

TOM

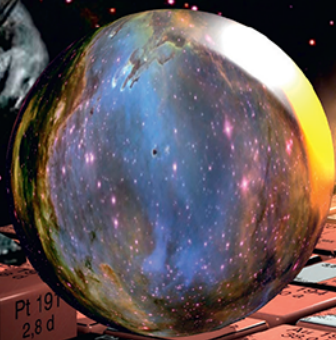
4

LATWY PODRĘCZNIK

Jerzy Grębosz

Opus magnum C++

# Misja w nadprzestrzeń C++14/17



Helion

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione.

Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki i opracowanie graficzne książki: Jerzy Grębosz

Zdjęcie Młławicy Orzeł w grafice na okładce oraz zdjęcia łazika Curiosity

i powierzchni Marsa – wykorzystane w tytułach rozdziałów – dzięki uprzejmości NASA

## Wydanie drugie B

ISBN: 978-83-8322-582-1

Copyright © Jerzy Grębosz 2020, 2023

Printed in Poland

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/op144v>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<https://ftp.helion.pl/przyklady/op144v.zip>

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

<b>0</b>	<b>Proszę tego nie czytać!</b> .....	<b>1</b>
0.1	Wyruszamy na kolejną wyprawę!.....	1
<b>1</b>	<b>Szablony o zmiennej liczbie parametrów</b> .....	<b>3</b>
1.1	Szablon funkcji o zmiennej liczbie parametrów (i argumentów).....	4
1.2	Jak dobrać się do argumentów tkwiących w pakiecie?.....	11
1.2.1	Ciekawe szablony zwracające rezultat .....	16
1.3	Szablon klas o dowolnej (zmiennej) liczbie parametrów.....	18
1.4	Trzy rodzaje pakietów parametrów szablonu .....	24
1.4.1	Pakiet szablonu będący pakietem wartości .....	25
1.5	Argumenty pakietu odbierane przez wartość, referencję, adres.....	30
1.6	Rozwinięcie według wzorca (czyli rozwinięcie „z kontekstem”).....	31
1.7	Rozwinięcie pakietu typów w klamrowej liście inicjalizatorów .....	34
1.7.1	Łatwe narzędzie do wypisania argumentów.....	34
1.8	Rozwinięcie pakietu na liście parametrów aktualnych innego szablonu .....	36
1.9	Gdzie można umieścić wyrażenia rozwijające pakiet parametrów .....	39
1.10	Cwiczenia .....	39
<b>2</b>	<b>Cechy języka wprowadzone do standardu C++14</b> .....	<b>43</b>
2.1	Zapis dwójkowy stałych dosłownych .....	43
2.2	Separatory cyfr w stałych dosłownych .....	44
2.2.1	Wypisywanie liczb w postaci binarnej.....	45
2.2.2	Wczytywanie liczb dwójkowych strumieniem wejściowym.....	48
2.3	Kompilator rozpoznaje typ rezultatu funkcji .....	50
2.4	Deklaracja typu rezultatu <i>decltype(auto)</i> .....	52
2.4.1	Przykład zastosowania konstrukcji <i>decltype(auto)</i> w szablonie funkcji.....	55
2.5	Szablon definicji zmiennej .....	59
2.5.1	Jak to drzewiej bywało, czyli świat bez szablonów zmiennych .....	64
2.5.2	Teraz zobaczysz, jak prosto się to robi z C++14.....	71
2.5.3	Ciekawe zastosowanie: sprawdzenie cech charakteru danego typu .....	72
2.5.4	Lubię, nie lubię... .....	77
2.5.5	Dwa usprawnienia.....	80
2.5.6	Realizacja tego pomysłu w programie .....	81
2.6	Przeładowanie globalnych operatorów <i>new</i> , <i>new [ ]</i> , <i>delete</i> i <i>delete [ ]</i> .....	86
2.7	Nowości C++14 w wyrażeniach lambda .....	94
2.7.1	Przykład uogólnionego wyrażenia lambda.....	94
2.7.2	Przykład definicji obiektu na liście wychwytywania i jego inicjalizacja .....	96
2.7.3	Przykład wychwylenia na zasadzie przeniesienia (move) .....	99



2.8	C++14 a funkcje <i>constexpr</i> .....	104
2.8.1	Zniesienie wielu ograniczeń w ciele funkcji <i>constexpr</i> .....	104
2.8.2	Funkcje składowe <i>constexpr</i> w C++14 nie są już automatycznie <i>const</i> .....	110
2.9	Atrybuty.....	111
2.9.1	Nowy atrybut <i>[[deprecated]]</i> wprowadzony w C++14.....	112
2.9.2	Oznaczenie wybranej funkcji jako przestarzałej.....	113
2.9.3	Argument funkcji uznany za przestarzały.....	114
2.9.4	Przestarzałe niestacyjne składniki klasy: funkcja składowa i dana składowa.....	114
2.9.5	Obiekt oznaczony jako przestarzały.....	115
2.9.6	<i>deprecated</i> a zbiorcza definicja kilku zmiennych (z ewentualną inicjalizacją).....	116
2.9.7	Typy, które uznajemy za przestarzałe.....	116
2.9.8	Przestarzałe synonimy typów (w instrukcjach <i>typedef</i> i <i>using</i> ).....	118
2.9.9	Oznaczanie atrybutem <i>[[deprecated]]</i> specjalizacji szablonu klasy.....	118
2.9.10	Oznaczanie atrybutem <i>[[deprecated]]</i> specjalizacji szablonu funkcji.....	119
2.10	Przewrotu nie było.....	119
2.11	Ćwiczenia.....	119

