# Doskonałe przekazywanie argumentów

dr inż. Ireneusz Szcześniak

jesień 2017 roku

# Przekazywanie argumentów

Piszemy funkcję `f`, która wywołuje funkcję `g` z jednym parametrem, ale o kwalifikatorach (const, volatile) i typie parametru funkcji `g` nic nie wiemy. Chcemy napisać tylko jedną wersję funkcji `f`.

```
template<typename T>
void
f(<kwalifikatory> <typ> a)
{
  g(a);
};
```

PYTANIE: jakie mają być kwalifikatory i jaki typ parametru `a` funkcji `f`? Czy kwalifikatorem może być `const`, czy musi być `const`? Czy typem ma być `T`, `T &`, czy `T &&`?

ODPOWIEDŹ: można, ale tylko z C++11. O tym później.

# Motywacja: fabryki obiektów

Funkcje `make_unique` i `make_shared` są fabrykami obiektów. Tworzą one obiekty i potrzebują przekazać swoje argumenty do konstruktora klasy w niezmienionej postaci.

To jest przykład dla dwóch parametrów:

```
template<typename T, typename A1, typename A2>
unique_ptr<T>
make_unique(<kwalifikator> <typ A1> a1,
            <kwalifikator> <typ A2> a2)
{
  return unique_ptr<T>(new T(a1, a2));
};
```

# Parametry i argumenty funkcji

- Parametry i argumenty funkcji to nie to samo!
- Parametr to zmienna dostępna w ciele funkcji.
- Argument to wyrażenie w wywołaniu funkcji.
- Parametry są inicjalizowane na podstawie argumentów.
- `foo(int x)` - `x` jest parametrem funkcji
- `foo(a)` - `a` jest argumentem wywołania funkcji

Argument może być l-wartością albo r-wartością, a parametr zawsze jest l-wartością, bo ma nazwę (możemy pobrać jego adres).

# Możliwe rozwiązania

Możliwe rozwiązania z pominięciem kwalifikatora `volatile`.

- przez wartość: `T`
- przez stałą wartość: `const T`
- przez referencję typu lvalue: `T &`
- przez referencję stałą typu lvalue: `const T &`
- przez referencję typu rvalue: `T &&`
- przez referencję stałą typu rvalue: `const T &&`

# Rozwiązanie: przekazywanie przez wartość

Chodzi o `T` i `const T`. Na przykład dla `T`:

```
template<typename T>
void
f(T a)
{
  g(a);
};
```

Gdy wykonamy `f(1)`, a funkcja `g` będzie pobierać argumenty przez referencję, to nie otrzyma referencji na oryginalny obiekt, a referencję na parametr funkcji `f`, który jest kopią oryginalnego obiektu.

TO JEST ZŁE ROZWIĄZANIE!

# Pozostałe rozwiązania do rozważenia

Bierzemy pod uwagę wyłącznie rozwiązania z referencjami, ale bez `const T &&`, bo jest ona bezużyteczna.

Zostają nam trzy przypadki do rozważenia:

- `T &`
- `const T &`
- `T &&`

# Rozwiązanie: T &

Rozwiązanie z T & wygląda tak:

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

PROBLEM: jeżeli argument jest kategorii rvalue, np. f(1), to będzie błąd kompilacji. Dlaczego?

TO JEST ZŁE ROZWIĄZANIE!

# Rozwiązanie: `const T &`

Rozwiązanie z `const T &` wygląda tak:

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

Teraz będzie się kompilować dla r-wartości, np. `f(1)`.

PROBLEM: Jeżeli funkcja będzie `g(int &)`, to kod nie będzie się kompilować. Dlaczego?

TO JEST ZŁE ROZWIĄZANIE!

# Rozwiązanie: `T &` razem z `const T &`

Możemy mieć dwie definicje, jedna dla `T &`, a druga dla `const T &`. Czyli mamy dwie definicje dla jednego parametru.

PROBLEM: Dla $n$ parametrów potrzebujemy $2^n$ definicji funkcji!

TO JEST ZŁE ROZWIĄZANIE!

# Prawidłowe rozwiązanie

Z C++11, typem parametru powina być r-referencja bez kwalifikatorów, czyli tak:

```
template<typename T>
void
f(T &&t)
{
  g(std::forward<T>(t));
};
```

Problem w tym, że parametr `t` jest l-wartością (bo ma nazwę `t`), nawet jeżeli argumentem wywołania funkcji `f` była r-wartość. W ten sposób tracimy informację o kategorii wartości wyrażenia, które było argumentem funkcji `f`.

# Referencja do referencji

W C++ nie ma typu referencji do referencji, ale takie typy mogą się pojawić, jako efekt definicji typów szablonowych z użyciem `typedef` czy `using`.

```
template<typename T>
class A
{
  typedef typename T &my_type;
  // ...
};
```

Jeżeli kompilator wywnioskuje typ T jako, na przykład, `int &`, to wtedy otrzymamy typ `my_type` jako `int & &`. Co wtedy?

# Spłaszczanie typów referencji

Jeżeli pojawi się typ referencji do referencji, to kompilator zamieni taki typ na referencję według zasady:

- `cv1 T & cv2 T &` → `cv12 T &`
- `cv1 T & cv2 T &&` → `cv12 T &`
- `cv1 T && cv2 T &` → `cv12 T &`
- `cv1 T && cv2 T &&` → `cv12 T &&`

Zbiory cv1, cv2, cv12 oznaczają zbiory kwalifikatorów, do których mogą należeć const i volatile. Zbiór cv12 jest sumą zbiorów cv1 i cv2, czyli $\mathrm{cv12} = \mathrm{cv1} \cup \mathrm{cv2}$.

# Dopasowywanie typu argumentu szablonu

Jaki będzie dopasowany typ `T` w szablonie? Przykład:

```
template<typename T>
void
f(T &&a)
{
};
```

Dwie zasady:

- jeżeli argumentem funkcji `f` jest r-wartość typu A, to `T = A`
- jeżeli argumentem funkcji `f` jest l-wartość typu A, to `T = A &`

# `std::forward`

Funkcja szablonowa `std::forward` przyjmuje l-wartość `a` typu $T$ i zależności od sposobu wywołania (typu argumentu szablonu) zwraca:

- r-referencję na `a` dla `std::forward<T>(a)`
- l-referencję na `a` dla `std::forward<T &>(a)`

Funkcji `std::forward` używa się w definicji pewnej funkcji szablonowej `f`, gdzie trzeba odzyskać kategorię wyrażenia, które było argumentem wywołania funkcji szablonowej `f`.

# Podsumowanie

- Problem doskonałego przekazywania argumentów występuje w programowaniu generycznym z użyciem szablonów.
- Jako typ parametru funkcji używamy: `T &&t`, gdzie `T` jest parametrem szablonu, a `t` parametrem funkcji.
- Aby przekazać parametr do innej funkcji, używamy funkcji `std::forward`, która odzyskuje kategorię wartości wyrażenia.

Dziękuję za uwagę.