

Niestandardowe kontenery w C++ z naciskiem na kontenery z biblioteki Boost

dr Grzegorz Bazior

16 grudnia 2024



Prezentacja zrobiona na bazie artykułu:
[cpp0x.pl/artykuly/Inne-artykuly/Kontenery-niestandardowe-w-C+
+-z-naciskiem-na-biblioteke-boost/100](http://cpp0x.pl/artykuly/Inne-artykuly/Kontenery-niestandardowe-w-C+-z-naciskiem-na-biblioteke-boost/100)

Plan prezentacji

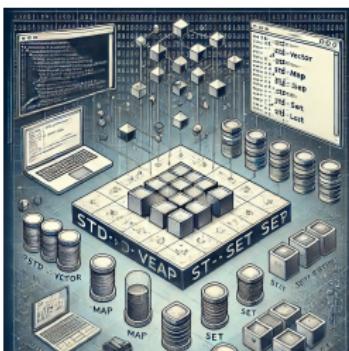
- Wprowadzenie do kontenerów (co to jest, wymagania)
- Kontenery w kolejnych standardach
- Problemy z kontenerami
- Problem wyboru kontenera
- Ciekawostki o kontenerach standardowych
- Kontenery niestandardowe w ramach biblioteki boost
- Inne kontenery m.in. **plf::colony**

Czyli gdzie po świecie C++ się poruszamy?



Rysunek: Źródło fearlesscoder.blogspot.com, autor Elena Sagalaeva, dostęp 9 grudnia 2024, licencja na umieszczenie na slajdzie: "Yes, you can!"

Biblioteka Kontenerów



Definicja kontenera (standard)

Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations

+ kontener musi spełniać wymagania konceptu Container
(w praktyce 7 stron w standardzie)

Wymagania względem konceptu wg StackOverflow

```
1 template <class ContainerType>
2 concept Container = requires(ContainerType a, const ContainerType b) {
3     requires std::regular<ContainerType>;
4     requires std::swappable<ContainerType>;
5     requires std::destructible<typename ContainerType::value_type>;
6     requires std::same_as<typename ContainerType::reference, typename
7         ContainerType::value_type &>;
8     requires std::same_as<typename ContainerType::const_reference, const
9         typename ContainerType::value_type &>;
10    requires std::forward_iterator<typename ContainerType::iterator>;
11    requires std::forward_iterator<typename ContainerType::
12        const_iterator>;
13    requires std::signed_integral<typename ContainerType::
14        difference_type>;
15    requires std::same_as<typename ContainerType::difference_type,
16        typename std::iterator_traits<typename ContainerType::iterator>::
17        difference_type>;
18    requires std::same_as<typename ContainerType::difference_type,
19        typename std::iterator_traits<typename ContainerType::
20            const_iterator>::difference_type>;
21    { a.begin() } -> std::same_as<typename ContainerType::iterator>;
22    { a.end() } -> std::same_as<typename ContainerType::iterator>;
23    { b.begin() } -> std::same_as<typename ContainerType::const_iterator
24        >;
25    { b.end() } -> std::same_as<typename ContainerType::const_iterator>;
26    { a.cbegin() } -> std::same_as<typename ContainerType::
27        const_iterator>;
28    { a.cend() } -> std::same_as<typename ContainerType::const_iterator
29        >;
30    { a.size() } -> std::same_as<typename ContainerType::size_type>;
31    { a.max_size() } -> std::same_as<typename ContainerType::size_type>;
32    { a.empty() } -> std::same_as<bool>; }
```



Kontenery standardowe wprowadzane w kolejnych standardach C++

C++	sekwencyjne	asocjacyjne	adapty	widoki
03	<code>vector</code> _{RA} <code>list</code> _B <code>deque</code> _{RA}	<code>set</code> _B <code>map</code> _B <code>multiset</code> _B <code>multimap</code> _B	<code>stack</code> _{brak} <code>queue</code> _{brak} <code>priority_queue</code> <i>brak</i>	-
11	<code>array</code> _{RA} <code>forward_list</code> _F	<code>unordered_set</code> _F <code>unordered_multiset</code> _F <code>unordered_map</code> _F <code>unordered_multimap</code> _F	-	-
20	-	-	-	<code>span</code> _{RA}
23	-	-	<code>flat_map</code> _{RA} <code>flat_multimap</code> _{RA} <code>flat_set</code> _{RA} <code>flat_multiset</code> _{RA}	<code>mdspan</code> _{RA}

Nie w pełni kontenery

- `std::string`_{RA} (`basic_string<...>`)
 - `std::string_view`_{RA} (`basic_string_view<...>`)
 - `std::vector<bool>` RA, ale operujący na obiektach proxy
 - `ranges/*/+`
 - `std::valarray` brak `begin()`, jednakże `std::begin(valarray<T>)`
 - `std::bitset` *brak*
 - `std::pair`, `std::tuple` *brak*

Kontenery standardowe wciąż są modernizowane



Rysunek: Zestawienie metod kontenerów standardowych.

Kolory: C++03, C++11, C++17, C++20, C++23.

Źródło screena: cppreference.com. Licencja: CC-BY-SA

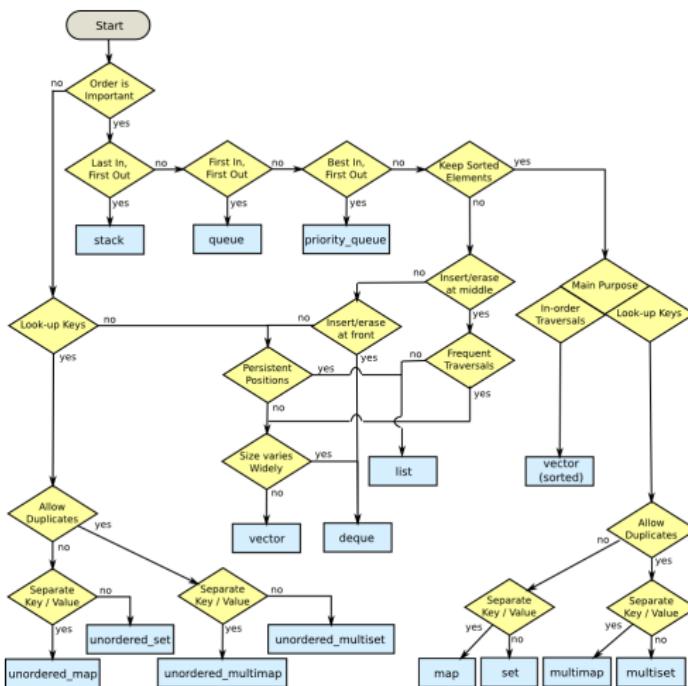
Ciekawostki o kontenerach standardowych



Rysunek: Alexander Stepanov, źródło [Wikibooks](#), autor: Paul R. McJones, CC BY-SA 3.0

