Instytut Informatyki UMCS
Zakład Technologii Informatycznych

Wykład 10

Biblioteka kontenerów

STL

dr Marcin Denkowski

Lublin, 2019

AGENDA

1. Co to jest STL
2. Kontenery
3. Iteratory

STANDARD TEMPLATE LIBRARY

- **STL** – **Standard Template Library** – biblioteka generyczna C++ zawierająca kontenery, iteratory, obiekty funkcyjne i algorytmy w formie szablonów
- Zapoczątkowana przez Hewlett Packard (~1994)
- Weszła do biblioteki standardowej C++
- Obecnie nazwa STL ma znaczenie jedynie historyczne

BIBLIOTEKA STL

- **STL** – Standard Template Library
  – Standardowa Biblioteka Szablonów

- **Główne składowe:**

1. **Kontenery** – obiekty zbiorcze, generyczne i homogeniczne, które przechowują w pewien konkretny sposób inne obiekty

2. **Iteratory** – uogólnione wskaźniki umożliwiające poruszanie się po kontenerach

3. **Algorytmy** – zestaw szablonów funkcji wykonujących pewne operacje na kontenerach

TYPY KONTENERÓW

- **Kontenery sekwencyjne** – dostęp sekwencyjny – elementy są ułożone jeden za drugim (logicznie)
  - `array, vector, list, forward_list, dequeue`
- **Kontenery asocjacyjne** – elementy są posortowane, szybki dostęp
  - `set, map, multiset, multimap`
- **Kontenery asocjacyjne nieuporządkowane** – nieuporządkowane dane, szybki dostęp
  - `unordered_set, unordered_map, unordered_multiset, unordered_multimap`
- **Adaptery kontenerów** – inny interfejs dla kontenerów sekwencyjnych
  - `stack, queue, priority_queue`

ITERATORY

1. **Iteratory** – specjalne obiekty przeznaczone do poruszania się po kontenerach

2. Koncepcyjnie jest rozwinięciem idei wskaźnika

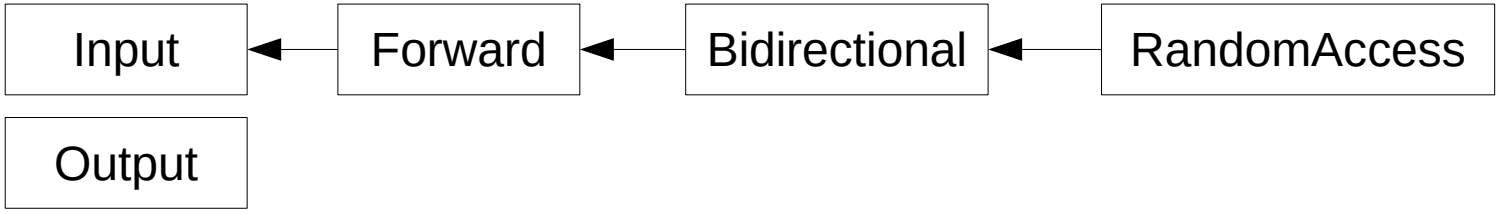
3. Posiada operator dereferencji (*), który umożliwia pobranie wartości obiektu z kontenera, na który wskazuje

4. W zależności od typu kontenera, odpowiedni iterator może posiadać operacje arytmetyczne umożliwiające przesuwanie go po obiektach kontenera, np.:
   - `operator++`
   - `operator--`
   - `operator+ | operator-`
   - `operator+= | operator-=`
   - `operatory= | == | < |  >`

5. Każdy kontener może definiować swój własny typ iteratora

ITERATORY

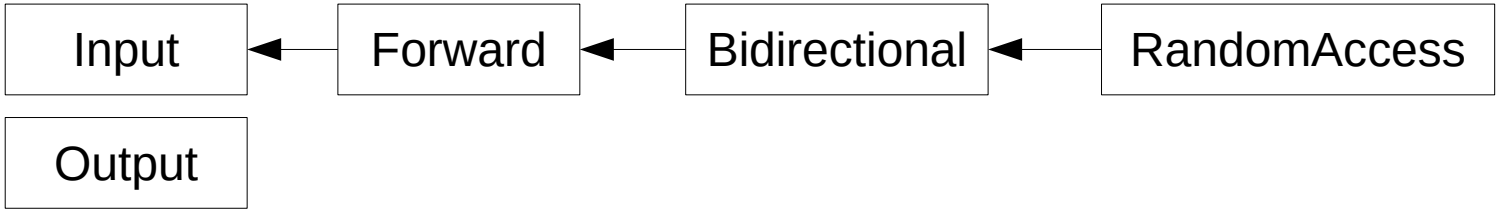
Zdefiniowanych jest 5 kategorii iteratorów:



- **Iteratory wejściowe** (*Input Iterator*) – odczyt wartości z kontenera
- **Iteratory wyjściowe** (*Output Iterator*) – zapis wartości z kontenera
- **Iteratory postępujące** (*Forward Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element do przodu (operator++)
- **Iteratory dwukierunkowe** (*Bidirectional Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element w obu kierunkach (operator++, operator--)
- **Iteratory dostępu swobodnego** (*Random Access Iterator*) – umożliwiają dostęp swobodny do elementów kontenera (operator+=, -=, ...)

ITERATORY

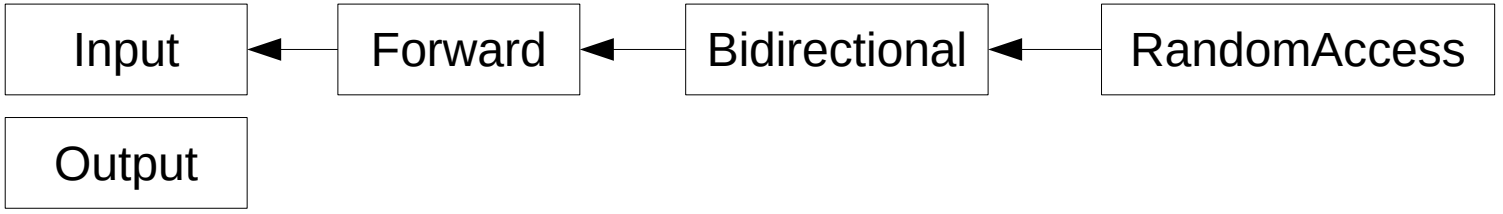
Zdefiniowanych jest 5 kategorii iteratorów:



- **Iteratory wejściowe** (*Input Iterator*) – odczyt wartości z kontenera
- **Iteratory wyjściowe** (*Output Iterator*) – zapis wartości z kontenera
- **Iteratory postępujące** (*Forward Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element do przodu (`operator++`)
- **Iteratory dwukierunkowe** (*Bidirectional Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element w obu kierunkach (operator++, operator--)
- **Iteratory dostępu swobodnego** (*Random Access Iterator*) – umożliwiają dostęp swobodny do elementów kontenera (operator+=, -=, ...)

ITERATORY

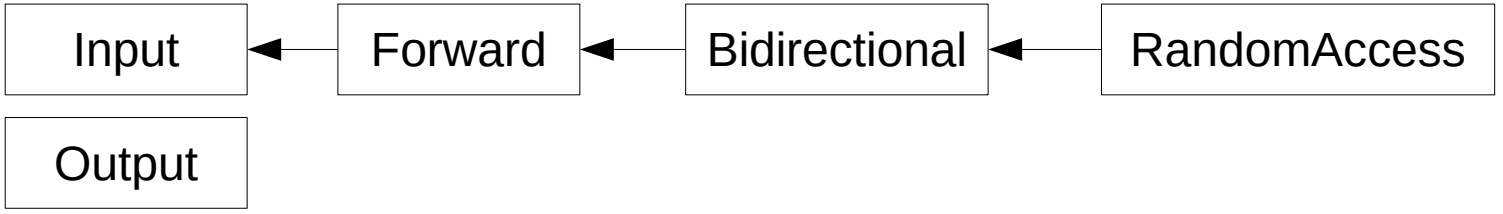
Zdefiniowanych jest 5 kategorii iteratorów:



- **Iteratory wejściowe** (*Input Iterator*) – odczyt wartości z kontenera
- **Iteratory wyjściowe** (*Output Iterator*) – zapis wartości z kontenera
- **Iteratory postępujące** (*Forward Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element do przodu (operator++)
- **Iteratory dwukierunkowe** (*Bidirectional Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element w obu kierunkach (`operator++, operator--`)
- **Iteratory dostępu swobodnego** (*Random Access Iterator*) – umożliwiają dostęp swobodny do elementów kontenera (operator+=, -=, ...)

