



Uniwersytet
Wrocławski

Move semantics

Zbigniew Koza

Wydział Fizyki i Astronomii

Motywacja

```
using X = std::vector<double>
```

- ```
X x {f()}; // C++
```

```
X * y = malloc(...); // C
```

```
init(y, f);
```

} Nieeleganckie  
wskaźniki są  
efektywne
- ```
X v0, v1, v2;
```

```
X y = (v0 + (2*v1)) - 3*v2; // C++
```

```
X* z; // C
```

```
add1_add2_sub3(z, &v0, &v1, &v2);
```

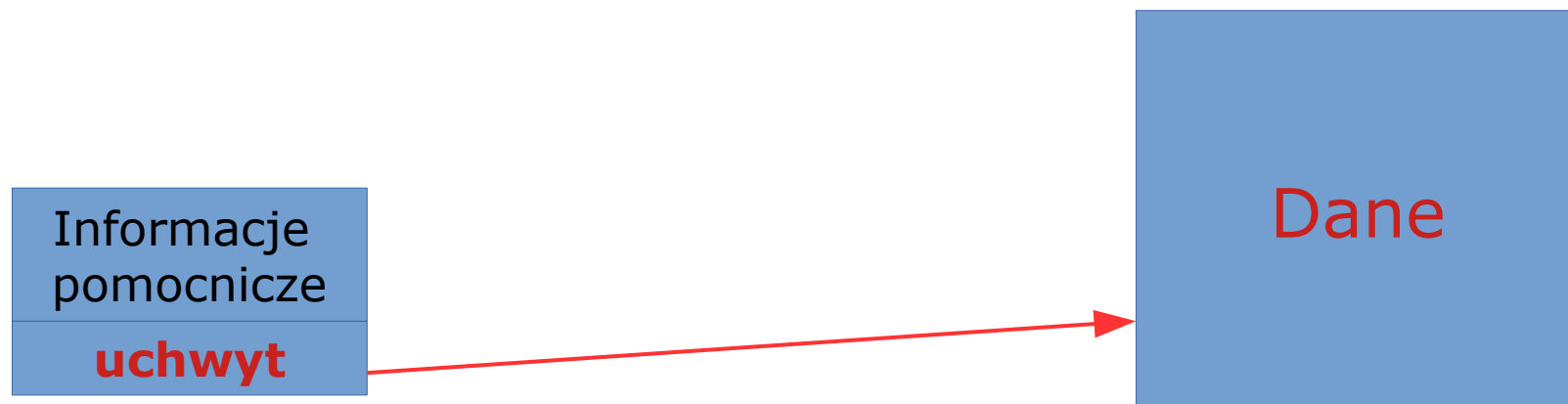
Uchwyty do zasobów

np. `std::vector`



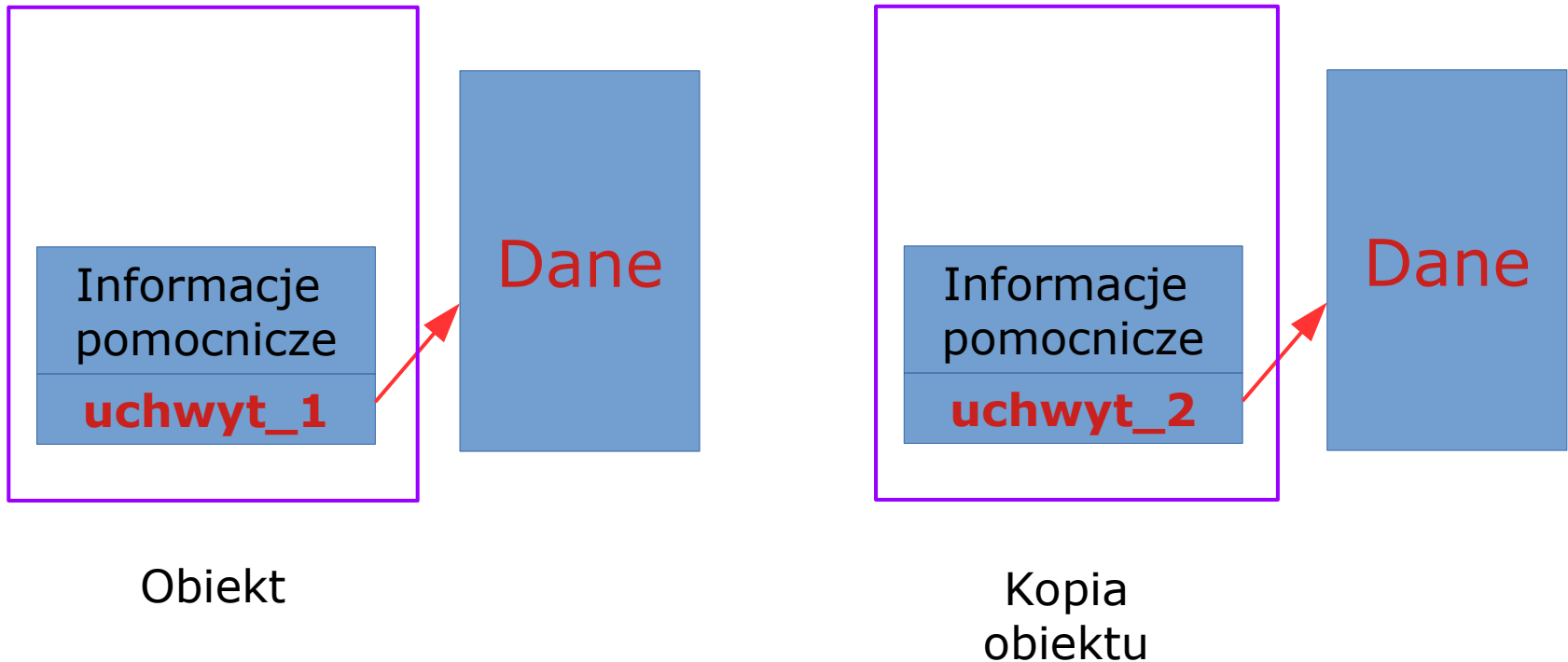
- Taka konstrukcja:
 - Ułatwia zarządzanie zasobami,
 - Umożliwia zmniejszenie użycie stosu programu
 - Ułatwia optymalizację pewnych operacji, np.
`swap`