- Zaczęło się w 1993, gdy [Alexander Stepanov](#) zaprezentował swoją implementację przed [ANSI/ISO](#).
- Następnie implementację przejęła firma [SGI \(Silicon Graphics International\)](#) na licencji [MIT](#).
- Implementacja zawierała również kontenery, które nie weszły do standardu, m.in. [rope](#).
- **SGI STL** była pierwszą szeroko stosowaną implementacją STL i wpłynęła na wiele bibliotek C++ w przyszłości.
- W ramach STL zdefiniowano także mechanizm iteracji, który do dziś jest podstawą wielu operacji na kontenerach w C++.

Otwarty problem - algorytm wyboru kontenera



Rysunek: Algorytm doboru kontenerów wg odpowiedzi użytkownika Mikael Persson na Stack Overflow (podejście naiwne).

Licencja: CC BY-SA 4.0. Czas dostępu: 9 grudnia 2024.

Szybkie jego rozwiązanie

- Bjarne Stroustrup w **The C++ Programming Language*** (4th Edition):
"In general, a 'vector' should be your default choice of container. It is flexible, dynamic, and provides fast access to its elements. Other containers are useful in more specialized cases, but 'vector' is the most common."

- Scott Meyers w **Effective STL**:
"When you need a container, your default choice should be a 'vector'. That's because 'vector' offers the best balance of performance and functionality for most applications. It efficiently supports fast random access, is usually compact in memory, and is generally faster than other sequence containers."

Kontenery biblioteki Boost 1.8.6 - przegląd



- 24 podprojekty kategorii "containers" (na 176 w sumie)
- Wiele z nich już weszło do standardu, m.in.: `boost::array`,
`boost::string_view`, `boost::unordered_*`
- Zgodność interfejsu z kontenerami standardu C++
- Oprócz nowych kontenerów, niedostępnych (jeszcze) w standardzie, Boost oferuje alternatywy dla standardowych kontenerów, np. `boost::container::vector`,
`boost::container::list`.

Kontenery w ramach Boost konkurencyjne dla standardu

```
#include <boost/container/*.hpp>  
  
using namespace boost::container;
```

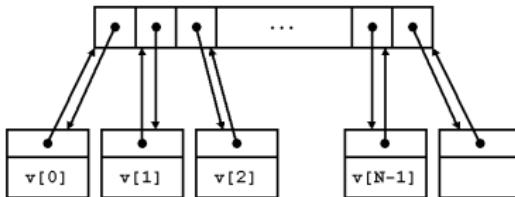
Kontener	Metody	Zachowanie
vector	nth index_of stable_emplace_back	- konfigurowalny typ capacity - konfigurowalne rozszerzanie
deque	nth index_of	konfigurowalny rozmiar pamięci ciągłej
list		operacja slice w czasie O(1)
set multiset map multimap		możliwość ustawienia różnych algorytmów: AVL, SPLAY, SCAPEGOAT, i TREAP
flat_*		lepsze zarządzanie pamięcią szybsze przeszukiwanie ¹

¹Więcej informacji na temat kontenerów flat_* można znaleźć w dokumentacji Boost: [link](#), gdzie zasugerowano się wytycznymi Alexandrescu.

Kontenery biblioteki Boost niedostępne jeszcze w standardzie (boost::container)

- **static_vector** - wektor o statycznie zaalokowanej pamięci, po której przepełnieniu nie jest możliwa ponowna alokacja.
- **slist** - lista jednokierunkowa (jak std::forward_list z C++11), ale zawiera metodę `size()`, metodę tę wprowadzono dopiero w standardzie C++23.
- **small_vector** - wektor, który zawiera wstępnie zaalokowaną statyczną pamięć, a w razie potrzeby dynamicznie alokuje dodatkową. Dane są zawsze w pamięci ciągłej (albo na stosie, albo na stercie, nigdy w dwóch miejscach).
- **stable_vector** - łączy funkcjonalności std::vector i std::list, zapewniając stabilność iteratorów (nawet po realokacji pamięci) oraz dostęp swobodny. ↓
- **devector** - hybryda std::vector i std::deque, oferująca wstawianie elementów w czasie stałym zarówno z przodu, jak i na końcu kontenera. ↓

boost::stable_vector - stabilny kontener hybrydowy łączący std::vector i std::list

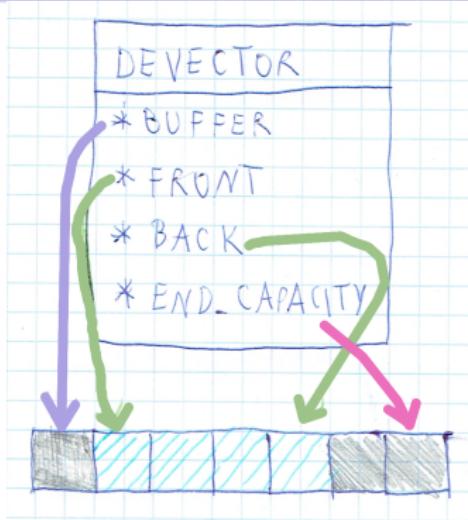


Rysunek: Źródło [dokumentacja boost::stable_vector](#), licencja: Boost Software License 1.0.

