Wojciech Olech (2017)

Dlaczego using namespace to zła praktyka

Zacznijmy od tego - co to jest przestrzeń nazw

Otóż przestrzeń nazw jest to **przestrzeń, w której trzymane są nazwy** - obviously. Nazwy zmiennych, funkcji, obiektów, typów i wielu innych rzeczy. W C nie było przestrzeni nazw. Wszystkie funkcje i struktury oraz typy były w przestrzeni globalnej, co powodowało że łatwo było o ich konflikt. Wystarczyło dołączyć bibliotekę, a potem - nie zdając sobie z tego sprawy - stworzyć funkcję która miała by jednakową nazwę (na przykład funkcja time z biblioteki time.h, a nuż uznamy że chcemy mieć funkcję albo strukturę time w swoim programie, co spowoduje potencjalny konflikt nazw). Problem ten został rozwiązany w C++'ie, gdzie wprowadzono pojęcie przestrzeni nazw. Dzięki temu programiści, pisząc biblioteki, nie muszą martwić się o to że gdzieś w kodzie może być jakaś funkcja która zupełnie przypadkiem będzie miała taką samą nazwę jak nasza. Wystarczy trzymać się konwencji stosowania przestrzeni nazw (nie tylko poprzez namespace ale również class, klasy też wprowadzają przestrzenie nazw), i nagle kod mógł stać się ładniejszy i bardziej zrozumiały. W C++ie mamy również keyword using. Ma on kilka różnych zastosowań, może na przykład wprowadzić nam w aktualną przestrzeń nazw nazwy z innej przestrzeni - using namespace. Wydaje się to być bardzo wygodne - możemy w końcu olać prefiks przestrzeni nazw, magiczne kilka literek. Ale czy na pewno jest to takie wygodne? Otóż, nie dość że nie do końca, to dodatkowo może nam przysporzyć wielu problemów, których początkujący programiści nie są świadomi.

Konflikty nazw

Przyjmijmy taki kod jako przykład:

```
#include <vector>
using namespace std;

struct vector { int x, y; };

int main() {
    vector vec;
}
```

Co się stanie kiedy go spróbujemy go skompilować? Otóż, *spróbujemy* to słowo klucz - kompilator powie nam, że nazwa vector jest dwuznaczna. Dlaczego? Dlatego, że dołączyliśmy do naszego kodu bibliotekę vector ze standardu, która zawiera klasę std::vector. Z racji, że using namespace wprowadza nam przestrzeń nazw std do przestrzeni globalnej, kompilator jest w stanie znaleźć nazwę std::vector jako vector. A my mamy strukturę vector w naszym kodzie. Kompilator nie wie którym vectorem ma się posłużyć i wyrzuca błąd. Zaznaczę tutaj, że std::vector jest typem szablonowym, czyli prawidłowa deklaracja jego obiektu to na przykład std::vector<int> vec;, ale kompilator na tym etapie kompilacji olewa to czy jest to typ szablonowy, czy nie, widzi dwie takie same nazwy i dostaje lekkiego pierdolca.

Czytelność kodu

Inną rzeczą, która powoduje że using namespace nie jest dobrą praktyką jest to, że psuje nam czytelność kodu. Przyjmijmy taki kod:

```
#include <iostream>
using namespace std;

int pow2(int x) {
    return x*x;
}

int main() {
    int x{};
    cin >> x;
    cout << "x to " << x << '\n';
    << "pow2 x to " << pow2(x) << endl;
}
```

Okej, wygląda legitnie. Tyle, że nie odróżniamy naszego kodu, od kodu z biblioteki standardowej. Dlaczego jest to problemem? Dlatego, że czytelność kodu to jeden ze wskaźników jego jakości. Dobry kod jest czytelny i jednoznaczny, oraz można go zrozumieć po prostu go czytając. Fakt, nie zawsze jesteśmy w stanie taki kod pisać, ale powinniśmy do tego dążyć. Jednoznaczny kod oznacza, że patrząc na nazwę funkcji czy zmiennej nie musimy domyślać się do czego ona służy albo co robi, bo powinno to być łatwe do zrozumienia. Powinniśmy również zdawać sobie sprawę z tego, skąd ta funkcja pochodzi. A tutaj, wprowadzając w przestrzeń globalną nazwy z biblioteki standardowej nie jest tak kolorowo, bo musimy się zastanawiać - czy nazwa funkcji, na przykład copy albo sort jest nasza, czy znajduje się w bibliotece standardowej (to porównanie jest trochę z dupy, patrz wyżej, ale jeśli dwie funkcje miały by taki sam zwracany typ i nazwę to kompilator nie zwróci błędu, tylko potraktuje to jako przeładowania - dlatego ta opcja może wystąpić). Nie jest to nic dobrego, dlatego też używanie using namespace w większych i poważniejszych kawałkach kodu nie jest najrozsądniejszym wyborem.

Alternatywy

No dobra, ale - co w przypadku jeśli przestrzenie nazw są troche dłuższe, niż std - na przykład losowa przestrzeń nazw z biblioteki boost - boost:multiprecision. Na szczęście, using jest w stanie nam tutaj pomóc, bez wprowadzania przestrzeni nazw w zakres globalny. Otóż, możemy zrobić alias - using mp = boost::multiprecision. W ten sposób możemy odwoływać się do mt zamiast boost::multiprecision w zakresie w którym zadeklarowaliśmy ten alias. Jest to przyjemne, bezpieczniejsze niż using namespace i praktyczne - taki sam manewr możemy na przykład zastosować z typami, na przykład żeby zastosować zasadę pojedynczej definicji (ODR - One Definition Rule). Na przykład using size_type = std::size_t, i mamy ładną pojedynczą definicję na typ, którą - w momencie w którym uznamy, że chcemy używać na przykład long zamiast std::size_t, zamieniamy w jednym miejscu i nie martwimy się o mozolny refractoring kodu. Innym wyjściem jest również wprowadzenie pojedynczej nazwy do zakresu, na przykład using std::cin, żeby odnosić się do std::cin bez std::. Jest to na pewno bezpieczniejsza opcja, niż wprowadzanie całej przestrzeni nazw, ale i tak sugeruję trzymać sie

stosowania std:: i innych prefiksów przestrzeni nazw, ze względu na czytelność kodu.

Oczywiście przy małych projektach, samplach kodu i tego typu rzeczach nie musimy się zazwyczaj martwić o wyżej wymienione rzeczy, ale - niestety - wiele osób pokazuje tą technikę nie omawiając problemów, jakie może przysporzyć, oraz alternatyw.