

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

## Wykład 10 - szablony, trejty, STL

Robert Nowak

24L

## Szablon (powtórzenie) - pojęcia niezależnie od typu

- ▶ polimorfizm: statyczny (szablony), dynamiczny (f. wirtualne)
- ▶ kod nie jest kompilowany, jeżeli nie został użyty
- ▶ brak narzutów czasowych w czasie wykonania
- ▶ obsługuje typy wbudowane
- ▶ definicja (a nie deklaracja) ma być widoczna w miejscu użycia

nagłówek nazwa.hpp	<pre>template&lt;typename T&gt; class Foo {     /* Definicja klasy i metod inline */ }; #include "kontener.hpp"</pre>
impl. szablonu nazwa.hpp	<pre>template&lt;typename T&gt; T&amp; Foo&lt;T&gt;::get(int index) {     /* kod szablonu */ }</pre>
implementacja nazwa.cpp	<pre>/* kod, który nie jest szablonem */</pre>

## Listy inicjujące (C++11) - przykład szablonu

```
int tab[] = { 2, 3, 5, 8 }; //poprawne w C++03
struct X {string s; int i; };
X tabX[] = { { "A", 2}, {"B", 3} }; //poprawne w C++03

struct Foo {
    Foo(std::initializer_list<int> val);
};
Foo f = { 2, 3, 5, 8 }; //obiekt inicjowany listą wartości

void fun(std::initializer_list<double> val) {
    for (double x : val)
        std::cout << x << std::endl; //print all values
}
fun({2.8, 4.7, 9.12}); //przekazanie listy wartości do funkcji
```

# TMP - template metaprogramming

## Konkretyzacja i specjalizacja szablonu równoważna Maszynie Turinga

```
/* Przykład - funkcja obliczająca silnię w TMP */  
template<unsigned n>  
struct Silnia {  
    static const int value = n*Silnia<n-1>::value;  
};  
template<> struct Silnia<0> {  
    static const int value = 1;  
};  
cout << Silnia<0>::value << endl; //1  
cout << Silnia<5>::value << endl; //120
```

## TMP - template metaprogramming (2)

```
//szablon oblicza potęgę o wykładniku całkowitym
template <unsigned n> double int_power(double x) {
    return int_power<2>( int_power<n/2>(x) )*int_power<n%2>(x);
}
//specjalizacja dla kwadratu
template <> double int_power<2>(double x) { return x*x; }

template <> double int_power<1>(double x) { return x; }

template <> double int_power<0>(double x) { return 1.0; }
```

Przykład:

$$x^{35} = x^{32+2+1} = (((((x^2)^2)^2)^2)^2 * (x^2) * x$$

# Szablony kontra hierarchia klas

Używaj szablonów (a nie hierarchii klas) gdy:

- ▶ nie można utworzyć wspólnej klasy bazowej, ważne są typy wbudowane
- ▶ bardzo ważna jest efektywność
- ▶ akceptowalna powtórna kompilacja przy nowym typie

W przeciwnym wypadku: hierarchia klas (funkcje wirtualne)

## Szablon czy hierarchia klas (2)

- ▶ szablony: typ obiektu nie modyfikuje działania klasy
- ▶ funkcje wirtualne: typ obiektu modyfikuje działanie klasy

### Działanie stosu nie jest zależne od typu

```
template<typename T> class Stack {  
    void push(const T& t);  
    T pop();  
    /* ... */  
};
```

```
class Stack {  
    void push(Base* b);  
    Base* pop();  
    /* ... */  
};
```

## Szablon czy hierarchia klas (3)

- ▶ szablony: typ obiektu nie modyfikuje działania klasy
- ▶ funkcje wirtualne: typ obiektu modyfikuje działanie klasy

### Działanie figury jest zależne od jej typu

```
template<class T> class Figure {
    double getArea();
    /* ... */
};
template<> class Figure<Rect> {
    double getArea() {
        return t.getW()*t.getH();
    }
    /* ... */
};
```

```
class Figure {
    virtual double getArea() = 0;
    /* ... */
};
class Rect : public Figure {
    virtual double getArea() {
        return getW() * getH();
    }
};
```