Uchwyty do zasobów



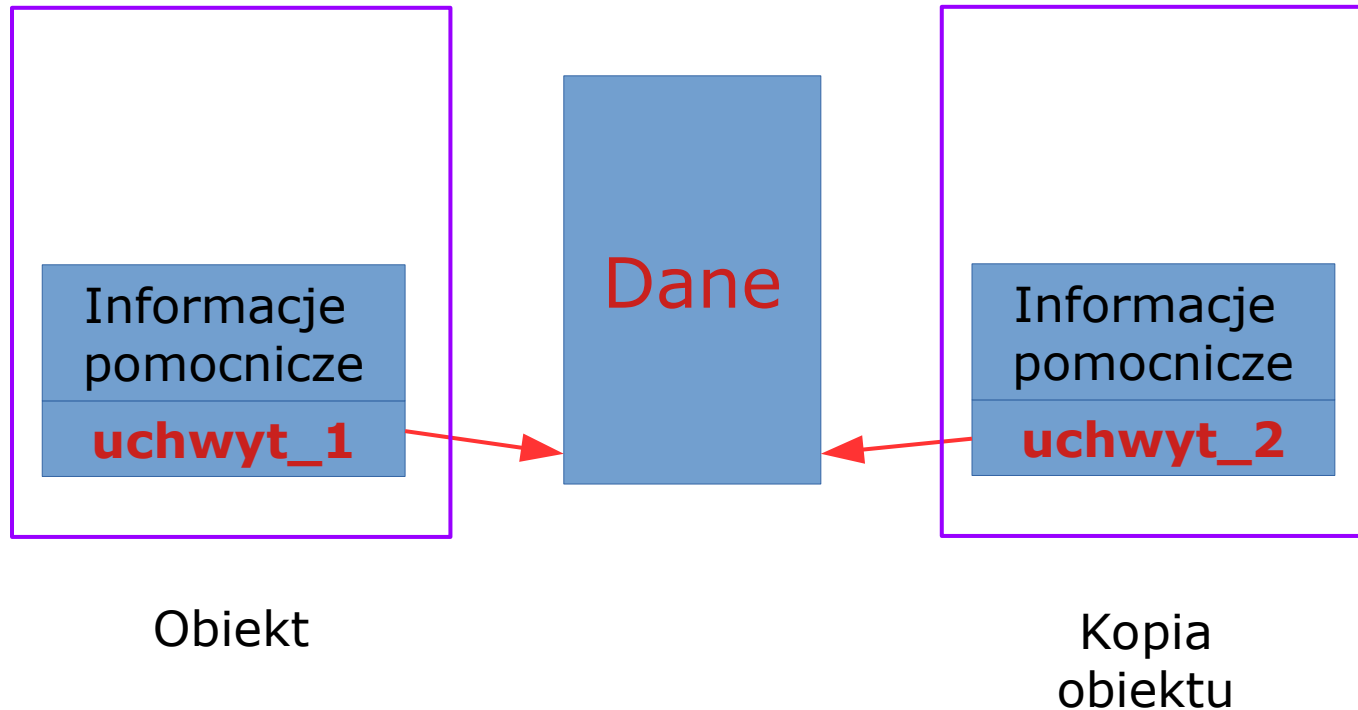
- Taką konstrukcję ma większość kontenerów STL (vector, list, deque,...)

Kopia głęboka (a = b)



- Kopiowanie głębokie (= „całkowite”) jest kosztowne (identyczne dane w kilku miejscach)
- Ale zawsze bezpieczne

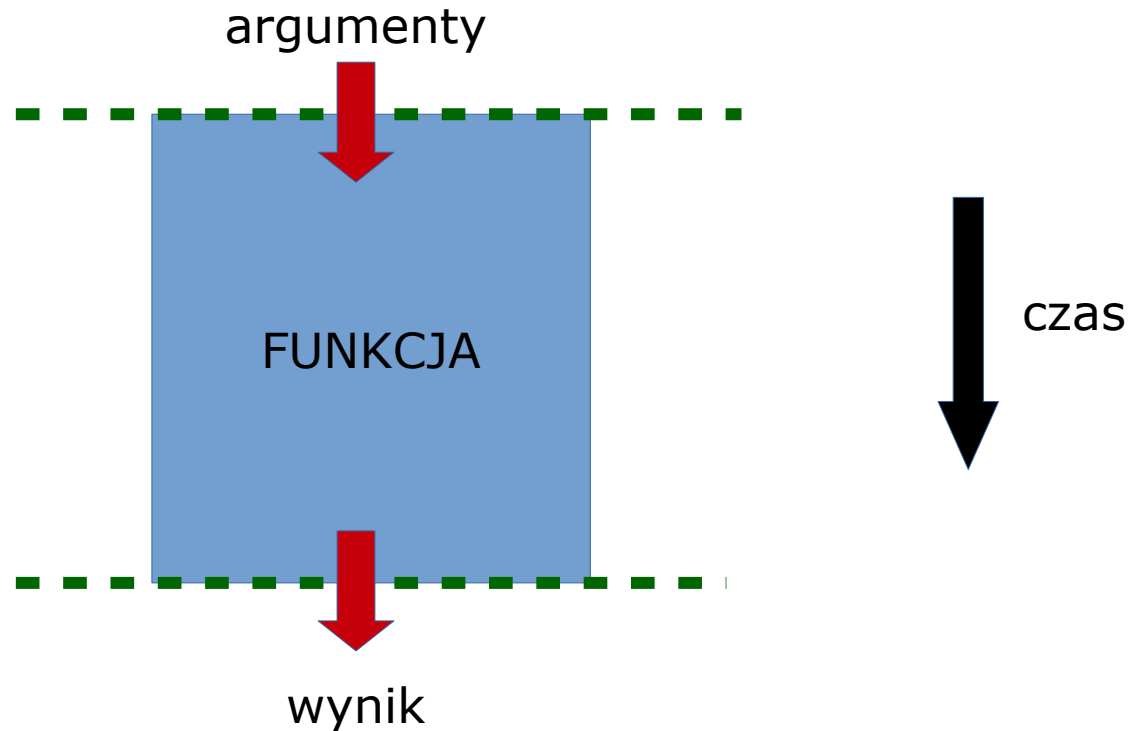
Kopia płytka



- Kopiowanie płytke jest tanie
- Ale czasami niebezpieczne
- Podstawa mechanizmu tzw. *reference counting*

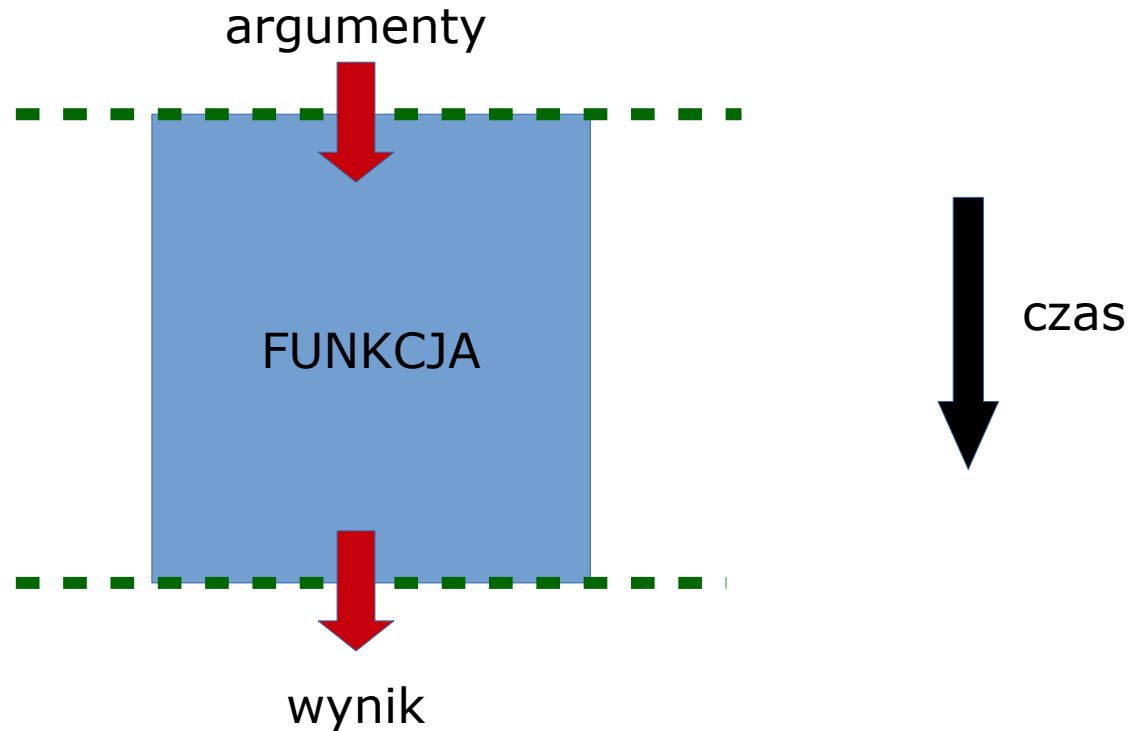
Interfejsy funkcji

- Jak prawidłowo i możliwie tanio przekazać dane do i z funkcji?



