

(Średnio) zaawansowane programowanie w C++ (ZPR)

Wykład 13. C++14, C++17, C++20, C++23. Boost. Przetwarzanie tekstu.

Robert Nowak

25L

Plan wykładu

- ▶ standardy C++, Python, Rust.
- ▶ biblioteka standardowa C++, uzupełnienie;
- ▶ biblioteki Boost, Boost Graph Library (BGL);
- ▶ przetwarzanie tekstu w C++;

Standard C++, język C++ jest zgodny wstecz

- ▶ ISO/IEC 14882, opublikowany w 1998 (C++98)
- ▶ ISO/IEC 14882:2003, zmodyfikowany w 2003 (C++03)
- ▶ ISO/IEC 14882:2011, C++11, c++0x (III 2011), draft:N3290
<http://www.open-std.org/jtc1/sc22/wg21>
- ▶ ISO/IEC 14882:2014, C++14, c++1y (VIII 2014), draft:N3797
- ▶ ISO/IEC 14882:2017, C++17, c++1z (XII 2017), draft:N4661
- ▶ ISO/IEC 14882:2020, C++20, c++2a (XII 2020), draft:N4878
- ▶ ISO/IEC 14882:2024 C++23, c++2b (XII 2023) draft:N4950
- ▶ C++26 draft:N5008 (III 2025)
`git clone https://github.com/cplusplus/draft.git`

Standard Python, Python Software Foundation

3 wersje standardu (niezgodne wstecz!):

Python 1 (nie używana), Python 2 (wycofana od 2020), Python 3

Definicja standardu w Python Enhancement Proposals (PEP),
<http://python.org>, przykłady:

- ▶ PEP 0 - definicja PEP, lista PEP (jest ich ponad 8000)
- ▶ PEP 8 - styl kodowania
- ▶ PEP 20 - 'The Zen of Python' (przesłanie)

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Konferencja PyCon: <http://pycon.org>

Standard Rust, rust-lang.org

obecnie stabilna wersja 1.87 (od 15.05.2025)

- ▶ codziennie nowa wersja,
- ▶ co 6 tygodni wersja beta, która następnie jest ogłaszana jako stabilna
- ▶ nie ma gwarancji zgodności

Zmiany w git:

<https://github.com/rust-lang/rust>

Uzgadnianie zmian: RFC (request for comments),

<https://rust-lang.github.io/rfcs>

np: 0002-rfc-process.md, opisuje RFC dla Rust

Dokumentacja

Dokumentacja w kodzie zmniejsza ryzyko braku spójności

Komentarze - poprawiają czytelność kodu.

Komentarz mówi DLACZEGO, kod mówi JAK.

- ▶ każdy byt powinien mieć pojedynczą odpowiedzialność,
- ▶ dokumentacja projektowa powinna być generowana z kodu.

Hierarchia komentarzy:

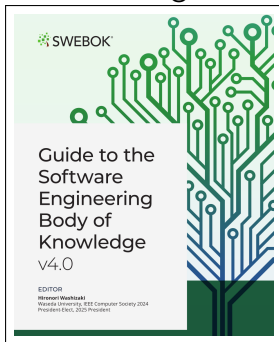
1. odpowiedzialność bibliotek, pakietów, przestrzeni nazw, katalogów
2. odpowiedzialność modułów (plików), klas,
3. odpowiedzialność metod publicznych,
4. niebanalne algorytmy

SWEBOK

IEEE Computer Society,

Guide to the Software Engineering Body of Knowledge

<https://www.computer.org/education/bodies-of-knowledge/software-engineering/v4>



- ▶ zarządzanie wymaganiami,
- ▶ projektowanie i implementacja oprogramowania,
- ▶ testowanie,
- ▶ dostarczanie, konfiguracja, pielęgnacja,
- ▶ zarządzanie jakością,
- ▶ zarządzanie zespołem.

std::ref <functional>

`std::ref(X)` obiekt zaw. wskaźnik do X, możliwość kopiowania, itp.

