KURS JĘZYKA C++

7. PRZESTRZENIE NAZW

SPIS TREŚCI

CZYM JEST PRZESTRZEŃ NAZW

- Przestrzeń nazw to obszar, w którym umieszcza się różne deklaracje i definicje.
- Przestrzeń nazw definiuje zasięg, w którym dane nazwy będą obowiązywać i będą dostępne.
- Przestrzenie nazw rozwiązują problem kolizji nazw.
- Przestrzenie nazw wspierają modularność kodu.

DEFINICJA PRZESTRZENI NAZW

- Przestrzeń nazw tworzymy za pomocą słowa kluczowego `namespace`, ograniczając zawartość klamrami:

```
namespace przestrzeń
{
    // deklaracje i definicje
}
```

- Aby odnieść się do typu, funkcji albo obiektu umieszczonego w przestrzeni nazw musimy stosować kwalifikator zakresu *`przestrzeń`*`::` poza tą przestrzenią.
- Funkcja `main()` musi być globalna, aby środowisko uruchomieniowe rozpoznało ją jako funkcję specjalną.
- Do nazw globalnych odnosimy się za pomocą pustego kwalifikatora zakresu `::`, na przykład `::wspolczynnik`.
- Jeśli w przestrzeni nazw zdefiniujemy klasę to do składowej statycznej w takiej klasie odnosimy się kwalifikując najpierw nazwą przestrzeni a potem nazwą klasy *`przestrzeń`*`::`*`klasa`*`::`*`składowa`*.

DEFINICJA PRZESTRZENI NAZW

- Przykład przestrzeni nazw:

```
namespace wybory
{
    int min2 (int, int);
    int min3 (int, int, int);
}

int wybory::min2 (int a, int b)
    { return a<b ? a : b; }
int wybory::min3 (int a , int b , int c)
    { return min2(min2(a,b),c); }

int min4 (int a, int, b, int c, int d)
{
    return wybory::min2(
        wybory::min2(a,b),
        wybory::min2(c,d));
}
```

DEKLARACJA UŻYCIA

- Deklaracja użycia wprowadza lokalny synonim nazwy z innej przestrzeni nazw (wskazanej nazwy można wówczas używać bez kwalifikowania jej nazwą przestrzeni).
- Deklaracja użycia `using` ma postać:
  `using` *`przestrzeń::symbol;`*
- Deklaracja użycia obowiązuje do końca bloku, w którym wystąpiła.
- Deklaracje użycia stosujemy w celu poprawienia czytelności kodu.
- Deklaracje użycia należy stosować tak lokalnie, jak to jest możliwe.
- Jeśli większość funkcji w danej przestrzeni nazw korzysta z jakiejś nazwy z innej przestrzeni, to deklaracje użycia można włączyć do przestrzeni nazw.

DYREKTYWA UŻYCIA

- Dyrektywa użycia udostępnia wszystkie nazwy z określonej przestrzeni nazw.
- Dyrektywa użycia `using namespace` ma postać:
  `using namespace` *`przestrzeń;`*
- Dyrektywy użycia stosuje się najczęściej w funkcjach, w których korzysta się z wielu symboli z innej przestrzeni nazw przestrzeń niż ta funkcja jest zdefiniowana.
- Globalne dyrektywy użycia są stosowane do transformacji kodu i nie powinno się ich stosować do innych celów.
- Globalne dyrektywy użycia w pojedynczych jednostkach translacji (w plikach `.cpp`) są dopuszczalne w programach testowych czy w przykładach, ale w produkcyjnym kodzie jest to niestosowne i jest uważane za błąd.
- Globalnych dyrektyw użycia nie wolno stosować w plikach nagłówkowych!

ANONIMOWE PRZESTRZENIE NAZW

- Anonimową przestrzeń nazw tworzymy za pomocą słowa kluczowego `namespace` bez nazwy, ograniczając zawartość klamrami:

```
namespace
{
    // deklaracje i definicje
}
```

- Anonimowa przestrzeń nazw zastępuje użycie deklaratora `static` przy nazwie globalnej – dostęp do nazw zdefiniowanych w przestrzeni anonimowej jest ograniczony do bieżącego pliku.
- Dostęp do anonimowej przestrzeni nazw jest możliwy dzięki niejawnej dyrektywie użycia.

```
namespace $$$
{
    // deklaracje i definicje
}
using namespace $$$;
```

W anonimowej przestrzeni nazw `$$$` jest unikatową nazwą w zasięgu, w którym jest zdefiniowana ta przestrzeń.