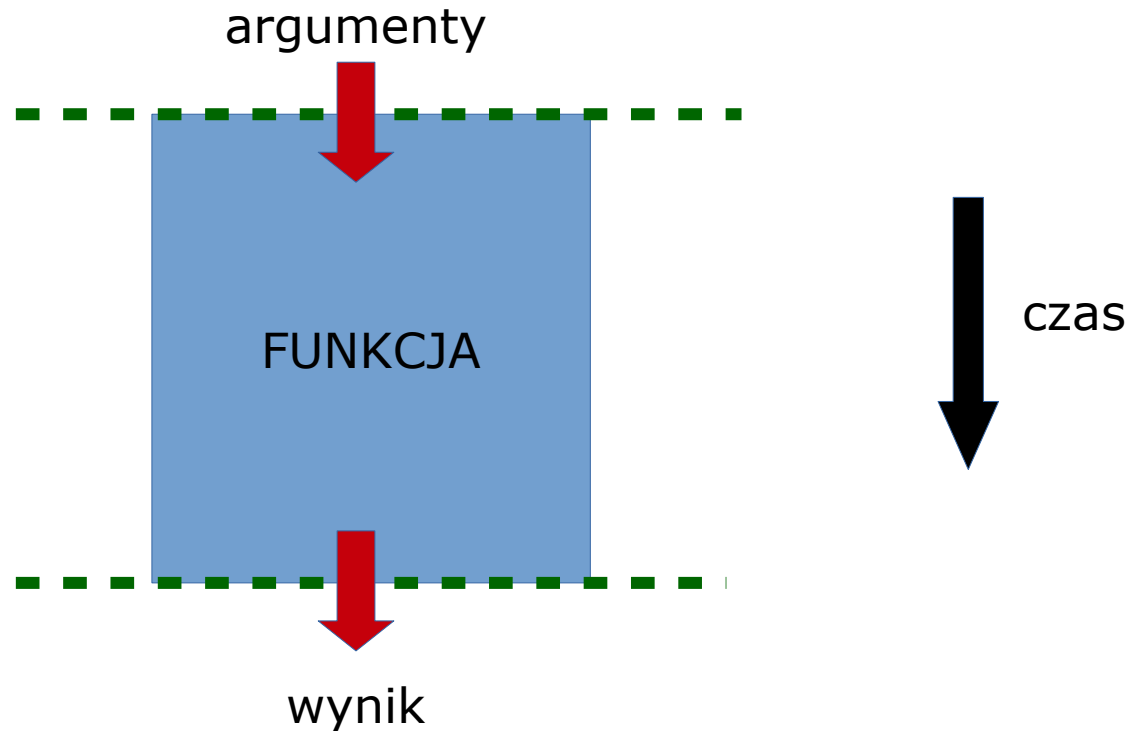
Interfejsy funkcji

- Kopiowanie głębokie bywa kosztowne



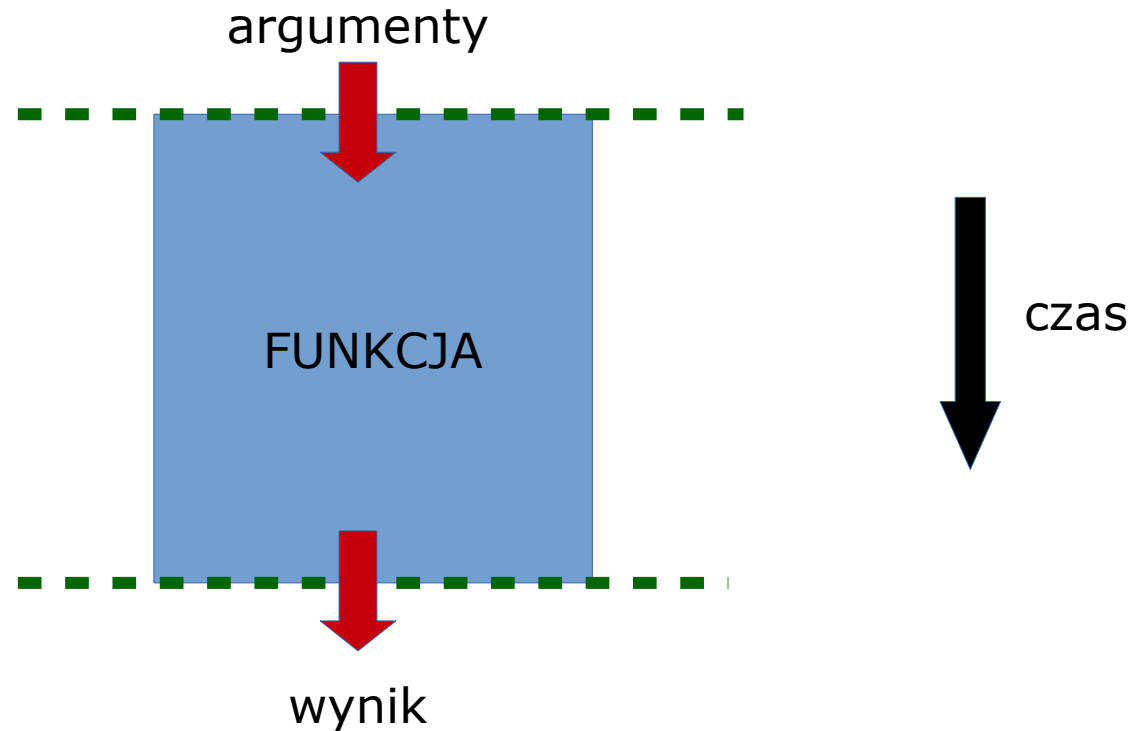
Interfejsy funkcji

- Wynik funkcji „ginie” zaraz po zakończeniu funkcji
→ czy kopiowanie głębokie jest potrzebne (?)



Znane interfejsy funkcji w C++

- Przez **wartość**, **referencję** (**stałą referencję**), **wskaźnik**



Interfejsy funkcji (C++98)

- Przez **wartość**: kosztowne **kopiowanie**
- Przez **referencję**: tylko do **obiektów mających adres (l-wartości)**, bardzo kłopotliwe przy zwracaniu wartości funkcji
- Przez **stałą referencję**: uniwersalne, ale przy przekazaniu **obiektu bezadresowego (r-wartości)** po cichu wykonywana jest jego **kopia**, co kosztuje
- **Wskaźnik** – jak referencja, ponadto nie lubimy nagich wskaźników!

