301

C++ - przestrzenie nazw

Czasem w plikach C++ zakładanych automatycznie używana jest *dyrektywa using*:

using namespace std;

Po co?

Ponieważ w C++ wszystko, co programista utworzy i wszystko, co zawarte jest w bibliotekach funkcjonuje w jakiejś *przestrzeni nazw*.

A my najczęściej chcemy używać tego, co należy do przestrzeni nazw **std**

Co to jest przestrzeń nazw?

302

C++ - przestrzenie nazw

- Przestrzenie nazw dzielą kod na logiczne jednostki. Stąd:
  przestrzeń nazw ↔ fragment kodu programu.
- Oczekiwania:
  wewnątrz jednej przestrzeni nazw wszystkie funkcje i klasy powinny logicznie uzupełniać się tworząc całość skierowaną na realizacje określonego celu, np. kompletną implementację złożonego algorytmu, bądź np. jeżeli mamy program łączący się przez internet – kompletną funkcjonalność związaną z tą łącznością:

```
namespace net_connect
{
  int make_connection();
  int test_connection();
  // itd...
}
```


303

C++ - przestrzenie nazw

- Poza tą przestrzenią można odwoływać się do jej funkcji i klas korzystając z prefixu takiego, jak nazwa przestrzeni oraz operatora zakresu **::**
- Przykład:

```
net_connect::make_connection();
```

Czemu ma służyć taki podział kodu?

- Podziałowi składowych programu na grupy, podobne do klas albo struktur.
- Przestrzenie nazw nie tworzą jednak żadnych obiektów, nie potrzebują tworzenia swoich instancji, żeby być używane. Po prostu aby użyć określonej funkcji piszemy tylko nazwę przestrzeni i nazwę tej funkcji.
- Ten podział pozwala na nadawanie tych samych nazw funkcjom należącym do różnych przestrzeni nazw, a więc chroni przed potencjalnymi konfliktami nazw w przypadku łączenia bibliotek.

304

C++ - przestrzenie nazw

Jeżeli nie chcemy za każdym razem pisać nazwy przestrzeni możemy się do niej odwołać przez słowo kluczowe *using*

Jeżeli w bloku kodu napiszemy

```
using namespace nazwa_przestrzeni
```

w obrębie tego bloku możemy odwoływać się do wszystkich funkcji należących do tej przestrzeni bez potrzeby podawania jej nazwy jako prefixu (jeżeli napiszemy tak na początku pliku – dostęp do funkcji obowiązuje w obrębie całego pliku)

Jeżeli chcemy odwoływać się tylko do wybranej funkcji z pewnej przestrzeni nazw możemy napisać:

```
using nazwa_przestrzeni::nazwa_funkcji
```


305

C++ - przestrzenie nazw

```
namespace A {
   const int two = 2;
   void napisz() {
   printf("A\n"); };
}
namespace B {
   void napisz()
   {printf("B\n"); };
}
namespace C {
   const int two = 22;
}
```

```
int main () {
   using A::napisz;
   using namespace C;
   napisz();
   B::napisz();
   printf("%i\n",two);

   return 0;
}
```

// co się pojawi w oknie konsoli?

306

C++ - przestrzenie nazw

A jeżeli chcemy w obrębie całego pliku korzystać ze wszystkich funkcji bibliotecznych, piszemy:

```
using namespace std;
```

ponieważ właśnie te funkcje są zadeklarowane w przestrzeni nazw **std**

C++ - przestrzenie nazw

Przestrzenie nazw to nowy element w C++.

W C też używamy plików nagłówkowych bibliotek standardowych, ale tam wszystkie nazwy w nich zadeklarowane są jednakowo dostępne bez żadnych kwalifikacji – nie ma tam mechanizmu przestrzeni nazw.

Aby zapewnić możliwość kompilowania programów w C przez kompilatory C++ przyjęto umowę:

jeżeli włączymy plik w sposób tradycyjny, podając pełną nazwę pliku nagłówkowego, jest on włączany oraz automatycznie otwierana jest nowa przestrzeń nazw **std**. Jeżeli ten sam plik włączymy pod nową nazwą, to przestrzeń std nie jest automatycznie otwierana.

