

## IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

## KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

## TWÓJ KOSZYK

DODAJ DO KOSZYKA

## CENNIK I INFORMACJE

ZAMÓW INFORMACJE  
O NOWOŚCIACH

ZAMÓW CENNIK

## CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

# STL. Leksykon kieszonkowy

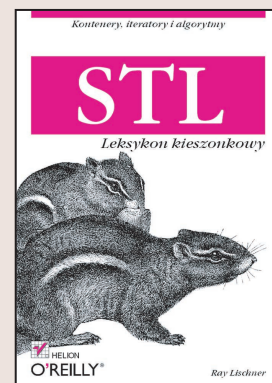
Autor: Ray Lischner

Tłumaczenie: Wojciech Moch

ISBN: 83-7361-438-9

Tytuł oryginału: [STL Pocket Reference](#)

Format: B5, stron: 132



Czy do wyszukania wartości w danym zakresie należy użyć funkcji `search()` czy `find()`? Jakie argumenty ma funkcja `list::splice`? Kiedy wywoływać `mem_fun`, a kiedy `mem_fun_ref`? Zapewne jak wielu innych programistów masz kłopot z zapamiętaniem tych wszystkich szczegółów, nawet jeśli codziennie używasz biblioteki STL. Książka Raya Lischnera „STL. Leksykon kieszonkowy” będzie dla Ciebie nieocenioną pomocą – w prosty sposób odpowiada na wszystkie takie pytania.

W tej książce znajdują się opisy interfejsów kontenerów, iteratorów, algorytmów i obiektów funkcyjnych zawartych w bibliotece STL. Można w niej znaleźć szczegóły dotyczące wywołań funkcji, typów zwracanych przez te funkcje, parametrów szablonów i wiele więcej. W połączeniu z książką „C++. Leksykon kieszonkowy”, książka ta pozwala na spore oszczędności czasu. Na pewno przyda się w czasie pisania programów.

„STL to skondensowana wiedza i doświadczenie, a ta książka to skondensowana biblioteka STL. Nie, nie pożyczę swojego egzemplarza... kup sobie własny!”

*Andrew Duncan, Senior Software Engineer, Expertcity Inc.*



# Spis treści

<b>Wprowadzenie</b> .....	<b>5</b>
<b>Kontenery</b> .....	<b>8</b>
Kontenery standardowe .....	9
Adaptory kontenerów .....	11
Wartości .....	11
Typowe składowe .....	12
Wyjątki .....	22
Kolejki .....	23
Listy .....	24
Mapy .....	28
Kolejki priorytetowe .....	29
Kolejki .....	31
Zbiory .....	32
Stosy .....	34
Ciągi znaków .....	35
Wektory .....	47
<b>Iteratory</b> .....	<b>49</b>
Kategorie iteratorów .....	50
Stosowanie iteratorów .....	51
Cechy iteratorów .....	54
Iteratory do stałej .....	55
Iteratory wstawiające .....	57
Iteratory strumieni wejścia-wyjścia .....	58
Iterator typu raw storage .....	65
Iteratory wsteczne .....	65
Szablony funkcji iteratorów .....	70
<b>Algorytmy</b> .....	<b>70</b>
Operacje niemodyfikujące .....	73
Porównania .....	73
Wyszukiwanie .....	76

Wyszukiwanie binarne.....	77
Operacje modyfikujące sekwencje.....	79
Operacje na niezainicjowanych sekwencjach .....	85
Sortowanie.....	86
Łączenie .....	88
Operacje na zbiorach .....	89
Operacje na stercie .....	91
Permutacje.....	93
Różne .....	93
Numeryczne.....	94
<b>Funktory .....</b>	<b>95</b>
Stosowanie funktorów .....	96
Podstawy funktorów .....	98
Adaptory .....	99
Funktory wiążące.....	102
Funktory arytmetyczne i logiczne .....	103
Funktory porównujące .....	105
<b>Różne.....</b>	<b>106</b>
Alokatory.....	106
Pola bitowe (bitset) .....	109
Pary .....	113
Sprytne wskaźniki.....	114
<b>Boost .....</b>	<b>117</b>
Tablice.....	118
Dynamiczne pola bitowe .....	118
Funktory wiążące.....	119
Składanie funkcji .....	120
Adaptory .....	122
Zmiana pliku nagłówkowego Functional .....	122
Funkcje lambda .....	123
Sprytne wskaźniki.....	124
<b>Skorowidz .....</b>	<b>125</b>

## Różne

W tym rozdziale opisane zostaną szablony klas `allocator`, `auto_ptr` i `bitset`, których nie można przyporządkować do innych kategorii.

## Alokatory

Alokator jest rozwinięciem wyrażeń `new` i `delete`. Standardowe kontenery stosują alokatory do alokowania i zwalniania pamięci a także do konstruowania i niszczenia obiektów zapisanych w kontenerze.

Biblioteka standardowa definiuje szablon klasy `allocator`, który jest domyślnym alokatorem wszystkich standardowych kontenerów. Możliwe jest zastosowanie innego alokatora pod warunkiem, że udostępni on ten sam interfejs co alokator standardowy.

Implementowanie nowego alokatora jest trudniejsze niż można by z początku sądzić i wykracza poza ramy tej książki. W tym podrozdziale zostaną opisane tylko sposoby zastosowania standardowego szablonu klasy `allocator`.

Poniżej opisane zostały typy składowe szablonu `allocator`:

```
typedef const T* const_pointer  
    Typ wskaźnika na stały element.
```

```
typedef const T* const_reference  
    Typ stałej l-wartości.
```

`typedef ptrdiff_t difference_type`

Typ reprezentujący różnicę między dowolnymi dwoma wskaźnikami zwróconymi przez alokator po wywołaniu funkcji `allocate()`.

`typedef T* pointer`

Typ wskaźnika.

`template <class U> struct rebind`

Wiąże obiekt alokatora z innym typem wartości. Klasa `rebind` posiada jedną deklarację `typedef` — `other` — będącą instancją szablonu `allocator` z typem `U`, podanym w parametrze szablonu. Standardowe kontenery alokujące obiekty pomocnicze, takie jak węzły łączące, zamiast bezpośredniego alokowania wartości wykorzystują szablon `rebind`. Osoby, które nie muszą implementować standardowych kontenerów, najprawdopodobniej nie będą też musiały znać sposobu działania tego szablonu.

`typedef T& reference`

Typ l-wartości.

`typedef size_t size_type`

Typ, który może reprezentować rozmiar największego żądania alokacji.

`typedef T value_type`

Typ alokowanych wartości.

Poniżej opisane zostały metody szablonu `allocator`:

`allocator() throw()`

`allocator(const allocator&) throw()`

`template<class U> allocator(const allocator<U>&) throw()`

Tworzy nowy obiekt alokatora kopiując, jeżeli to możliwe, istniejący alokator.

`pointer address(reference x) const`

`const_pointer address(const_reference x) const`

Zwraca adres elementu `x`, czyli wartość `&x`.

```
pointer allocate(size_type n, allocator<void>::  
                const_pointer hint = 0)
```

W celu zaalokowania pamięci wystarczającej do przechowania  $n$  elementów typu  $T$ , wywołuje globalny operator `new`. Argument `hint` musi mieć wartość `0` lub wartość wskaźnika uzyskanego z innego wywołania funkcji `allocate`, którego nie przekazano jeszcze do funkcji `deallocate`. Zwracany jest wskaźnik na zaalokowaną właśnie pamięć. Jeżeli nie można zaalokować wystarczającej ilości pamięci, zgłaszany jest wyjątek `bad_alloc`.

```
void construct(pointer p, const T& val)
```

Za pomocą globalnego operatora `new` tworzy kopię wartości `val` i umieszcza ją pod adresem `p`.

```
void deallocate(pointer p, size_type n)
```

Zwalnia pamięć wskazywaną przez `p` poprzez wywołanie globalnego operatora `delete`. Argument `n` przechowuje liczbę elementów typu  $T$  — ta sama wartość przekazana była do funkcji `allocate`.

```
void destroy(pointer p)
```

Wywołuje destruktorka obiektu umieszczonego pod adresem `p`. Oznacza to, że wykonywane jest wywołanie `reinterpret_cast<T*>(p)->~T()`.

```
size_type max_size() const throw()
```

Zwraca maksymalny rozmiar, który można przekazać funkcji `allocate`.

Standard definiuje specjalizowany szablon `allocator<void>`, który nie deklaruje funkcji `allocate`, `construct` itd., ponieważ nie jest możliwe utworzenie obiektu typu `void`. Możliwe jest jednak stosowanie jego składowych `pointer`, `const_pointer` i `rebind`.

Operatory równości (`==` i `!=`) są przeciążane w ten sposób, że wszystkie obiekty typu `allocator` są sobie równe niezależnie od typu alokowanych wartości.

## Pola bitowe (*bitset*)

Obiekty typu `bitset` są spakowanymi sekwencjami bitów o stałej wielkości. Nie są to standardowe kontenery, nie udostępniają też iteratorów.

Szablon klasy `bitset` (zadeklarowany jest w pliku nagłówkowym `<bitset>`) pobiera pojedynczy parametr `N` określający liczbę przechowywanych bitów.

Pojedynczy bit może zostać ustawiony na wartość jedynki (funkcja `set`) lub zera (funkcja `reset`). Możliwa jest też zmiana wartości bitu z jedynki na zero i z zera na jedynkę, umożliwia to funkcja `flip`. Poniżej zostały opisane metody szablonu `bitset`:

`bitset()`

Tworzy obiekt typu `bitset` z wszystkimi bitami wyzerowanymi.

`bitset(unsigned long value)`

Tworzy obiekt typu `bitset` inicjalizując pierwszych  $m$  bitów wartością `value`, gdzie  $m == \text{CHAR\_BITS} * \text{sizeof}(\text{unsigned long})$ . Jeżeli  $N > m$ , to wszystkie pozostałe bity są ustawiane na wartość zero. Jeżeli  $N < m$ , to nadmiarowe bity są ignorowane.

```
template<typename CharT, typename Traits, typename Alloc>  
explicit bitset(const basic_string<CharT,Traits,Alloc>& s,  
               typename basic_string<CharT,Traits,Alloc>::size_type p=0,  
               typename basic_string<CharT,Traits,Alloc>::size_type n=  
               basic_string<CharT,Traits,Alloc>::npos)
```

Tworzy obiekt typu `bitset` i inicjalizuje go znakami z ciągu znaków `s`, rozpoczynając od znaku `p` i wykorzystując następujących `n` znaków (albo znaki do końca ciągu, jeżeli jest on krótszy od `n`). Domyślnie wykorzystywany jest cały ciąg znaków. Znak o wartości `'0'` powoduje wyzerowanie bitu a znak o wartości `'1'` powoduje ustawienie bitu na jedynkę. Znaki o innej wartości powodują zgłoszenie wyjątku `invalid_argument`.

Znak znajdujący się najbardziej po prawej stronie podciągu (czyli znak `s[p+n-1]` albo ostatni znak ciągu `s`) inicjalizuje najmniej znaczący bit pola, czyli bit o indeksie 0. Kolejne bity pola inicjowane są znakami z poprzednich indeksów ciągu `s`. Bity niezainicjowane przez ciąg znaków są zerowane. Wszystkie pola bitowe z poniższych przykładów otrzymują wartość `000111`:

```
bitset<6> a(string("111"));
bitset<6> b(string("000111"));
bitset<6> c(string("10110011100"), 5, 4);
bitset<6> d(string("111111"), 3, 42);
```

Tak długa deklaracja szablonu wynika z zastosowania szablonu klasy `basic_string`. W typowych zastosowaniach wykorzystujących tylko klasę `string`, można tą deklarację odczytywać następująco:

```
bitset(const string& s, size_t p=0,
       size_t n=string::npos)
```

```
bool any() const
```

Zwraca wartość `true`, jeżeli którykolwiek bit jest ustawiony na wartość jeden a wartość zera, gdy wszystkie bity są zerami.

```
size_t count() const
```

Zwraca liczbę bitów o wartości jeden.

```
bitset<N>& flip()
```

Odwraca wartość wszystkich bitów. Zwraca wartość `*this`.

```
bitset<N>& flip(size_t pos)
```

Odwraca wartość bitu na pozycji `pos`. Jeżeli wartość `pos` jest nieprawidłowa, zgłaszany jest wyjątek `out_of_range`. Zwraca wartość `*this`.

```
bool none() const
```

Zwraca wartość `true`, gdy wszystkie bity zerami, a wartość `false`, jeżeli którykolwiek bit jest ustawiony na wartość jeden.

reference operator[](size\_t pos)

Zwraca obiekt `bitset::reference` będący referencją na bit na pozycji `pos`. Jeżeli wartość `pos` jest z poza zakresu, zachowanie funkcji jest niezdefiniowane. Klasa `bitset::reference` przechowuje referencję na obiekt `bitset` i pozycję `pos`. Przeciąża ona operator przypisania (`=`) w ten sposób, że przypisania do obiektu typu `reference` zmieniają wartość pola bitowego `bitset`. Klasa `reference` definiuje też metodę `flip`, odwracającą wartość bitu, którego dotyczy referencja.

bool operator[](size\_t pos) const

Zwraca wartość bitu na pozycji `pos`. Jeżeli wartość `pos` jest spoza zakresu, zachowanie funkcji jest niezdefiniowane.

bitset<N>& reset()

Zeruje wszystkie bity. Zwraca wartość `*this`.

bitset<N>& reset(size\_t pos)

Zeruje bit na pozycji `pos`. Jeżeli wartość `pos` jest nieprawidłowa zgłaszany jest wyjątek `out_of_range`. Zwraca wartość `*this`.

bitset<N>& set()

Ustawia wszystkie bity. Zwraca wartość `*this`.

bitset<N>& set(size\_t pos, int val = true)

Jeżeli wartość `val` jest różna od zera, ustawia bit na pozycji `pos`. Jeżeli wartość `pos` jest nieprawidłowa, zgłaszany jest wyjątek `out_of_range`. Zwraca wartość `*this`.

size\_t size() const

Zwraca wartość `N`.

bool test(size\_t pos) const

Zwraca wartość bitu na pozycji `pos`. Jeżeli wartość `pos` jest nieprawidłowa, zgłaszany jest wyjątek `out_of_range`.

```
template <class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const
```

Zwraca ciąg znaków reprezentujący zawartość obiektu `bitset`. Każdy wyzerowany bit jest zamieniany na znak '0', a bity ustawione na jedynkę zamieniane są na znak '1'. Bit z pozycji 0 jest zapisywany w ciągu na pierwszej pozycji od prawej strony (pozycji  $N-1$ ).

W czasie wywoływania funkcji `to_string` kompilator nie jest w stanie wykryć parametrów szablonu, dlatego trzeba je podać jawnie:

```
std::bitset<64> bits(std::string("101000111101010101"));
std::string str = bits.template to_string<char,
std::char_traits<char>,
std::allocator<char> >());
```

```
unsigned long to_ulong() const
```

Przetwarza zawartość obiektu `bitset` na wartość całkowitą. Jeżeli  $N$  jest zbyt duże, żeby można było ją zapisać jako `unsigned long`, zgłaszany jest wyjątek `overflow_error`.

Dla obiektów `bitset` definiowane są również operatory bitowe, przesunięcia i równości stosujące zwyczajową semantykę. Operandów operatorów bitowych muszą być tej samej wielkości. Operatory przesunięć uzupełniają brakujące bity zerami.

Przeciążane są też operatory wejścia-wyjścia. Operator wyjścia (`<<`) zapisuje zawartość obiektu `bitset` do ciągu znaków w ten sam sposób jak robi to funkcja `to_string`. Operator wejścia (`>>`) odczytuje z ciągu znaków zawartość obiektu `bitset`, tak samo jak robi to konstruktor obiektu.

## Pary

Szablon klasy `pair` reprezentuje parę związanych ze sobą obiektów. Pary są najczęściej stosowane w szablonach klas `map` i `multi-map`, które przechowują w nich klucze i związane z nimi obiekty. Szablon `pair` i związane z nim szablony funkcji są zadeklarowane w pliku nagłówkowym `<utility>`.

Deklaracji szablonu `pair` prawie nie trzeba objaśniać:

```
template <typename T1, typename T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y);
    template<typename U, typename V>
        pair(const pair<U, V> &p);
};
```

Konstruktory obiektów `pair<T1, T2>` nie są zbyt skomplikowane:

`pair()`

Inicjalizuje element `first` wartością `T1()` a element `second` wartością `T2()`.

`pair(const T1& x, const T2& y)`

Inicjalizuje element `first` wartością `x` a element `second` wartością `y`.

`template<typename U, typename V>`

`pair(const pair<U, V> &p);`

Inicjalizuje element `first` wartością `p.first` a element `second` wartością `p.second`. Jeżeli to konieczne, są wykonywane odpowiednie konwersje.

Niektóre szablony funkcji ułatwiają pracę z parami:

```
template<typename T1, typename T2>
```

```
pair<T1,T2> make_pair(T1 a, T2 b)
```

Tworzy obiekt `pair<T1, T2>` i inicjalizuje go wartościami `a` i `b`. Zastosowanie funkcji `make_pair` zamiast konstruktora szablonu `pair<>` pozwala kompilatorowi rozpoznać typy `T1` i `T2` wartości `a` i `b`.

```
template<typename T1, typename T2>
```

```
bool operator==(const pair<T1,T2>& a, const pair<T1,T2>& b)
```

Zwraca wartość `true`, gdy pary `a` i `b` są sobie równe, czyli gdy `a.first == b.first` i `a.second == b.second`.

```
template<typename T1, typename T2>
```

```
bool operator<(const pair<T1,T2>& a, const pair<T1,T2>& b)
```

Zwraca wartość `true`, gdy para `a` jest mniejsza od pary `b`, przy założeniu, że element `first` jest bardziej znaczący niż element `second`. To znaczy, że zwracany jest wynik wyrażenia `a.first < b.first || (!(b.first < a.first) && a.second < b.second)`.

