Dzisiejszy wykład

- Przestrzenie nazw (namespaces)
- Funkcje o zmiennej liczbie argumentów

Przestrzenie nazw

- Globalna przestrzeń nazw jest jedna
- W programach pisanych przez wiele osób, lub korzystających z bibliotek napisanych przez innych, istnieje ryzyko kolizji nazw

```
// library1.h
const double LIB_VERSION = 1.204;
```

```
// library2.h
const int LIB_VERSION = 3;
```

```
// main.c
#include <library1.h>
#include <library2.h>
```

- Tradycyjnie problem rozwiązuje się przy pomocy prefiksów unikatowych dla każdej biblioteki

```
// library1.h
const double library1_LIB_VERSION
                          = 1.204;
```

```
// library2.h
const int library2_LIB_VERSION = 3;
```

Przestrzenie nazw

- Zasadniczo przestrzenie nazw zastępują prefiksy identyfikatorów
- Zamiast

```
const double sdmBOOK_VERSION = 2.0;          // in this library,
class sdmHandle { ... };                     // each symbol begins
sdmHandle& sdmGetHandle();                   // with "sdm"
```

piszemy:

```
namespace sdm {
    const double BOOK_VERSION = 2.0;
    class Handle { ... };
    Handle& getHandle();
}
```

Użycie przestrzeni nazw

```
void f1() {
    using namespace sdm;        // make all symbols in sdm
                                // available w/o qualification
                                // in this scope
    cout << BOOK_VERSION;       // okay, resolves to sdm::BOOK_VERSION ...
    Handle h = getHandle();     // okay, Handle resolves to
                                // sdm::Handle, getHandle
    ...                         // resolves to sdm::getHandle
}
void f2() {
    using sdm::BOOK_VERSION;    // make only BOOK_VERSION
                                // available w/o qualification
                                // in this scope
    cout << BOOK_VERSION;       // okay, resolves to
                                // sdm::BOOK_VERSION
    ...
    Handle h = getHandle();     // error! neither Handle
                                // nor getHandle were
    ...                         // imported into this scope
}
void f3() {
    cout << sdm::BOOK_VERSION;// okay, makes BOOK_VERSION
                                // available for this one use
    ...                         // only
    double d = BOOK_VERSION;    // error! BOOK_VERSION is
                                // not in scope
    Handle h = getHandle();     // error! neither Handle
                                // nor getHandle were
    ...                         // imported into this scope
}
```

Użycie przestrzeni nazw

- W przypadku niejednoznaczności można jawnie użyć kwalifikatora zakresu ::

```
namespace AcmeWindowSystem {
     ...
     typedef int Handle;
     ...
}

void f() {
     using namespace sdm;                 // import sdm symbols
     using namespace AcmeWindowSystem; // import Acme symbols
     ...                                  // freely refer to sdm
                                          // and Acme symbols
                                          // other than Handle
     Handle h;                            // error! which Handle?
     sdm::Handle h1;                      // fine, no ambiguity
     AcmeWindowSystem::Handle h2;         // also no ambiguity
     ...
}
```

Anonimowe przestrzenie nazw

- Anonimowa przestrzeń nazw zastępuje użycie słowa kluczowego *static* przy nazwie globalnej

```
#include "header.h"

namespace {
   int a;
   void f() {/* ... */}
   int g() {/* ... */ }
}
```

- Dostęp z zewnątrz do anonimowej przestrzeni nazw jest możliwy dzięki niejawnej dyrektywie użycia. Powyższa deklaracja jest równoważna następującej:

```
#include "header.h"

namespace $$$ {
   int a;
   void f() {/* ... */}
   int g() {/* ... */ }
}

using namespace $$$;
```

- $$$ jest pewną nazwą unikatową w zasięgu, w którym zdefiniowana jest ta przestrzeń nazw

Przeszukiwanie nazw

- 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

```
namespace Chrono {
   class Date { /* ... */ };
   bool operator==(const Date&, const std::string&) ;
   std::string format(const Date&) ; // make string representation
   // ...
}

void f(Chrono::Date d, int i)
{
   std::string s = format(d) ; // Chrono::format()
   std::string t = format(i) ; // error: no format() in scope
}
```

- Reguła ta umożliwia pominięcie jawnego kwalifikowania nazw, co jest szczególnie istotne w przypadku argumentów funkcji operatorowych i wzorców, gdzie może to być szczególnie kłopotliwe.