- dostarcza większość cech vectora poza ciągłą pamięcią: RA, zamortyzowany czas dodania/usuwania na końcu
- elementy trzymane w oddzielnych węzłach, z wskaźnikami na pamięć ciągłą

Operacja	std::vector	stable_vector
insert	silne, chyba że konstrukcja/przenoszenie T zgłasza wyjątek (podstawowe)	silne
erase	no-throw, chyba że konstrukcja/przenoszenie T zgłasza wyjątek (podstawowe)	no-throw

boost::container::devector - hybryda std::vector i std::deque



- Łączy std::vector (dostęp swobodny, pamięć ciągła) i std::deque (efektywne wstawianie na początku i końcu).
- Zapewnia "amortized constant time" na dodawanie/usuwanie z przodu i tyłu kontenera.
- W odróżnieniu od std::vector umożliwia dynamiczną alokację miejsca również przed początkiem wektora.
- Rozszerzanie w tę stronę, z której miejsca zabrakło, ale unika "nieograniczonego marnowania pamięci"
- Zawiera dodatkowy wskaźnik w stosunku do std::vector, zajmuje około $4 * \text{sizeof}(T^*)$

Rysunek: Rysunek własny
zainspirowany rysunkiem z [Blogu PVS-Studio](#). Szare komórki obrazują zaalokowaną aczkolwiek nieużywaną pamięć, niebieskie używaną pamięć.

Pozostałe kontenery biblioteki Boost (boost::*)

- **boost::bimap** - podobny do std::map, umożliwiający dwukierunkowe mapowanie klucz-wartość.
- **boost::circular_buffer** - bufor cykliczny, dane w pamięci o stałym rozmiarze. Nowe dane zastępują najstarsze, utrzymując stały rozmiar bufora, RA, stały czas dodawania/usuwania na końcach.
 - circular_buffer - z możliwością zmiany capacity na żądanie (nie automatycznie),
 - circular_buffer_space_optimized - alokuje pamięć tylko wtedy, gdy jest to konieczne, ale nie więcej niż capacity.
- **boost::dynamic_bitset** - w przeciwnieństwie do std::bitset z dynamicznie zmienialnym rozmiarem (a nie statycznym).
- **boost::multi_array** - wielowymiarowa tablica w ciągłym obszarze pamięci: multi_array, multi_array_ref i const_multi_array_ref.
- **boost::multi_index** - kontener umożliwiający dostęp do elementów według wielu indeksów (niekoniecznie unikalnych), inspirowany bazami danych, gdzie możemy odwoływać się do wartości przez różne indeksy.↓

boost::multi_index - wieloindeksowe kontenery

- Indeks drzewiasty (`ordered_index`) - podobny do `std::set`, zapewnia szybkie wyszukiwanie i dostęp.
- Indeks listowy (`sequenced_index`) - podobny do `std::list`, zachowuje kolejność wstawiania.
- Indeks haszujący (`hashed_index`) - podobny do `std::unordered_set`, szybkie wyszukiwanie przez funkcję haszującą.

Zalety:

- Umożliwia jednoczesne korzystanie z wielu indeksów na jednym zbiorze danych, co pozwala na efektywne zarządzanie złożonymi strukturami danych.
- Obsługuje dodatkowe funkcje: wyszukiwanie obiektów po podobiektach, zapytania o zakres oraz aktualizację elementów na miejscu.

• Zastosowania:

- Gdy wymagane są różne sposoby dostępu do danych w jednym kontenerze, np. wyszukiwanie według klucza oraz według wartości.
- Zastępuje `std::set` lub `std::multiset`, oferując dodatkowe możliwości indeksowania.

boost::multi_index - wieloindeksowe kontenery cz2

```
1 struct Employee {
2     int id;
3     std::string name;
4     int age;
5
6     Employee(int id_, std::string name_, int age_)
7         : id(id_), name(name_), age(age_)
8     {}
9
10    friend std::ostream& operator<<(std::ostream& os, const Employee& e)
11    {
12        return os << e.id << " " << e.name << " " << e.age << std::endl;
13    }
14};
15
16 namespace EmployeeTags {
17     struct Id {};
18     struct Name {};
19     struct Age {};
20 };
21
22 using employee_set = boost::multi_index_container<
23     Employee,
24     indexed_by<
25         ordered_unique<tag<EmployeeTags::Id>, BOOST_MULTI_INDEX_MEMBER(
26             Employee, int, id)>,
27         ordered_non_unique<tag<EmployeeTags::Name>,
28             BOOST_MULTI_INDEX_MEMBER(Employee, std::string, name)>,
29         ordered_non_unique<tag<EmployeeTags::Age>,
30             BOOST_MULTI_INDEX_MEMBER(Employee, int, age)>>>;
```

boost::multi_index - wieloindeksowe kontenery cz3

```
1 template<typename Tag, typename MultiIndexContainer>
2 void print_out_by(const MultiIndexContainer& s) {
3     const auto& index = get<Tag>(s);
4
5     using value_type = MultiIndexContainer::value_type;
6     /* dump the elements of the index to cout */
7     std::copy(index.begin(), index.end(), std::ostream_iterator<
8         value_type>(std::cout));
9 }
10 int main() {
11     employee_set es;
12
13     es.insert(Employee(0, "Joe", 31));
14     es.insert(Employee(1, "Robert", 27));
15     es.insert(Employee(2, "John", 40));
16     es.insert(Employee(2, "Aristotle", 2387)); // fails
17     es.insert(Employee(3, "Albert", 20));
18     es.insert(Employee(4, "John", 57));
19
20     std::cout << "employees sorted by ID" << std::endl;
21     print_out_by<EmployeeTags::Id>(es);
22     std::cout << std::endl;
23
24     std::cout << "employees sorted by name" << std::endl;
25     print_out_by<EmployeeTags::Name>(es);
26     std::cout << std::endl;
27
28     std::cout << "employees sorted by age" << std::endl;
29     print_out_by<EmployeeTags::Age>(es);
30     std::cout << std::endl;
31 }
```

```
employees sorted by ID
0 Joe 31
1 Robert 27
2 John 40
3 Albert 20
4 John 57

employees sorted by name
3 Albert 20
0 Joe 31
2 John 40
4 John 57
1 Robert 27

employees sorted by age
3 Albert 20
1 Robert 27
0 Joe 31
2 John 40
4 John 57
```

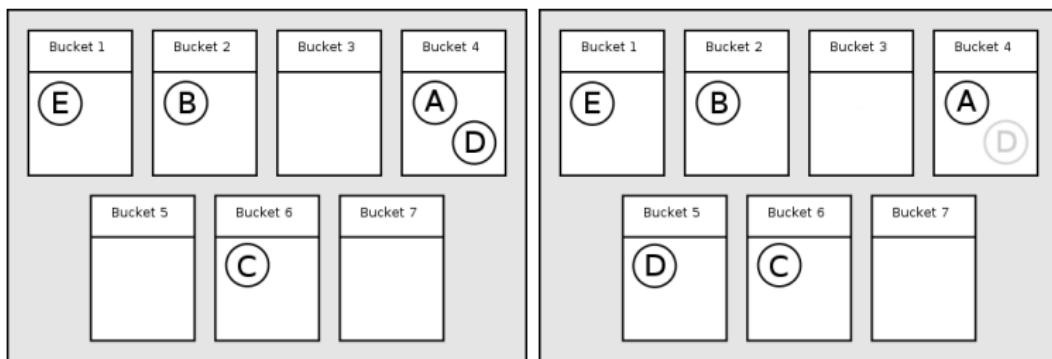


Kontenery hashujące Boost.unordered

```
#include <boost/unordered/* >  
using namespace boost::unordered;
```

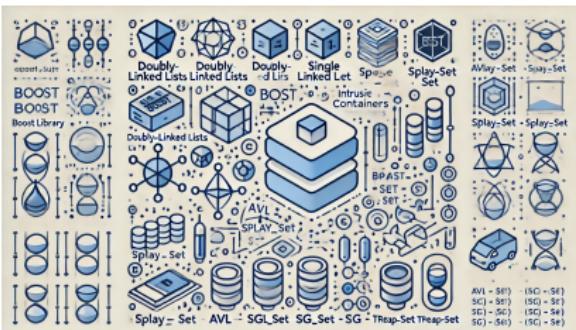
	Bazujące na węzłach	Płaskie (ang. flat)
Adresowanie zamknięte	unordered_set unordered_map unordered_multiset unordered_multimap	
Adresowanie otwarte	unordered_node_set unordered_node_map	unordered_flat_set unordered_flat_map
Współbieżne		concurrent_flat_set concurrent_flat_map

Kontenery hashujące Boost.unordered - porównanie algorytmów



Rysunek: Adresowanie zamknięte (po lewej) vs adresowanie otwarte (po prawej). Źródło: [Dokumentacja boost.unordered](#), licencja: Boost Software License 1.0.

Kontenery intruzyjne biblioteki Boost: dostępne kontenery



- boost::intrusive::list - lista dwukierunkowa
 - boost::intrusive::slist - lista jednokierunkowa
 - boost::intrusive::set, multiset
 - boost::intrusive::unordered_set, unordered_multiset
 - boost::intrusive::avl_set, avl_multiset
 - boost::intrusive::splay_set, splay_multiset
 - boost::intrusive::sg_set, sg_multiset
 - boost::intrusive::treap_set, treap_multiset

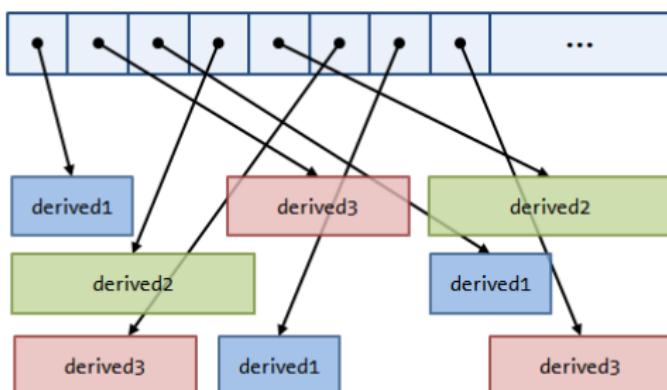
Kontenery intruzyjne biblioteki Boost

```
1 struct MyClass : public list_base_hook<> {
2     // boost::intrusive::list_member_hook<>
3     member_hook_;
4     int v;
5     MyClass(int v) : v{v} {}
6 };
7 int main() {
8     std::vector<MyClass> values = {3, 5, 8};
9
10    boost::intrusive::list<MyClass> l;
11    for (auto& v : values) l.push_back(v); // 
12    referencje
13    for (auto it=l.begin(); it!=l.end(); ++it)
14        std::cout << it->v << std::endl;
15 }
```

Kontenery wskaźników: Boost.ptr_container

Kontenery przechowujące elementy przez wskaźnik zamiast wartości. Gdy kontener jest usuwany, zwalnia również pamięć swoich elementów (działanie jak kontener intelligentnych wskaźników, chociaż wygodniej)

- ptr_vector
- ptr_list
- ptr_deque
- ptr_map
- ptr_multimap
- ptr_set
- ptr_multiset
- ptr_array
- ptr_set_adapter
- ptr_multiset_adapter



Rysunek: Źródło: [dokumentacja boost](#), licencja: Boost Software License 1.0.