Interfejsy funkcji (C++98)

- `int x = 9;`

Przez wartość:

`void f(int n);`

`f(10);` `f(x);` `f(10 + x)`

- Przez referencję:

`void f(int& n)`


~~`f(10);`~~ `f(x);` ~~`f(10 + x)`~~

- Przez stałą referencję:

`void f(const int& n)`

`f(10);` `f(x);` `f(10 + x)`

„ciche” kopiowanie



Interfejsy funkcji (C++98)

- `int x = 9;`

Przez wartość:

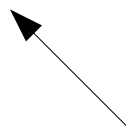
`int f(); return 10; return x; return 10 + x;`

- Przez referencję:

`int& f() return ???`

- Przez stałą referencję:

`const& f() return ???`



Może, ale nie musi wystąpić
„ciche” kopiowanie

Interfejsy funkcji (C++98)

- Jak działa stała referencja?
 - Jeżeli **argument ma adres**, to jest on przekazany do funkcji (czyli to działa jak zwykła referencja lub wskaźnik).
 - Jeżeli argument **nie ma adresu**, to najpierw w pamięci konstruowana jest **kopia** jego wartości, a następnie funkcja dostaje jej adres

L-wartości i r-wartości

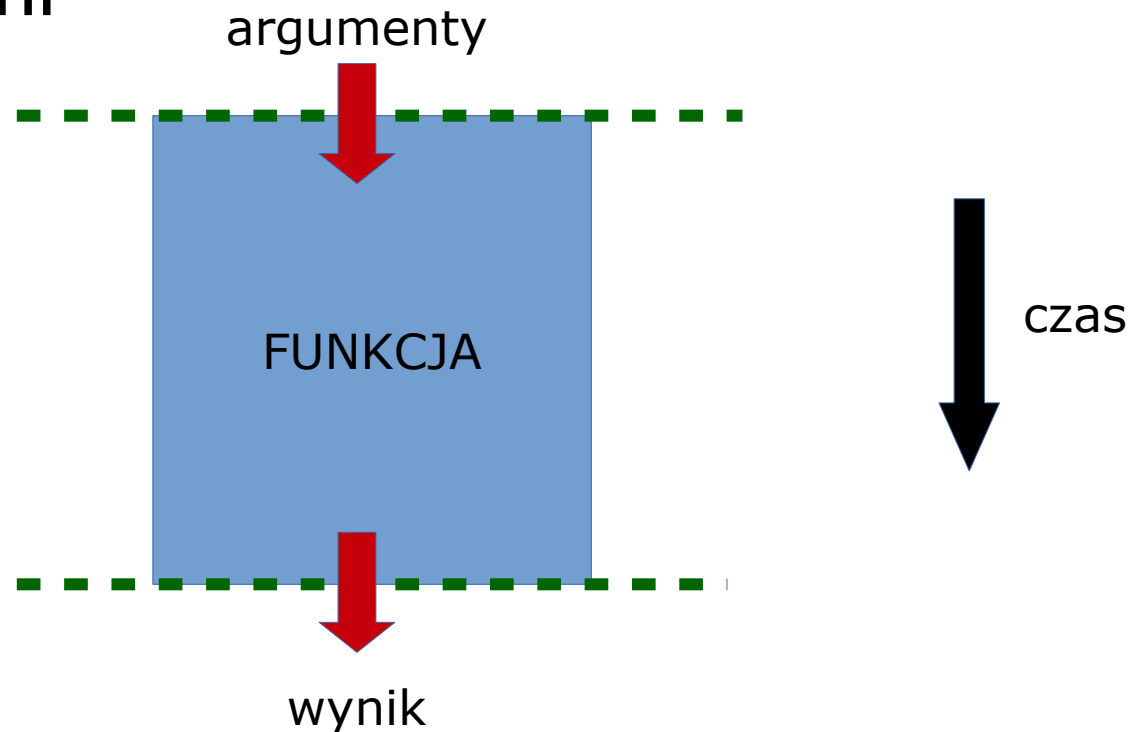
- L-wartości („lewe strony operatora=”) mają adresy (miejsce w RAM)
- R-wartości nie mają adresów (**zmienne tymczasowe**, np. przechowywane w rejestrach, cache’u, stałe wkompilowywane w kod)

Tylko l-wartości mają adres

- `int x = 9;` // x jest l-wartością
- `x + 9;` // r-wartość
- `sin(x)` // wartością tej funkcji jest r-wartość
- `std::cout` // l-wartość
- `sin` // l-wartość
- `std::vector<int> v;` // v jest l-wartością
- `5` // 5 jest r-wartością

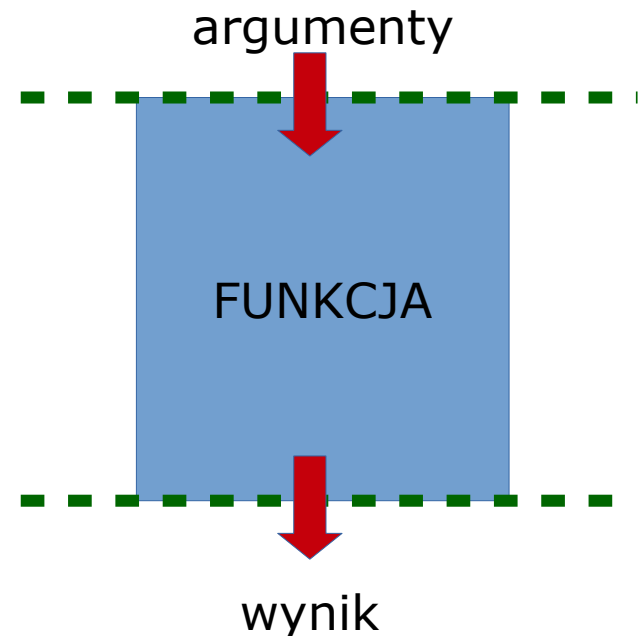
Czy można automatycznie ominąć zbędne kopiowanie głębokie?

- Problem dotyczy obiektów **tymczasowych** (np. wartości funkcji) z uchwytami



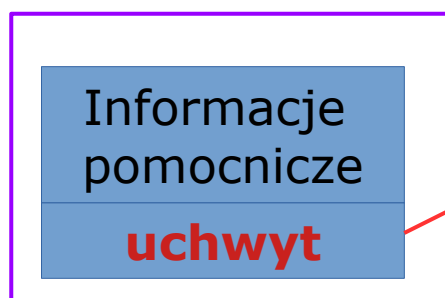
Czy można automatycznie ominąć zbędne kopiowanie głębokie?

- Problem dotyczy obiektów **tymczasowych** (np. wartości funkcji) z uchwytami
- Po co kopiować coś, co zaraz „zginie”?



