
# C++11/14

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
AGH University of Science and Technology

2018-11-15

autor Miłosz Filus

# Plan prezentacji

- Semantyka przenoszenia
- Dedukcja typów
- Perfect Forwarding
- Zmiany w STL
- Inteligentne wskaźniki

# Semantyka przenoszenia

# Wstęp

Semantyka przenoszenia wprowadza udogodnienie w nowym standardzie pozwalając programiście na łatwiejszą kontrole pamięci z dostarczonego już interfejsu. Ponadto w pewnych miejscach kompilator sam ją wykorzystuje.

Operacje przenoszenia pozwalają na oszczędność pamięci poprzez brak składowania w pamięci niepotrzebnych kopii oraz na optymalizacje szybkości programu, gdzie zamiast kopiowania użyjemy przenoszenia.

# Kategorie wartości w C++

r – wartość (rvalue)

» Nie możemy pobrać z nich adresu w pamięci &

» Obiekt tymczasowy (bez nazwy)

```
int(5); x + 1; 7;
&int(5) - błąd!; &(x + 1) - błąd!; &7 - błąd!;
```

l – wartość (lvalue)

» Możemy pobrać adresy w pamięci &

» Możemy do nich przypisach wartości gdy nie są const (użyc operatora przypisania)

```
int x;
&x - OK!
```

# Rodzaje referencji

Referencja do l-wartości

```
int x;
int& rx = x;
```

Referencja do r-wartości

```
int&& rvalueRef = 7;
int&& rvalueRef2 = int(1);
```

Referencja uniwersalna (TYLKO PRZY PEŁNEJ DEDUKCJI TYPÓW)

```
auto&& universal1 = x;
auto&& universal2 = 7;
```

Wyjątek! const T& - przedłużanie żywotności obiektów tymczasowych

```
const int& e1 = x; // OK!
const int& e2 = 7; // TEŻ OK!
```

# Nowość w C++11 - Przenoszenie

- Konstruktor przenoszący
- Przenoszący operator przypisania
- Główna idea

```
struct A {
A(A&& a);
A& operator=(A&& a);
};
```

```
A(A&& a) {
   m_memoryPointer =     a.m_memoryPointer;
   a.m_memoryPointer = nullptr;
}
```

```
A& operator=(A&& a) {
   if (this != &a) {
      m_memoryPointer = a.m_memoryPointer;
      a.m_memoryPointer = nullptr;
   }
}
```

# Jak wywołać operacje przenoszenia?

- Musimy operować na obiekcie r-wartościowym – czyli automatycznie obiekty tymczasowe
- A co jeśli chcemy ‘przenieść’ obiekt który jest l-wartościowy?
- Musimy zrzutować nasz obiekt na obiekt r-wartościowy
- Z pomocą przychodzi funkcja szablonowa std::move zawarta w pliku nagłówkowym <utility>
- Jej działanie polega na rzutowaniu dostarczonego obiektu na obiekt r-wartościowy

# Zasada 3 (Rule of three) (do C++03)

Kompilator automatycznie generuje dla nas (jeśli go sami nie zdefiniujemy)

- Operator przypisania
- Konstruktor kopiujący
- Destruktor

```
struct A {
    A(int val):m_val(val){}
    int m_val;
};
```

```
A a(5);
A ac(a); //konstruktor kopiujący
a = ac; //operator przypisania
```

# Zasada 5 (Rule of five) (od C++11)

Dodatkowo generowane

- Konstruktor przenoszący
- Przenoszący operator przypisania

Uwaga!

- Jeśli zdefiniujemy własny konstruktor kopiujący lub przenoszący operator przypisania – reszta funkcji się nie wygeneruje.
- Jeżeli również, zdefiniujemy jedną z funkcji wymienionych w zasadzie 3, elementy przenoszące również nie zostaną wygenerowane
- Konstruktor kopiujący i przenoszący operator przypisania nie są od siebie niezależne – Gdy zdefiniujemy jeden z nich, również musimy drugi… Albo?

# …Albo jawna deklaracja funkcji domyślnych

- Jeśli w naszej klasie nie ma potrzeby pisania własnej definicji jednej z funkcji z zasady 5 – np. gdy używamy sprytnych wskaźników i nie musimy się martwić destruktorem
- Możemy wtedy kazać kompilatorowi wygenerować domyślne funkcje lub zabronić kompilatorowi ich generowania
- Służą do tego słowa kluczowe default i delete

```
~A() = default;
A(const A&) = delete;
```

# Dedukcja typów

# Dedukcja typów w szablonach

```
template <typename T>
void f(ParametrType param);
```

- Trzy przypadki
- 1. ParametrType – Wskaźnik lub Referencja

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

- 2. ParametrType – Referencja Uniwersalna

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

- 3. Ani referencja, ani wskaźnik ->wartość przekazywanie przez kopie

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

# Typ tablicowy?

- Gdy przekazujemy do funkcji tablice, a funkcja jest pierwszego przypadku ze wskaźnikiem, typ jest sprowadzany do wskaźnika
- W przypadku przekazywania przez referencje, typ zostaje jako typ tablicowy

```
int tablica[5];
template <typename T>
void f(T& param); //-> ParametrType to int(&)[5]
template <typename T>
void f(T* param); //-> ParametrType to int*
```