Kontenery grupujące typy: boost.poly_collection

Kontenery grupując elementy zgodnie z ich rzeczywistym typem.
Szybsze przetwarzanie i optymalizacja pamięci

- **base_collection**

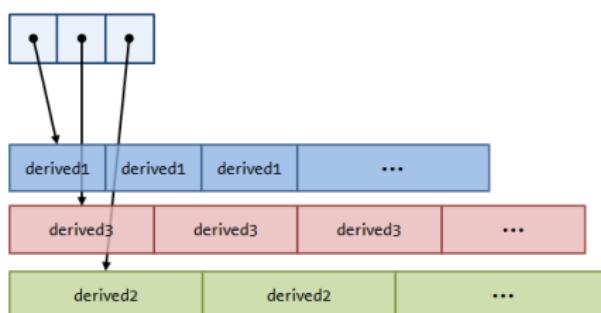
wspólna hierarchia
dziedziczenia, sortowane
automatycznie wg typów,
metoda `insert`.

- **function_collection**

Przechowuje funkcje o
różnych sygnaturach (nie
wymagają wspólnej
hierarchii).

- **any_collection**

Przechowuje elementy o
wspólnych cechach bez
wymogu hierarchii.



Rysunek: Porównanie zarządzania pamięcią
używając boost.poly_collection (źródło:
podstrona Boost), licencja: [Boost Software
License 1.0](#).

Przykład boost.base_collection część 1

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4 #include <utility>
5 #include <boost/poly_collection/base_collection.hpp>
6
7 struct sprite {
8     sprite(int id) : id{id} {}
9     virtual ~sprite() = default;
10    virtual void render(std::ostream& os) const = 0;
11
12    int id;
13};
14 struct warrior : sprite {
15     using sprite::sprite;
16     warrior(std::string rank, int id) : sprite{id}, rank{std::move(rank)}
17     {}
18     void render(std::ostream& os) const override { os << rank << " " <<
19         id; }
20
21     std::string rank = "warrior";
22 };
23 struct juggernaut : warrior {
24     juggernaut(int id) : warrior{"juggernaut", id} {}
25 };
26 struct goblin : sprite {
27     using sprite::sprite;
28     void render(std::ostream& os) const override { os << "goblin " << id
29     ; }
30 };
```

Przykład boost.base_collection część 2

```
1 int main() {
2     boost::base_collection<sprite> c;
3     c.insert(warrior{1});
4     c.insert(juggernaut{2});
5     c.insert(goblin{3});
6     c.insert(warrior{4});
7     c.insert(juggernaut{5});
8     c.insert(goblin{6});
9
10    for (const sprite& s : c) {
11        s.render(std::cout);
12        std::cout << "\n";
13    }
14
15    auto it = std::find_if(c.begin(),c.end(),[](const sprite& s){return
16        s.id==3;});
17    c.erase(it);
```

```
juggernaut 2
juggernaut 5
goblin 3
goblin 6
warrior 1
warrior 4
```

Przykład boost.function_collection

```
1 int main() {
2     std::vector<std::unique_ptr<sprite>> sprs;
3     sprs.push_back(std::make_unique<warrior>(1));
4     sprs.push_back(std::make_unique<juggernaut>(2));
5     sprs.push_back(std::make_unique<warrior>(4));
6
7     std::vector<std::string> msgs = {"\"stamina: 10,000\"", "\"game over
8
9     using render_callback = void(std::ostream&);
10    boost::function_collection<render_callback> c;
11    auto render_sprite = [](&const sprite& s) { return [&](std::ostream&
12        os) { s.render(os); }; };
13    auto render_message = [](&const std::string& m) { return [&](std::
14        ostream& os) { os << m; }; };
15
16    for(const auto& ps : sprs)
17        c.insert(render_sprite(*ps));
18    for(const auto& m : msgs)
19        c.insert(render_message(m));
20
21    for(const auto& cbk : c) {
22        cbk(std::cout);
23        std::cout << "\n";
24    }
25 }
```

```
"stamina: 10,000"
"game over"
warrior 1
juggernaut 2
warrior 4
```

Przykład boost.any_collection

```
1 #include <boost/poly_collection/any_collection.hpp>
2 #include <boost/type_erasure/operators.hpp> // ...
3 struct window {
4     window(std::string caption): caption{std::move(caption)} {}
5     void display(std::ostream& os) const {os<<"["<<caption<<"]";}
6     std::string caption;
7 };
8 std::ostream& operator<<(std::ostream& os, const sprite& s) {
9     s.render(os);
10    return os;
11 }
12 std::ostream& operator<<(std::ostream& os, const window& w) {
13     w.display(os);
14    return os;
15 }
16
17 int main() {
18     using boost::type_erasure::ostreamable;
19     using renderable=ostreamable<>;
20     boost::any_collection<renderable> c;
21
22     c.insert(warrior{1});
23     c.insert(juggernaut{2});
24     c.insert(warrior{4});
25     c.insert(goblin{6});
26     c.insert(std::string{"\nstamina: 10,000\n"});
27     c.insert(std::string{"\ngame over\n"});
28     c.insert(window{"pop-up 1"});
29     c.insert(window{"pop-up 2"});
30
31     for(const auto& r:c)
32         std::cout << r << '\n';
33 }
```

[pop-up 1]
[pop-up 2]
"stamina: 10,000"
"game over"
juggernaut 2
goblin 6
warrior 1
warrior 4



Kontenery zawierające tuple compile-time: boost::fusion

- **vector** - wektor elementów
- **cons** - lista jednokierunkowa
- **list** - lista
- **deque** - deque
- **front_extended_deque** - deque rozszerzany tylko z przodu
- **back_extended_deque** - deque rozszerzany tylko z tyłu
- **set** - zbiór
- **map** - mapa

```
1 #include \
2 <boost/fusion/container/vector.hpp>
3 #include \
4 <boost/fusion/container/list.hpp>
5 // ...
6 vector<int, float> v(12, 5.5f);
7 cout << at_c<0>(v) << endl;
8 cout << at_c<1>(v) << endl;
9
10 list<int, float, string> l(12,
11     5.5f, "text");
12 cout << at_c<0>(l) << endl;
13 cout << at_c<1>(l) << endl;
14 cout << at_c<2>(l) << endl;
```

boost::fusion::vector - tupla w czasie kompilacji