Przenoszenie: płytka kopia

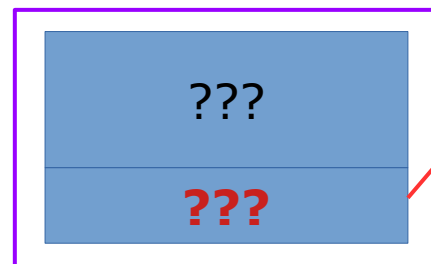
Funkcja wywołująca



Obiekt



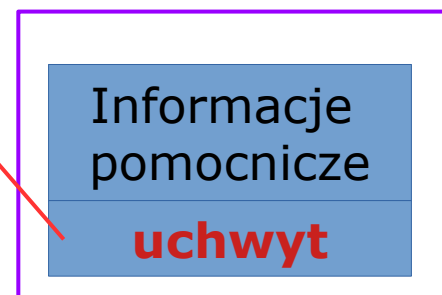
Funkcja wywoływana



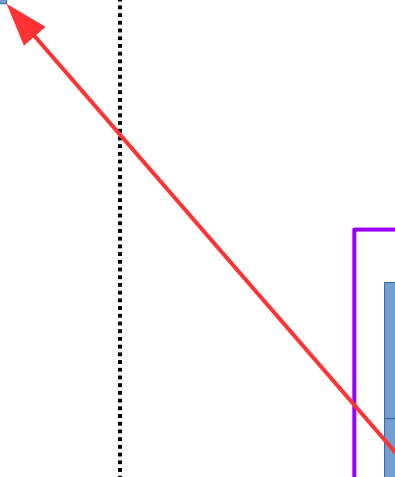
Kopia



Obiekt



Kopia



Rozwiązanie:

referencja do r-wartości (&&)

- `int x = 9;`
- `int & b = x;` //OK, l-referencja do obiektu
- `int && c = 9;` // OK, r-referencja do stałej
- `int & d = 9` // niemożliwe, 9 nie ma adresu
- `int && e = x` // niemożliwe, x ma adres

Jak działa referencja do r-wartości?

- `int && c = 9; // r-referencja do stałej`
 - Kompilator generuje kopię wartości (tu: 9) i umieszcza w zmiennej `c` jej adres
 - Przypomina to obsługę stałych referencji, (**`const&`**)

Po co referencje do r-wartości?

- Mając do dyspozycji referencje do l-wartości (&) i r-wartości (&&), można na poziomie języka, czyli **automatycznie**, odróżnić przekazywanie do i z funkcji parametrów mających adres i nie mających adresu, co z kolei umożliwia automatyczną optymalizację obsługi tych drugich poprzez zastosowanie przeniesienia zamiast głębokiej kopii.

Przykład klasa Wektor

- Dane są dostępne przez uchwyt:

```
class Wektor
{
    int* data;
    size_t size;
public:
    Wektor(size_t n)
    {
        data = new int[n];
        size = n;
    }
    ~Wektor() { delete [] data; }
    int operator[](int n) const { return data[n]; }
    int& operator[](int n) { return data[n]; }
};
```

Operator+

Przez **wartość**!

Przez (stałą)
referencję!

```
Wektor operator+(Wektor lhs, const Wektor& rhs)
{
    for (size_t i = 0; i < lhs.size(); i++)
        lhs[i] += rhs[i];
    return lhs;
}
```

Przez **wartość**!

– Możliwe użycie:

```
int main()
{
    Wektor v{5};
    Wektor w = v + v + v + v;
}
```


Konstruktor przenoszący:

```
Wektor(Wektor&& rhs)
{
    _size = rhs._size;
    _data = rhs._data;
    rhs._size = 0;
    rhs._data = nullptr;
}
```

**Przeniesienie
„prawa własności”
do danych**

**„wyzerowanie”
oryginału (tu: rhs)**

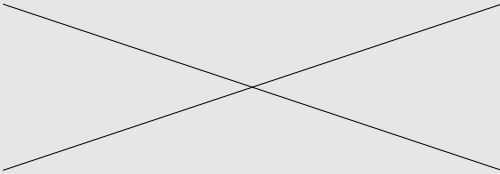
- Zawsze dwie fazy:
 - Zmiana właściciela danych
 - „wyzerowanie” oryginału (dlatego nie używa się `const &&`)

Efekt:

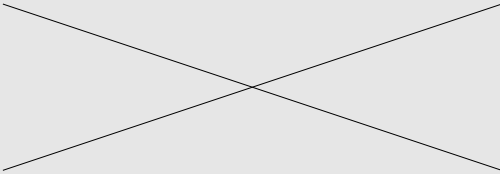
```
int main()
{
    Wektor v{5};
    Wektor w = v + v + v + v;
}
```

- Z użyciem konstruktora &&:
 - 1 wywołanie konstruktora kopii
 - 3 wywołania konstruktora przenoszącego
- Bez konstruktora &&:
 - 4 wywołania konstruktora kopii

Jeśli zdefiniuję konstruktor przenoszący

	<i>Argument ma adres (l-wartość), np. x</i>	<i>Argument nie ma adresu (r-wartość), np. wartość zwracana przez funkcję</i>
Przez wartość	Kopiowanie	Przenoszenie
Przez &	Przekazanie adresu	
Przez const&	Przekazanie adresu	Przenoszenie i przekazanie adresu

Jeśli nie zdefiniuję konstruktora przenoszącego

	<i>Argument ma adres (l-wartość), np. x</i>	<i>Argument nie ma adresu (r-wartość), np. wartość zwracana przez funkcję</i>
Przez wartość	Kopiowanie	Kopiowanie Przenoszenie
Przez &	Przekazanie adresu	
Przez const&	Przekazanie adresu	Kopiowanie Przenoszenie i przekazanie adresu

Przypisanie jest podobne do tworzenia

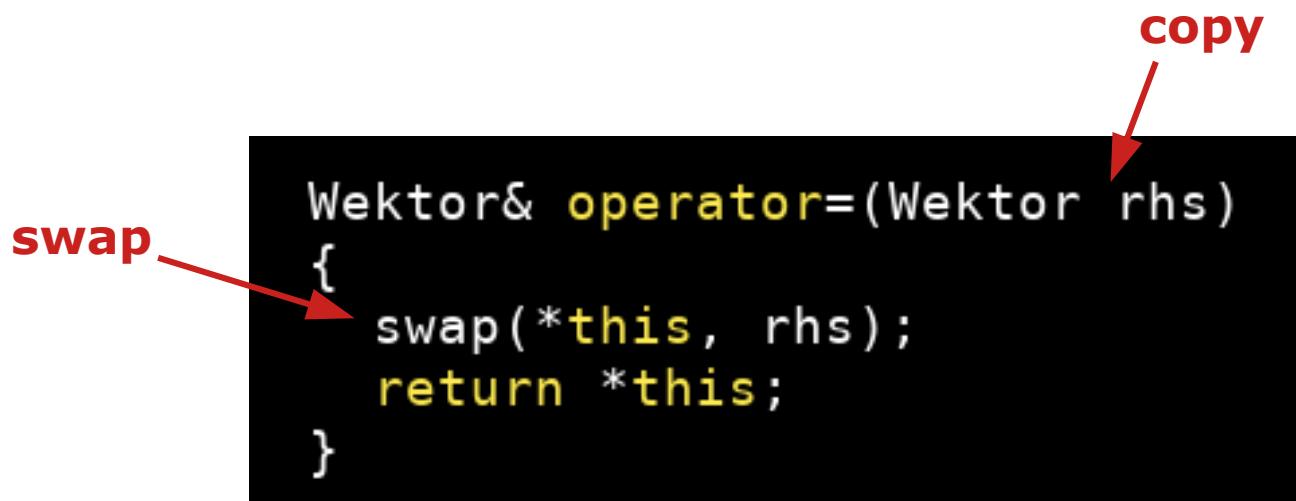
```
int x;
```

```
int y = x; // konstruktor kopiujacy
```

```
y = x;      // przypisanie
```

- Przeniesienie zasobów z obiektu tymczasowego miałoby tu sens

Idiom kopiuj-i-zamień (copy-and-swap)



The diagram illustrates the copy-and-swap idiom implementation for a C++ vector. It features a code block with the following C++ code:

```
Wektor& operator=(Wektor rhs)
{
    swap(*this, rhs);
    return *this;
}
```

Two red arrows point to specific parts of the code to highlight the steps of the idiom:

- A red arrow labeled **copy** points to the parameter `rhs` in the function signature `Wektor rhs`.
- A red arrow labeled **swap** points to the `swap(*this, rhs);` line inside the function body.