# auto

- Działa w analogiczny sposób jak dedukcja typów w szablonach
- Również podział na 3 przypadki
- Musi być nadanie wartości przy deklaracji (z czegoś ten typ musi być wydedukowany)
- Pełna kontrola typów zachowana, kompilator dopasowuje typ
- Wyjątek initializer_list

# Zalety auto

- Odporność na zmiany kodu

```
//std::vector<int> container;
std::set<int> container;
for (auto it = container.cbegin(); it != container.cend(); ++it);
```

- Ułatwienie pracy z wyrażeniem lambda oraz z długimi, trudnymi typami

# decltype

- Pozwala dokładnie określić typ zmiennej
- Efektywne połączenie z auto
- Dedukcja typu zwracanego przez funkcje
- Nie traci kategorii typu wartości, rvalue przekazane do decltype pozostaje rvalue, tak samo z lvalue

# Perfect Forwarding

# Perfect Forwarding

- Szablonowa funkcja std::forward – działanie zbliżone do funkcji std::move
- W przeciwieństwie do std::move gdzie, zawsze zachodzi rzutowanie do r-wartości, w std::forward jest podejmowana decyzja czy rzutować do l-wartości czy r-wartości w zależności od kategorii wartości dostarczonej w argumencie
- Ma to zastosowanie tam gdzie argumenty są pobierane poprzez referencje uniwersalną, a następnie wywołujemy inną funkcje z tymi argumentami. Może to mieć kluczowe znaczenie, np. czy zostanie wywołany konstruktor kopiujący czy konstruktor przenoszący oraz to co się stanie z dostarczonymi obiektami

# parameter pack

- Jest to typ parametru szablonu, który może przyjąć od 0 do dowolnej liczby argumentów
- Możliwość zmiany dostarczonych typów oraz modyfikacja wartości dostarczonych parametrów operując na całym packu
- Brak trywialnego dostępu do argumentów (brak mechanizmu rozpakowywania)

# działanie operatora …

- Po typename informuje że parametrem szablonu jest parameter pack
- Po nazwie typu

```
template <typename... T>
class A : public T... // -> class A : public T1, public T2,...,public TN
{}
```

- Po nazwie typu, gdy jest nazwa tego parameter packa (args)

```
template <typename... T>
void f(T... args) // -> f(T1 arg1, T2 arg2, T3 arg3,..., TN argN)
{}
```

- W innych przypadkach powtarza pattern przed …

```
    (args...); // -> (arg1, arg2, arg3,...,argn)
((args...) + args...) // -> ((arg1, arg2,...,argn) + arg1 ,
(arg1,arg2,...,argn) + arg2 , ... , (arg1,arg2,...,argn) + argn)
```

# Variadic templates

- Opiera się na parameter packu
- Najczęściej dostarczamy minimum jeden argument, by parameter pack nie był pusty
- Użycie variadic templates głównie opiera się na rekurencji
- Możemy używać variadic templates również przy dziedziczeniu

# Zmiany w STL

# emplace w kontenerach

- Tworzy obiektu ‘w miejscu’ z pobranych parametrów ‘przepuszczonych’ przez perfect forwarding
- Nie tworzy obiektu tymczasowego, który by było trzeba przenieść – tak działa push
- Metoda szybsza od zwykłego push, ponieważ gdy w push tworzymy nowy obiekt, musi się to odbyć przez obiekt tymczasowy, nie mamy możliwości przesłania parametrów do stworzenia go w miejscu

# std::chrono

- Nowoczesne rozwiązania co do mierzenia czasu
- Dostarcza większą dokładność
- Większe możliwości

Podział na zawarte klasy

- durations – miary czasu
- time points – punkty czasowe, można używać duration do policzenia ile czasu minęło między danymi punktami
- zegary
- Upływ czasu możemy odczytywać w wygodnym dla nas formacie, co dostarcza nagłówek

# std::tuple

- Zasada działania bardzo podobna do krotki pythonowej
- Możliwość trzymania różnych typów w jednej strukturze
- Dostęp do elementów poprzez funkcje szablonową
- Implementacja oparta na varadic templates
- Kiedy używamy krotek?