Pozostałe operatory porównań definiowane są na podstawie operatorów `==` i `<`.

## *Sprytne wskaźniki*

Szablon klasy `auto_ptr` (zadeklarowanej w pliku nagłówkowym `<memory>`) implementuje sprytne wskaźniki będące właścicielami wskaźników. Właściwe stosowanie klasy `auto_ptr` daje pewność, że dany wskaźnik posiada tylko jednego właściciela (co pozwala na uniknięcie podwójnych usunięć wskaźnika), który automatycznie zwalnia pamięć, gdy właściciel wyjdzie poza dopuszczalny zakres (dzięki czemu unika się wycieków pamięci). Przypisanie wartości typu `auto_ptr` zmienia właściciela wskaźnika z obiektu źródłowego na obiekt docelowy operacji przypisania.

---

## Uwaga

Szablon `auto_ptr` nie ma semantyki wartości, ponieważ przypisanie lub kopiowanie obiektu typu `auto_ptr` powoduje modyfikację obiektu źródłowego (zrzeczenie się własności), dlatego obiekty typu `auto_ptr` nie mogą być przechowywane w kontenerach.

---

Poniżej znajdują się opisy składowych szablonu `auto_ptr`:

`typedef T element_type`

Synonim typu bazowego.

`explicit auto_ptr(T* p = 0) throw()`

Inicjalizuje obiekt `auto_ptr`, tak żeby był on właścicielem wskaźnika `p`.

`auto_ptr(auto_ptr& x) throw()`

`template<class U> auto_ptr(auto_ptr<U>& x) throw()`

Inicjalizuje obiekt `auto_ptr` wskaźnikiem zwróconym przez `x.release()`. W drugiej wersji konstruktora typ `U*` musi być pośrednio konwertowalny na `T*`. Należy zauważyć, że `x` nie jest oznaczony jako `const`. Nie jest możliwe kopiowanie obiektu typu `const auto_ptr`, ponieważ doprowadziłoby to naruszenia zasad własności.

`auto_ptr(auto_ptr_ref<T> r) throw()`

Inicjalizuje obiekt `auto_ptr` wskaźnikiem uzyskanym z wywołania funkcji `release` z obiektu `r`.

`~auto_ptr() throw()`

Usuwa wskaźnik posiadany przez obiekt np. przez wywołanie `delete get()`.

`T* get() const throw()`

Zwraca posiadany wskaźnik.

`T* release() throw()`

Zwraca wynik działania funkcji `get()` a posiadany wskaźnik ustawia na 0.

```
void reset(T* p = 0) throw()
```

Usuwa posiadany wskaźnik (jeżeli nie jest on równy p) i zapisuje p jako nowo posiadany wskaźnik.

```
template<class U> operator auto_ptr_ref<U>() throw()
```

Zwraca tymczasowy obiekt typu auto\_ptr\_ref posiadający wskaźnik. Wskaźnik musi być konwertowalny na typ U\*. Własność jest przenoszona na nowy obiekt auto\_ptr\_ref.

Typ auto\_ptr\_ref jest typem definiowanym w implementacji, ułatwiającym stosowanie obiektów auto\_ptr jako typów zwracanych przez funkcję. W większości przypadków, można ignorować typ auto\_ptr\_ref i deklarować parametry funkcji i wartości przez nie zwracane z wykorzystaniem typu auto\_ptr, pozwalając kompilatorowi wykonać odpowiednie konwersje.

```
template<class U> operator auto_ptr<U>() throw()
```

Zwraca nowy obiekt typu auto\_ptr. Posiadany wskaźnik jest konwertowany na typ U\* a nowy obiekt auto\_ptr przejmuje go na własność.

```
auto_ptr& operator=(auto_ptr& x) throw()
```

```
template<class U>
```

```
auto_ptr& operator=(auto_ptr<U>& x) throw()
```

```
auto_ptr& operator=(auto_ptr_ref<T> r) throw()
```

Własność wskaźnika przenoszona jest na obiekt \*this z obiektu x albo z obiektu auto\_ptr przechowywanego przez parametr r. To znaczy, że wykonywane jest wywołanie reset(x.release()).

```
T& operator*() const throw()
```

Zwraca wynik wywołania \*get(). Jeżeli przechowywany jest wskaźnik null, zachowanie operatora jest niezdefiniowane.

```
T* operator->() const throw()
```

Zwraca wynik wywołania get().

W rozdziale „Boost” opisano inne sprytne wskaźniki, w tym również takie, które mogą wskazywać tablice i być przechowywane w kontenerach.