- Uniwersalna implementacja (jeśli mamy swap)

swap

```
friend void swap(Wektor & lhs, Wektor & rhs)
{
    std::swap(lhs._data, rhs._data);
    std::swap(lhs._size, rhs._size);
}
```

- swap na pewno nie zgłosi wyjątku
 - swap może być definiowane jako funkcja swobodna, jako friend w klasie, jako składowa

Strong exception safety

swap: nie zgłosi wyjątku

```
Wektor& operator=(Wektor rhs)
{
    swap(*this, rhs);
    return *this;
}
```

copy: może zgłosić wyjątek

- Nawet jeżeli podczas wykonywania
 $v = w;$
zgłoszony zostanie wyjątek, to żaden
z argumentów nie ulegnie modyfikacji

Dygresja: konstruktor przenoszący i *swap*

```
Wektor(Wektor&& rhs) noexcept  
: Wektor()  
{  
    swap(*this, rhs);  
}
```

Inicjalizacja **this*
konstruktorem
bezargumentowym

```
Wektor()  
:_data(nullptr), _size(0)  
{ }
```

- swap można użyć do **idiomatycznej** definicji obu podstawowych operacji przenoszenia: konstrukcji i kopiowania.
- Powyższy konstruktor jest nieco mniej wydajny od napisanego „ręcznie”

Dygresja: wyjątki i gwarancje

- **No-throw guarantee**: operacja nie zgłosi wyjątku
- **Strong exception safety**:
 - Jeśli podczas operacji zostanie zgłoszony wyjątek, to zostaje przywrócony stan programu sprzed próby wykonania tej operacji
- **Basic exception safety**:
 - Zgłoszenie wyjątku nie spowoduje wycieku zasobów, a zaangażowane obiekty będą w jakimś poprawnym stanie
- **No exception safety**: brak gwarancji

resize, push_back etc.

- Operacja `resize` może wymagać zmiany lokalizacji danych
- To kolejne miejsce, w którym warto zastosować przenoszenie
- Standard nakłada na `push_back`, `resize`, etc. spełnianie ***strong exception safety***
- To jest możliwe tylko wtedy, gdy **konstruktor przenoszący** daje gwarancję ***no-throw***

Dygresja: wyjątki i gwarancje

- Żadna klasa **zarządzająca zasobami** nie może spełnić **no-throw guarantee** (bo zasoby z definicji są skończone)
- Kontenery biblioteki standardowej gwarantują jedynie **strong exception safety**

Konstruktor przenoszący: **noexcept**

```
Wektor(Wektor&& rhs) noexcept
{
    _size = rhs._size;
    _data = rhs._data;
    rhs._size = 0;
    rhs._data = nullptr;
}
```

- Modyfikator **noexcept** daje gwarancję umowną: kompilator może założyć, że funkcja nie zgłosi wyjątku, a jeśli zgłosi, to jest to problem programisty.

Gdzie stosuje się przenoszenie?

- Przekazywanie „tłustych” obiektów jako wartości funkcji przez wartość
`return std::vector<int> (1000000);`
- Przekazywanie „tłustych” obiektów do funkcji przez wartość, jeżeli w funkcji potrzebujemy kopii argumentu
- Konstruktor przenoszący
`X(X&& other)`
- operator= `(class X &&)`

Reguła trzech i reguła pięciu

- Jeżeli klasa zarządza zasobem, to zdefiniuj w niej:

		C++98	C++11
Konstruktor kopiujący	T(const T&)	tak	tak
Kopiujący operator=	T& operator=(const T&)	tak	tak
Konstruktor przenoszący	T(T&&)	—	tak
Przenoszący operator=	T& operator=(T&&)	—	tak
Destruktor	~T()	tak	tak

Reguła trzech i reguła pięciu

- Reguła trzech – poprawność działania
- Reguła pięciu – efektywność

		C++98	C++11
Konstruktor kopiujący	T(const T&)	tak	tak
Kopiujący operator=	T& operator=(const T&)	tak	tak
Konstruktor przenoszący	T(T&&)	—	tak
Przenoszący operator=	T& operator=(T&&)	—	tak
Destruktor	~T()	tak	tak

The rule of zero

- Jeśli klasa nie zarządza zasobami, to nie powinna mieć zdefiniowanej żadnej składowej z „wielkiej piątki”
- Kompilator wygeneruje automatycznie odpowiednie funkcje z „wielkiej piątki”

```
class X
{
    int x;
    std::vector<int> v;
    std::strings;
public:
    ...
}
```

Copy elision

- Jest to całkowite pomijanie wywołania konstruktora kopiującego i/lub przenoszącego
- *"zero-copy pass-by-value semantics"*
- W niektórych kontekstach wymagane przez standard, w niektórych – opcjonalne
 - https://en.cppreference.com/w/cpp/language/copy_elision

Copy elision

```
#include <iostream>
```

```
#include <vector>
```

```
struct X {
```

```
    X() { std::cout << "constructed\n"; }
```

```
    X(const X&) { std::cout << "copy-constructed\n"; }
```

```
    X(X&&) { std::cout << "move-constructed\n"; }
```

```
    ~X() { std::cout << "destroyed\n"; }
```

```
};
```

```
std::vector<X> f() {
```

```
    std::vector<X> v = std::vector<X>(3); // copy elision
```

```
    return v; // NRVO (prawdopodobnie)
```

```
}
```

```
void g(std::vector<X> arg) {
```

```
    std::cout << "arg.size() = " << arg.size() << '\n';
```

```
}
```

```
int main() {
```

```
    std::vector<X> v = f(); // copy elision
```

```
    g(f()); // copy elision
```

```
}
```

Copy elision (i NRVO)

```
#include <iostream>
#include <vector>

struct X {
    X() { std::cout << "constructed\n"; }
    X(const X&) { std::cout << "copy-constructed\n"; }
    X(X&&) { std::cout << "move-constructed\n"; }
    ~X() { std::cout << "destroyed\n"; }
};

std::vector<X> f() {
    std::vector<X> v = std::vector<X>(3); // copy elision
    return v; // NRVO (prawdopodobnie)
}

void g(std::vector<X> arg) {
    std::cout << "arg.size() = " << arg.size() << '\n';
}

int main() {
    std::vector<X> v = f(); // copy elision
    g(f()); // copy elision
}
```