`std::cref(X)` obiekt zaw. stały wskaźnik do X

```
int foo = 10;
auto bar = std::ref(foo);
++bar; //działa na obiekcie, foo == 11
```

```
void thrd_fun(Data& data) { /* nieistotne */ }
Data data;
auto thr1 = std::thread( thread_fun, data ); //pracuje na kopii
auto thr2 = std::thread( thread_fun, ref(data) ); //pracuje na oryginale
```


rekordy, std::pair

```
template<class _T1, class _T2>
struct pair {
    typedef _T1 first_type;
    typedef _T2 second_type;
    _T1 first; //Pierwsza składowa
    _T2 second; //Druga składowa
    pair() : first(), second() { }
    pair(const _T1& __a, const _T2& __b)
        : first(__a), second(__b) { }
};

//Przykłady użycia
using StringCount = std::pair<std::string, int>;
StringCount f(); //funkcja zwraca dwie wartości
//Tworzy parę o typach dedukowanych z argumentów
std::make_pair(24, true);
```

rekordy, std::tuple

```
//struktura zawierająca trzy obiekty (typów A,B,C)
using triple = std::pair<A,std::pair<B,C> >;
tuple<A,B,C> //Zawiera trzy obiekty (typów A,B,C)
```

► dostęp do składowej - get

```
int i = 3;
tuple<int,double,int&> t(2,1.5,i);
t.get<0>() = 4; //dostęp do elementu
double d = get<1>(t); //dostęp do elementu
```

► make_tuple - nie trzeba podawać typów obiektów

► operatory: ==, !=, <, >, <=, >=

► funkcja tie

```
int i; double d;
tie(i,d); //make_tuple(ref(i),ref(d)), zwraca tuple<int&,double&>
```

std::function (C++11)

Pozwala przechowywać funkcje i obiekty funkcyjne

- ▶ obiekt który akcje (np. wskaźnik do funkcji lub metody)
- ▶ możliwość kopiowania, przypisywania itp.

```
std::function<void (int, int)> pf; //funkcja dwuargumentowa
//boost::function2<void, int, int> pf; - to samo
void f(int x, int y) { cout << x + y << endl; }
pf = f;
pf(2,3); //wywołanie f dla x = 2, y = 3
struct F {
    void operator()(int x, int y);
};
pf = F();
pf(2,3); //wywołanie F.operator() dla x = 2, y = 3
pf = [](int x, int y){ cout << x + y << endl; };
pf(2,3);
```

std::function - przykłady

```
using PF = std::function<void (int)>;
PF pf; //Obiekt, który przechowuje komendę
pf(3); //Wyjątek: boost::bad_function_call
class Foo { public: void f(int x); /* ... */ };
Foo foo;
//Związanie obiektu dla którego będzie wołana metoda
pf = std::bind(Foo::f,ref(foo),_1);
pf(4); //Woła foo->f(4)

vector<PF> callbacks; //Możliwość przechowywania w kolekcjach
//Woła wszystkie metody w kolekcji (z argumentem = 7)
for_each(callbacks.begin(), callbacks.end(), [](PF& pf){ pf(7);});
```

Zastosowanie : wszędzie tam, gdzie wzorzec komendy

- ▶ separacja tworzenia akcji od jej wołania
- ▶ kolekcjonowanie poleceń

inne typy pomocnicze

- ▶ `std::variant` (C++11) – unia z wyróżnikiem bieżącego typu
- ▶ `std::optional` (C++11) – obiekt z sygnalizacją braku wartości

`std::any` (C++17)

kontener na wartość dowolnego typu (jak `void*`)

```
#include <any>

any a; //dla pustych obiektów wartość przechowywanego typu to void
assert(a.has_value() == false);
a = string("Hello"); //a przechowuje wartość string
assert(a.type() == typeid(string) );
string s = any_cast<string>(a); //dostęp do obiektu

//wyjątek bad_any_cast, gdy niezgodne typy
template <typename Val> Val any_cast(const any& a);

//nullptr gdy niezgodne typy
template <typename Val> const Val* any_cast(const any* a);
template <typename Val> Val* any_cast(any* a);
```

Tablice wielowymiarowe, Boost.Multi_Array

Wygodne do reprezentacji macierzy, tensorów, itp.