# std::random

- Profesjonalne narzędzie do generacji liczb pseudo losowych
- Możliwość wybrania dystrybucji i np. generowanie liczb po dystrybucji rozkładu normalnego
- Dużo większe możliwości od ctime, eliminacja niedokładności która występuje w ctime

# A no dlatego

- Zakres losowania dla rand() to [0,32767]
- Modulo doprowadza do następującej sytuacji

```
int number = rand();
int newNumber = number % 100;
// number znajduje sie w przedziale [0,99], modulo dziala dobrze,
zwraca liczbe od 0 do 99
// number znajduje sie w przedziale [100,199], modulo dziala dobrze,
zwraca liczbe od 0 do 99
// i tak dalej az
// number znajdue sie w przedziale [32700,32767], modulo sie
wywraca, zwraca liczbe od 0 do 67
```

- Przez co rozkład nie jest jednostajny
- Więcej wad… Link do całego wykładu na ten temat w źródłach

# Sprytne wskaźniki

# std::unique_ptr

- Lekki i szybki
- Jego instancja, może zajmować tyle pamięci co wskaźnik surowy!
- Własny destruktor dostarczamy szablonowo
- Nie można go skopiować, tylko przenoszenie
- Może wskazywać na tablicę
- Znajduje się w pliku nagłówkowym <memory>
- Semantyka wyłącznego posiadania
- Konwertowalny na shared_ptr

```
void deleter(int* ptr) {
   delete ptr;
}
```

```
std::unique_ptr<int> ptr(new int(5)); //rozmiar pojedynczego wskaźnika
std::unique_ptr<int[]> arr_ptr(new int[5]); //rozmiar pojedynczego wskaźnika
std::unique_ptr<int, void(*)(int*)> ptr_withDeleter(new int(5), deleter);
//rozmiar dwóch wskaźników
```

# Dobre praktyki używania unique_ptr

- Używanie std::make_unique do tworzenia obiektów, pozwala na używanie perfect forwardingu – minusem jest to, że nie obsługuje customowego deletera oraz nie może przetrzymywać tablic
- Brak potencjalnego wycieku pamięci, przy użycie make_unique
- Uwaga! std::make_unique został dodany w C++14

```
std::unique_ptr<int> ptr = std::make_unique<int>(5); //
tworzenie nowego obiektu poprzez perfect forwarding, w
argumentach dostarczamy argumenty do wywolania konstruktora
```

# std::shared_ptr

- Można go kopiować
- Nieco wolniejszy od unique_ptr, ponieważ zlicza swoje kopie
- Jego rozmiar to dwa wskaźniki (surowy wskaźnik + wskaźnik na blok kontrolny)
- Posiada blok kontrolny, gdzie przechowuje licznik oraz deleter
- Własny destruktor przekazywany jako argument funkcji, nie szablonowo jak w unique_ptr
- Pamięć niszczona, gdy licznik osiągnie 0
- Możliwość stworzenia z weak_ptr lub unique_ptr

```
std::shared_ptr<int> ptr(new int(5));
std::shared_ptr<int> ptr(new int(5), deleter);
```

# Dobre praktyki używania shared_ptr

- Tak jak w przypadku unique_ptr, używanie std::make_shared, gdzie jest jeszcze lepiej zoptymalizowany, jeśli chodzi o blok kontrolny
- Tworzenie obiektów w miejscu przy użyciu perfect forwardingu
- Jedna alokacja pamięci przy użyciu make_shared
- Należy się zastanowić czy użyć make_shared, jeśli chcemy tworzyć wskaźnik na bardzo duży obiekt

```
std::shared_ptr<int> ptr = std::make_shared<int>(5);
//w argumentach parametry do konstruktora
```

# std::weak_ptr

- Słaby wskaźnik? A po co to komu?
- Nie jest samodzielnym inteligentnym wskaźnikiem
- Uzupełnienie wskaźnika shared_ptr
- Powinien być tworzony przy przy pomocy shared_ptr
- Służy do sprawdzania czy wskazywana wartość przez shared_ptr istnieje, czy już została skasowana – czyli czy shared_ptr wisi
- Ma swój licznik w bloku kontrolnym shared_ptr'a
- Jego licznik decyduje o dealokacji pamięci na blok kontrolny, gdy dojdzie do 0
- Powyższy warunek oczywiście sprawdzany po sprawdzeniu licznika shared_ptr
- Możemy z jego użyciem shared_ptr, funkcja lock()

```
std::shared_ptr<int> ptr = std::make_shared<int>(5);
std::weak_ptr<int> weakPtr(ptr);
ptr.reset();
std::cout << weakPtr.expired() << std::endl; // true
```

# A jak już o wskaźnikach mowa

- Stała NULL – nieścisłości, literał 0
- kluczowe słowo nullptr działa na zasadzie klasy z dopasowaniem typu wskaźnikowego za pomocą szablonowych operatorów
- Implementacja na zasadzie

```
struct nullptr{
template <typename T>
operator T*() const { return 0; }
};
```

- Używanie nullptr pozwala uniknąć nieścisłości

# Źródła

Cały napisany kod, w przykładach jak i na slajdach został napisany przeze mnie

Materiały z których korzystałem

http://www.cplusplus.com

https://en.cppreference.com

https://stackoverflow.com/

Scott Meyers – Skuteczny i nowoczesny C++

Wykład o random

https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful