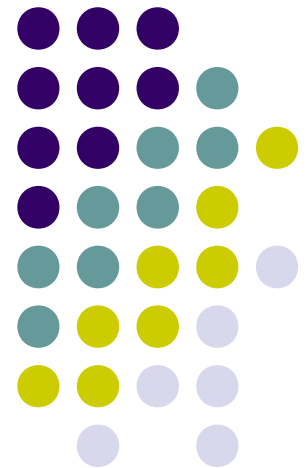


Średniozaawansowane programowanie w C++

Wykład #1
13 października 2016 r.



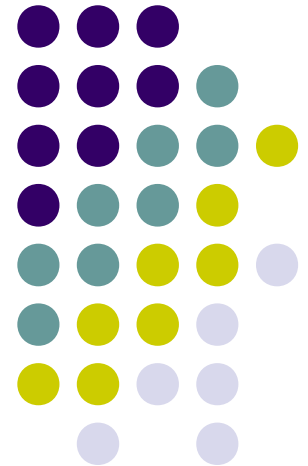
Plan



1. Kontenery STL (vector, map)
2. Sprytne wskaźniki
3. Algorytmy STL
4. Bindowanie funkcji
5. Lambda wyrażenia

Kontenery STL

Lepiej
Szybciej
Bezpieczniej
Wygodniej



Zajawka problemu



```
// Tablica na stosie  
double tablica [333];
```

```
// Zalety: niszczona automatycznie (nie ma wycieków pamięci), łatwa w obsłudze
```

```
// Wady: niszczona automatycznie (funkcja nie może jej zwrócić), nierozszerzalna
```

```
// Tablica na sterckie  
double *tablica = new double [333];
```

```
// Zalety: przydział dynamiczny (tworzona i niszczona na żądanie), może być zwrócona przez funkcję
```

```
// Wady: przydział dynamiczny (potencjalny wyciek pamięci), nieskalowalna, nie przechowuje informacji o liczbie elementów
```

Najważniejsze kolekcje



Sekwencyjne:

vector - jednowymiarowa tablica

list - lista dwukierunkowa

deque - kolejka o dwu końcach

Asocjacyjne:

map - tablica asocjacyjna (słownik)

set - zbiór

multiset - wartość może występować wielokrotnie

multimap - klucz może występować wielokrotnie

vector <class T> – mądra tablica



```
#include <vector>
```

```
// Tablica elementów typu double
```

```
std::vector<double> tablica;
```

```
// Zalety: niszczona automatycznie (nie ma wycieków pamięci), łatwa w  
obsłudze, rozszerzalna
```

```
// Wady: rozszerzanie i wstawianie elementów w środek jest kosztowne
```

```
tablica.push_back (3.14159);    // dodawanie elementu do tablicy
```

```
// Liczenie średniej wartości („przeoranie” całej tablicy)
```

```
double srednia = 0.0;
```

```
for (std::vector<double>::const_iterator it = tablica.begin();
```

```
    it != tablica.end(); ++it)
```

```
    srednia += *it;
```

```
srednia /= tablica.size();      // FANTASTYCZNE: tablica zna swój rozmiar!
```

vector <class T> – mądra tablica



```
// Tablicę można przekazywać jako parametr funkcji:

// a) przez wartość – kopiuje zawartość całej tablicy (NIEZALECANE)
void moja_funkcja (const std::vector<double> tablica);

// b) przez referencję – szybko i sprawnie
void moja_funkcja (const std::vector<double> &tablica);

// Funkcja może zwrócić tablicę przez wartość (NIEZALECANE)
std::vector<double> wczytaj_tablice (std::ifstream &plik);

// Czasami warto z góry zarezerwować miejsce dla danych:
tablica.reserve (10000);

// Można także wstawiać elementy w dowolne miejsce:
tablica.insert (tablica.begin()+10, 137.99);
// Albo usuwać:
tablica.erase (tablica.begin()+13, tablica.begin()+17);

// Zawartość wektora jest niszczone automatycznie!

// W wektorze można trzymać elementy dowolnej klasy:
std::vector<MojaFantastycznaKlasa> jupi;
```

auto – wytrych dla leniwych



```
// Liczenie średniej wartości)
double srednia = 0.0;

// „Męczący” zapis w C++03
for (std::vector<double>::const_iterator it = tablica.begin();
     it != tablica.end(); ++it)
    srednia += *it;

// Alternatywa w C++0x (GCC >= 4.4)
for (auto it = tablica.begin(); it != tablica.end(); ++it)
    srednia += *it;

srednia /= tablica.size();
```


initializer_list <class T>



```
// Parametrem funkcji może być zbiór!
double srednia (std::initializer_list <double> liczby)
{
    double sr = 0.0;
    for (auto it = liczby.begin(); it != liczby.end(); ++it)
        srednia += *it;
    return (sr / initializer_list.size ());
}

double moja_srednia = srednia ({4.5, 5.0, 3.5, 4.5, 3.0});

// To samo tyczy się metod
class Zbior
{
    public:
    Zbior (std::initializer_list <int> liczby); // konstruktor
};

Zbior wynik_losowania = {3, 16, 27, 31, 39, 41};
```

Pętla *for* oparta na zakresie



```
double srednia (std::initializer_list <double> liczby)
{
    double sr = 0.0;
    for (double &x : liczby)
        srednia += x;
    return (sr / initializer_list.size ());
}

double moja_srednia = srednia ({4.5, 5.0, 3.5, 4.5, 3.0});
```

map <class Ta, class Tb> – mapa (niekoniecznie drogowa)



```
#include <map>
```

```
std::string wyraz;  
std::cin >> wyraz;
```

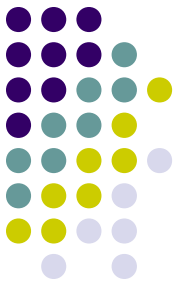
```
std::map<char, int> statystyka; // „tabela” z dwiema kolumnami
```

```
for (int i = 0; i < wyraz.size(); ++i)  
    statystyka [wyraz[i]] ++;          // MAGIA?!  
for (std::map<char, int>::const_iterator it = statystyka.begin();  
     it != statystyka.end(); ++it)  
    std::cout << „Litera „ << it->first << „ występuje w wyrazie „ <<  
        it->second << „ razy” << std::endl;
```

Efekt działania programu:

```
> tatarak  
Litera a występuje w wyrazie 3 razy  
Litera k występuje w wyrazie 1 razy  
Litera r występuje w wyrazie 1 razy  
Litera t występuje w wyrazie 2 razy
```

map <class Ta, class Tb> – jak to działa?



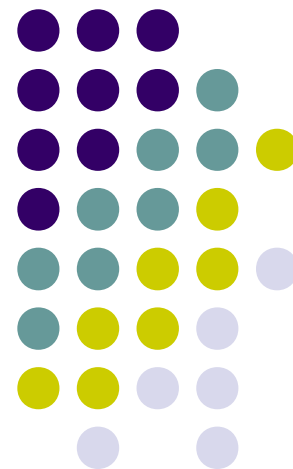
```
std::map<char, int> statystyka; // „tabela” z dwiema kolumnami
```

<char>	<int>
a	3
k	1
r	1
t	2

```
statystyka ['a'] ++;  
std::map<char, int>::iterator it = statystyka.begin()+1;  
  
std::cout << it->first << std::endl;  
std::cout << it->second << std::endl;
```

Sprytne wskaźniki

Sprytny nie musi oznaczać
„cfaniak”



Czemu warto być sprytnym?



```
try
{
    misiaczek *x = new misiaczek ();
    x->miziaj ();    // funkcja może rzucić wyjątek
    delete x;
}
catch (std::exception) // coś się popsło przy mizianiu
{
    // napraw sytuację
}
```

// ŻŁE: jeśli zostanie rzucony wyjątek, nie zwolnimy pamięci,
na którą wskazuje x!

Podstawowe sprytne wskaźniki



- `boost::scoped_ptr` – niekopiowalny
- `std::auto_ptr` – kopiowanie = przekazywanie wskaźnika
- `std::unique_ptr` – jw., ale w C++11
- `boost::shared_ptr` – wskaźnik z licznikiem
- `std::shared_ptr` – C++11

boost::scoped_ptr



```
#include <boost/scoped_ptr.hpp>
```

```
try
{
    boost::scoped_ptr<misiaczek> x (new misiaczek);
    x->miziaj ();    // funkcja może rzucić wyjątek
}
catch (std::exception)    // coś się popsło przy mizianiu
{
    // napraw sytuację
}
```

```
// DOBRZE: scoped_ptr wywoła destruktor ~misiaczek() przed swoją śmiercią
```


std::auto_ptr



```
#include <memory>
```

```
std::auto_ptr<BardzoDuzaKlasa> fabryka_bdk (int n)
{
    std::auto_ptr<BardzoDuzaKlasa> bdk (new BardzoDuzaKlasa (n));
    // (...) miziamy bdk
    return bdk;
}
```

```
int main ()
{
    fabryka_bdk (18); // DOBRZE: zwrócony wskaźnik jest niszczoney!
    // (...)
    std::auto_ptr<BardzoDuzaKlasa> bdk_33 = fabryka_bdk (33);
    // (...)
    return 0;
}
```

```
// DOBRZE: bdk_33 wywoła destruktor dla przechowywanego wskaźnika
```

boost::shared_ptr



```
#include <boost/shared_ptr.hpp>
```

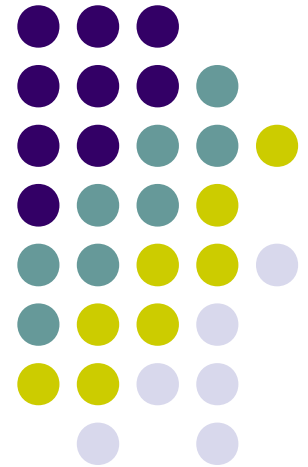
```
void miziadełko (boost::shared_ptr<NaszaKlasa> nk)
{
    nk->miziaj();
}
```

```
int main ()
{
    boost::shared_ptr<NaszaKlasa> nasza_klasa (new NaszaKlasa (id));
    // licznik (nasza_klasa) = 1
    miziadełko (nasza_klasa);           // licznik (nasza_klasa) = 2
    // licznik (nasza_klasa) = 1
    return 0;
    // licznik (nasza_klasa) = 0
}
```

```
// DOBRZE: nasza_klasa wywoła destruktora dla przechowywanego wskaźnika
```

Algorytmy STL

Nie wyważa się
otwartych drzwi
Nie odkrywa się Ameryki
po raz wtóry



Najfajniejsze algorytmy



```
#include <algorithm>
```

for_each – mizia wszystkie elementy w ten sam sposób

find – znajduje element w kontenerze

count – liczy wystąpienia elementów

copy – kopiuje wskazane elementy

generate – wypełnia zakres wartościami z generatora

sort - sortuje

min_element – znajduje najmniejszy element

max_element – znajduje największy element

Więcej algorytmów i szczegółów:

<http://www.cplusplus.com/reference/algorithm/>

for_each



```
template <class InputIterator, class Function>
    Function for_each (InputIterator first, InputIterator last, Function f);

void dodaj_piec (double &x)
{
    x += 5;
}

std::vector<double> tablica;

// (...) wypełnienie tablicy

// zwiększenie każdego elementu tablicy o 5
std::for_each (tablica.begin(), tablica.end(), dodaj_piec);

// UWAGA! Zamiast funkcji jako trzeciego argumentu może być użyty funktor.
```

count



```
template <class InputIterator, class T>
    typename iterator_traits<InputIterator>::difference_type
    count ( ForwardIterator first, ForwardIterator last, const T& value );

int liczba_trafien;
int moje_liczby[] = {10,20,30,30,20,10,10,20};    // 8 elementów
liczba_trafien = (int) count (moje_liczby, moje_liczby+8, 10);
std::cout << "Liczba 10 pojawia sie " << liczba_trafien << " razy.\n";
```

generate



```
template <class ForwardIter, class Generator>
    void generate (ForwardIter first, ForwardIter last, Generator gen );

double rozklad_gaussa ();          // generuje liczby z rozkładu Gaussa
std::vector<double> liczby (100);

// Wypełniamy całą tablicę liczbami z rozkładu normalnego
generate (liczby.begin(), liczby.end(), rozklad_gaussa);
```

sort, max_element, min_element



```
template <class RandomAccessIterator>
    void sort ( RandomAccessIterator first, RandomAccessIterator last );

template <class ForwardIter>
    ForwardIterator max_element ( ForwardIter first, ForwardIter last );
template <class ForwardIter>
    ForwardIterator min_element ( ForwardIter first, ForwardIter last );

std::vector<double> liczby;
// (...) wczytujemy liczby z pliku/klawiatury

std::cout << „Największa liczba:” <<
    *max_element (liczby.begin(), liczby.end()) << std::endl;
std::cout << „Najmniejsza liczba:” <<
    *min_element (liczby.begin(), liczby.end()) << std::endl;
sort (liczby.begin(), liczby.end());    // już posortowane :)
```

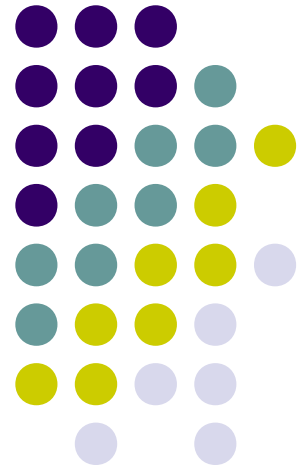

Bindowanie funkcji

Funktory

`std::bind`

`std::ref`

`std::function`



Funktor



```
class ObliczaczSredniej
{
    public:
        ObliczaczSredniej () : n_ (0), suma_ (0.0) {}
        void operator() (double liczba) {suma_ += liczba; ++n_;}
        double obliczSrednia () const {return suma_ / n_;}
    private:
        int n_;
        double suma_;
};

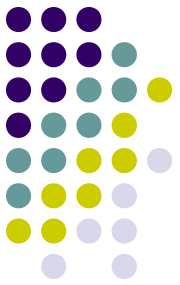
int main ()
{
    ObliczaczSredniej srednia;
    std::vector<double> tab;
    // (...) wczytujemy liczby do tab

    // FANTASTYCZNE: Obliczenie średniej z tabeli w jednej linijce!
    std::for_each (tab.begin(), tab.end(), srednia);

    std::cout << „Srednia liczb wynosi” << srednia.obliczSrednia();
}
```

std::bind

– czyli jak przytulić wszystkie misie?



```
#include <functional>

using namespace std::placeholders; // _1, _2, _3...

class Misio
{
    public:
        void przytul ();
        // (...)
};

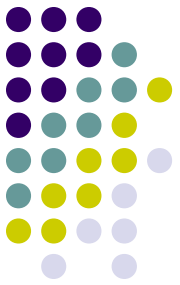
int main ()
{
    std::vector<Misio> misie;
    // (...)

    std::for_each (misie.begin(), misie.end(),
        std::bind (&Misio::przytul, _1));
    // FANTASTYCZNE: Wszystkie misie zostały przytulone!

    std::cout << "Misie się cieszą!" << std::endl;
}
```

std::bind

– wszystkie misie przytulamy mocno!



```
#include <functional>
```

```
class Misio  
{
```

```
    public:
```

```
        void przytul (bool czy_mocno);  
        // (...)
```

```
};
```

```
int main ()
```

```
{
```

```
    std::vector<Misio> misie;  
    // (...)
```

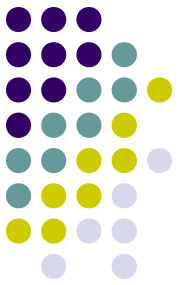
```
    std::for_each (misie.begin(), misie.end(),  
                  std::bind (&Misio::przytul, _1, true));  
    // FANTASTYCZNE: Wszystkie misie zostały mocno przytłone!
```

```
    std::cout << "Misie się cieszą jeszcze bardziej!" << std::endl;
```

```
}
```

std::ref

– dobra referencja nie jest zła!



```
#include <functional>
```

```
class ObliczaczSredniej
{
    public:
        ObliczaczSredniej () : n (0), suma (0.0) {}
        void dodaj (double liczba) {suma_ += liczba; ++n_;}
        double obliczSrednia () const {return suma_ / n_;}
    // private: (...)
};

int main ()
{
    ObliczaczSredniej srednia;
    std::vector<double> tab;
    // (...) wczytujemy liczby do tab

    // FANTASTYCZNE: Obliczenie średniej z tabeli w jednej linijce!
    std::for_each (tab.begin(), tab.end(),
        std::bind (&ObliczaczSredniej::dodaj, std::ref(srednia), _1));

    std::cout << "Srednia liczb wynosi" << srednia.obliczSrednia();
}
```

C++11

std::function

– przechowujemy funkcje



```
#include <functional>
```

```
void licz_od_do (int ood, int doo);
```

```
int main()
```

```
{
```

```
    std::function <void (int, int)> pf = licz_od_do;
```

```
    pf (3, 7);      // równoważne licz_od_do (3, 7);
```

```
    std::function <void (int)>
```

```
        licz_do = std::bind (licz_od_do, 1, _1);
```

```
    licz_do (8);    // równoważne licz_od_do (1, 8);
```

```
}
```

std::function jest obiektem klasy, zatem może być przechowywany w kontenerze, np. w wektorze:

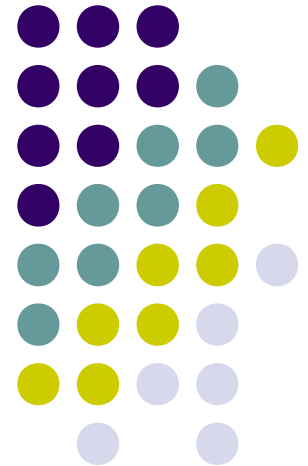
```
typedef std::function <void (int, int)> func_i_i;
```

```
std::vector <func_i_i> wektor_funkcji;  // FANTASTYCZNE!
```

Funkcje i wyrażenia lambda

Bardziej

λ



Funkcje anonimowe (*lambda*)



[domknięcie](parametry) -> zwracany_typ { ciało_funkcji }

```
int main ()
{
    // Wypisuje Brzydki kaczor!
    []() {std::cout << "Brzydki kaczor!";} ();

    // Uniwersalna funkcja obrażająca :)
    auto gbur = [] (std::string kogo) {std::cout << "Brzydki " <<
        kogo << "!";};
    gbur ("koczkodan");    // obraża koczkodana
    gbur ("kaczor");       // obraża kaczora

    // Zwraca x2
    auto kwadrat = [] (int x) {return x*x;};
    std::cout << kwadrat (7);

}
```

Obszerna lektura uzupełniająca: https://pl.wikipedia.org/wiki/Funkcja_anonimowa

std::lambda

– wyświetlenie całej tablicy



```
int main ()
{
    std::vector<int> liczby;
    // (...) wypełniamy tablicę

    std::for_each (liczby.begin(), liczby.end(),
        [] (int &x) {std::cout << x << ' ';} );

    // FANTASTYCZNE: Cała tablica wyświetlona na cout
}
```

boost::lambda

– wyświetlenie całej tablicy



```
#include <boost/lambda/lambda.hpp>
```

```
int main ()  
{
```

```
    std::vector<int> liczby;  
    // (...) wypełniamy tablicę
```

```
    std::for_each (liczby.begin(), liczby.end(),  
                  std::cout << boost::lambda::_1 << ' ');  
    // FANTASTYCZNE: Cała tablica wyświetlona na cout
```

```
}
```

std::lambda

– obliczanie średniej



```
int main ()
{
    std::vector<float> liczby;
    // (...) wypełniamy tablicę
    float srednia = 0.0;

    std::for_each (liczby.begin(), liczby.end(),
        [&srednia] (int &x) {srednia += x;} );

    srednia /= liczby.size();
    // FANTASTYCZNE: Średnia obliczona błyskawicznie

    std::cout << "Średnia liczb:" << srednia << std::endl;
}
```

boost::lambda

– obliczanie średniej



```
#include <boost/lambda/lambda.hpp>

int main ()
{
    std::vector<float> liczby;
    // (...) wypełniamy tablicę
    float srednia = 0.0;

    std::for_each (liczby.begin(), liczby.end(),
                  srednia += boost::lambda::_1);

    srednia /= liczby.size();
    // FANTASTYCZNE: Średnia obliczona błyskawicznie

    std::cout << "Średnia liczb:" << srednia << std::endl;
}
```

Są jeszcze funkcje lambda:

http://en.wikipedia.org/wiki/Anonymous_function#C.2B.2B_.28since_C.2B.2B11.29

Programowanie jest fantastyczne!!!