### 3 Cechy języka wprowadzone do standardu C++17.....123

3.1	Specyfikacja wyjątków staje się częścią typu funkcji.....	123
3.2	Technika „pomijanie kopiowania” bywa teraz obowiązkiem kompilatora.....	128
3.3	Przydomek <i>alignas</i> a operatory <i>new</i> i <i>delete</i> .....	134
3.3.1	Przeładowanie globalnych <i>new</i> i <i>delete</i> uwzględniające wyrównanie adresów.....	135
3.3.2	Jak przeładować wyrównujące operatory <i>new/delete</i> na użytek wybranej klasy.....	141
3.4	Porządek obliczania składników w złożonych wyrażeniach – nareszcie ustalony.....	144
3.5	Stała znakowa typu <i>u8</i> .....	146
3.6	Szesnastkowy zapis liczb zmiennoprzecinkowych.....	147
3.6.1	Wypisywanie i wczytywanie zmiennoprzecinkowych liczb szesnastkowych.....	149
3.7	Wyrażenia poskładane w harmonijkę – ułatwienie pracy z pakietem argumentów.....	152
3.7.1	Pierwszy przykład użycia wyrażenia harmonijkowego w szablonie.....	152
3.7.2	Harmonijka z dodatkowym wyrażeniem początkowym.....	155
3.7.3	Cztery formy wyrażenia harmonijkowego.....	157
3.7.4	Kontekst wyrażenia harmonijkowego – przykład.....	159
3.8	Dozwolone słowo <i>auto</i> w deklaracji <i>template &lt;auto&gt;</i> .....	161
3.9	Kompilator rozpoznaje typ parametrów klasy szablonej.....	166
3.9.1	Wektory czego innego niż widać.....	170
3.10	Instrukcja <i>if constexpr</i> – prawie jak kompilacja warunkowa.....	173
3.10.1	Instrukcja <i>if constexpr</i> rozwiązuje problem „lubianych” i „nie lubianych” typów.....	176
3.11	Wyrażenia inicjalizujące w instrukcjach <i>if</i> i <i>switch</i> .....	179
3.12	Dowiązania strukturalne, czyli łatwe „sięganie do składników”.....	182
3.12.1	Dowiązanie do tablic zbudowanych na bazie klasy <i>std::array</i> .....	184
3.12.2	Łatwe sięganie do składników struktur/klas.....	185
3.12.3	Przystosowanie naszej klasy do obsługi deklaracji dowiązań.....	192
3.12.4	Przystosowanie cudzej klasy do obsługi deklaracji dowiązań.....	198
3.13	Operator preprocesora zwany <i>__has_include</i> .....	201
3.14	Nowe atrybuty: <i>maybe_unused</i> , <i>fallthrough</i> i <i>nodiscard</i> .....	203
3.14.1	Atrybut <i>[[maybe_unused]]</i> .....	205
3.14.2	Atrybut <i>[[fallthrough]]</i> używany w instrukcji <i>switch</i> .....	208
3.14.3	Atrybut <i>[[nodiscard]]</i> – nie zlekceważ mnie.....	210
3.15	Typ <i>std::byte</i> do operacji na surowych blokach pamięci.....	213
3.16	Modyfikacje istniejących cech języka.....	222
3.17	Rozluźnienie zasady inicjalizowania typów wyliczeniowych.....	223
3.18	Modyfikacja deklaracji <i>static_assert</i> .....	224
3.19	Prostszy sposób zapisu zagnieżdżonych przestrzeni nazw.....	225
3.20	Dozwolone słowo <i>typename</i> w parametrze określającym inny szablon.....	229

3.21	Dla zakresowej pętli <i>for</i> funkcje <i>begin</i> i <i>end</i> mogą zwracać odmienne typy .....	233
3.22	Rozwinięcie pakietu możliwe nawet w deklaracji <i>using</i> .....	239
3.23	Nowe zasady <i>auto</i> -rozpoznawania typu obiektów mających inicjalizację klamrową .....	245
3.24	W C++17 agregat może być nawet klasą pochodną .....	247
3.25	Zmiana typu rezultatu funkcji <i>std::uncaught_exception</i> .....	250
3.26	Ćwiczenia .....	254

## **4** **Posłowie – czyli C++20 *ante portas*.....264**

## **Skorowidz .....265**



## 2.7 Nowości C++14 w wyrażeniach lambda

O wyrażeniach lambda rozmawialiśmy w *Opus magnum* w rozdziale 30. Teraz zapoznamy się z dwoma ciekawymi cechami wprowadzonymi przez standard C++14. Są to:

- 1) uogólnione wyrażenia lambda,
- 2) definicja (na liście wychwytywania) lokalnego obiektu, mającego swoją inicjalizację.

Zanim zobaczymy te cechy w przykładowych programach, kilka zdań wyjaśnienia.

### 2.7.1 Przykład uogólnionego wyrażenia lambda

Mówiąc najprościej, w C++11 argumenty formalne wyrażenia lambda musiały być ściśle określonego typu. Tymczasem...



Standard C++14 pozwala, by typ argumentu formalnego naszego wyrażenia lambda był parametrem. Robimy to, określając typ danego parametru słowem `auto`. Dzięki temu nasze wyrażenie lambda staje się jakby szablonem funkcji lambda o danej nazwie.

W poniższym programie zobaczysz, jakie to proste.



```

#include <iostream>
using namespace std;
//*****
int main()
{
    // wytworzenie zwykłego wyrażenia lambda C++11
    auto czy_mniejsze_int = [] (int m, int k) { return m < k; };

    cout << boolalpha;
    cout << "Czy (5 < 2) ? " << czy_mniejsze_int(5, 2) << endl;
    cout << "Czy (5.1 < 5.9) ? " << czy_mniejsze_int(5.1, 5.9) << endl; // niestety!

    // C++14
    // wytworzenie uogólnionego wyrażenia lambda
    auto czy_mniejsze_uniwersalne = [] (auto m, auto k) { return m < k; };

    cout << "Czy (5 < 2) ? " << czy_mniejsze_int(5, 2) << endl;
    cout << "Czy (5.1 < 5.9) ? " << czy_mniejsze_int(5.1, 5.9) << endl;
    // to działa nawet na znakach
    cout << "Czy ('a' < 'b') ? " << czy_mniejsze_uniwersalne('a', 'b') << endl;
    cout << "Czy ('b' < 'a') ? " << czy_mniejsze_uniwersalne('b', 'a') << endl;

    cout << "Wolno nawet nawet porownac wartosci roznych typow\n";
    cout << "Czy (3 < 3.14) ? " << czy_mniejsze_uniwersalne(3, 3.14) << endl;

    cout << "Znak 'p' ma kod liczbowy: " << int('p') << endl;

    cout << "Czy ('p' < 111) ? " << czy_mniejsze_uniwersalne('p', 111) << endl;
    cout << "Czy ('p' < 113.5) ? " << czy_mniejsze_uniwersalne('p', 113.5) << endl;
}

```

❶

❷

❸

❹

❺

❻

❼

❽

❾

❿

⓫



### Na ekranie pojawi się taki tekst:

Czy (5 < 2) ? false	3
Czy (5.1 < 5.9) ? false	4
Czy (5 < 2) ? false	6
Czy (5.1 < 5.9) ? false	7
Czy ('a' < 'b') ? true	
Czy ('b' < 'a') ? false	
Wolno nawet nawet porównac wartosci roznych typow	
Czy (3 < 3.14 ) ? true	9
Znak 'p' ma kod liczbowy: 112	
Czy ('p' < 111) ? false	10
Czy ('p' < 113.5) ? true	11



### Najpierw przypomnienie

Jak pamiętamy z C++11, wyrażenie lambda najczęściej wpisujemy wprost w potrzebną nam instrukcję. Zostaje ono użyte przez tę instrukcję, a potem staje się niepotrzebne. Gdybyśmy jednak chcieli skorzystać z danego wyrażenia lambda kilkakrotnie (w kilku innych instrukcjach), możemy to zrobić, nadając mu nazwę. W praktyce nadanie nazwy polega na tym, że definiujemy „obiekt lambda” o określonej nazwie i inicjalizujemy go wymyślonym przez siebie wyrażeniem lambda.

```
auto nazwa_obiektu_lambda = wyrażenie_lambda;
```

Abyśmy się nie musieli głowić, jakiego typu jest nasze wyrażenie lambda, stosujemy zastępcze słowo kluczowe auto. Nie będę rozwijał tego zagadnienia, bo o tej sprawie rozmawialiśmy w *Opusie* (§30.5.1). Koniec przypomnienia, teraz o nowościach.

- 1 W naszym programie widzimy tak właśnie zdefiniowane wyrażenie lambda. Obiekt lambda, w którym będziemy je przechowywać, nazywamy długą, ale obrazową nazwą czy\_mniejsze\_int. Patrząc na ciało tego wyrażenia lambda, łatwo zauważysz, że otrzymuje ono dwa argumenty (dwie wartości typu int) i sprawdza, czy pierwsza wartość jest mniejsza od drugiej. Jeśli „tak”, to zwraca ono wartość true, jeśli „nie”, wówczas zwraca wartość false.

- 3 Oto wywołanie tego wyrażenia lambda dla dwóch wartości typu int: 5 oraz 2. Na ekranie pojawia się odpowiedź false.

*Odpowiedź jest „s³owna” dzięki manipulatorowi boolalpha, którym powiedziliamy strumieniowi cout, że wolimy „s³ownie” 2.*

A co będzie, jeśli to samo wyrażenie lambda zastosujemy wobec dwóch wartości typu double?

- 4 Oto taka sytuacja. Niestety nie zadziała to poprawnie. Skoro wartości 5.1 oraz 5.9 (czyli typu double) zostały wysłane wyrażeniu lambda obsługującemu wartości typu int, to nastąpiło ich obcięcie do typu int. Zatem wyrażenie lambda porównało, czy (5 < 5). Wartością tego wyrażenia jest false, bo przecież 5 nie jest mniejsze od 5.

Wniosek? Nasze wyrażenie lambda nie nadaje się do porównywania wartości typu double. Nie jest uniwersalne.

Ten kłopot rozwiązuje możliwość wytworzenia uogólnionego wyrażenia lambda, na które pozwala nam standard C++14.

- 5 Oto jego definicja. Zacytujmy:

```
auto czy_mniejsze_uniwersalne = [] (auto m, auto k) { return m < k; };
```

Jak widać, zamiast określać typ argumentów `m i k`, postawiliśmy tam słowa `auto`. Dzięki temu stworzyliśmy jakby szablony wyrażen lambda o danej nazwie. Takie uogólnione wyrażenie lambda można zastosować do wartości różnych innych typów. (Oczywiście takich, wobec których operator `<` ma sens). Oto kilka takich sytuacji.

- ⑥ Użycie wyrażenia lambda czy `_mniejsze_uniwersalne` wobec argumentów typu `(int, int)`. Działa, podobnie jak to poprzednie (③).
- ⑦ Użycie wobec argumentów typu `(double, double)`. Działa poprawnie, czego poprzednia lambda (④) nie potrafiła.
- ⑧ Użycie wobec argumentów typu `(char, char)`. Także działa poprawnie.
- ⑨ Dwa argumenty naszego uogólnionego wyrażenia lambda wcale nie muszą być tego samego typu. Przecież każdy z nich ma swoje własne określenie `auto`. Oto użycie tego wyrażenia wobec pary argumentów `(int, double)`. Jak widać, także i ono działa poprawnie.
- ⑩ A oto bardziej karkołomne zastosowanie. Porównanie argumentów `(char, int)`, a potem (11) argumentów `(char, double)`. (Patrzac na ekran i sprawdzając, czy i jak to działa, pamiętaj że kod ASCII znaku 'p' to 112).



Pora na podsumowanie: uogólnione wyrażenie lambda to narzędzie bardzo wygodne i bardzo proste w użyciu. Sprawia, że wyrażenie lambda, które właśnie tworzymy, może być bardziej uniwersalne, bo można je zastosować do wielu różnych typów argumentów.

Dociekliwym wyjaśniam, że cały mechanizm uogólnionego wyrażenia lambda polega na tym, że kompilator zamienia je nie na obiekt funkcyjny mający funkcję składową `operator()`, ale na obiekt funkcyjny mający *szablon funkcji składowej* `operator()`. W zależności od tego, jakie podajemy argumenty aktualne naszemu wyrażeniu lambda, kompilator produkuje taką lub inną przeładowaną wersję tego `operator()`.

Jak widać, cały spryt uogólnionego wyrażenia lambda wynika z tego, że używa ono szablonu. Szablony zaś, jak to już kiedyś ustaliliśmy, to nie:

- ❖ *run-time* polimorfizm,
  - innymi słowami, wielopostaciowo zrealizowana już w trakcie pracy programu (za pomocą funkcji wirtualnej),*
- ❖ *lecz compile-time* polimorfizm
  - inaczej: wielopostaciowo zrealizowana w trakcie kompilacji (za pomocą szablonu).*

Przyznam się, że nie lubię tych uogólnionych wyrażen lambda nazywać polimorficznymi. Niepotrzebnie kojarzy się to z funkcjami wirtualnymi, a co tu dużo mówić, ten mechanizm nie dorasta funkcjom wirtualnym do pięt.

## 2.7.2 Przykład definicji obiektu na liście wychwytywania i jego inicjalizacja

Jak wiadomo, wyrażenie lambda może wychwycić jakieś lokalne obiekty automatycznie dostępne w zakresie, w którym ono nastąpiło. Dzięki temu może z nich korzystać w swoim ciele.





C++14 daje nam dodatkową możliwość:

Na liście wychwytywania możemy zdefiniować jakiś dodatkowy obiekt, który nam się przyda w ciele wyrażenia lambda. Od razu inicjalizujemy go jakimś wyrażeniem.

Mówiąc ściślej – jeśli na liście wychwytywania umieścimy nazwę z inicjalizacją:

*nazwa = wyrażenie\_inicjalizacyjne;*

to tak, jakbyśmy zdefiniowali zmienną i zainicjalizowali ją. Jej zakres to zakres ciała wyrażenia lambda. Jej typ jest taki, jakby przed jej nazwą stało słowo auto (czyli wynika on z typu wyrażenia inicjalizacyjnego).

Co ciekawe i bardzo wygodne – w wyrażeniu, które inicjalizuje zmienną, wolno nam nawet wykorzystać nazwy lokalnych (automatycznych) obiektów dostępnych w zakresie, w którym nasze wyrażenie lambda tworzymy.

Jeśli potrafisz sobie wyobrazić wyrażenie lambda jako obiekt funkcyjny, to przedstawiony tu mechanizm możesz rozumieć jako możliwość dodania do wyrażenia lambda nowych danych składowych.

Te i dalsze interesujące szczegóły zobaczymy w programie poniżej.



```

#include <iostream>
#include <memory>
using namespace std;
//*****
int main()
{
    int h = 1;
    int k = 12;
    double e = 2.71;

    // definicja wyrażenia lambda
    auto impakt = [obj = 5, h = h + 400, zmienna = k * (k + 1), &ref = e] ()
    {
        // k = 1;      błąd, bo k nie było wychwycone (służy jedynie do inicjalizacji)
        // e = 1.0;    błąd (jak wyżej)
        ref = ref + 10;
        // zmienna++;      modyfikacja dozwolona tylko wtedy, jeśli lambda...
        //                                     ...ma przydomek mutable

        cout << "obj " << obj
              << ", h = " << h
              << ", zmienna = " << zmienna
              << ", ref = " << ref
              << endl;

        return;
    };

    impakt();
    cout << "Po wykonaniu lambda impakt h = " << h << endl;
}

```



### Tak wyglądał będzie ekran po wykonaniu tego programu:

obj 5, h = 401, zmienna = 156, ref = 12.71  
Po wykonaniu lambdy impact h = 1



### Omówienie

- ② W naszym programie definiujemy wyrażenie lambda. Definiujemy je (dla uproszczenia) nie w wywołaniu jakiejś funkcji algorytmu, ale – że tak powiem – w postaci „wolno stojącej”. Zostaje ono zapisane w obiekcie lambda o nazwie impact. Fakt, że tak właśnie postąpiliśmy, nie ma nic wspólnego z przedmiotem obecnego paragrafu. Po prostu dzięki temu uniknąłem konieczności definiowania funkcji-algorytmu.

Jak widzisz, lista wychwytywania (czyli to, co jest zamknięte w kwadratowym nawiasie) jest dość rozbudowana; jest w niej kilka członów oddzielonych przecinkami:

```
[obj = 5, h = h + 400, zmienna = k * (k + 1), &ref = e]
```

Jest ich aż tyle, bo chcę pokazać różne warianty nowego typu zapisu wychwytywania z inicjalizacją. Nie przerażaj się jednak, zwykle zastosujesz tylko jeden, no, może dwa takie człony. Omówimy je po kolei.

człon:  $obj = 5$

Taki zapis na liście wychwytywania odpowiada jakby zapisowi:

```
auto obj = 5;           // pseudokod
```

co oznacza następujące polecenie dla kompilatora: zdefiniuj w ramach tego wyrażenia lambda lokalną zmienną o nazwie obj i inicjalizuj ją wartością wyrażenia (5). Wyrażenie to jest (jak wiadomo) typu int, więc niech zmienna obj będzie właśnie takiego typu.

człon:  $h = h + 400$

Taki zapis mówi kompilatorowi, że chcielibyśmy, aby zdefiniował nam on lokalny obiekt o nazwie h i inicjalizował go wyrażeniem (h + 400). Ale uwaga: to h stojące na prawo od znaku = (czyli w wyrażeniu inicjalizacyjnym) to zupełnie inne h niż to po lewej. Dotyczy ono jakiegoś h, które istnieje i jest dostępne w zakresie, w którym akurat naszą lambda definiujemy. W tym przypadku będzie to obiekt ① o nazwie h, zdefiniowany w zakresie funkcji main.



Zapamiętaj to. Jeśli w wyrażeniu wychwytyjącym mamy nawet tę samą nazwę po jednej i drugiej stronie znaku =, np.:

```
h = h
```

owo h z lewej oznacza lokalny obiekt, który definiujemy na użytek ciała wyrażenia lambda, zaś to h z prawej jest nazwą jakiegoś obiektu, który powinien być dostępny w zakresie, w którym naszą lambda definiujemy.

Aby uniknąć pomyłek i nieporozumień, zwykle daję tym lokalnym obiektom inne nazwy, niż mają obiekty stojące po prawej. Przykład tego widzisz na następnej pozycji listy wychwytywania, czyli...

człon:  $zmienna = k * (k + 1)$

Tutaj sprawa jest jasna i czytelna, więc nie trzeba tego objaśniać. Dodam więc tylko, że obiekt o nazwie k został tu tylko użyty w wyrażeniu inicjalizującym, ale to wcale nie znaczy, że został wychwycony. Zresztą zobacz, próba użycia go w ciele wyrażenia

lambda (3) wywoła błąd kompilatora. (Dlatego tę instrukcję musiałem opatrzyć znakami komentarza).

człon: `&ref = e`

A to jest definicja referencji (przezwiseka) o nazwie `ref`, która od razu dowiaduje się (dzięki inicjalizacji), że jest przezwiskiem obiektu `e`. Sam obiekt `e` nie jest wychwycony, więc użycie `go` w ciele (4) byłoby błędem. Ponieważ jednak wytworzyliśmy właśnie referencję do tego obiektu, to można z niego korzystać za jej pomocą (5).

- 6 Przypominam, że wprawdzie wyrażenia lambda pozwalają nam wychwycić jakieś obiekty przez wartość (czyli przez kopię), ale kompilator nie pozwoli nam tych kopii modyfikować. Aby to było możliwe, powinniśmy naszą lambda opatrzyć przydomkiem `mutable` (zob. *Opus*, §30.3.5). Ponieważ akurat tutaj tego nie zrobiliśmy, musiałem instrukcję 6 ująć w komentarz.
- 7 Oto jest instrukcja, w której wypisujemy na ekranie wartości różnych obiektów dostępnych nam w ramach ciała naszego wyrażenia lambda. Na ekranie możesz zobaczyć na przykład wartość lokalnego obiektu `h` (401).
- 9 Potem, po zakończeniu definicji wyrażenia lambda i po wykonaniu `go` (8), wypisujemy wartość tego `h`, które było zdefiniowane w funkcji `main`. Jak widać, ta wartość nie została zmieniona i nadal wynosi 1.



Podsumujmy. C++14 wzbogacił możliwości listy wychwytywania wyrażeń lambda. Możemy tam wytwarzać (i inicjalizować) nowe pomocnicze obiekty, z których chcemy skorzystać w ciele wyrażenia lambda.

Inicjalizacja tych pomocniczych obiektów może być dowolnym wyrażeniem – nawet korzystającym z obiektów dostępnych w zakresie, w którym naszą lambda definiujemy. O pewnym szczególnym zastosowaniu takiej inicjalizacji porozmawiamy w następnym paragrafie.

### 2.7.3 Przykład wychwycenia na zasadzie przeniesienia (move)

Jak wiemy, lista wychwytywania pozwala wyrażeniu lambda uzyskać dostęp do obiektów przez wartość, czyli przez kopię (albo przez sporządzenie referencji do danego obiektu). A co będzie, jeśli interesującego nas obiektu nie da się kopiować, wolno go tylko przenieść?

*To znaczy, co będzie, gdy informację zawartą w tym obiekcie można jedynie wyjąć z jednego obiektu i włożyć do drugiego? Nie może ona być obecna w obu obiektach równocześnie.*

Wyrażenie lambda nie może takiego obiektu wychwycić przez wartość (kopię).

Przykładem takich niekopiowalnych obiektów są „sprytnie wskaźniki” realizowane przez biblioteczną klasę `std::unique_ptr`. (*Opus*, §27.7.1). Jeśli w programie posługujemy się nimi, to jak sprawić, żeby nasze wyrażenie lambda było zdolne wychwycić taki wskaźnik? W C++11 takiego obiektu wychwycić się nie dało. C++14 już nam to umożliwi. O tym teraz porozmawiamy.