# Problemy przy stosowaniu szablonów

Implementacja i testowanie: kod nie jest kompilowany

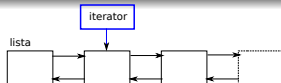
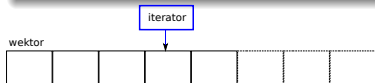
- ▶ utworzyć algorytm lub kontener dla konkretnego typu (np. `int`)
- ▶ przetestować go
- ▶ przekształcić go w szablon
- ▶ wygenerować i przetestować kilka różnych typów

Używanie:

- ▶ brak mechanizmów rozróżniania typu w szablonach
- ▶ nieczytelne komunikaty o błędach

# Iterator (powtórzenie), STL kategorie iteratorów

Iterator - wzorzec projektowy, obiekt pośredniczy przy dostępie do elementów kolekcji.



- ▶ `input_iterator`, `output_iterator`
  - ▶ przesuwanie się do przodu o jeden element (inkrementacja)
  - ▶ jednokrotny odczyt (`input_iterator`) lub zapis (`output_iterator`) wartości wskazywanej
- ▶ `forward_iterator`
  - ▶ przesuwanie się do przodu o jeden element
  - ▶ wielokrotny odczyt i zapis wartości wskazywanej przez iterator
- ▶ `bidirectional_iterator`
  - ▶ j.w oraz przesuwanie się do tyłu o jeden element
- ▶ `random_access_iterator`
  - ▶ j.w oraz przesuwanie się do przodu i do tyłu o 'n' elementów

# Wykorzystanie „trejtów” (trait) do informacji o typie

```
//Przykład funkcji, która wykorzystuje informację o typie
```

```
template<typename I> void advance(I& iter, int d) {  
    if(/* iter jest typu random_access */)   
        iter += d;  
    else {  
        if( d >= 0) { while (d--) ++iter; }  
        else { while (d++) --iter; }  
    }  
}
```

```
//Klasy do rozróżniania kategorii iteratora
```

```
struct input_iterator_tag {};  
struct output_iterator_tag {};  
struct forward_iterator_tag : public input_iterator_tag {};  
struct bidirectional_iterator_tag : public forward_iterator_tag {};  
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

## Wykorzystanie „trejtów” (trait) do informacji o typie (2)

```
template</* ... */> class vector {
public:
    class iterator {
    public:
        using iterator_category = random_access_iterator_tag; //istotne!
    };
};

template</* ... */> class list {
public:
    class iterator {
    public:
        using iterator_category = bidirectional_iterator_tag;
    };
};

//odczyt kategorii
template<typename I> struct iterator_traits {
    using iterator_category = I::iterator_category;
};
```

## Wykorzystanie „trejtów” (trait) do informacji o typie (3)

```
template<typename I>
inline void doAdvance(I& it, int d, random_access_iterator_tag) {
    it += d;
}
template<typename I>
inline void doAdvance(I& iter, int d, bidirectional_iterator_tag) {
    if( d >= 0) { while (d--) ++it; }
    else { while (d++) --it; }
}
template<typename I>
inline void doAdvance(I& it, Dist d, input_iterator_tag) {
    if(d < 0) throw out_of_range(/* ... */);
    while (d--) ++it;
}
//Implementacja - dodatkowy parametr określa typ iteratora
template<typename I> void advance(I& it, int d) {
    doAdvance(it, d, iterator_traits<I>::iterator_category() )
}
```

## Trejty dostarczane przez bibliotekę standardową

iterator_traits	
char_traits	eq, lt, int_type
numeric_limits	min, max, quiet_NaN, infinity, is_signed, is_integer, digits, digits10, has_infinity, has_quiet_NaN

```
std::numeric_limits<double>::min() //Zamiast __DBL_MIN__
std::numeric_limits<int>::min() //Zamiast INT_MIN
!(x == x); //badanie czy liczba jest nan
x == std::numeric_limits<double>::quiet_NaN(); //ŹLE!
```

```
template<typename T> T findMax(const T const * data, const size_t num) {
    T largest = std::numeric_limits< T >::min(); //inicjacja
    for (unsigned int i=0; i < num; ++i)
        if (data[i] > largest) largest = data[i];
    return largest;
}
```

## <type\_traits> (boost::type\_traits), ok. 50 trejtów

```
std::cout << std::is_void<void>::value; //true  
std::cout << std::is_void<int>::value; //false
```

podstawowe	is_void, is_null_pointer, is_integral, is_floating_point, is_array, is_enum, is_union, is_class, is_function, is_pointer, is_lvalue_reference, is_rvalue_reference, ...
kompozycja	is_fundamental, is_arithmetic, is_object, ...
właściwości	is_const, is_volatile, is_pod, is_empty, ...
relacje	is_base_of, is_convertible, is_same
pomocnicze	is_constructible, is_default_constructible, ...

## nieczytelne komunikaty o błędach

```
std::set<Foo> s; //Klasa Foo nie dostarcza operatora '<'  
s.insert( Foo(3) ); //ta linia jest oznaczana jako błędna
```

```
/usr/include/c++/4.2/bits/stl_function.h:  
In member function 'bool std::less<Tp>::operator()(const Tp&, const Tp&) const  
[with Tp = Foo]':  
/usr/include/c++/4.2/bits/stl_tree.h:980: instantiated from '  
std::pair<typename std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare, _Alloc>::iterator, bool>  
std::_Rb_tree<_Key, _Val, _KeyOfValue, _Compare, _Alloc>::_M_insert_unique(const _Val&)  
[with _Key = Foo, _Val = Foo, _KeyOfValue = std::_Identity<Foo>, _Compare = std::less<Foo>,  
_Alloc = std::allocator<Foo>']  
/usr/include/c++/4.2/bits/stl_set.h:307: instantiated from '  
std::pair<typename std::_Rb_tree<_Key, _Key, std::_Identity<_Key>, _Compare,  
typename _Alloc::rebind<_Key>::other>::const_iterator, bool>  
std::set<_Key, _Compare, _Alloc>::insert(const _Key&)  
[with _Key = Foo, _Compare = std::less<Foo>, _Alloc = std::allocator<Foo>']  
/usr/include/c++/4.2/bits/stl_function.h:227: error:  
no match for 'operator'< in 'x < 'yy
```

## Rozwiązanie: asercja czasu kompilacji (od C++11)

```
static_assert( predicate, message );  
static_assert( predicate );
```

pozwalą poprawnie wskazywać błędny kod



# Koncepty (dawniej Boost.ConceptCheck)

concepts (od C++20) - rozwinięcie warunków dla szablonów

```
BOOST_CONCEPT_ASSERT(( koncepcja )); //podwójne nawiasy
```

podstawowe	ConvertibleConcept AssignableConcept EqualityComparableConcept LessThanComparableConcept ...
iteratory	BidirectionalIteratorConcept RandomAccessIteratorConcept ...
funktory	GeneratorConcept UnaryFunctionConcept ...
kontenery	ForwardContainerConcept RandomAccessContainerConcept ...

```
//Przykład - typ musi dostarczać operatora porównania
template<typename T>
void funkcja(T t) {
    BOOST_CONCEPT_ASSERT(( LessThanComparableConcept<T>));
    /* ... */
}
```

## Boost.ConceptCheck - definiowanie własnych wymagań

```
//przykład, wymagany operator== i metody get i set
template<typename T> struct MyConcept : boost::EqualityComparable<T> {
    BOOST_CONCEPT_USAGE(MyConcept) {
        t.get(); //musi mieć metodę get
        t.set(1); //musi mieć metodę set z argumentem int
    }
    T t;
};

template<typename T> void fxx(T t) { //Przykład użycia
    BOOST_CONCEPT_ASSERT((MyConcept<T>)); //przykład użycia
}

fxx(3); //linia 48, użycie szablonu dla typu, który nie spełnia wymagań
pr.cpp: In member function 'void MyConcept<T>::constraints() [wi ...
...
pr.cpp:48: instantiated from 'void fxx(T) [with T = int]'
pr.cpp:40: error: request for member 'get in '((MyConcept<int>*)...
pr.cpp:41: error: request for member 'set in '((MyConcept<int>*)...
```

## Curiously recurring template pattern - efekt jak f. wirtualne