Typowe implementacje:

- ▶ tablice wielowymiarowe z C, np. `int tab[2][3]`;
- ▶ wektory wektorów, np. `vector<vector<int>> tab`;
- ▶ `boost::multi_array <boost/multi_array.hpp>`

konstruktor

```
//liczba elementów - kontener  
array< array_type::index, 2> shape = { 2, 3 };  
//wymiar - parametr szablonu  
multi_array<double, 2> matrix( shape );
```

dostęp

```
matrix[1][1] = 2.56; //operator indeksowania  
//kontener z indeksami  
array< array_type::index, 2> index = { 1, 1 };  
matrix( index ) = 2.56;
```

Tablice wielowymiarowe `std::mdspan` (C++23)

- ▶ `std::span` (C++20) - widok na ciągłą sekwencję obiektów
- ▶ `std::mdspan` (C++23) – widok na sekwencję jak na wielowymiarową tablicę

```
std::array a = {0,1,2,3,4,5,6,7,8,9,10,11};  
mdspan(a.data(), 2, 6 ); //widok na tablice 2D  
//[ [ 0, 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10, 11 ] ]  
mdspan(a.data(), 3, 4 ); //widok na tablice 2D  
//[ [ 0, 1, 2, 3], [ 4, 5, 6, 7], [ 8, 9, 10, 11] ]  
mdspan(a.data(), 2, 2, 3 ); //widok na tablice 3D: 2 x 2 x 3  
//[ [ [ 0, 1, 2], [3, 4, 5] ], [ 6, 7, 8], [9, 10, 11] ] ]
```

Pomocnicze: `std::rank` - zwraca liczbę wymiarów, `std::extent` - liczba elementów dla danego wymiaru

C++11, uzupełnienie

- ▶ **constexpr**, wyrażenia, funkcje i in. obliczane w czasie kompilacji

- ▶

```
class Foo { //delegacja konstruktora
public:
    Foo(const std::string& s) { /* ... */ }
    Foo(const char* c) : Foo(std::string(c)) {} //delegacja
    /* ... */
```

//Wskazuje, aby nie tworzyć instancji w bieżącym module
`external template class std::vector<Foo>;`

- ▶ operator typu wyrażenia **decltype**

```
using size_t = decltype(sizeof(0));
using nullptr_t = decltype(nullptr);
```


Standard C++14, uzupełnienie

```
//dedukcja typu zwracanego, kompilator określa, że zwracamy double
auto getValue() {
    return 1.0;
}
```

przydatna do redukcji modyfikacji kodu:

```
struct Record {
    int id; //identyfikator
    std::string name;
};

//auto find_id(...) pozwoli uniknąć modyfikacji kodu po zmianie typu
//identyfikatora w klasie Record
int find_id(const vector<Record>& records, const string& name) {
    auto it = find_if(records.begin(), records.end(),
        [&](const Record& r){ return r.name==name; } );
    if(it == records.end())
        return -1;
    return it->id;
}
```

Standard C++14, uzupełnienie (2)

► atrybut `[[deprecated]]`

```
class [[deprecated]] Foo {};
```

//kompilacja kodu, który używa Foo dostarczy ostrzeżenie, np.

```
test.cpp:4:6: warning: 'Foo' is deprecated ...
```

► literały binarne: `int val = 0b10101010;`

► separator przy definiowaniu ciągu cyfr (nie ma wpływu na wartość)

```
int mask = 0b1111'0000'1111'0000;
```

► zmienne szablonowe, np. `pi<T>`

```
cout << pi<int> << endl; //3
```

```
cout << pi<std::string> << endl; //napis pi
```

► generyczne funkcje anonimowe, (użycie `auto` w `lambda`) - podobne do szablonów

```
auto add = [](auto x, auto y) { return x + y ; }
```

Standard C++17, uzupełnienie

- ▶ `std::string_view` (referencja tylko do odczytu do części napisu)
- ▶ biblioteka do systemu plików (bazuje na `Boost.Filesystem`)
- ▶ usunięcie alternatywnych reprezentacji znaków specjalnych (trigraph),
tzn `??=` (reprezentuje `#`), `??/` (reprezentuje `\`) itp.
- ▶ inicjacja (opcjonalna) dla `if`, `switch` i `while`

```
if( int c = get(); c != SUCCESS ) { //'c' istnieje w bloku  
    /* ... */  
}
```

- ▶ zmienne `inline`
- ▶ dodany typ `std::void_t`
- ▶ dodany typ `std::byte` - reprezentacja bajtów, niedozwolone operacje arytmetyczne, nie jest to typ znakowy
- ▶ literały szesnastkowe do reprezentacji zmiennopozycyjnej (IEEE 754)

Standard C++17, uzupełnienie (2)