Przykład:

C: `#include <string.h>` C++: `#include <cstring>`

C++ - przestrzenie nazw

Przyjęto konwencję, że pliki nagłówkowe o tradycyjnej nazwie „**nazwa.h**" są w C++ nazywane „**cnazwa**":

- `assert.h` `cassert`
- `limits.h` `climits`
- `stdio.h` `cstdio`
- `time.h` `ctime`
- `ctype.h` `cctype`
- `stdlib.h` `cstdlib`
- `math.h` `cmath`
- `...` `iostream (!)`
- …

W kodzie programu pisanego w C++ należy używać wersji przygotowanej dla C++.

C++ - przestrzenie nazw

- Cały kod, który został napisany poza jakąkolwiek przestrzenią nazw w rzeczywistości należy do tzw. globalnej przestrzeni nazw
- Jeżeli chcemy tylko zamknąć kod w swojej przestrzeni nazw, możemy napisać:

```
namespace
 {
  class Car
  {
   .... // składowe klasy Car
  }
  // inne składowe „przestrzeni nazw bez nazwy" (unnamed namespace)
 }
```


C++ - przestrzenie nazw

Kiedy używamy przestrzeni nazw bez nazwy, kompilator sam wymyśla dla niej unikalną nazwę

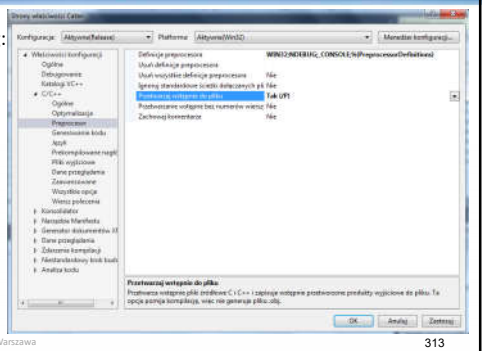
Po co używać takiej przestrzeni nazw?

Jeżeli deklarujemy struktury lub klasy, które będą używane tylko w lokalnym fragmencie kodu, mamy pewność, że ich nazwa nie wejdzie w kolizję z nazwą zadeklarowaną gdziekolwiek w globalnej przestrzeni nazw.

STANDARDOWE DYREKTYWY PREPROCESORA

Dyrektywy Preprocesora

Preprocessing:
Zbiór działań wykonywanych tuż przed kompilacją wskazanego pliku.
Aby zobaczyć wynik działania preprocesora należy (VS2017):



313

Dyrektywy Preprocesora

Standardowe dyrektywy dzielą się na trzy rodzaje:

1. Dołączające plik we wskazanym miejscu:
   `#include`
2. Makra:
   `#define`
3. Definiujące warunkową kompilację:
   `#if, #ifdef, #ifndef, #elif, #else, #endif`

Składnia i zasady stosowania

1. Zawsze na początku linii zaczynają się od #
2. Mogą wystąpić w każdym miejscu w programie
3. Mogą za nimi wystąpić komentarze
4. Zajmują jedną linię, chyba że jawnie wprowadzono znak kontynuacji

314

Dyrektywy Preprocesora

Dołączanie plików

Dołączane pliki powinny być do tego odpowiednio przygotowane. Przyjmuje się, że tak przygotowany przez programistę jest każdy plik z rozszerzeniem „h"

Polecenie:

`#include <filename>`

mówi preprocesorowi, aby zastąpił w/w linię zawartością wskazanego pliku.

Są dwa formaty:

1. **`#include "filename"`** – szuka pliku w bieżącym folderze, a jeżeli nie znajdzie, w folderach wskazanych w systemie (w systemowych zmiennych środowiskowych, np. PATH, INCLUDE, itp.) – *dla plików h użytkownika*
2. **`#include <filename>`** – szuka pliku tylko w folderach wskazanych w systemie (w systemowych zmiennych środowiskowych, np. PATH, INCLUDE, itp.) – *dla plików h bibliotecznych i systemowych*

315

Dyrektywy Preprocesora

Dyrektywa #define

Polecenie:

`#define nazwa` ***`dużo_różnego_tekstu`***

mówi preprocesorowi, aby każde wystąpienie **`nazwa`** zastąpił przez ***`dużo_różnego_tekstu`***.