ITERATORY

Zdefiniowanych jest 5 kategorii iteratorów:



- **Iteratory wejściowe** (*Input Iterator*) – odczyt wartości z kontenera
- **Iteratory wyjściowe** (*Output Iterator*) – zapis wartości z kontenera
- **Iteratory postępujące** (*Forward Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element do przodu (operator++)
- **Iteratory dwukierunkowe** (*Bidirectional Iterator*) – umożliwiają poruszanie się po kontenerze o jeden element w obu kierunkach (operator++, operator--)
- **Iteratory dostępu swobodnego** (*Random Access Iterator*) – umożliwiają dostęp swobodny do elementów kontenera (`operator+=, -=`, ...)

ARRAY

```
template <class T, size_t N> class array;
```

1. Kontener sekwencyjny o stałej liczbie elementów
2. Implementacja „jest” tak prosta/szybka jak zwykła tablica automatyczna
3. Mogą być traktowane jak obiekty klasy `tuple` – działa na nich funkcja `get<>()`
4. Obsługiwany przez *Random Access Iterator*

ARRAY

```
template <class T, size_t N> class array;
```

1. Kontener sekwencyjny o stałej liczbie elementów
2. Implementacja „jest” tak prosta/szybka jak zwykła tablica automatyczna
3. Mogą być traktowane jak obiekty klasy `tuple` – działa na nich funkcja `get<>()`
4. Obsługiwany przez *Random Access Iterator*
5. Zestaw metod/funkcji:
   - `iterator begin()/end()` – random access iterator
   - `int size()`
   - `bool empty()`
   - `T& opeartor[](int)` – dostęp do elemetu i-tego
   - `T& at(int)` – dostęp do elemetu i-tego (sprawdza zakres i rzuca wyjątek `out_of_range`)
   - `T& front()/back()` – dostęp do pierwszego/ostatniego elementu
   - `T* data()` – wskaźnik na pierwszy element
   - `void fill(T&)` – ustawia wszystkie elementy na podany

VECTOR

```
template <class T, class Alloc=allocator<T>>
class vector;
```

1. Kontener sekwencyjny o dynamicznej ilości elementów
2. Tak jak tablica automatyczna używa ciągłej przestrzeni pamięci, co umożliwia używanie zwykłych wskaźników na elementach
3. Używa dynamicznego zarządzania pamięcią – co może być „drogie” przy niektórych operacjach
4. Zapewnia bardzo szybki *O(1)* dostęp do elementów i dodawanie/usuwanie na końcu
5. Obsługiwany przez *Random Access Iterator*

VECTOR

```
template <class T, class Alloc=allocator<T>>
class vector;
```

1. Zestaw metod/funkcji:
   - `iterator begin()/end()` – random access iterator
   - `int size()`
   - `int capacity()` – ilość zaalokowanej przestrzeni
   - `void resize(int)` – wymusza zmianę rozmiaru
   - `void reserve(int)` – wymusza zmianę zaalokowanej przestrzeni
   - `bool empty()`
   - `T& opeartor[](int)` – dostęp do elemetu i-tego
   - `T& at(int)` – dostęp do elemetu i-tego (sprawdza zakres i rzuca wyjątek `out_of_range`)
   - `T& front()/back()` – dostęp do pierwszego/ostatniego elementu
   - `T* data()` – wskaźnik na pierwszy element

VECTOR

```
template <class T, class Alloc=allocator<T>>
class vector;
```

1. Zestaw metod/funkcji:
   - `void assign(InputIterator first, InputIterator last)` – przypisuje nową zawartość
   - `void push_back(T&)` – wstawia element na koniec kontenera (możliwa realokacja)
   - `void pop_back()` – usuwa ostatni element
   - `iterator insert(iterator pos, ...)` – wstawia element lub grupę elementów
   - `iterator erase(iterator pos, ...)` – usuwa element lub grupę elementów
   - `void clear()` – usuwa wszystkie elementy
   - `iterator emplace(iterator pos, ...)` – wstawia konstruowany element na pozycję
   - `iterator emplace_back(...)` – wstawia konstruowany element na koniec kontenera

LIST

```
template <class T, class Alloc=allocator<T>>
class list;
```

1. Kontener sekwencyjny o dynamicznej ilości elementów
2. Zapewnia stały czas *O(1)* wstawiania i usuwania elementów gdziekolwiek w kontenerze
3. Nie udostępnia iteratora o dostępie swobodnym
4. Elementy w kontenerze mogą być rozmieszczone gdziekolwiek w pamięci
5. Realizowany na zasadzie listy dwukierunkowa (wiązanej dwukierunkowo) (*double-linked list*)
6. Obsługiwany przez *Bidirectional Iterator*