Sprawę rozwiązuje bardzo prosto poznana w poprzednim paragrafie *inicjalizacja na liście wychwytywania*. Zobaczmy to na przykładzie. Będą w nim dwa wyrażenia

lambda. Jedno wychwyci sprytny wskaźnik do pojedynczego obiektu, a drugie – do tablicy obiektów.



```

#include <iostream>
#include <memory>
using namespace std;
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// ❶
struct Tklasa
{
    int skladnik = 0;
    ~Tklasa() {
        cout << "Destruktor ~Tklasa" << endl; ❷
    }
};
//*****
int main()
{
    cout << "Definicja sprytnego wskaźnika do pojedynczego obiektu\n";
    unique_ptr<Tklasa>  spryt { new Tklasa }; ❸

    { // lokalny zakres w celach szkoleniowych ----- ❹

        auto lambda1 = [sw = std::move(spryt) ] () ❺
        {
            sw->skladnik = 4;
            cout << "Wykonuje sie lambda1" << endl; ❻
        }; ❻

        if(!spryt) ❼
            cout << "A teraz obiekt spryt NIE ma juz prawa wlasnosci " << endl; ❼

        cout << "Przed wykonaniem lambdy" << endl;
        lambda1(); // 1. wykonanie lambdy ❸
        lambda1(); // 2. wykonanie lambdy ❸

        cout << "--- Teraz nastapi koniec zakresu istnienia lambdy1\n";
    } // koniec lokalnego zakresu ----- ❹
    cout << "--- Skonczył sie zakres istnienia lambdy1\n";

    cout << "PROBY Z TABLICA" << endl;

    // sprytny wskaźnik do tablicy 5-elementowej (z rezerwacją tej tablicy)
    constexpr int rozmiar = 5;
    unique_ptr<Tklasa[]>  wsktab { new Tklasa[rozmiar] }; ❿

    { // drugi lokalny zakres (w celach szkoleniowych) ===== ❶

        auto lamTab = [stab = std::move(wsktab), rozmiar] () ❶
        {
            for(int i = 0 ; i < rozmiar ; ++i){
                stab[i].skladnik = i;
            }
            cout << "Pracuje lamTab\n";
        }
    }
}

```

```

};
cout << "Przed wywołaniem lamTab\n";
lamTab();
lamTab();
cout << "=== Zakonczył sie obszar istnienia lamTab" << endl;
} // koniec lokalnego zakresu =====
cout << "=== Koniec funkcji main" << endl;
}

```



### Na ekranie pojawi się tekst:

```

Definicja sprytnego wskaźnika do pojedynczego obiektu
A teraz obiekt spryt NIE ma już prawa własności
Przed wykonaniem lambdy
Wykonuje się lambda1
Wykonuje się lambda1
--- Teraz nastąpi koniec zakresu istnienia lambdy1
Destruktor ~Tklasa
--- Skonczył się zakres istnienia lambdy1
PROBY Z TABLICA
Przed wywołaniem lamTab
Pracuje lamTab
Pracuje lamTab
=== Zakonczył się obszar istnienia lamTab
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
Destruktor ~Tklasa
=== Koniec funkcji main

```



### Omówienie sposobów wychwytywania na zasadzie przenoszenia

O bibliotecznej klasie „sprytnych wskaźników” `unique_ptr` rozmawialiśmy w *Opusie* (§27.7.1). Służy ona do tworzenia obiektów zachowujących się podobnie jak wskaźniki. W sprytnym wskaźniku możesz przechować adres jakiegoś obiektu, który wytworzyłeś w zapasie pamięci operatorem `new`. Sprytny wskaźnik ma tę dodatkową zdolność, że gdy kończy się czas jego życia, to – że tak powiem – „pociąga on za sobą do grobu” obiekt, którego adres przechowywał. Jak sprytny wskaźnik to robi? Po prostu w jego destruktorze użyta jest instrukcja `delete` kasująca obiekt, którym się do tej pory opiekował.

Jak wiemy, może istnieć wiele wskaźników wskazujących na ten sam obiekt. W przypadku sprytnych wskaźników mogłoby to być groźne. Wyobraź sobie, że na jeden obiekt stworzony w zapasie pamięci operatorem `new` wskazuje kilka sprytnych wskaźników. Gdy jednemu z tych sprytnych wskaźników skończy się czas życia, jego destruktor zlikwiduje cenny obiekt w zapasie pamięci. Jeśli pozostałe wskaźniki nadal będą chciały z tym „cennym” obiektem pracować – wywoła to katastrofę.

Aby tej katastrofy uniknąć, sprytnie wskaźniki `unique_ptr` stosują zasadę, że tylko jeden z nich może w danej chwili znać adres naszego obiektu z zapasu pamięci. Jeśli zrobimy przypisanie takiego wskaźnika do jakiegoś innego:

```
spryciarz_nowy = spryciarz_stary;
```



odbędzie się to nie na zasadzie *skopiowania* cennego adresu, ale na zasadzie *przeniesienia* go. Innymi słowy, `spryciarz_stary` przekaże cenny adres `spryciarzowi_nowemu`, a sam o tym adresie zapomni.

Niby to genialne, ale w przypadku, gdybyśmy chcieli pracować z takim obiektem w ramach wyrażenia lambda, to jak zorganizować jego wychwycenie? Przecież nie można tego zrobić na zasadzie wychwycenia przez wartość (czyli przez kopię). Zatem jak taki sprytny wskaźnik może zostać wychwycony przez wyrażenie lambda?

### Jak wychwycić: „sprytny wskaźnik opiekujący się pojedynczym obiektem”

❸ `unique_ptr<Tklasa> spryt { new Tklasa ;`

To jest definicja sprytnego wskaźnika o nazwie `spryt`. Jest on typu `unique_ptr<Tklasa>`, czyli może się on opiekować adresem obiektu typu `Tklasa`. (To taka pomocnicza klasa zdefiniowana na górze programu ❶). W nawiasie klamrowym widzimy, że nasz sprytny wskaźnik jest od razu inicjalizowany. Czym? Adresem obiektu typu `Tklasa`, który wytwarzamy operatorem `new` w zapasie pamięci.

❷ i ❸ W celach szkoleniowych definiujemy sobie w programie pewien lokalny zakres. Oczywiście w Twoim programie nie musisz tego robić. Mnie jest on potrzebny, by pokazać Ci, kiedy dokładnie nastąpi „pociągnięcie do grobu”.

❹ Oto definicja prostego wyrażenia lambda. Jego ciało jest rozpisane na kilka linijek.

*Dla uproszczenia naszej rozmowy ta definicja nie jest zapakowana do wywołania jakiejś funkcji-algorytmu, ale jest umieszczona w obiekcie lambda o nazwie lambda1. (Podobnie robiliśmy w poprzednich przykładach, aby sobie dodatkowo nie zaprzętałyśmy algorytmami).*



Zwróć uwagę na listę wychwytywania. Tu jest kwintesencja tego programu.

```
auto lambda1 = [sw = std::move(spryt)] () // itd.
```

Definiujemy tutaj jakiś obiekt o nazwie `sw` i inicjalizujemy go wartością wyrażenia `move(spryt)`. Innymi słowy, sprawiamy tutaj, że adres przechowywany w obiekcie `spryt` zostaje przeniesiony do obiektu `sw`.