- ▶ algorytmy STL mogą być wykonywane równoległe (współbieżnie lub wektorowo)

```
vector<double> v(1024);  
fill( execution::unseq, v.begin(), v.end(), 1.0 );  
for_each( execution::unseq, v.begin(), v.end(),  
          [](double& d) { return sin(d) + 2.0*cos(d);});
```

- ▶ `std::execution::sequenced_policy`
- ▶ `std::execution::parallel_policy` - można używać wielu wątków
- ▶ `std::execution::unsequenced_policy` - można stosować operacje wektorowe
- ▶ `std::execution::parallel_unsequenced_policy`

Standard C++20, uzupełnienie

operator `<=>`, porównanie trójstronne (three-way comparison)

$$(a <=> b) \begin{cases} < 0 & \text{jeżeli } a < b \\ == 0 & \text{jeżeli } a == b \\ > 0 & \text{jeżeli } a > b \end{cases}$$

podobnie jak `strcmp` w 'C'

```
std::strong_ordering::equal  
std::strong_ordering::less  
std::strong_ordering::greater  
std::strong_ordering::unordered
```

Dostarczone dla:

- ▶ typów liczbowych (całkowite, zmiennopozycyjne)
- ▶ wskaźników (arytmetyka adresowa)
- ▶ tablic (napisów)
- ▶ możliwość dostarczanie dla typów użytkownika

Standard C++20, uzupełnienie (2)

- ▶ nowa postać pętli for dla zakresu, `for (init; decl : expr)`

```
vector<Foo> vec;  
for(int i = 0; Foo f : vec) {  
    bar(f, i);  
    ++i;  
}
```

- ▶ napisy jako parametry szablonów (podobnie jak liczby całkowite, napis nie jest typem)

```
Foo<"foobar"> f;
```

C++20 dostępny w następujących narzędziach:

kompilator	język	biblioteka standardowa
GNU gcc	10	12
Visual Studio	2022, (oraz 2019 od ver. 16.9)	2022
CLang	11	13

BGL - generyczna reprezentacja grafów

Boost

Biblioteki Boost: biblioteki eksperymentalne, które kandydują do wejścia do standardu C++, kod bardzo dobrej jakości.

<http://boost.org>

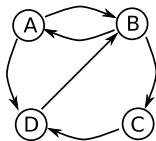


Obecnie 88 wersja, \approx 100 bibliotek, wiele jest już w standardzie C++11, C++14, C++17, C++20, C++23.

BGL - generyczna reprezentacja grafów

Listy sąsiedztwa

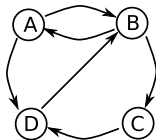
```
adjacency_list<VertexList,  
               OutEdgeList,  
               Directed,  
               VertexProperties,  
               EdgeProperties,  
               GraphProperties>
```



vertex	edges	
A	→ B	D
B	→ A	C
C	→ D	
D	→ B	

Macierz sąsiedztwa

```
adjacency_matrix<Directed,  
                 VertexProperty,  
                 EdgeProperty,  
                 GraphProperty,  
                 Allocator>
```



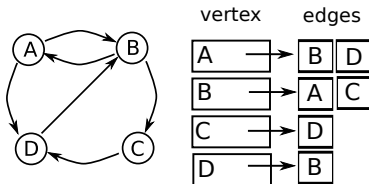
	A	B	C	D
A		1		1
B	1		1	
C				1
D		1		

Graf skompresowany (compressed_sparse_row_graph)

BGL - parametry szablonów

- ▶ pierwszy parametr - kontener przechowujący wierzchołki
 - ▶ `vecS = std::vector`
 - ▶ `listS = std::list`
 - ▶ `setS = std::set`
 - ▶ `multisetS = std::multiset`
- ▶ drugi parametr - kontener przechowujący krawędzie
- ▶ trzeci parametr - typ grafu
 - ▶ `directedS` = skierowany, dostęp do krawędzi wychodzących
 - ▶ `bidirectionalS` = skierowany, krawędzie wych. i wchodzące
 - ▶ `undirectedS` = nieskierowany