LIST

```
template <class T, class Alloc=allocator<T>>
class vector;
```

1. Zestaw metod/funkcji:

- `iterator begin()/end()` – bidirectional iterator

- `int size()`
- `void resize(int)` – wymusza zmianę rozmiaru
- `bool empty()`

- `void sort(Compare)` – sortuje elementy w porządku rosnącym
- `void unique()` – usuwa sąsiednie duplikaty
- `void merge(list)` – łączy dwie listy
- `void reverse()` – zamienia kolejność elementów
- `void remove(UnaryPred)` – usuwa elementy, które spełniają kryterium
- `void splice(iterator po, list)` – przenosi elementy

LIST

```
template <class T, class Alloc=allocator<T>>
class list;
```

1. Zestaw metod/funkcji:
   - `T&` `front()/back()` – dostęp do pierwszego/ostatniego elementu
   - `void` `push_back(T&)` – wstawia element na koniec kontenera
   - `void` `push_front(T&)` – wstawia element na początek kontenera
   - `void` `pop_back()` – usuwa ostatni element
   - `void` `pop_front()` – usuwa pierwszy element
   - `iterator insert(iterator pos, ...)` – wstawia element lub grupę elementów
   - `iterator erase(iterator pos, ...)` – usuwa element lub grupę elementów
   - `void` `clear()` – usuwa wszystkie elementy
   - `iterator emplace(iterator pos, ...)` – wstawia konstruowany element na pozycję pos
   - `iterator emplace_back(...)` – wstawia konstruowany element na koniec kontenera
   - `iterator emplace_front(...)` – wstawia konstruowany element na początek kontenera

		Sequence containers				
Header		<array>	<vector>	<deque>	<forward_list>	<list>
Container		array	vector	deque	forward_list	list
	(constructor)	(implicit)	vector	deque	forward_list	list
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list
	operator=	(implicit)	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign
Iterators	begin	begin	begin	begin	begin	begin
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin
	end	end	end	end	end	end
	cend	cend	cend	cend	cend	cend
	rbegin	rbegin	rbegin	rbegin		rbegin
	crbegin	crbegin	crbegin	crbegin		crbegin
	rend	rend	rend	rend		rend
	crend	crend	crend	crend		crend
Element access	at	at	at	at		
	operator[]	operator[]	operator[]	operator[]		
	data	data	data			
	front	front	front	front	front	front
	back	back	back	back		back
Capacity	empty	empty	empty	empty	empty	empty
	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	resize		resize	resize	resize	resize
	capacity		capacity			
	reserve		reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
Modifiers	clear		clear	clear	clear	clear
	insert		insert	insert	insert_after	insert
	insert_or_assign					
	emplace		emplace	emplace	emplace_after	emplace
	emplace_hint					
	try_emplace					
	erase		erase	erase	erase_after	erase
	push_front			push_front	push_front	push_front
	emplace_front			emplace_front	emplace_front	emplace_front
	pop_front			pop_front	pop_front	pop_front
	push_back		push_back	push_back		push_back
	emplace_back		emplace_back	emplace_back		emplace_back
	pop_back		pop_back	pop_back		pop_back
	swap	swap	swap	swap	swap	swap
	merge				merge	merge

SET

```
template <class Key, class Compare=std::les<Key>,
          class Allocator = std:allocator<Key> >
class set;
```

1. Kontener asocjacyjny o dynamicznej ilości elementów
2. Zawiera posortowane, unikatowe obiekty typy Key
3. Wyszukiwanie, wstawianie i usuwanie mają złożoność logarytmiczną
4. Zwykle realizowane na bazie drzew czerwono-czarnych
5. Obsługiwany przez *Bidirectional Iterator*

SET

```
template <class Key, class Compare=std::les<Key>,
          class Allocator = std:allocator<Key> >
class set;
```

1. Zestaw metod/funkcji:

- `iterator begin()/end()` – bidirectional access iterator

- `int size()`
- `bool empty()`

- `void clear()` – usuwa wszystkie elementy
- `iterator insert(const T&)` – wstawia obiekt
- `void erase(iterator pos)` – usuwa obiekt na pozycji
- `int erase(const Key&)` – usuwa obiekt
- `void merge(set)` – łączy dwa zbiory
- `pair<K,bool> emplace(Args...)` – konstruuje i wstawia obiekt