```
1 vector<int , string , double> v(10, "Fusion", 3.14);
2 cout << "element0: " << at_c<0>(v) << endl;
3 at_c<0>(v) = 44;
4 cout << "element_front: " << front(v) << endl;
5 cout << "rozmiar:" << size(v) << endl << endl;
6
7 auto it_double = find<double>(v);
8 if (it_double != end(v))
9     cout << "element_double: " << *it_double << endl;
10
11 cout << "Przed usunięciem:" << endl;
12 for_each(v, [](const auto& element)
13 { cout << element << endl; });
14
15 cout << "\nPo usunięciu elementu std::string:" << endl;
16 auto it_begin = begin(v);
17 auto it_to_erase = advance_c<1>(it_begin);
18 auto v2 = erase(v, it_to_erase);
19 for_each(v2, [](const auto& element)
20 { cout << element << endl; });
21
22 cout << "\nPo dodaniu elementu std::string:" << endl;
```

	element0: 10
	element_front: 44
	rozmiar:3
	element double: 3.14
	Przed usunięciem:
	44
	Fusion
	3.14
	Po usunięciu elementu std::string:
	44
	3.14
	Po dodaniu elementu std::string:
	44
	3.14
	text
	boost::fusion::map:
	at_key: 10
	false

boost::fusion::map - mapa w czasie kompilacji

```
1 void printMap( const auto& map) {
2     for_each(map, []<typename T>(const T& p) {
3         using KeyType = typename T::first_type;
4         using ValueType = typename T::second_type;
5         cout << demangle(typeid(KeyType).name()) << " ->" << p.second
6             << " (" << demangle(typeid(ValueType).name()) << ")");
7     });
8 }
9 int main() {
10    map<pair<int, char>, pair<double, std::string>> m(make_pair<int>('X'),
11        make_pair<double>("C++"));
12    printMap(m); // int->X (char), double->C++ (basic_string...)
13
14    auto m1 = make_map<int, std::string, bool, double>("Boost", 'X', 3.14, 444);
15    printMap(m1); // int->boost (char[6]), string->X (char)
16    // bool->3.14 (double), double->444(int)
17    if (has_key<std::string>(m1))
18        cout << "at key<std::string>: " << at_key<std::string>(m1) << '\n'; // X
19    cout << "klucz bool?: " << std::boolalpha << has_key<bool>(m1) << '\n'; // T
20    cout << "klucz float?: " << has_key<float>(m1) << '\n'; // F
21
22    auto m2 = erase_key<std::string>(m1);
23    cout << "Po usunięciu klucza std::string: " << has_key<std::string>(m2) <<
24        '\n'; // F
25
26    auto m3 = push_back(m2, make_pair<float>('X'));
27    auto m4 = push_back(m3, make_pair<char>('Z'));
28    cout << "Czy klucz float istnieje w m3? " << has_key<float>(m3) << '\n'; // F
29    cout << "Czy klucz char istnieje? " << has_key<char>(m4) << '\n'; // F
30
31    cout << "\nElementy mapy m4:" << endl;
32    printMap(m4); // int->boost, bool->3.14, double->444, float->X, char->X
33 }
```

Kontenery wielowątkowe: boost::lockfree

Boost.Lockfree oferuje implementacje adapterów kontenerów, podobnych do tych dostępnych w standardzie C++, lecz przeznaczonych do zastosowań wielowątkowych, bez użycia blokad.

- **queue** - kolejka bez blokad obsługująca wielu producentów i konsumentów
- **stack** - stos bez blokad obsługujący wielu producentów i konsumentów
- **spsc_queue** - kolejka "wait-free" obsługująca jednego producenta i jednego konsumenta (ang. "ringbuffer")



Struktury danych typu "wait-free"

Struktury danych są "wait-free" (wolne od oczekiwania), jeśli każda równoczesna operacja jest zapewniona, że zakończy się w skończonej liczbie kroków. To pozwala zagwarantować liczbę operacji w najgorszym przypadku.

Kontenery trzymające zakresy czasu boost::icl

Kontenery Boost.Icl odpowiadają std::set i std::map, ale trzymają wartości jako zakresy (np. daty)

```
1 // Switch on my favorite telecasts using an
   interval_set
2 interval<seconds>::type news(make_seconds("20:00:00"), make_seconds("20:15:00"));
3 interval<seconds>::type talk_show(make_seconds("22:45:30"), make_seconds("23:30:50"));
4 interval_set<seconds> myTvProgram;
5 myTvProgram.add(news).add(talk_show);
6
7 // Iterating over elements (seconds) would be
   silly ...
8 for (interval_set<seconds>::iterator telecast =
9     myTvProgram.begin();
10    telecast != myTvProgram.end(); ++telecast)
11    // ...so this iterates over intervals
12    TV.switch_on(*telecast);
```



boost::fusion::icp - przykład

```
1 ptime make_seconds(const std::string& time_str)
2 {
3     return time_from_string("2024-01-01 " + time_str);
4 }
5
6 auto timeRange(const string& secondsFrom, const string& secondsTo)
7 {
8     return interval<ptime>::right_open(make_seconds(secondsFrom), make_seconds(
9         secondsTo));
10 }
11
12 int main()
13 {
14     using guests = std::set<std::string>;
15     interval_map<ptime, guests> party;
16
17     party += make_pair(timeRange("20:00:00", "22:00:00"), guests{"Mary"});
18     party += make_pair(timeRange("21:00:00", "23:00:00"), guests{"Harry"});
19     party += make_pair(timeRange("22:30:00", "23:30:00"), guests{"Tom"});
20
21     cout << "Mapa przedzialow czasowych:\n";
22     for (const auto& entry : party) {
23         cout << "[" << entry.first.lower() << ", " << entry.first.upper() << "]>=";
24         for (const auto& guest : entry.second)
25             cout << guest << " ";
26         cout << "]\n";
27     }
28 }
```

Mapa przedziałów czasowych:
[2024-Jan-01 20:00:00, 2024-Jan-01 21:00:00)->{Mary }
[2024-Jan-01 21:00:00, 2024-Jan-01 22:00:00)->{Harry Mary }
[2024-Jan-01 22:00:00, 2024-Jan-01 22:30:00)->{Harry }
[2024-Jan-01 22:30:00, 2024-Jan-01 23:00:00)->{Harry Tom }
[2024-Jan-01 23:00:00, 2024-Jan-01 23:30:00)->{Tom }

boost::fusion::icp - przykład część 2