BGL - przykład grafu



```
using Graph = adjacency_list<vecS, vecS, bidirectionalS> Graph;
enum { A, B, C, D, N }; //identyfikatory wierzchołków
const int NUM_VERTICES = N; //liczba wierzchołków
typedef std::pair<int, int> Edge;
const Edge EDGE_ARRAY[] = { Edge(A,B), Edge(A,D),
    Edge(B,A), Edge(B,C),
    Edge(C,D), Edge(D,B) };
const int NUM_EDGES = sizeof(EDGE_ARRAY)/sizeof(EDGE_ARRAY[0]);

Graph g(EDGE_ARRAY, EDGE_ARRAY + NUM_EDGES, NUM_VERTICES); //tworzy graf
```

BGL - przeglądanie grafu

oznaczenia:

oznaczenie	nazwa	typ
g	graf	
e	krawędź	
v	wierzchołek	<code>boost::graph_traits<G>::vertex_descriptor</code>

- ▶ dostęp do wierzchołków
 - ▶ `std::pair<vertex_iter, vertex_iter> vertices(g)`
 - ▶ `vertices_size_type num_vertices(g)`
- ▶ dostęp do krawędzi
 - ▶ `std::pair<edge_iter, edge_iter> edges(g)`
 - ▶ `edges_size_type num_edges(g)`
- ▶ dostęp do krawędzi dla danego wierzchołka
 - ▶ `std::pair<edge_iter, edge_iter> out_edges(v, g)`
- ▶ dostęp do wierzchołków dla danej krawędzi
 - ▶ `vertex_descriptor source(e, g)`
 - ▶ `vertex_descriptor target(e, g)`

Własne dane związane z wierzchołkiem i z krawędzią

- ▶ obiekty zarządzane (dodawane, usuwane) przez graf
- ▶ obiekty zarządzane na zewnątrz (property_map)

```
struct CityData { //dane związane z wierzchołkiem
    CityData() : name(""), population(0) {}
    string name;
    int population;
};

struct RoadData { //dane związane z krawędzią
    RoadData() : length(0) {}
    int length;
};

using Graph = adjacency_list<vecS, vecS, directedS, CityData, RoadData>;
Graph g(3); //grafu (trzy wierzchołki, brak krawędzi)
Graph::vertex_descriptor v = *vertices(g).first; //pierwszy wierzchołek
g[v].name = "nazwa"; //g[v] zwraca referencję
cout << g[v].population; // na obiekt CityData dla wierzchołka v
```

BGL - modyfikacja grafu

oznaczenia:

oznaczenie	nazwa	typ
g	graf	
e	krawędź	
u, v	wierzchołek	boost::graph_traits<G>::vertex_descriptor

► modyfikacja wierzchołków

- `vertex_descriptor add_vertex(g)`
- `void clear_vertex(v, g)` - usuwa krawędzie wychodzące z wierzchołka i dochodzące do wierzchołka
- `void remove_vertex(v, g)` - usuwa wierzchołek

► modyfikacja krawędzi

- `std::pair<edge_descriptor, bool> add_edge(u, v, g)` - zwraca false jeżeli krawędź już istnieje i graf nie pozwala na wiele krawędzi pomiędzy tymi samymi wierzchołkami
- `void remove_edge(u, v, g)`
- `void remove_edge(e, g)`

Algorytmy grafowe

<code>copy_graph</code>	tworzy kopię
<code>transpose_graph</code>	zmienia kierunki krawędzi
<code>breadth_first_search</code>	przełgda wszcz
<code>depth_first_search</code>	przełgda w głab
<code>astar_search</code>	przełgda – algorytm A*
<code>topological_sort</code>	przełgda topologicznie
<code>dijkstra_shortest_path</code> <code>bellman_ford_i in.</code>	najkrótsze ścieżki
<code>kruskal_spanning_tree</code>	minimalne drzewa rozpinające
<code>connected_components</code>	badanie spójności
algorytmy badania maksymalnych przepływów, maksymalnych skojarzeń, algorytmy dla grafów planarnych, rzadkich, metryki i in	

