

---

# C++ Optymalizacja kodu

*Kurt Guntheroth*

*przekład: Natalia Chounlamany*

APN Promise  
Warszawa 2016

**O'REILLY®**

---

# Spis treści

<i>Przedmowa</i> .....	xv
1 Wprowadzenie do optymalizacji.....	1
Optymalizacja to część procesu rozwoju oprogramowania.....	2
Optymalizacja jest efektywna.....	3
Optymalizacja jest OK.....	4
Nanosekunda tu, nanosekunda tam.....	6
Podsumowanie strategii optymalizacji kodu C++.....	7
Użyj lepszego kompilatora, lepiej używaj kompilatora.....	7
Użyj lepszych algorytmów.....	9
Użyj lepszych bibliotek.....	10
Zredukuj alokację pamięci i kopiowanie.....	11
Usuń obliczenia.....	12
Użyj lepszych struktur danych.....	12
Zwiększ równoległość.....	13
Zoptymalizuj zarządzanie pamięcią.....	13
Podsumowanie.....	13
2 Wpływ działania komputera na optymalizację.....	15
Nieprawdziwe przekonania języka C++ o komputerach.....	16
Prawda o komputerach.....	17
Pamięć jest powolna.....	17
Dostęp do pamięci nie zamyka się w bajtach.....	18
Nie wszystkie operacje dostępu do pamięci są równie wolne.....	19
Słowa mają najbardziej i najmniej znaczący koniec.....	20
Pamięć ma ograniczoną pojemność.....	21
Wykonanie instrukcji zabiera dużo czasu.....	21
Komputery mają trudności z podejmowaniem decyzji.....	22
Istnieje wiele strumieni wykonania programu.....	22
Wywoływanie systemu operacyjnego jest kosztowne.....	24
C++ również kłamie.....	24
Różne instrukcje mają różny koszt.....	25
Instrukcje nie są wykonywane kolejno.....	25
Podsumowanie.....	26

3	Mierzenie wydajności .....	27
	Mentalność optymalizatora .....	28
	Wydajność musi być mierzona .....	28
	Optymalizatorzy polują na grubą zwierzynę .....	29
	Reguła 90/10 .....	29
	Prawo Amdahla .....	31
	Przeprowadzanie eksperymentów .....	32
	Prowadź notatnik laboratoryjny .....	34
	Mierzenie bazowej wydajności i wyznaczanie celów .....	35
	Można poprawić tylko to, co zostało zmierzone .....	37
	Działanie programu profilującego .....	37
	Pomiary czasowe długotrwałych zadań .....	40
	„Odrobina wiedzy” o mierzeniu czasu .....	41
	Mierzenie czasu przy użyciu komputerów .....	46
	Pokonywanie trudności w mierzeniu .....	55
	Tworzenie klasy stopera .....	58
	Mierzenie czasu aktywnych funkcji w warunkach testowych .....	63
	Szacowanie kosztu kodu w celu znalezienia aktywnego kodu .....	64
	Szacowanie kosztu pojedynczych instrukcji C++ .....	64
	Szacowanie kosztu pętli .....	65
	Inne sposoby odnajdowania aktywnych punktów .....	67
	Podsumowanie .....	68
4	Optymalizowanie użycia ciągów: przykład .....	69
	Dlaczego ciągi są tak problematyczne .....	69
	Ciągi są dynamicznie alokowane .....	70
	Ciągi to wartości .....	71
	Ciągi wymagają wiele kopiowania .....	71
	Pierwsze podejście do optymalizacji ciągów .....	73
	Użyj operacji mutujących do eliminowania wartości tymczasowych .....	74
	Redukuj realokację poprzez rezerwację obszaru .....	75
	Eliminuj kopiowanie argumentów std::string .....	75
	Eliminuj wyłuskania wskaźników przy użyciu iteratorów .....	77
	Eliminuj kopiowanie wartości zwrotnych .....	78
	Użyj tablic znaków zamiast ciągów .....	79
	Podsumowanie pierwszego podejścia do optymalizacji .....	81
	Drugie podejście do optymalizacji ciągów .....	81
	Użyj lepszego algorytmu .....	81
	Użyj lepszego kompilatora .....	84
	Użyj lepszej biblioteki ciągów .....	84
	Użyj lepszego alokatora .....	88