```
1 ptime check_time = make_seconds("21:30:00");
2 auto it = party.find(check_time);
3 if (it != party.end()) {
4     cout << "\nOsoby obecne o " << check_time << ":" ;
5     for (const auto& guest : it->second)
6         cout << guest << " ";
7     cout << "\n";
8 }
9 else
10    cout << "\nO " << check_time << " nikogo nie ma.\n";
11
12 interval<ptime>::type check_interval = interval<ptime>::right_open(
13     make_seconds("21:00:00"), make_seconds("22:30:00"));
14 cout << "\nSprawdzanie obecności w przedziale [" << check_interval.lower()
15 << ", " << check_interval.upper() << "):\n";
16 for (auto iter = party.begin(); iter != party.end(); ++iter) {
17     if (intersects(iter->first, check_interval)) {
18         cout << "W przedziale [" << iter->first.lower() << ", " << iter->
19         first.upper() << ") są: ";
20         for (const auto& guest : iter->second)
21             cout << guest << " ";
22         cout << "\n";
23     }
24 }
```

Osoby obecne o 2024-Jan-01 21:30:00: Harry Mary

Sprawdzanie obecności w przedziale [2024-Jan-01 21:00:00, 2024-Jan-01 22:30:00):
W przedziale [2024-Jan-01 21:00:00, 2024-Jan-01 22:00:00) są: Harry Mary
W przedziale [2024-Jan-01 22:00:00, 2024-Jan-01 22:30:00) są: Harry

Konfigurowalna kolejna priorytetowa boost::heap



- Modyfikacje priorytetów po dodaniu elementów do kolejki.
- Możliwość przeglądania elementów przy pomocy iteratorów.
- Możliwość łączenia wielu kolejek.
- Możliwość ustawienia stabilnego sortowania.
- Porównywanie kolejek.

Oto lista szablonów kontenerów dostarczonych przez Boost.Heap:

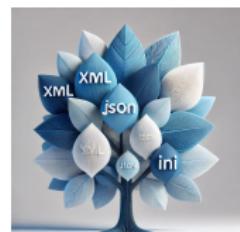
- `boost::heap::priority_queue` - wrapper na funkcje stlowe dotyczące kopca, adapter na `std::vector`, niemodyfikowalny.
- `boost::heap::d_ary_heap` - wrapper na `std::vector`, implementujący kopiec a-arny, dane mogą być modyfikowalne.
- `boost::heap::binomial_heap` - implementacja kopca dwumianowego.
- `boost::heap::fibonacci_heap` - implementacja kopca Fibonacciego.
- `boost::heap::pairing_heap` - implementacja pairing heap.
- `boost::heap::skew_heap` - implementacja skew heap.

Kontenery (?) właściwości boost::property_map i boost::property_tree

boost::property_map (lub boost::property_map_parallel) umożliwia dodawanie właściwości do obiektów (użyteczne w bibliotece grafów BGL, np. nazwy wierzchołków).

boost::property_tree:

- Kontener w postaci drzewa.
- Każdy wierzchołek może mieć dowolną liczbę dzieci.
- Ułatwia generowanie i parsowanie: XML, JSON, INI.
- Iteracja po drzewie jest możliwa, ale nie ma prostego sposobu na jednoczesną iterację po wszystkich elementach przy użyciu pętli ranged-based-for.
- Elementy w drzewie są przechowywane w kolejności wstawienia, a nie według klucza.



Podsumowanie: Boost czy nie boost - oto jest pytanie



- Boost dostarcza szeroki wachlarz kontenerów i algorytmów rozszerzających C++.
- Ich znajomość może przynieść korzyści w specjalistycznych zastosowaniach.
- Warto znać ich ograniczenia i zalety w porównaniu do standardowych kontenerów.

Kontenery w bibliotece Qt

- udostępnia zestaw szablonowych klas kontenerów do ogólnego użytku
- Wg dokumentacji Qt6/Containers: zaprojektowane na lżejsze, bezpieczniejsze i łatwiejsze w użyciu niż kontenery z STL
- Można ich używać zamiast STL, jeśli preferujesz "Qt-owy" sposób pracy.
- Klasy kontenerów są współdzielone w sposób niejawny oraz zoptymalizowane pod kątem:
 - szybkości działania,
 - niskiego zużycia pamięci,
 - minimalnej liczby wstawianego kodu, co prowadzi do mniejszych plików wykonywalnych.
- Są również bezpieczne w środowisku wielowątkowym w sytuacjach, gdy są używane wyłącznie do odczytu przez wszystkie wątki.
- Kontenery Qt dostarczają iteratory do przechodzenia po elementach: w stylu STLa oraz stylu Javy

Porównanie kontenerów Qt z odpowiednikami w STL

typ	kontener	odpowiednik std::	opis
Asocjacyjno-frekwencyjne	QList (= QVector)	vector	Nie jest listą powiązaną.
	QVarLengthArray	array i vector	Łączy cechy array i vector.