Algorytmy grafowe - przykład

```
void breadth_first_search(const Graph& g, Vertex s ) {
    for_each v in g //oznacz każdy wierzchołek jako biały
    color[v] = WHITE;
    color[s] = GRAY;
    q.push(s); //wstaw do kolejki wierzchołek początkowy
    while (! q.empty() ) { //jeżeli są wierzchołki nierozpatrzone
        u = q.pop(); //pobierz pierwszy wierzchołek
        for_each e in out_edges(u) {
            v = e.target(); //v to wierzchołek sąsiedni do u
            if(color[v] == WHITE) {
                color[v] = GRAY; //wstaw do kolejki, jeżeli nieodwiedzony
                q.push(v);
            }
        }
        color[u] = BLACK; //oznacz wierzchołek jako rozpatrzony
    }
}
```


przeszukiwanie wszere - przykłąd użycia

```
class MyVisitor : public default_bfs_visitor { //wizytator
public:
    template <typename Vertex, typename Graph> //własny kod, nadpisywanie
    void discover_vertex(Vertex u, const Graph & g) const {
        cout << "discover vertex:" << u << endl;
    }
};

int main() {
    using Graph = adjacency_list<vecS, vecS, directedS>;
    Graph g(/* pominięto tworzenie grafu */);
    MyVisitor vis; //obiekt wizytatora
    breadth_first_search(g, A, visitor(vis) ); //woła algorytm
    return 0;
}
```

wizytator wołany przy przeglądaniu grafu

<code>initialize_vertex(v,g)</code>	wołana dla każdego wierzchołka zanim rozpoczęte zostanie przeszukiwanie
<code>discover_vertex(v,g)</code>	wierzchołek odwiedzony po raz pierwszy (jest wstawiany do kolejki)
<code>examine_vertex(v,g)</code>	wierzchołek jest rozpatrywany (pobierany z kolejki), żadna krawędź wychodząca z wierzchołka nie została zbadana
<code>examine_edge(e,g)</code>	rozpatrywana krawędź
<code>finish_vertex(v,g)</code>	wierzchołek jest rozpatrzony, tzn. wszystkie wierzchołki sąsiednie zostały odwiedzone

BGL - podsumowanie

- ▶ reprezentacja grafów skierowanych i nieskierowanych
- ▶ przeglądanie grafu, modyfikacja topologii grafu
- ▶ własne dane związane z krawędzią, z wierzchołkiem lub z grafem
- ▶ kilkadziesiąt algorytmów

Przetwarzanie tekstu w C++

przetwarzanie tekstu : konwersje za pomocą strumieni

```
//C++11 <string> definiuje konwersję dla wszystkich typów wbudowanych
string to_string(int val);
string to_string(double val);

//konwersja za pomocą strumieni, obsługuje typy wbudowane i użytkownika
#include <sstream>

std::ostringstream os;
os << 1234;
os.str();//Zwraca std::string

std::string napis("1234");
std::istringstream is(napis);
int i;
is >> i;

//ale ma niewygodny interfejs (potrzebne 3 linie kodu)
```

Boost.lexical_cast

```
//Szablon wykorzystuje strumienie do przekształceń
template<typename Target, typename Source>
Target lexical_cast(const Source & arg) {
    std::stringstream sout;
    Target result;
    if(!(sout << arg && sout >> result) )
        throw bad_lexical_cast( typeid(Source), typeid(Target) );
    return result;
}
```

- ▶ 'Source' może być zapisywane do strumienia
- ▶ 'Target' może być czytany ze strumienia
- ▶ 'Source' i 'Target' mają konstruktory kopiujące
- ▶ 'Target' ma konstruktor bezparametrowy

wykorzystanie boost::lexical_cast

```
#include <boost/lexical_cast.hpp>
//Przekształcenie napisu na inny typ
int i = lexical_cast<int>(string("1234") );
//Przekształcenie typu na napis
string s = lexical_cast<string>(1234);
Foo f;
string s = lexical_cast<string>(f); //podobnie dla typu użytkownika
```

- ▶ obsługuje typy wbudowane i typy użytkownika,
- ▶ wygodny interfejs, obsługa błędów,
- ▶ zgłoszona do umieszczenia w bibliotece standardowej (TR2), ale nie została umieszczona w C++11, C++14, C++17, C++20

lokalizm (C++)

klasyfikacja znaków, porządek napisów, format daty, itp.

```
#include <locale>
class ios_base {
    locale imbue(const locale& loc); //ustawia lokalizm
    locale getloc() const; //pobiera lokalizm
};
```

- ▶ `locale::classic()` standardowy US English ASCII
- ▶ globalna
 - ▶ `locale::locale()` konstr. domyślny tworzy taką jak globalna
 - ▶ inicjowana na classic
 - ▶ zmiana na inną: `locale::global()`
- ▶ `locale::locale("");` tworzy lokalizm domyślny dla systemu
- ▶ `locale::locale("nazwa");` tworzy lokalizm o podanej nazwie. Może zgłosić wyjątek `'runtime_error'`