SET

```
template <class Key, class Compare=std::les<Key>,
          class Allocator = std:allocator<Key> >
class set;
```

1. Zestaw algorytmów:
   - `int count(const Key&)` – ilość podanych obiektów (0 lub 1)
   - `iterator find(const Key&)` – iterator na podany obiekt lub na end()
   - `iterator lower_bound(const Key&)` – iterator na pierwszy podany obiekt
   - `iterator upper_bound(const Key&)` – iterator na podany obiekt lub na end()
   - `pair<iterator, iterator> equal_range(const Key&)` – para iteratorów – pierwszy ustawiony na podany obiekt, drugi na obiekt następny za podanym

MAP

```
template <class Key, class T,
          class Compare=std::les<Key>,
          class Allocator =
          std:allocator<std::pair<const Key, T>>>
class map;
```

1. Kontener asocjacyjny o dynamicznej ilości elementów
2. Zawiera posortowane pary klucz-wartość o unikatowych kluczach typu Key
3. Wyszukiwanie, wstawianie i usuwanie mają złożoność logarytmiczną
4. Zwykle realizowane na bazie drzew czerwono-czarnych lub tablic z haszowaniem
5. Obsługiwany przez *Bidirectional Iterator*

MAP

```
template <class Key, class T, class Compare=std::les<Key>,
          class Allocator = std:allocator<std::pair<const Key, T>>>
class map;
```

1. Zestaw metod/funkcji:
   - `iterator begin()/end()` – bidirectional access iterator
   - `int size()`
   - `bool empty()`
   - `T& at(const Key&)` – zwraca referencję na wartość pod podanym kluczem, jeżeli istnieje lub rzuca `std::out_of_range`
   - `T& operator[](const Key&)` – zwraca referencję na wartość pod podanym kluczem, jeżeli istnieje w przeciwnym razie wstawia do mapy podany klucz
   - `void clear()` – usuwa wszystkie elementy
   - `iterator insert(const T&)` – wstawia obiekt
   - `void erase(iterator pos)` – usuwa obiekt na pozycji
   - `int erase(const Key&)` – usuwa obiekt
   - `void merge(set)` – łączy dwa zbiory
   - `pair<K,bool> emplace(Args...)` – konstruuje i wstawia obiekt

MAP

```
template <class Key, class T, class Compare=std::les<Key>,
          class Allocator = std:allocator<std::pair<const Key, T>>>
class map;
```

1. Zestaw algorytmów:
   - `int count(const Key&)` – ilość podanych obiektów (0 lub 1)
   - `iterator find(const Key&)` – iterator na podany obiekt lub na end()
   - `iterator lower_bound(const Key&)` – iterator na pierwszy podany obiekt
   - `iterator upper_bound(const Key&)` – iterator na podany obiekt lub na end()
   - `pair<iterator, iterator> equal_range(const Key&)` – para iteratorów – pierwszy ustawiony na podany obiekt, drugi na obiekt następny za podanym

		Associative containers			
Header		<set>		<map>	
Container		set	multiset	map	multimap
	(constructor)	set	multiset	map	multimap
	(destructor)	~set	~multiset	~map	~multimap
	operator=	operator=	operator=	operator=	operator=
	assign				
Iterators	begin	begin	begin	begin	begin
	cbegin	cbegin	cbegin	cbegin	cbegin
	end	end	end	end	end
	cend	cend	cend	cend	cend
	rbegin	rbegin	rbegin	rbegin	rbegin
	crbegin	crbegin	crbegin	crbegin	crbegin
	rend	rend	rend	rend	rend
	crend	crend	crend	crend	crend
Element access	at			at	
	operator[]			operator[]	
	data				
	front				
	back				
Capacity	empty	empty	empty	empty	empty
	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size
	resize				
	capacity				
	reserve				
	shrink_to_fit				
Modifiers	clear	clear	clear	clear	clear
	insert	insert	insert	insert	insert
	insert_or_assign			insert_or_assign	
	emplace	emplace	emplace	emplace	emplace
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
	try_emplace			try_emplace	
	erase	erase	erase	erase	erase
	push_front				
	emplace_front				
	pop_front				
	push_back				
	emplace_back				
	pop_back				
	swap	swap	swap	swap	swap
	merge	merge	merge	merge	merge
	extract	extract	extract	extract	extract