```
template<typename T> struct base {
    //wybór w czasie kompilacji!
    int calc { return static_cast<T*>(this)->doCalc(); }
};
struct derived : base<derived> { //dziedziczy i jest parametrem
    int doCalc() { /* implementacja */ }
};
//Przykład użycia (Barton-Nackman trick)
template<typename T> struct less_comparable {
    friend bool operator>=(const T& a, const T& b) { return !(a<b); }
    friend bool operator>(const T& a, const T& b) { return b<a; }
    friend bool operator<=(const T& a, const T& b) { return !(b<a); }
};
class value_type : less_comparable<value_type> { //ma: <, <=, >, >=
public:
    bool operator<(const value_type& v) { /* ... */ }
};
value_type a, b;
a > b; //szablon klasy bazowej generuje odpowiednie funkcje
```

## szablony jako parametry szablonów

```
template<typename T, typename Contener = std::deque<T> > class Stack {
public:
    class EmptyStackException : public std::exception {};
    bool empty() const { return c.empty(); }
    T pop() {
        if( empty() ) throw EmptyStackException();
        T ret = c.back();
        c.pop_back();
        return ret;
    }
    void push(const T& t) { c.push_back(t); }
private:
    Contener c;
};

Stack<int> s; //wykorzystuje std::queue
Stack<int, std::list<int> > s2; //wykorzystuje std::list
Stack<int, std::list<std::string> > s2; //Błąd!
```

## szablony jako parametry szablonów (2)

```
template<typename T,  
        template <typename E, typename A> class Contener = std::deque>  
class Stack {  
public:  
    class EmptyStackException : public std::exception {};  
    bool empty() const { return c.empty(); }  
    T pop() {  
        if (empty() ) throw EmptyStackException();  
        T ret = c.back();  
        c.pop_back();  
        return ret;  
    }  
    void push(const T& t) { c.push_back(t); }  
private:  
    Contener<T, std::allocator<T> > c;  
};  
Stack<int> s; //wykorzystuje std::queue  
Stack<int, std::list > s2; //wykorzystuje std::list
```

# Biblioteka STL (standardowa w C++ od 1997)

## Skład:

- ▶ kontenery
- ▶ iteratory
- ▶ algorytmy
- ▶ funktory
- ▶ adaptery
- ▶ alokatory

## Cechy:

- ▶ wykorzystuje mechanizm szablonów
  - ▶ kod wielokrotnego użycia
  - ▶ można stosować do typów wbudowanych
- ▶ wykorzystuje wzorzec iteratora (zmniejsza liczbę algorytmów)
- ▶ bezpieczna w aplikacjach wielowątkowych
- ▶ bardzo efektywna

Omówiona wcześniej na ZPR

# Kontenery - lista kolekcji standardowych

sekwencyjne	<b>basic_string</b> <b>vector</b> <b>list</b> <b>deque</b>	jednowymiarowa tablica jednowymiarowa tablica lista dwukierunkowa kolejka o dwu końcach
asocjacyjne	<b>set</b> <b>map</b> <b>multiset</b> <b>multimap</b>	usuwa elementy równoważne tablica asocjacyjna (słownik) nie usuwa el. równoważnych klucz może wyst. wielokrotnie
haszujące	<b>unordered_set</b> <b>unordered_map</b> <b>...multiset</b> <b>...multimap</b>	zbiór słownik wielozbiór wielosłownik

# Adaptery

adaptery	
<code>queue&lt;T&gt;</code>	kolejka
<code>priority_queue&lt;T&gt;</code>	kolejka priorytetowa
<code>stack&lt;T&gt;</code>	stos

Inny podział kontenerów:

**bazujące na tablicach:** vector, basic\_string, deque

**bazujące na węzłach:** list, set, map, multiset, multimap, unordered\_set, unordered\_map, unordered\_multiset, unordered\_multimap



# Podstawowe cechy kontenerów

**operuje na kopiach, kontener przechowuje kopie elementów**

obiekty przechowywane w kontenerach:

- ▶ powinny mieć publiczny konstruktor kopiujący
- ▶ powinny mieć publiczny operator przypisania

gdy kopiowanie kosztowne: przechowywanie wskaźników,  
`std::unique_ptr`, `std::shared_ptr`, `std::ref`

współbieżność:

bezpiecznie czytanie	wiele wątków może równocześnie czytać z kontenera
bezpieczne pisanie do różnych kontenerów	różne wątki mogą równocześnie pisać do różnych kontenerów

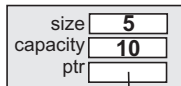
# std::vector (#include <vector>) - omówiony wcześniej