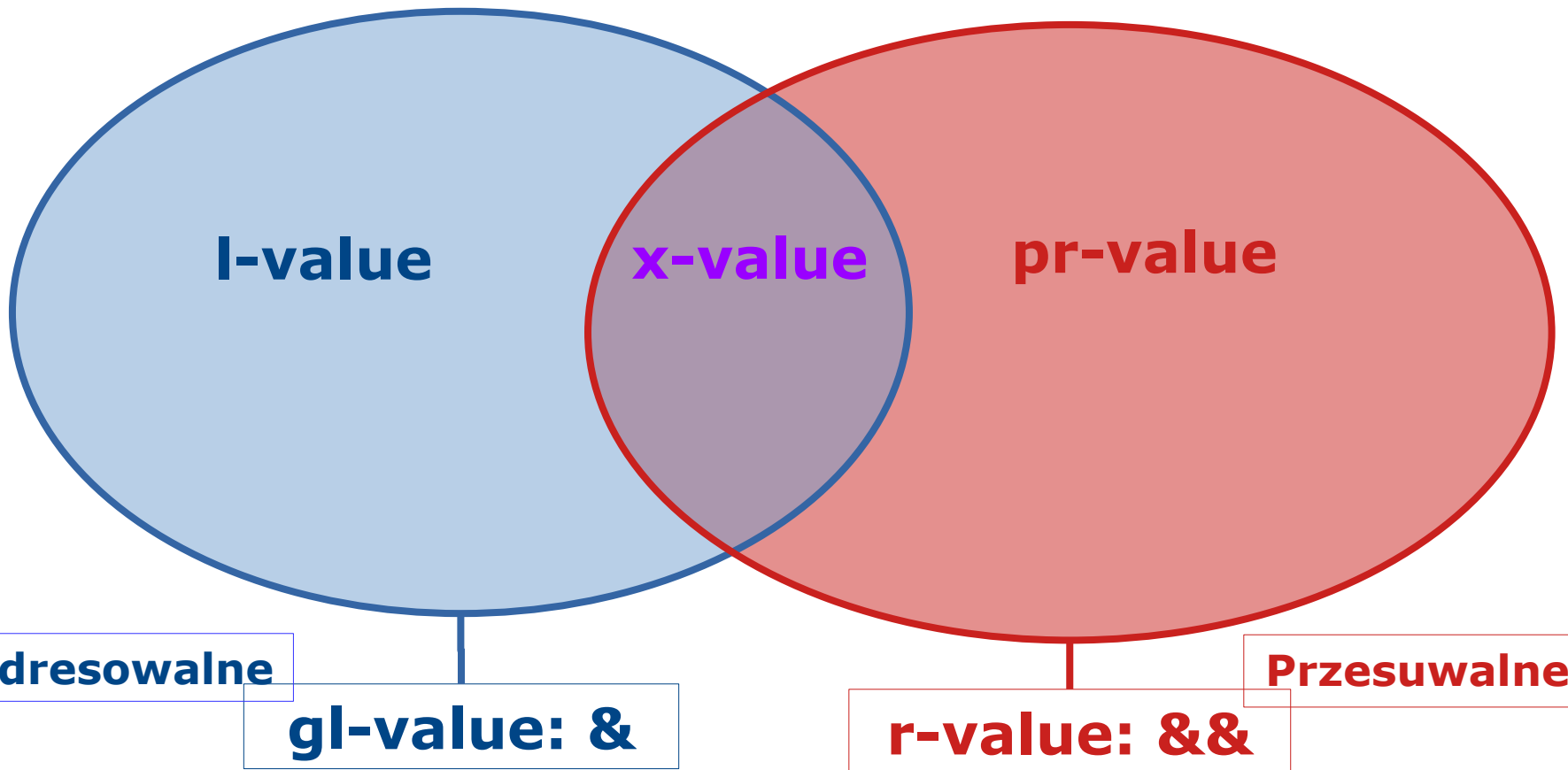
0 wywołań

constructed
constructed
constructed
constructed
constructed
constructed
arg.size() = 3
destroyed
destroyed
destroyed
destroyed
destroyed
destroyed