Przeszukiwanie nazw

- W przypadku, kiedy funkcja przyjmuje argumenty z więcej niż jednej przestrzeni nazw, funkcje są wyszukiwane w przestrzeni nazw każdego argumentu i w zwykły sposób rozstrzyga się przeciążenie wszystkich znalezionych funkcji.
- Jeżeli funkcję wywołuje metoda klasy, to pierwszeństwo przed funkcjami znalezionymi przez typy argumentów mają metody tej samej klasy i jej klas bazowych.

```
namespace Chrono {
   class Date { /* ... */ };
   bool operator==(const Date&, const std::string&);
   std::string format(const Date&) ; // make string representation
   // ...
}

void f(Chrono::Date d, std::string s)
{
   if (d == s) {
        // ...
   }
   else if (d == "August 4, 1914") {
        // ...
   }
}
```

Aliasy przestrzeni nazw

- Jeżeli użytkownicy nadają przestrzeniom nazw krótkie nazwy, to mogą one spowodować konflikt

```
namespace A { // short name, will clash (eventually)
   // ...
}
A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

- Długie nazwy są niewygodne w użyciu

```
namespace American_Telephone_and_Telegraph{ // too long
   // ...
}
American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

- Dylemat ten można rozwiązać za pomocą krótkiego aliasu dla długiej nazwy przestrzeni nazw

```
// use namespace alias to shorten names:
namespace ATT = American_Telephone_and_Telegraph;
ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

Aliasy przestrzeni nazw

- Dzięki aliasom przestrzeni nazw użytkownik może także odwoływać się do biblioteki

```
namespace Lib = Foundation_library_v2r11;
// ...
Lib::set s;
Lib::String s5 = "Sibelius";
```

- Upraszcza to wymianę jednej biblioteki na inną. Używanie *Lib* zamiast *Foundation_library_v2r11* ułatwia przejście do wersji *v3r5* - wystarczy zmienić inicjowanie aliasu *Lib* i ponownie skompilować kod.
- Nadużywanie aliasów może prowadzić do nieporozumień

Komponowanie przestrzeni nazw

- Często chcemy utworzyć interfejs z istniejących już interfejsów, np.:

```
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&) ;
}
```

- Teraz przy pisaniu programu można posługiwać się My_lib:

```
void f()
{
   My_lib::String s = "Byron"; // finds My_lib::His_string::String
   // ...
}
using namespace My_lib;
void g(Vector<String>& vs)
{
   // ...
   my_fct(vs[5]) ;
   // ...
}
```

Komponowanie przestrzeni nazw

- Definiując funkcję lub daną, musimy znać prawdziwą nazwę elementu:

```
void My_lib::fill() // error: no fill() declared in My_lib
{
   // ...
}
void His_string::fill() // ok: fill() declared in His_string
{
   // ...
}
void My_lib::my_fct(My_lib::Vector<My_lib::String>& v) // ok
{
   // ...
}
```

- Idealna przestrzeń nazw powinna:
  - wyrażać logicznie spójny zbiór cech
  - nie dawać użytkownikom dostępu do innych cech
  - nie stanowić znaczącego obciążenia notacyjnego dla użytkowników

Wybór

- Czasami jest potrzebny dostęp jedynie do kilku nazw z danej przestrzeni. Można napisać deklarację przestrzeni nazw, zawierającą tylko te potrzebne nazwy

```
namespace His_string{ // part of His_string only
  class String { /* ... */ };
  String operator+(const String&, const String&) ;
  String operator+(const String&, const char*) ;
}
```

- Jeżeli nie jest się projektantem przestrzeni *His_string*, łatwo można wprowadzić bałagan. Zmiana w rzeczywistej definicji przestrzeni nie znajdzie odbicia w tej deklaracji. Wyboru elementów z przestrzeni nazw można dokonać jawnie za pomocą deklaracji użycia:

```
namespace My_string {
  using His_string::String;
  using His_string::operator+; // use any + from His_string
}
```

- Deklaracja użycia wprowadza do zasięgu każdą deklarację o danej 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

```
namespace His_lib {
  class String { /* ... */ };
  template<class T> class Vector { /* ... */ };
  // ...
}
namespace Her_lib {
  template<class T> class Vector { /* ... */ };
  class String { /* ... */ };
  // ...
}
namespace My_lib {
  using namespace His_lib;    // everything from His_lib
  using namespace Her_lib;    // everything from Her_lib
  using His_lib::String;      // resolve potential clash in favor of His_lib
  using Her_lib::Vector;      // resolve potential clash in favor of Her_lib
  template<class T> class List { /* ... */ };      // additional stuff
  // ...
}
```

Komponowanie i wybór

- 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 dyrektywy *typedef* lub poprzez dziedziczenie

```
namespace Lib2 {
  using namespace His_lib;    // everything from His_lib
  using namespace Her_lib;    // everything from Her_lib
  using His_lib::String;      // resolve potential clash in favor of His_lib
  using Her_lib::Vector;      // resolve potential clash in favor of Her_lib
  typedef Her_lib::String Her_string;      // rename
  template<class T> class His_vec          // "rename"
      : public His_lib::Vector<T> { /* ... */ };
  template<class T> class List { /* ... */ };      // additional stuff
  // ...
}
```

Przestrzenie nazw i przeciążanie

- Przeciążanie działa między przestrzeniami nazw. Dzięki temu przekształcenie istniejących bibliotek do postaci z użyciem przestrzeni nazw wymaga minimalnych zmian kodu źródłowego, np.