	QStack	stack	Dziedziczy po QList.
	QQueue	queue	Dziedziczy po QList.
	QSet	unordered_set	Wewnątrz używa QHash.
	QMap	map	ale zachowuje kolejność
	QMultiMap	multimap	-
	QHash	unordered_map	API podobne do QMap, ale szybsze wyszukiwanie.
	QMultiHash	unordered_multimap	-
Specjalne	QCache	-	Pamięć K-V, szbkie wyszukiwanie, ma ustalony rozmiar cache
	QContiguousCache	-	jw., dane w sposób ciągły.

Wybrane niestandardowe kontenery w stylu STL

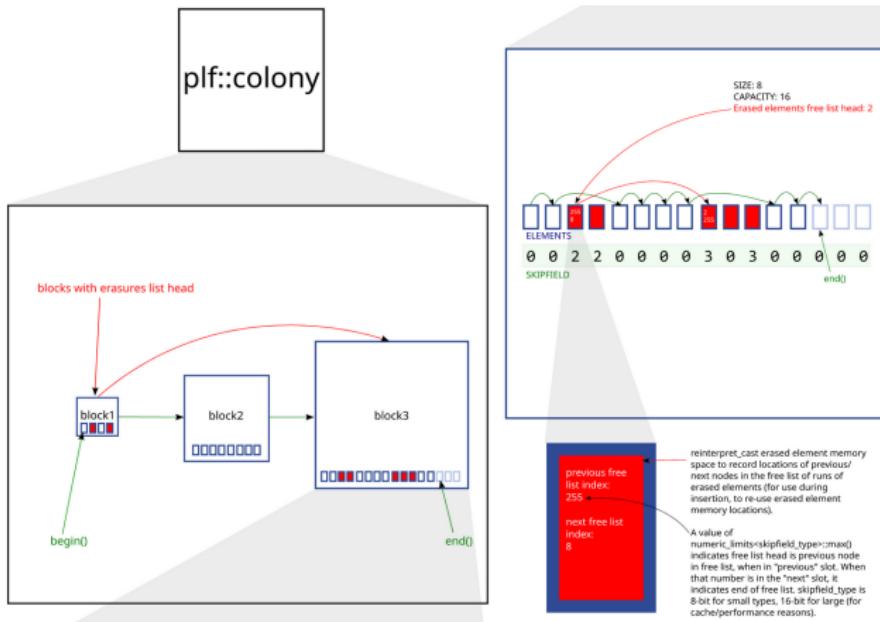
- **Concurrent Data Structures (libcdds)**: Kolekcja współbieżnych kontenerów (wersja intruzywna i nieintruzywna), niewymagających zewnętrznej synchronizacji dla dostępu współdzielonego. Zawiera stosy, kolejki, `unordered_map` i `unordered_set`.
- **EASTL (Electronic Arts Standard Template Library)**: Wydajna alternatywa dla STL, zaprojektowana z myślą o przewidywalności i wydajności w środowisku tworzenia gier. Oferuje kontenery zoptymalizowane pod kątem niskiej latencji i deterministycznego zachowania.
- **Folly (fbvector, fbstring)**: Autorstwa Facebooka, poprawiające wydajność w typowych zastosowaniach.
- **Abseil Containers**: Zbiór kontenerów `*_map` i `*_set`, które wykorzystują struktury bazujące na B-drzewach. Szybsze w specyficznych przypadkach, ale nieco wolniejsze w ogólnych.
- **STXXL (Standard Template Library for Extra Large Data Sets)**: Implementacja STL zoptymalizowana pod kątem przetwarzania ogromnych zbiorów danych, które nie mieścią się w podręcznej pamięci.

plf::colony - propozycja do standardu jako std::hive

plf::colony: Kontener do przechowywania nieuporządkowanych danych, zapewniający szybkie iterowanie, wstawianie i usuwanie. Zaprojektowany tak, by nie unieważniać wskaźników, referencji ani iteratorów do elementów, które nie zostały usunięte.

- Dane przechowywane są w sekwencji bloków pamięci, które rosną w sposób wykładniczy (domyślnie współczynnik wzrostu wynosi 2).
- Używany mechanizm *skipfield*, umożliwiający pomijanie usuniętych elementów, co eliminuje konieczność realokacji przy usuwaniu i wstawianiu.
- Lokacje usuniętych elementów są pamiętane i ponownie używane przy kolejnych wstawieniach, co optymalizuje wykorzystanie pamięci.
- Struktura opiera się na "grupach" elementów (bloki pamięci z metadanymi), połączonych w listę dwukierunkową.
- Brak konieczności realokacji oznacza, że wszystkie wskaźniki, iteratory i referencje pozostają ważne po operacjach wstawiania.
- Skipfield korzysta z tzw. mechanizmu *low complexity jump-counting*, co redukuje złożoność operacji iteracji przez pomijanie zbędnych pozycji.

plf::colony - implementacja



Rysunek: Implementacja `plf::colony`, grafika z [oficjalnej strony](#).
Licencja: zlib License (© 2019 Matt Bentley).

PS: `plf::list` wg twórców jest szybsza o 333% (wstawianie), 81% (usuwanie), 16% (iteracja).

Bibliografia



Rysunek: Książki niekoniecznie użyte bezpośrednio w tej prezentacji.

Widoczne okładki: tytuły i autorzy należą do ich właścicieli.

- Artykuł mojego autorstwa – "Kontenery niestandardowe w C++ z naciskiem na bibliotekę Boost" (w tym bibliografia użyta w artykule)
- Benchmark kontenerów standardowych i `plf::colony` – Baptiste Wicht
- Alternatywny artykuł na temat wydajności kontenerów niestandardowych – PVS-Studio Blog
- Zestaw tutoriali Boost – The Boost C++ Libraries

źródła grafik:

- Niepodpisane własne
- Obrazki generowane za pomocą AI: DALL·E w ramach ChatGPT (wersja bezpłatna).

Kontenery w ramach standardów
oooooooo

Kontenery biblioteki boost
oooooooooooooooooooooooooooo

Inne niestandardowe kontenery
oooooooo●○

Pytania?



Reklamy: auto-reklama && reklama

RuchyKosciola.pl - jak
Bóg da za rok o tej
samej porze.
**Póki co potrzebuję
Twojej pomocy:**

- Dodaj dane swojej wspólnoty
- Dodaj co najmniej jedną lokalizację Twojej wspólnoty (jeśli ma ona więcej niż jedną lokalizacje)



Rysunek: Aplikacja Drogowskaz - oficjalna ulotka
(to nie jest aplikacja mojego autorstwa, ale
polecam!) - **tam też potrzebne są dane**