Named return
value optimization

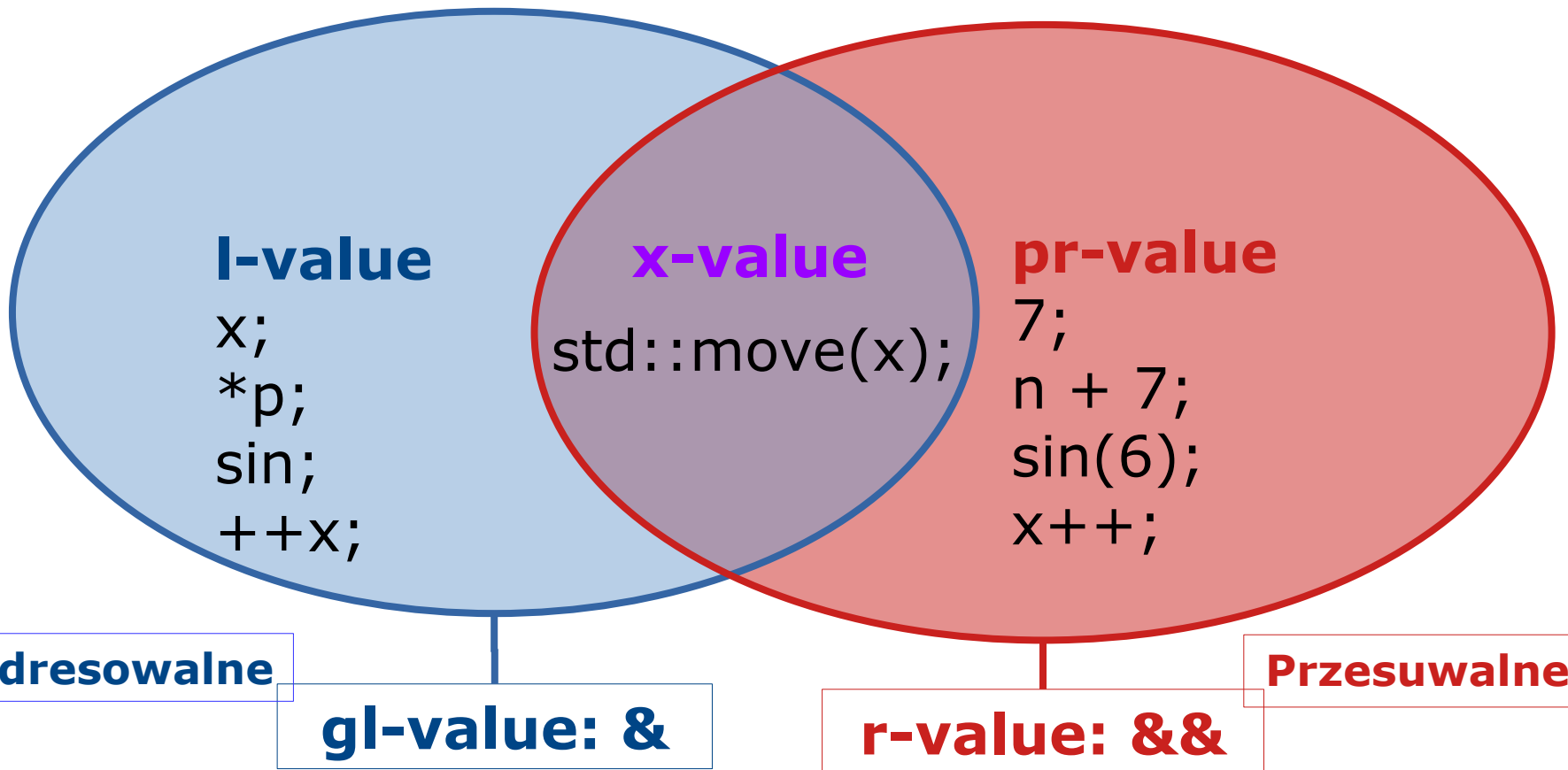
5 kategorii

Każde **wyrażenie** ma **typ**, np. **int**, i **kategorię**, np. **l-value**



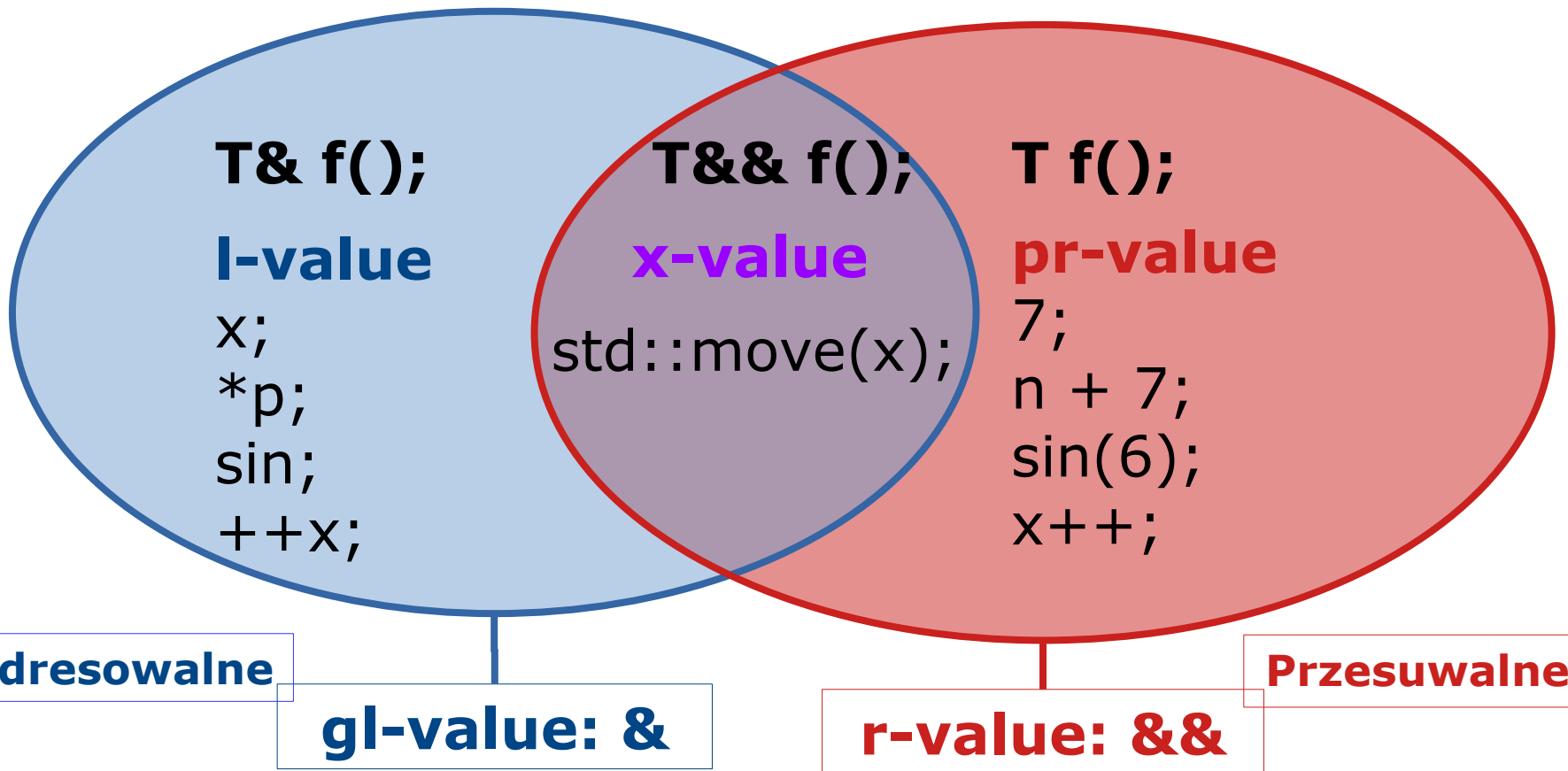
5 kategorii

Każde **wyrażenie** ma **typ**, np. int, i **kategorię**, np. **l-value**



5 kategorii

Każde **wyrażenie** ma **typ**, np. `int`, i **kategorię**, np. **l-value**



std::move

- `std::move(x)` oznacza: można traktować `x` jak obiekt bezadresowy (tymczasowy)

```
int main()
{
    std::string s = "Ala ma kota";
    std::string ss = std::move(s);
    std::cout << "s = \"\" << s << "\"\n";
    std::cout << "ss = \"\" << ss << "\"\n";
}
```



```
s = ""
ss = "Ala ma kota"
```


&& jako referencja uniwersalna

- `void f(auto && n);`
`f(7);` // n jest r-referencją do 7 (&&)
`f(m);` // n jest l-referencją do m (&)
- `template<typename T>`
`void f(T && n);`
`f(7);` // n jest r-referencją do 7 (&&)
`f(m);` // n jest l-referencją do m (&)
- Referencja && jest uniwersalna, gdy typ jest dedukowany

Zmiana „paradygmatu”

- C++98:
 - Nigdy nie zwracaj „tłustych” obiektów przez wartość
 - Obiekty przekazuj wyłącznie przez (najlepiej stałą) referencję
- C++11:
 - Śmiało zwracaj „tłuste” obiekty przez wartość
 - Przekazanie obiektu przez wartość może prowadzić do najprostszego i najbardziej efektywnego kodu

Podsumowanie

- R-referencja (&&) ułatwia:
 - **Optymalizację działań na obiektach tymczasowych** (wartości operatorów i funkcji, obiekty nienazwane, wyrażenia lambda, literały, etc.)
 - Optymalizacja polega na **przenoszeniu uchwytów** do zasobów a nie samych zasobów
 - Przenoszenie wykonywane jest automatycznie tylko wtedy, gdy **z pewnością jest bezpieczne**
 - Optymalizację „niepewną” można wymusić przez **std::move**, które zmienia kategorię (typ) wyrażenia na „przenoszalny” ("x-value")

Podsumowanie

- Nie musisz używać r-referencji (&&)
 - Programy sprzed C++11 wciąż się kompilują
 - Po prostu twoje programy mogą działać nieco wolniej i potrzebować nieco więcej pamięci
 - Mimo wszystko będziesz widzieć && w komunikatach kompilatora i dokumentacji języka

Podsumowanie

- Używaj `&&`, gdy zależy ci na wydajności
 - Rozszerz „wielką trójkę” do „wielkiej piątki” (przenoszący konstruktor i operator=), jeżeli twoja klasa bezpośrednio zarządza zasobami
 - Można przeciągać `&&` dla innych funkcji niż konstruktor przenoszący i operator=, ale w praktyce są to niezwykle rzadkie przypadki
 - Pamiętaj o „swap idiom” w operator=
 - Konstruktor przenoszący: z atrybutem `noexcept`
 - Pamiętaj o *copy elision* i NRVO
 - Używaj funkcji zwracających obiekty przez wartość

Podsumowanie (c.d.)

- Nie używaj `std::move` w instrukcji `return`
- `std::move` niczego nie przenosi (zmienia typ)
- Pamiętaj, co to są referencje uniwersalne