Drugi argument, tj. ***`dużo_różnego_tekstu`*** jest opcjonalny. Jeżeli go nie ma, identyfikator **`nazwa`** istnieje, ale nie ma wartości. Taki identyfikator też może być przydatny – jako flaga (można sprawdzać, czy jest, albo czy go nie ma)

Polecenie:

`#undef nazwa`

mówi preprocesorowi, aby usunął („*oddefiniował*") **`nazwa`**.

316

Dyrektywy Preprocesora

Dyrektywa #define

1. Zastępuje tekst powtarzany w wielu miejscach kodu innym tekstem.
2. Najczęściej używana do definiowania stałych.
3. Stałe zwykle pisane są DRUKOWANYMI literami (*taka konwencja*), aby programistom było łatwiej je rozpoznać.

Przykład: to może powodować komunikat ostrzeżenia (*warning*):

```
#define SOME_VALUE 5
#define SOME_VALUE 6
```

A to pójdzie gładko:

```
#define SOME_VALUE 5
#undef  SOME_VALUE
#define SOME_VALUE 6
```


317

Dyrektywy Preprocesora

Makra

Tutaj nie wolno dać spacji (!)

Ogólna forma definicji makra:

```
#define macro(param1, param2, …) value
```

Przykład:

```
#define SUM(a, b) a + b
...
x = SUM(2, 6); /* zostanie zamienione na: "x = 2 + 6;" */
```

Jeżeli makro jest dłuższe, niż zaplanowana szerokość wiersza, można je łamać:

```
#define MIN(a, b) a < b \
      ? a : b
```


318

Dyrektywy Preprocesora

Makra

Łamanie wierszy:

```
#define INC(A)      \
   if ((A) < 100)   \
      (A)++;        \
   else             \
      printf ("Błąd: przepełnienie.\n");
```


Dyrektywy Preprocesora

Makra – przykłady:

```
#define SQUARE(x) ((x) * (x))

s = SQUARE(5);
```

staje się:

```
s = ((5) * (5));

s = SQUARE(a + 1);
```

staje się:

```
s = ((a + 1) * (a + 1));
```


Dyrektywy Preprocesora

Makra – przykłady:

```
#define MAX(a, b) \
((a) > (b) ? (a) : (b))

x = 4;
y = 9;
z = MAX(x, y);
```

staje się:

```
z = ((x) > (y) ? (x) : (y));
```


Dyrektywy Preprocesora

Makra – przykłady:

Stosowanie makra działa jak polecenie „znajdź i zastąp" w edytorze tekstu:

```
#define SQUARE1(x) x * x
#define SQUARE2(x) (x * x)
#define BAD_SUM(x, y) (x) + (y)
s = SQUARE1(a + 1);
/* s = a + 1 * a + 1; */
s = SQUARE2(a + 1);
/* s = (a + 1 * a + 1); */
s = BAD_SUM(a + 1, b + 2) * 3;
/* s = (a + 1) + (b + 2) * 3; */
```

Dlatego stosowanie nawiasów jest takie ważne (!)

Dyrektywy Preprocesora

Makra

Działanie preprocesora – zamiana symboli na wartości:

1. Argumenty wywołanych makr sprawdzane są, czy same nie reprezentują symboli zdefiniowanych za pomocą **`#define`**
2. Argumenty będące zdefiniowanymi symbolami są zastępowane wartościami
3. Zmodyfikowany program jest sprawdzany, czy występują w nim nadal symbole zdefiniowane za pomocą **`#define`**, i jeżeli tak – powrót do pkt 1.

Wyjątek: literały nie są skanowane, np.:

```
#define MAX 10
printf("Wartość MAX wynosi %d.\n", MAX);
```

Wyjście: Wartość MAX wynosi 10.

Dyrektywy Preprocesora

Makra to nie funkcje!