*O funkcji `move` rozmawialiśmy w Opusie w §22.10.*

Pytanko: jakiego typu jest obiekt `sw`? Odpowiedź: takiego, jakby przed nim stało słowo zastępcze `auto`.

```
auto sw = std::move(spryt); // czyli typu: unique_ptr<Tklasa>
```

W ten sposób sprawiliśmy, że od tej pory obiektem wytworzonym operatorem `new` (w instrukcji ❸) opiekuje się wyłącznie sprytny wskaźnik `sw`.

W ciele wyrażenia `lambda1` stawiamy jakieś przykładowe instrukcje korzystające z obiektu wskazywanego przez `sw`, następnie stawiamy klamrę `}` kończąca definicję tego ciała. Potem ❹ następuje średnik, bo to przecież koniec długiej instrukcji, która rozpoczęła się w ❸.

*Ta druga instrukcja:*

1) zdefiniowała wyrażenie lambda i...

2) zapamiętała je w obiekcie lambda o nazwie lambda1.

❺ Jesteśmy znowu w zakresie funkcji `main`. Dla ciekawości sprawdzamy teraz sprytny wskaźnik o nazwie `spryt`. Jeśli teraz znajduje się w nim już `nullptr`, to znaczy, że naprawdę oddał on swój cenny skarb sprytnemu wskaźnikowi `sw`. Na ekranie widzimy ❺, że tak rzeczywiście się stało.

- 8 Oto uruchomienie naszego wyrażenia lambda. Robię to dwukrotnie, abyś zobaczył, że sprytny wskaźnik `sw` istnieje cały czas, dopóki istnieje obiekt `lambda1`. Nie znika więc po wykonaniu funkcji lambda. Przecież ktoś mógłby wywołać `lambda1` po raz kolejny.
- 9 Nadchodzi koniec naszego „szkoleniowego” lokalnego zakresu, w którym zdefiniowaliśmy nasz obiekt `lambda1`. Tutaj więc obiekt `lambda1` przestaje istnieć. Skoro tak, to przestanie również istnieć jego (lokalny) sprytny wskaźnik `sw`. W chwili swojej „śmierci” tenże biblioteczny sprytny wskaźnik `sw` wywołuje swój destruktor, w którym (uwierz mi na słowo) wykonywana jest instrukcja `delete` kasująca nasz obiekt `Tklasa` z zapasu pamięci. To właśnie moment „pociągnięcia do grobu”. Czy to działa? Działa, potwierdza to gadający destruktor 2, którego wypis pojawia się na ekranie 9.

### Jak wychwycić: „sprytny wskaźnik opiekujący się tablicą obiektów”?

Ponieważ w zapasie pamięci najczęściej rezerwuje się nie tyle pojedyncze obiekty, co tablice takich obiektów, zobaczmy i taką sytuację.

- 10 Oto definicja sprytnego wskaźnika mogącego wskazywać na tablicę obiektów typu `Tklasa`.

```
unique_ptr<Tklasa[]> wsktab { new Tklasa[rozmiar] };
```

Jest on typu `unique_ptr<Tklasa[]>`. Ten nawias kwadratowy sprawia, że sprytny wskaźnik dowiaduje się, że to, czym będzie się opiekować, jest tablicą, zatem w razie likwidacji ma posłużyć się operatorem `delete []` (a nie prostym `delete`).

Definiowany tutaj sprytny wskaźnik (o nazwie `wsktab`) jest od razu inicjalizowany. Co robi ta inicjalizacja? Wytwarza w zapasie pamięci (operatorem `new[]`) pięcioelementową tablicę obiektów typu `Tklasa` i jej adres przekazuje sprytnemu wskaźnikowi. Od tej pory `wsktab` wie, którą tablicą ma się opiekować.

- 11 Znowu w celach wyłącznie ilustracyjnych definiujemy lokalny zakres. Oczywiście w swoim programie nie musisz tego robić.
- 12 To jest definicja drugiego wyrażenia lambda w naszym przykładzie. Jej ciało jest rozpisane na kilka linijek. Po nim następuje średnik 13 kończący tę długą instrukcję definicji 12. Jak widzisz, to wyrażenie lambda inicjalizuje obiekt `lambda` o nazwie `lamTab`.

Dla nas najważniejsza jest tutaj lista wychwytywania. Oto jej pierwszy człon:

```
stab = std::move(wsktab) // czyli jakby: auto stab = std::move(wsktab)
```

Jest to definicja lokalnego (dla lambda) obiektu o nazwie `stab`. Jakiego jest on typu? Takiego jak inicjalizujące go wyrażenie. Wyrażeniem inicjalizującym jest wywołanie bibliotecznej funkcji `move`. Sprawia ona, że cenny adres, przechowywany w sprytnym wskaźniku `wsktab`, zostanie stamtąd wyjęty i włożony do sprytnego wskaźnika `stab`. (Od tej pory to wskaźnik `stab` odpowiada za ewentualne „pociągnięcie do grobu” tablicy, natomiast `wsktab` zostaje z tego obowiązku zwolniony).

Aby móc w wyrażeniu lambda pracować z tą tablicą, powinniśmy znać jej rozmiar. Załatwia to drugi człon listy wychwytywania (12). Jak widać, ten rozmiar wychwytywamy po prostu przez wartość.

- 14 Oto dwukrotne wywołanie funkcji lambda `lamTab`. Robię to dwukrotnie, abyś nie pomyślał, że to zakończenie wykonywania `lamTab` spowoduje likwidację tablicy. Nie! Przecież tablica może być potrzebna w sytuacji, gdybyśmy nasze wyrażenie lambda chcieli wywołać po raz trzeci.

Likwidacja tablicy nastąpi dopiero wtedy, gdy będzie „ginał” sam obiekt lambda lamTab. A kiedy się to stanie? Już za chwilę. To właśnie w tym celu w programie utworzyłem lokalny zakres 11 ↔ 15.

- 15 Ponieważ obiekt lambda lamTab jest zdefiniowany w lokalnym zakresie, więc teraz, gdy ten zakres się kończy, obiekt lambda zostaje zlikwidowany. Na ekranie możemy zobaczyć teksty potwierdzające, że sprytny wskaźnik (będący częścią wyrażenia lambda) na chwilę przed swą „śmiercią” wywołał operator delete[] likwidujący tablicę. Dowiadujemy się o tym znowu dzięki gadającemu destruktorowi elementów tablicy. To on wypisał na ekranie stosowne informacje 15.

## Podsumujmy



W tym paragrafie dowiedzieliśmy się, że wprowadzenie w C++14 nowości, jaką jest możliwość definiowania na liście wychwytywania nowych lokalnych obiektów (wraz z ich inicjalizacją), otworzyło dodatkowe drzwi. Można teraz wychwytywać obiekty na zasadzie ich przenoszenia (a nie tylko na zasadzie kopiowania).