```
//Deklaracja
template<class T, class A=allocator<T> > class std::vector;

//Najważniejsze metody dostępu
reference operator[] (size_type n);
reference at(size_type n);
front(); back();
//Modyfikacja kolekcji
void push_back(const&T);
T pop_back();

//Ważne funkcje pomocnicze
bool operator==
bool operator<
//Specjalizacje
vector<bool>
```

vector



# std::list (#include <list>)

//Deklaracja

```
template<class T, class A=allocator<T> > class std::list;
```

//Metody dostępu - brak operatora[]

front() back()

//Modyfikacja kolekcji

push\_front(), push\_back()

pop\_front(), pop\_back()

insert

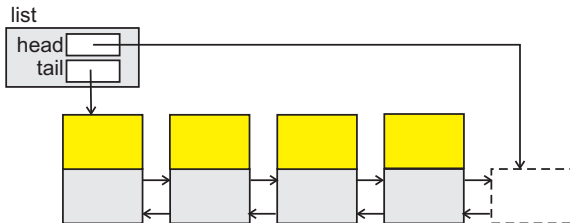
erase

//Inne metody

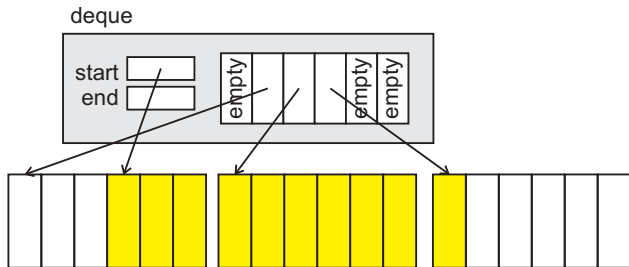
void remove(const T& val)

void reverse()

splice merge sort



# std::deque - kolejka o dwu końcach (`#include <deque>`)



Operacje:

- ▶ takie jak dla wektora (za wyjątkiem `capacity()`, `reserve()` )
- ▶ operacje z przodu ciągu (tak jak dla listy)

# kontenery asocjacyjne

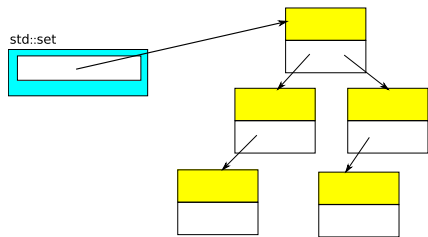
- ▶ przechowują elementy uporządkowane
- ▶ dostarczają metod wyszukiwania w czasie logarytmicznym
- ▶ **Alternatywa: posortowany wektor**

ale:

- ▶ porządek elementów inny niż kolejność wstawiania
- ▶ iterator typu `bidirectional`, algorytmy często wymagają `random_access`

## std::set - zbiór

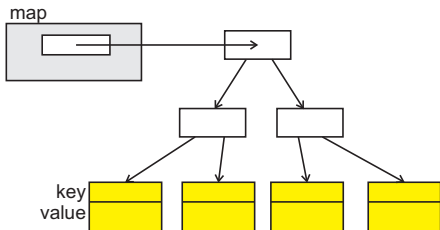
- ▶ eliminacja elementów równoważnych  
a jest równoważne b, jeżeli  $!(a < b) \ \&\& \ !(b < a)$