Wsparcie dla Unicode

Dostępne w standardzie C++03

```
"ASCII String"//Tablica obiektów typu char  
L"wchar_t String"//Tablica obiektów wchar_t
```

Dostępne w standardzie C++11

```
u8"UTF-8 String \u2018."//UTF-8, tablica obiektów typu char  
u"UTF-16 String \u2018."//UTF-16, tablica obiektów char16_t  
U"UTF-32 String \u2018."//UTF-32, tablica obiektów char32_t
```

Napisy ze znakami specjalnymi (raw string literals)

```
auto str = R"({  
    "Title":"C/C++",  
    "Id":6  
})";
```

Własne symbole ogranicznika, R"ddd(...)ddd"

```
auto str = R"ZZZ({"Title":"(C/C++)","Id":6})ZZZ";
```

Wyrażenia regularne

- ▶ element programowania deklaratywnego
- ▶ przydatne przy przetwarzaniu tekstów
- ▶ dostępne standardowo w wielu językach programowania

```
template <class charT, class traits = regex_traits<charT> >  
class basic_regex { //reprezentuje wyrażenie regularne  
    //  
};  
using regex = basic_regex<char>;  
using wregex = basic_regex<wchar_t>;
```

```
#include <regex>  
std::regex reg("a(a|b)*b"); //obiekt repr. wyrażenie regularne  
#include <boost/regex.hpp>  
boost::regex reg2("a(a|b)*b"); //obiekt repr. wyrażenie regularne
```

opis wyrażenia regularnego regex

<code>^ \$</code>	początek i koniec linii	
<code>.</code>	dowolny pojedynczy znak	<code>.la 1la ala bla cla dla</code>
<code>[aeo]</code>	zbiór znaków	<code>[aeo]la ala ela ola</code>
<code>[a-z]</code>	zakres znaków	
<code>[^0-9]</code>	znak spoza zakresu	
<code>\d</code>	klasy znaków <code>[[[:digit:]]</code>	<code>\dla 1la 2la 3la</code>
<code>\s</code>	biały znak <code>[[[:space:]]</code>	
<code>*</code>	dowolna liczba (zero lub więcej)	<code>a*b b ab aaaab</code>
<code>+</code>	jedno lub więcej	<code>a+b ab aaaab</code>
<code>?</code>	opcjonalność: zero lub jedno	<code>a?b b ab</code>
<code>{m,n}</code>	od m do n wystąpień	
<code>(wyr)</code>	tworzy grupę	<code>(ab)+ ab abab ababab</code>
<code> </code>	alternatywa	<code>a(a b)b aab abb</code>

Badanie, czy napis jest opisywany wyrażeniem regularnym

```
template </* ... */>
bool regex_match( //funkcja bada, czy napis jest opisywany wyrażeniem
    const basic_string</* ... */>& str,
    const basic_regex</* ... */>& e,
    match_flag_type flags = match_default);

//Przykład użycia - badanie czy napis jest typu NNN-NNN-NN-NN
bool poprawny_NIP(const std::string& nip_str) {
    static const boost::regex e("\\d{3}-\\d{3}-\\d{2}-\\d{2}");
    return regex_match(nip_str, e);
}
```

Typowe wykorzystanie wyrażeń regularnych

```
regex reg("a*b");  
//regex_search poszukuje napisu spełniającego wyrażenie  
regex_search("aba",reg); //Zwróci true  
//regex_match bada, czy napis spełnia wyr. regularne  
regex_match("aba",reg); //Zwróci false  
//regex_iterator iteracja po odnalezionych napisach  
//regex_replace zastępuje napisy opisane wyrażeniem
```

	zachłanne	niezachłanne
przykład wyrażenia:	<(.*>	<(.*?>
wynik dla <ala>ma<kota>:	ala>ma<kota	ala

```
//Przykład: przeszukuje html w poszukiwaniu adresów  
string html = /* wczytaj stronę html */;  
regex mail("\\href=\"mailto:(.*?)\">", regex_constants::icase);  
smatch what; //Przechowuje wyniki wyszukiwania  
if( regex_search(html,what,mail) )  
    //what[0] zawiera napis pasujący do wyrażenia  
cout << "wczytany email to" << what[1] << endl;
```

Dziękuję