Zwykle wystarcza nam kopiowanie. Bywają jednak obiekty, których informacji nie można kopiować; można ją tylko przenosić. Takie są na przykład sprytnie wskaźniki unique\_ptr. Tutaj zobaczyliśmy, jak praktycznie zrealizować takie wychwytywanie metodą przenoszenia.



# Skorowidz

## !

`__PRETTY_FUNCTION__` 8  
`__has_include` 201-202

## A

agregat  
 > może być klasą pochodną (C++17) 247-249  
`align_val_t` 136  
`alignas`  
 > a operatory `new` i `delete` 134-143  
 > w przeładowanych `new` i `delete` dla klasy 141  
`aligned_alloc (std::)` - funkcja bibliot. 140, 144  
`apostrof`  
 > jako separator cyfr 44  
`argument`  
 > funkcji  
 • deprecated 114  
`array` - klasa (std::) 184  
`atrybut` 111-118  
 > `carries_dependency` 111  
 > deprecated 112  
 > `fallthrough` 203-212  
 > `maybe_unused` 203-212  
 > `nodiscard` 203-212  
 > `noreturn` 111  
`auto`  
 > rozpoznaje typ rezultatu funkcji 50-51  
 > rozpoznawanie w inicjalizacji klamrowej 245-246  
 > w argumencie wyrażenia `lambda` 96  
 > w deklaracji `template auto` 161-165

## B

biblioteka  
 > szablonów 4  
 binarny zapis stałych dosłownych 43  
`bitset (std::)` - klasa biblioteczna 46, 217  
`byte (std::)` - klasa biblioteczna 223  
`byte (std::)` - klasa biblioteczna 213-221

## C

C++14 - nowe cechy języka 43-122  
 C++17 - nowe cechy języka 123-263  
`carries_dependency` - atrybut 111  
`class`  
 > deprecated 116  
`constexpr`  
 > a szablon zmiennej 61  
 > funkcja składowa w C++14 110  
 > funkcja w C++11 104  
 > funkcja w C++14 104-110

## D

`dana składowa`  
 > deprecated 114  
`decltype`  
 > użyte w parametrze szablonu 161  
`decltype(auto)`  
 > deklaracja typu rezultatu 52-58  
 > nie używać w alternatywnej deklaracji funkcji 55  
 > w szablonie funkcji 55  
`declval (std::)` 197  
`defaultfloat` manipulator 150  
`deklaracja`  
 > dostępu 28  
 > dowiązań  
 • funkcja `get` 193, 195  
 • klasa `std::tuple_element` 193, 196  
 • przystosowanie klasy cudzej 198  
 • przystosowanie klasy swojej 192  
 • `std::tuple_size` 193, 196  
`delete`  
 > a `alignas` 134-143  
 > globalny, przeładowanie 86-93  
 • a `alignas` 135  
 > z `alignas`  
 • przeładowanie w klasie 141  
`deprecated`  
 > argument funkcji 114  
 > atrybut 112  
 > `class`, `enum` 116

- > funkcja 113
- > funkcja składowa i dana składowa 114
- > obiekt 115
- > specjalizacji szablonu 118
- > typedef, using 118
- > z zbiorczą definicją zmiennych 116
- dostęp
  - > wybiórczo 28
- dostępu
  - > deklaracja 28
- dowiązanie strukturalne 182-200
  - > a const i volatile 183
  - > do elementów klasy std::array 184
  - > do obiektów klas i struktur 185
  - > gdzie stosować 191
  - > jako referencja l-wartości lub r-wartości 190
  - > jako referencji do l-wartości 183
  - > jako referencji do r-wartości 183
- dwójkowe liczby wczytane strumieniem 48
- dwójkowy zapis
  - > a klasa bitset 46
  - > stałych dosłownych 43

## E

- enum
- > deprecated 116

## F

- fallthrough - atrybut 208
- false\_type (std::)
- > pomocniczy typ biblioteczny 80
- free()
- > fun. bibl. do zwalniania pamięci 87
- funkcja
- > a noexcept 123-127
  - > auto rozpoznanie typu rezultatu 50-51
  - > constexpr
    - w C++11 104
    - w C++14 104-110
  - > constexpr w C++14 104
  - > decltype(auto) jako typ rezultatu 52-58
  - > deprecated 113
  - > składowa
    - constexpr w C++14 110
    - deprecated 114
  - > specyfikacja wyjątków 123-127

## G

- gl-wartość 134

## H

- harmonijka 152-160
- > składana w lewo 154
  - > składana w prawo 154
  - > z wyrażeniem początkowym 155
- has\_include (\_\_has\_include) 201-202
- hexfloat, manipulator 149

## I

- if - instrukcja sterująca
- > z wyrażeniem inicjalizującym 179-181
- if constexpr 16, 173-178
- > w szablonie 16
- inicjalizacja
- > agregatowa 247
  - > bezpośrednia 245
    - zmiana w standardzie C++17 246
  - > kopiująca 245
  - > w instrukcjach if oraz switch 179-181
  - > zbiorcza 247
- inicjalizator
- > klamrowy
    - auto rozpoznanie jego typu 245-246
- is\_const 174
- is\_integral 174
- is\_pointer 174
- is\_unsigned 174

## K

- kompilacja warunkowa
- > a instrukcja if constexpr 173-178
- kontekst
- > rozwnięcia pakietu 31-33
  - > wyrażenia harmonijkowego 157, 159
- konwersja
- > chwilowo materializująca 134

## L

- lambda (wyrażenie)
- > uogólnione 94
  - > wychwycenie obiektu na zasadzie przeniesienia (move) 99
- lista wychwytywania
- > na niej definicje obiektów lokalnych 96

## M

- makrodefinicja
- > \_\_PRETTY\_FUNCTION\_\_ 8
- malloc (std::) 87



manipulator  
 > hexfloat 149  
 maybe\_unused - atrybut 205  
 move 102

## N

new  
 > a alignas 134-143  
 > globalny, przeładowanie 86-93
 

- a alignas 135

 > z alignas
 

- przeładowanie w klasie 141

 niby-rekurencja 14  
 nodiscard - atrybut 210  
 noexcept  
 > w typie funkcji 123-127  
 noreturn - atrybut 111  
 nothrow\_t 136

## O

obiekt  
 > deprecated 115  
 operator  
 > decltype
 

- użyty w parametrze szablonu 161

 > delete
 

- globalny, przeładowanie w C++14 86-93

 > sizeof... 8

## P

pakiet argumentów sz. funkcji 6
 

- > a wskaźniki z przydomkami const 31
- > a wyrażenia harmonijkowe 152-160
- > jak dobrać się do argumentów 11-17
- > odbierane arg. przez wart., ref., adres 30
- > rozwinięta postać 7, 152
- > z referencjami l-wartości 30
- > z referencjami obiektów stałych 31
- > z referencjami r-wartości 30
- > ze wskaźnikami 31

 pakiet argumentów szablonu  
 > rozwinięcie z kontekstem 31-33  
 pakiet parametrów szablonu 6
 