```
// old A.h:
void f(int) ;
// ...

// old B.h:
void f(char) ;
// ...

// old user.c:
#include "A.h"
#include "B.h"
void g()
{
  f(´a´) ; // calls the f() from B.h
}
```

```
// new A.h:
namespace A {
   void f(int) ;
   // ...
}
// new B.h:
namespace B {
   void f(char) ;
   // ...
}
// new user.c:
#include "A.h"
#include "B.h"
using namespace A;
using namespace B;
void g()
{
   f(´a´) ; // calls the f()from B.h
}
```

Przestrzenie nazw są otwarte

- Przestrzeń nazw jest otwarta, tzn. można do niej dodawać nazwy w kilku deklaracjach. W ten sposób można umieszczać duże fragmenty programów w pojedynczej przestrzeni nazw.

```
namespace A {
  int f() ; // now A has member f()
}
namespace A {
  int g() ; // now A has two members, f() and g()
}
```

- Otwartość przestrzeni nazw jest pomocna w transformacji programów

```
// my header:
void f() ; // my function
// ...
#include<stdio.h>
int g() ; // my function
// ...
```

```
// my header:
namespace Mine {
  void f() ; // my function
  // ...
}
#include<stdio.h>
namespace Mine {
  int g() ; // my function
  // ...
}
```

Przestrzenie nazw są otwarte

- Definiując wcześniej zadeklarowaną składową przestrzeni nazw, bezpieczeniej jest użyć składni *Mine::* niż ponownie otwierać przestrzeń *Mine*

```
namespace Mine {
  void f();
}
void Mine::ff() // error: no ff() declared in Mine
{
  // ...
}
```

- Kompilator nie wykryje tego błędu przy próbie zdefiniowania *ff()* w ponownie otwartej przestrzeni nazw

```
namespace Mine { // reopening Mine to define functions
  void ff() // oops! no ff() declared in Mine;
            // ff() is added to Mine by this definition
  {
      // ...
  }
  // ...
}
```

Przestrzenie nazw i C

- Deklaracje funkcji i danych zdefiniowanych w plikach kompilowanych kompilatorem C należy poprzedzić w C++ deklaratorem *extern "C"*

```
extern "C" int printf(const char* ... );
```

```
extern "C" {
   ...
   int printf(const char* ... );
   ...
};
```

- W języku C++ zdefiniowano standardowe pliki nagłówkowe, w których funkcje są dostępne w przestrzeni nazw *std*

```
// cstdio
namespace std {
   // ...
   extern "C" int printf(const char* ... ) ;
   // ...
}
```

Funkcje o zmiennej liczbie argumentów

- W <cstdarg> znajduje się zbiór makrodefinicji, umozliwiających dostęp do funkcji przyjmujących zmienną liczbę argumentów
- Napiszmy funkcję obsługi błędów, która ma jeden argument całkowity, określający ważność błędu, po którym może wystąpić dowolna liczba łańcuchów tekstowych
- Komunikat o błędzie tworzony jest poprzez przekazanie każdego słowa jako osobnego argumentu łańcuchowego. Lista argumentów powinna się kończyć pustym wskaźnikiem do *char*

Funkcje o zmiennej liczbie argumentów

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

extern void error (int ...);
const char *Null_cp = 0;

int main (int argc, char *argv[])
{
  switch (argc)
    {
    case 1:
      error (0, argv[0], Null_cp);
      break;
    case 2:
      error (0, argv[0], argv[1],
        Null_cp);
      break;
    default:
      ostringstream s;
      s << argc-1;
      string ss=s.str();
      error (1, argv[0], "with",
        ss.c_str(), "arguments",
        Null_cp);
    }
  //...
}
```

```
void error (int severity ...)
  // "severity" followed by a
  // zero-terminated list of char *s
{
  va_list ap;
  va_start (ap, severity); //arg startup
  for (;;)
    {
      char *p = va_arg (ap, char *);
      if (p == 0)
        break;
      cerr << p << ' ';
    }
  va_end (ap); // arg cleanup
  cerr << '\n';
  if (severity)exit (severity);
}
```