POSZUKIWANIE NAZW W PRZESTRZENIACH NAZW

- Gdy definiujemy funkcję z jakiejś przestrzeni nazw (przed nazwą definiowanej właśnie funkcji stoi kwalifikator przestrzeni) to w jej wnętrzu dostępne są wszystkie nazwy z tej przestrzeni.
- Funkcja z argumentem typu `T` jest najczęściej zdefiniowana w tej samej przestrzeni nazw co `T`. Jeżeli więc nie można znaleźć funkcji w kontekście, w którym się jej używa, to szuka się jej w przestrzeniach nazw jej argumentów.
- Jeżeli funkcję wywołuje metoda klasy `K`, to pierwszeństwo przed funkcjami znalezionymi przez typy argumentów mają metody z klasy `K` i jej klas bazowych.

ALIASY PRZESTRZENI NAZW

- Jeżeli użytkownicy nadają przestrzeniom nazw krótkie nazwy, to mogą one spowodować konflikt. Długie nazwy są niewygodne w użyciu. Dylemat ten można rozwiązać za pomocą krótkiego aliasu dla długiej nazwy przestrzeni nazw.
- Aliasy dla przestrzeni nazw tworzymy za pomocą słowa kluczowego `namespace` z dwiema nazwami
  `namespace krótka = długa_nazwa_przestrzeni;`
- Przykład:

```
namespace American_Telephone_and_Telegraph
{
    // tutaj zdefiniowano Napis
}
namespace ATT = American_Telephone_and_Telegraph;
American_Telephone_and_Telegraph::Napis n = "x";
ATT::Napis nn = "y";
```

- Nadużywanie aliasów może prowadzić do nieporozumień!

KOMPONOWANIE I WYBÓR

- **Interfejsy** projektuje się po to, by zminimalizować zależności pomiędzy różnymi częściami programu. Minimalne interfejsy prowadzą do systemów łatwiejszych do zrozumienia, w których lepiej ukrywa się dane i implementację, łatwiej się je modyfikuje oraz szybciej kompiluje.
- Eleganckim narzędziem do konstruowania interfejsów są przestrzenie nazw.

KOMPONOWANIE I WYBÓR

- Gdy chcemy utworzyć interfejs z istniejących już interfejsów to stosujemy **komponowanie** przestrzeni nazw za pomocą dyrektyw użycia, na przykład:

```
namespace His_string {
  class String { /* ... */ };
  String operator+ (const String&, const String&);
  String operator+ (const String&, const char*);
  void fill (char);
  // ... }
namespace Her_vector {
  template<class T> class Vector { /* ... */ };
  // ... }
namespace My_lib {
  using namespace His_string;
  using namespace Her_vector;
  void my_fct(String&);
}
```

- Dyrektywa użycia wprowadza do zasięgu wszystkie deklarację z podanej przestrzeni nazw.

KOMPONOWANIE I WYBÓR

- Teraz przy pisaniu programu można posługiwać się `My_lib`:

```
void f () {
  My_lib::String s = "Byron";
  // znajduje My_lib::His_string::String
  // ...
}

using namespace My_lib;
void g (Vector<String> &vs) {
  // ...
  my_fct(vs[5]);
  // ...
}
```

KOMPONOWANIE I WYBÓR

- Gdy chcemy utworzyć interfejs i dołożyć do niego kilka nazw z innych interfejsów to stosujemy **wybór** za pomocą deklaracji użycia, na przykład:

```
namespace My_string {
   using His_string::String;
   using His_string::operator+;
   // …
}
```

- Deklaracja użycia wprowadza do zasięgu każdą deklarację o podanej nazwie. Pojedyncza deklaracja użycia może wprowadzić każdy wariant funkcji przeciążonej.

KOMPONOWANIE I WYBÓR

- Łączenie komponowania (za pomocą dyrektyw użycia) z wyborem (za pomocą deklaracji użycia) zapewnia elastyczność potrzebną w praktyce. Z użyciem tych mechanizmów możemy zapewnić dostęp do wielu udogodnień, a zarazem rozwiązać problem konfliktu nazw i niejednoznaczności wynikających z komponowania.
- Nazwy zadeklarowane jawnie w przestrzeni nazw (łącznie z nazwami wprowadzonymi za pomocą deklaracji użycia) mają pierwszeństwo przed nazwami wprowadzonymi za pomocą dyrektyw użycia.
- Nazwę w nowej przestrzeni nazw można zmienić za pomocą instrukcji `typedef` albo `using` lub poprzez dziedziczenie.