- > będących nazwami innych szablonów 24
- > będących typami 24
- > będących wartościami 24-25
- > trzy rodzaje 24-29

 pakietu rozwinięcie  
 > gdzie może wystąpić 39  
 > w nazwie innego szablonu 36-38  
 > z operatorem przecinek 36

parametr  
 > szablonu określający inny szablon 229-232  
 podwalina typu wyliczeniowego 214  
 pomijanie kopiowania 128-133  
 porządek obliczania składników wyrażen 144-145  
 postfix wyrażenia 145  
 pr-wartość 134  
 PRETTY\_FUNCTION 8  
 printf 87  
 przecinek (operator)  
 > a rozwinięcie pakietu 35  
 przestrzeń nazw  
 > a zagnieżdżenie
 

- nowy zapis w C++17 225-228

 przeładowanie  
 > globalnych new, delete 86-93  
 > operatora
 

- delete globalnego w C++14 86-93

 puts 87

## R

rekurencja 11, 14, 52  
 > niby-rekurencja 14  
 rozwinięcie pakietu  
 > argumentów 7  
 > dwukrotne 33  
 > gdzie może wystąpić? 39  
 > na liście inicjalizacyjnej konstruktora 23  
 > na liście parametrów aktualnych innego szablonu 36-38  
 > na liście pochodzenia klasy 22  
 > równoważne 33  
 > typów w { } 34-35  
 > w deklaracji using 239-244  
 > z kontekstem 31-33  
 > zapełnione 33  
 rozwinięta postać  
 > argumentów szablonu 10  
 > pakietu 13, 152

## S

separatory cyfr w stałych dosłownych 44-49, 148  
 silnia 107  
 sizeof... 8  
 specjalizacja  
 > deprecated 118  
 > szablonu o zmiennej liczbie parametrow 23  
 > szablonu zmiennej 63  
 specyfikacja wyjątków  
 > w C++17 123-127  
 sprytny wskaźnik  
 > unique\_ptr 99

static\_assert  
 > modyfikacja w C++17 224  
 stała  
 > dosłowna  
 • z separatorami cyfr 44-49  
 • zapisana dwójkowo 43  
 > znakowa typu u8 146  
 std::array 184  
 std::false\_type - pomocniczy typ 80  
 std::strtod 151  
 std::true\_type - pomocniczy typ 80  
 std::unique\_ptr 99  
 stdio 87  
 strtod - funkcja biblioteczna 151  
 strumienie a separatory cyfr 45  
 switch  
 > z wyrażeniem inicjalizującym 179-181  
 szablon definicji zmiennej 59-85  
 szablon funkcji  
 > deprecated 119  
 > zmienna liczba parametrów 4-10  
 szablon klas  
 > deprecated 118  
 > o zmiennej liczbie parametrów 18-23  
 szablon zmiennej 59-85  
 > a constexpr i const 61  
 > przykład zastosowania 72  
 > specjalizacja 63  
 szablony  
 > o zmiennej liczbie parametrów 3-42  
 szesnastkowy  
 > zapis liczb zmiennoprzecinkowych 147-151  
 • wypisyw. i wyczyt. ich strumieniami 149

## T

Taylora wzór 108  
 template auto 161-165  
 to\_integer (std::) - funkcja biblioteczna 214, 217  
 to\_ulong (std::) - funkcja biblioteczna 50  
 true\_type (std::)  
 > pomocniczy typ biblioteczny 80  
 trygonometryczne tablice 109  
 tuple\_element (std::) 193, 196  
 tuple\_size (std::) 193, 196  
 typ  
 > deprecated 116  
 > wyliczeniowy enum  
 • jego inicjalizacja liczbą 223  
 typedef  
 > deprecated 118  
 typename 229-232

## U

uncaught\_exception (std::) zmiana typu rezultatu (C++17) 250-253  
 uogólnione  
 > wyrażenie lambda 94  
 using  
 > deklaracja  
 • z rozwinięciem pakietu klas podst. 239-244  
 > deprecated 118

## V

valgrind - program diagnostyczny 86

## W

wielodziedziczenie 239  
 wielokropek  
 > w deklaracji argumentów fun. 6  
 > w ostrym nawiasie 6  
 > w wyrażeniu rozwijającym 32  
 wybiórcze udostępnianie 28  
 wychwycenie obiektu na zasadzie przeniesienia (move) 99  
 wyrażenia  
 > końcówkowe 145  
 > postfix 145  
 > złożone  
 • porządek ich obliczania 144-145  
 wyrażenia harmonijkowe 152-160  
 > cztery formy 157  
 > tzw. kontekst 159  
 wyrażenie  
 > incjalizujące  
 • w instrukcjach if oraz switch 179-181  
 > lambda  
 • uogólnione 94

## X

x-wartość 134

## Z

zagnieżdżona  
 > przestrzeń nazw - nowy zapis C++17 225-228  
 zakresowe for  
 > a funkcje begin, end - wg zasad C++17 233-238  
 > jak w C++11, adaptować klasę 233  
 > jak w C++17, adaptować klasę 234  
 zapis dwójkowy  
 > stałych dosłownych 43  
 znakowa stała u8 146

# PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA  
**Helion** 

# Misja w nadprzestrzeń C++14/17

## W tej książce:

- Najważniejsze informacje o nowych możliwościach języka C++
- Praktyczne przykłady zastosowania konstrukcji
- Ćwiczenia utrwalające zdobytą wiedzę

## C++ – mierz wysoko!

C++ to jeden z najpopularniejszych i najpotężniejszych języków programowania. Stanowi punkt wyjścia dla wielu innych języków, które odziedziczyły po nim składnię i liczne możliwości, dzięki czemu można śmiało stwierdzić, że znajomość C++ otwiera drzwi do świata nowoczesnego programowania i jest podstawą na wymagającym rynku pracy w branży informatycznej. Czasy się zmieniają, lecz to C++ jest wciąż wybierany wszędzie tam, gdzie liczą się możliwości, elastyczność, wydajność i stabilność.

Książka, którą trzymasz w rękach, to kontynuacja genialnego kompendium *Opus magnum C++11. Programowanie w języku C++*. Autor, wybitny specjalista z ogromnym doświadczeniem w międzynarodowych projektach i twórca niezwykle popularnego podręcznika *Symfonia C++*, postanowił uzupełnić swoje dzieło o zagadnienia, dla których zabrakło miejsca w poprzednich tomach. Jeśli chcesz poszerzyć wiedzę na temat szablonów oraz poznać możliwości najnowszych standardów języka C++, nie mogłeś lepiej trafić!

## Rusz w kolejną misję z C++ na pokładzie!

	<b>KOD KORZYŚCI</b> Sięgnij po więcej! ▶	
 <a href="http://helion.pl">helion.pl</a>	ISBN 978-83-8322-582-1	
 <b>HELION SA</b> ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	 9 788383 225821	
<b>Cena: 69,00 zł</b>		