) też się nie kompiluje:

```
void g(int &)
{
}

template <typename T>
void f(const T &p)
{
  g(p);
}

int main()
{
  int x = 1;
  // We can call "g" with an lvalue of non-const type.
  g(x);
  // We cannot call "f" with an lvalue of non-cost, because in "f"
  // it's bound to a const lvalue reference, which cannot be used to
  // initialize the parameter of g, which is a non-const lvalue
  // reference.
```

```
  f(x);
}
```

Jeżeli zdefiniujemy dwa szablony (jeden dla referencji niestałej, a drugi dla stałej), to wtedy problem jest rozwiązany, chociaż lepiej byłoby zdefiniować tylko jeden szablon:

```
#include <iostream>

void g(int &)
{
  std::cout << "int␣&\n";
}

void g(const int &)
{
  std::cout << "const␣int␣&\n";
}

template <typename T>
void f(T &p)
{
  g(p);
}

template <typename T>
void f(const T &p)
{
  g(p);
}

int main()
{
  int x = 1;
  f(x);
  f(1);
}
```

2 std::forward

Przykładowy program pokazujący działanie funkcji `std::forward`:

```
#include <iostream>
#include <utility>

using namespace std;

void
foo(int &)
{
  cout << "foo␣dla␣l-wartości\n";
}

void
foo(int &&)
{
  cout << "foo␣dla␣r-wartości\n";
}
```

```
int
main()
{
  int x = 1;
  foo(forward<int &>(x));
  foo(forward<int>(x));
}
```

3 Spłaszczenie referencji

Przykładowy program pokazujący spłaszczanie referencji:

```
#include <iostream>
#include <type_traits>

using namespace std;

int
main()
{
  cout << std::boolalpha;

  using lt = int &;
  using rt = int &&;

  using typ1 = lt &&;
  cout << is_same<int &, typ1>::value << endl;

  using typ2 = rt &&;
  cout << is_same<int &&, typ2>::value << endl;
}
```

4 Wnioskowanie typów

Przykładowy program pokazujący wnioskowanie typów. Jeżeli typem jest `auto &&`, to kompilator wywnioskuje, czy referencja powinna być typu l-wartość czy r-wartość w zależności od kategorii wartości wyrażenia, które jest inicjalizatorem referencji.

Żeby zobaczyć wywnioskowany typ w czasie kompilacji, w kodzie wprowadzono błąd, o którym kompilator informuje jednocześnie wypisując interesujący nas typ.

```
template <typename T>
class ER;

int
main()
{
  int x = 1;
  auto &&r1 = 1;
  auto &&r2 = x;

  ER<decltype(r1)> error1;
  ER<decltype(r2)> error2;
}
```

5 Dobre rozwiązanie

Zdefiniujemy różne przeciążenia dla funkcji g. Funkcja g będzie przyjmowała też drugi parametr, który będzie pozwalał nam stwierdzić, czy kompilator wybrał te przeciążenie, które się spodziewaliśmy. W funkcji main wywołujemy każde przeciążenie funkcji.

Potem piszemy funkcję f, która doskonale przekazuje swój argument do funkcji g.

```
#include <iostream>
#include <utility>

using namespace std;

void
test(const string &s1, const string &s2)
{
  if (s1 != s2)
    cout << "Expected:␣" << s1 << ",␣got:␣" << s2 << endl;
}

void
g(int &, const string &s)
{
  test(s, "int␣&");
}

void
g(const int &, const string &s)
{
  test(s, "const␣int␣&");
}

void
g(int &&, const string &s)
{
  test(s, "int␣&&");
}

void
g(const int &&, const string &s)
{
  test(s, "const␣int␣&&");
}

template<typename T>
void
f(T &&t, const string &s)
{
  g(forward<T>(t), s);
}

int main()
{
  // Test "int &"
  int x;
  g(x, "int␣&");
  f(x, "int␣&");
```

```
  // Test "const int &"
  const int &y = 1;
  g(y, "const␣int␣&");
  f(y, "const␣int␣&");

  // Test "int &&"
  g(1, "int␣&&");
  f(1, "int␣&&");
}
```

Usunąć funkcję `forward` z funkcji `f`. Wtedy będą przekazywane zawsze l-wartości do funkcji `g`. Funkcja forward załatwia nam poprawne przekazywanie kategorii wartości argumentu wywołania funkcji `f`.