ZAGNIEŻDŻONE PRZESTRZENIE NAZW I KLASY ZAGNIEŻDŻONE

- Wewnątrz przestrzeni nazw można zdefiniowań inną przestrzeń.
- Klasa tworzy lokalną przestrzeń nazw – domyślną dla składowych w tej klasie.
- W definicji klasy można umieścić definicję innego typu: klasy, struktury, wyliczenia czy też typu nazwanego za pomocą instrukcji `typedef`.

PRZESTRZENIE NAZW SĄ OTWARTE

- Przestrzeń nazw jest otwarta, co oznacza, że można do niej dodawać nowe pojęcia w kilku deklaracjach (być może rozmieszczonych w różnych plikach), na przykład:

```
namespace NS {
  int f(); // NS ma nową składową f()
}
namespace NS {
  int g(); // teraz NS ma dwie składowe
}          // f() i g()
```

- Definiując wcześniej zadeklarowaną składową w przestrzeni nazw, bezpieczniej jest użyć operatora zakresu niż ponownie otwierać przestrzeń (kompilator nie wykryje literówek w nazwie składowej), na przykład:

```
namespace NS {
  int h();
}
int NS::hhh () // błąd – brak NS::hhh
  { /*…*/ }
```

WERSJONOWANIE

- Szereg zmian w kolejnych wersjach w interfejsie jest kłopotliwy dla implementatorów.
- Rozwiązaniem tego problemu jest tworzenie podprzestrzeni dla każdej wersji i wybranie jednej (ostatniej) jako domyślnej za pomocą deklaracji `inline namespace`.
- Za pomocą śródliniowej przestrzeni nazw `inline namespace` można dokonać prostego wyboru między różnymi wersjami deklaracji w danej przestrzeni.
- Konstrukcja `inline namespace` jest intruzyjna, czyli zmiana domyślnej wersji (podprzestrzeni nazw) wymaga modyfikacji kodu nagłówka!

WERSJONOWANIE

- Przykład:

```
namespace Popular {
    // przestrzeń domyślna
    inline namespace v3 {
        int f(int);
        double f(double);
    }
    namespace v2 {
        int f(int);
        int g(double);
    }
    namespace v1 {
        int f(int);
    }
}
```

PRZESTRZEŃ NAZW `std`

- W języku **C++** wszystkie nazwy z biblioteki standardowej są umieszczone w przestrzeni nazw `std`.
- W języku **C** tradycyjnie używa się plików nagłówkowych i wszystkie nazwy w nich deklarowane są w przestrzeni globalnej (dostępne bez żadnych kwalifikacji).
- Aby zapewnić możliwość kompilowania przez kompilatory C++ programów napisanych w C przyjęto, że jeśli użyjemy tradycyjnej (pochodzącej z C) nazwy pliku nagłówkowego, to odpowiedni plik jest włączany i zadeklarowane w nim nazwy są dodawane do globalnej przestrzeni nazw. Jeśli natomiast ten sam plik nagłówkowy włączymy pod nową nazwą, to nazwy w nim deklarowane są dodawane do przestrzeni nazw `std`. Przyjęto przy tym konwencję, że pliki nagłówkowe z C nazwie `nazwa.h` są w C++ nazywane `cnazwa` (pary plików `<math.h>` i `<cmath>`, itp).

AUTOMATYZACJA KOMPILACJI - MAKE

AUTOMATYZACJA KOMPILACJI - MAKE

- Program **make** (program powłoki systemowej w systemie UNIX i Linux) automatyzuje proces kompilacji złożonych programów i bibliotek a przede wszystkim dużych projektów programistycznych; program **make** nadaje się również do innych prac, które wymagają przetwarzania wielu plików zależnych od siebie.
- Program **make** przetwarza dane w oparciu o reguły zapisane w pliku `makefile` albo `Makefile` albo wskazanego pliku z regułami za pomocą opcji `-f plik`.

LINKI DO MAKE

- https://cpp-polska.pl/