Ponieważ:

1. Są podmieniane tekstowo co może dać szybszy kod w przypadku prostych czynności.
2. Mogą być rekurencyjne.
3. Nie obejmuje ich kontrola typów podczas kompilacji.
4. Mogą generować bardzo skomplikowane błędy zachowania programu mimo poprawnej kompilacji.
5. Mogą mieć identyfikatory typów jako argumenty, np.:

```
#define PRINT_SIZE(type) \
printf("%d\n", (int) sizeof(type))
```

Dyrektywy Preprocesora

Makra łatwo dają efekty uboczne

Przykład:

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))

m = MIN(++a, --b);
/* m = ((++a) < (--b) ? (++a) : (--b)); */
```

- Co wtedy robić?
  – Nie używać wyrażeń dających efekty uboczne jako argumentów.
- Ale skąd wiadomo, że wywołuję makro?
  – Należy stosować konwencję nazewniczą dotyczącą makr i robi się łatwiej..

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Dyrektywy wspierające warunkową kompilację:

```
#if
#ifdef
#ifndef
#else
#elif
#endif
```


Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

`#if const-expr1`
Linijki kodu znajdujące się tutaj są dołączane przez procesor do pliku wynikowego tylko jeżeli **`const-expr1`** jest prawdziwe

`#elif const-expr2`
Linijki kodu znajdujące się tutaj są dołączane przez procesor do pliku wynikowego tylko jeżeli **`const-expr1`** jest fałszywe a **`const-expr2`** jest prawdziwe

`#else`
Linijki kodu znajdujące się tutaj są dołączane przez procesor do pliku wynikowego tylko jeżeli **`const-expr1`** jest fałszywe i **`const-expr2`** też jest fałszywe

`#endif`

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Najbardziej użyteczna dyrektywa:

`#ifdef identifier`

1. Cały kod występujący po niej zostanie dołączony przez preprocesor do pliku wynikowego w zależności od tego, czy **`identifier`** istnieje, czy nie.
2. Jej obszar działania ogranicza wystąpienie **`#endif`**. Następne linijki kodu traktowane są już jako konieczne do dołączenia chyba, że kolejne wywołanie dyrektywy to zmieni.

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Przykład użycia
– funkcja drukująca wartość tylko w trybie **DEBUG**:

```
void debug_print (int i) {
#ifdef DEBUG
   printf ("DEBUG PRINT: %d\n", i);
#endif
}
```

Ale w trybie „release" to nadal jest wywołanie funkcji. Czy można tego uniknąć?..

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Makro wykorzystujące warunkową kompilację:

```
#ifdef DEBUG
#define debug_print(a) \
          printf ("DEBUG PRINT: %d\n", (int)a);
#else
#define debug_print(a)
#endif
```

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

```
#ifdef NDEBUG
#define assert(EX)
#else
#define assert(EX) (void)((EX) || (__assert (#EX, __FILE__, __LINE__),0))
#endif

#ifdef __cplusplus
extern "C" {
#endif

extern void __assert (const char *msg, const char *file, int line);

#ifdef __cplusplus
};
#endif
```


C++ - strumienie

Motywacja wprowadzenia nowej biblioteki strumieni:

W języku C istnieje intepreter odpowiedzialny za analizę łańcucha formatującego podczas wykonywania programu oraz pobierający zmienną liczbę argumentów.

Wady interpretera formatowanego we/wy:

1. wystarczy użyć choćby niewielką część możliwości intepretera, a cały interpreter zostaje dołączony do nowego programu
2. interpretacja wykonuje się w trakcie wykonywania programu dlatego w tym momencie zawsze następuje pewne spowolnienie jego działania
3. interpretacja wykonuje się w trakcie wykonywania programu, dlatego dopiero wtedy okazuje się, czy w łańcuchu formatującym nie było błędów
4. funkcje z rodziny `printf` nie są rozszerzalne – operują tylko na typach wbudowanych

C++ - strumienie

W C++ operacje we/wy realizuje się za pomocą strumieni.

Wyrażenie „strumień" określa konstrukcję pozwalającą na wysyłanie lub odbieranie dowolnej liczby danych. Tak jak woda w strumieniu – odbieramy dane kiedy nadchodzą, a wysyłamy, kiedy tego potrzebujemy.

Strumień może mieć kierunek do i od naszego programu

- **strumień wyjściowy** – informacja płynie od naszego programu do ujścia. Ujściem może być terminal, plik, gniazdko internetowe, obszar w pamięci, nazwany potok, systemowa kolejka FIFO, modem, itp.
- **strumień wejściowy** – informacja płynie do naszego programu ze źródła. Źródłem może być terminal, plik, itp.

C++ - strumienie

Klasy reprezentujące strumienie wejściowe

- **`istream`** – podstawowa klasa reprezentująca strumienie wejściowe, w niej zdefiniowany jest przeciążony operator '>>'. Dziedziczą z tej klasy:
- **`istringstream`** – źródłem jest obiekt klasy 'string', czyli napis. Udostępniana po dołączeniu biblioteki 'sstream'
- **`istrstream`** – źródłem jest C-napis (czyli tablica znaków ze znakiem '\0' jako ostatnim). Udostępniana po dołączeniu biblioteki 'strstream'
- **`ifstream`** – źródłem jest plik. Udostępniana po dołączeniu biblioteki 'fstream'

C++ - strumienie

Klasy reprezentujące strumienie wyjściowe

- **`ostream`** – podstawowa klasa reprezentująca strumienie wyjściowe, w niej zdefiniowany jest przeciążony operator '<<'. Dziedziczą z tej klasy:
- **`ostringstream`** – ujściem jest obiekt klasy 'string', czyli napis. Udostępniana po dołączeniu biblioteki 'sstream'
- **`ostrstream`** – ujściem jest C-napis (czyli tablica znaków ze znakiem '\0' jako ostatnim). Udostępniana po dołączeniu biblioteki 'strstream'
- **`ofstream`** – ujściem jest plik. Udostępniana po dołączeniu biblioteki 'fstream'

C++ - strumienie

Uwaga na marginesie:

W C++ utworzono klasę 'string' aby łatwiej operować na łańcuchach tekstowych:

```
string napis1;
napis1 = "text1";
string napis2( "text2" );
napis1 = napis1 + napis2;
if (napis1 == napis2)
    cout << "takie same\n";
if (napis1 > napis2)
    cout << "napis1 jest wiekszy\n";
else
    cout << "napis2 nie jest wiekszy\n";
```


C++ - strumienie

Uwaga na marginesie c.d.:

Porównanie:

C :	C++:
`strcpy(a,b)`	`a = b`
`strcmp(a,b)`	`a == b`
`strcat(a,b)`	`a += b`
`strlen(a)`	`a.size()`
`strstr(a,b)`	`a.find(b)`

.. koniec uwagi na marginesie.

C++ - strumienie

Klasa **`iostream`** dziedziczy zarówno z **`istream`** jak i **`ostream`** (dołączamy tylko plik nagłówkowy <iostream>)

`fstream` dziedziczy z **`iostream`** (dołączamy plik nagłówkowy <fstream> i mamy przy okazji dostęp do całej biblioteki <iostream>)

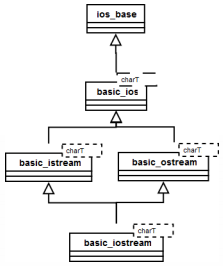
`stringstream` również dziedziczy z **`iostream`** (dołączamy plik nagłówkowy <sstream> i również dostajemy dostęp do całej biblioteki <iostream>)

Wszystkie te klasy mają jeden interfejs, niezależnie od tego, czy masz do czynienia z plikiem, terminalem, obszarem pamięci czy obiektem 'string'.

Wszystkie te klasy są specjalizacjami szablonu

C++ - strumienie

Hierarchia szablonów:



Przykładowo klasa 'istream' deklarowana jest jako:

```
typedef basic_istream<char> istream;
```


C++ - strumienie

Wszystkie pozostałe klasy są deklarowane analogicznie.

Istnieją też definicje typów dla wszystkich strumieni klas wykorzystujące zamiast '**`char`**' typ '**`wchar_t`**', który reprezentuje znaki wielobajtowe (*ten temat nie będzie rozwijany na tym wykładzie*)

C++ - strumienie

Strumienie predefiniowane

Po dołączeniu któregoś z plików nagłówkowych 'iostream', 'fstream' lub 'sstream', mamy do dyspozycji cztery już otwarte strumienie:

jeden wejściowy

- **`cin`** – standardowy strumień wejściowy przydzielony procesowi wykonującemu program, zwykle - klawiatura

trzy wyjściowe

- **`cout`** – standardowy strumień wyjściowy przydzielony procesowi wykonującemu program, zwykle – ekran terminala. Strumień jest buforowany, dlatego znaki mogą pojawiać się tam z opóźnieniem
- **`cerr`** – standardowy strumień komunikatów o błędach przydzielony procesowi wykonującemu program, zwykle – ekran terminala. Strumień nie jest buforowany
- **`clog`** – strumień komunikatów (rzadko używany), jest buforowany