```
set<int> s; //zbiór liczb całkowitych
s.insert(1); //dodaje elementy do kolekcji
s.insert(1); //ta operacja jest pusta (usuwa elementy równoważne)
s.insert(2);
assert( s.size() == 2); //bada liczbę elementów
assert( s.count(1) == 1 ); //zlicza liczbę wystąpień elementu
```

# std::map - słownik (tablica asocjacyjna)

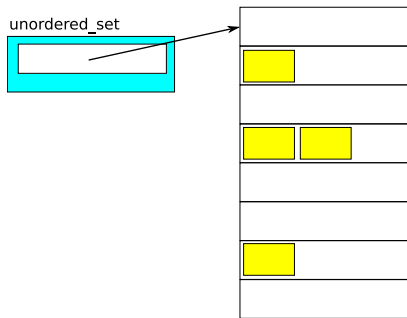
```
template<class K, class T, //typ klucza i wartości  
        class Cmp = less<Key>, //Porządek dla klucza  
        class A = allocator<Pair<const K, T> > > class std::map;
```



```
iterator find(const K& k);  
size_type  
count(const K& k) const;  
pair<iter,iter>  
equal_range(const K& k);
```

# std::unordered\_set (C++11) lub boost::unordered\_set

- ▶ nie zostały objęte standardem w 1999 roku
- ▶ dostawcy kompilatorów (własne rozwiązania): hash\_set, hash\_map, itd.
- ▶ standard C++11: unordered\_set itd.
- ▶ wykorzystują funkcję skrótów
- ▶ w przybliżeniu jednostkowy czas dostępu do elementów
- ▶ cena: zajętość pamięci (działa dobrze gdy wypełniony w ok. 25%)





## std::array (C++11) lub boost::array

- ▶ Adapter dla tablic z C
- ▶ dostarcza iteratorów oraz metod begin, end, rbegin, rend
- ▶ dostarcza metody: at, front, back i in.
- ▶ dostarcza operator==, operator<

```
template<typename T, std::size_t N>
class array {
public:
    //...
    T elems[N];
};
```

*//Przykłady użycia*

```
array<double, 4> tab = { 1.0, 1.1, 1.2, 1.3 };
array<double, 4> tab2(tab);
cassert( tab == tab2 );
tab2.assign(0.0); //wszystkie elementy będą miały wartość 0
```

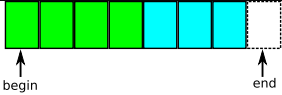
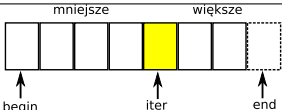
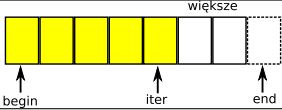
# Porównanie kolekcji

<b>cecha</b>	<b>kontener</b>
element w ustalonej pozycji kontenera	vector, string, list, deque
dostęp przez indeks	vector, string, deque
ważna szybkość wyszukiwania	posortowane wektory, kontenery asocjacyjne, kontenery haszujące
iteratory, wskaźniki nie mogą być unieważniane	lista, asocjacyjne
j.w. dla operacji modyfikującej koniec kolekcji	j.w. oraz deque
brak elementów równoważnych	set, map
nieistotny porządek w jakim elementy są przechowywane	kontenery haszujące

## algorytmy STL, omówione wcześniej

wzorzec iteratora, bardzo wydajne, wymagają funkcji lambda

przykład: algorytmy sortujące

partition	dzieli elementy używając predykatu	
stable_partition	jak wyżej	
nth_element	znajduje pozycję danego elementu	
partial_sort	sortuje częściowo	
sort	sortuje	
stable_sort	j.w, zachowuje porządek dla el. równoważnych	

## przykład użycia algorytmów: sortowanie

```
//Przykład
struct Prac {
    string nazw;
    int wiek;
    int pensja;
};
vector<Prac> p;

//Sortuje po wieku, bind do wiązania składowej
sort( p.begin(), p.end(),
      bind( less<int>(), bind(&Prac::wiek,_1), bind(&Prac::wiek,_2)) );

//Sortuje po wieku, lambda
sort( p.begin(), p.end(),
      [](const Prac& p, const Prac& r){ return p.wiek < r.wiek; });

//Sortowanie po pensji
sort( p.begin(), p.end(),
      [](const Prac& p, const Prac& r){ return p.pensja < r.pensja; });
```

*Dziękuję*