Eliminowanie konwersji ciągów .....	90
Konwersja z ciągu C do std::string. ....	90
Konwersja między kodowaniami znaków .....	91
Podsumowanie.....	91
<b>5 Optymalizowanie algorytmów .....</b>	<b>93</b>
Koszt czasowy algorytmów .....	95
Koszt czasowy w najlepszym, średnim i najgorszym przypadku .....	97
Amortyzowany koszt czasowy .....	98
Inne koszty .....	98
Zestaw narzędzi do optymalizacji wyszukiwania i sortowania.....	98
Efektywne algorytmy wyszukiwania. ....	99
Koszt czasowy algorytmów wyszukiwania.....	99
Wszystkie wyszukiwania są równie dobre dla małych n .....	100
Efektywne algorytmy sortowania .....	101
Koszt czasowy algorytmów sortowania .....	102
Zastąpienie algorytmów sortowania o niewydajnym najgorszym przypadku.....	102
Eksplotowanie znanych właściwości danych wejściowych. ....	103
Wzorce optymalizacji .....	103
Wstępne obliczanie .....	104
Opóźnione obliczanie .....	105
Przetwarzanie wsadowe.....	106
Buforowanie .....	106
Specjalizacja .....	107
Pobieranie większych porcji .....	107
Wskazówki .....	108
Optymalizowanie oczekiwanej ścieżki .....	108
Mieszanie.....	108
Podwójne sprawdzanie .....	109
Podsumowanie.....	109
<b>6 Optymalizacja zmiennych dynamicznych.....</b>	<b>111</b>
Powtórzenie wiadomości o zmiennych C++.....	112
Czas magazynowania.....	112
Własność zmiennych.....	115
Obiekty wartości i obiekty encji.....	116
Powtórzenie wiadomości o API zmiennych dynamicznych w C++.....	117
Sprytnie wskaźniki automatyzują zarządzanie własnością zmiennych dynamicznych .....	120
Dynamiczne zmienne wpływają na koszt czasowy wykonania.....	123

Redukowanie użycia zmiennych dynamicznych. . . . .	124
Twórz instancje klas statycznie . . . . .	124
Używaj statycznych struktur danych. . . . .	125
Użyj <code>std::make_shared</code> zamiast <code>new</code> . . . . .	129
Nie dziel się własnością niepotrzebnie . . . . .	130
Użyj „głównego wskaźnika” jako właściciela zmiennych dynamicznych	131
Redukowanie realokacji zmiennych dynamicznych. . . . .	132
Wstępnie alokuj zmienne dynamiczne, aby zapobiec realokacji. . . . .	132
Twórz zmienne dynamiczne poza pętlą . . . . .	133
Eliminowanie niepotrzebnego kopiowania . . . . .	134
Wyłącz nieumyślne kopiowanie w definicji klasy . . . . .	135
Wyliminuj kopiowanie podczas wywoływania funkcji. . . . .	136
Wyliminuj kopiowanie podczas powrotów z funkcji . . . . .	138
Biblioteki bez kopiowania. . . . .	140
Implementuj idiom „kopiowanie przy zapisie” . . . . .	141
Stosuj wycinki struktur danych . . . . .	142
Implementowanie semantyki przenoszenia. . . . .	143
Niestandardowa semantyka kopiowania: desperackie rozwiązanie . . . . .	143
<code>std::swap()</code> : semantyka przenoszenia dla ubogich. . . . .	144
Współwłasność encji . . . . .	145
Przenoszone części semantyki przenoszenia . . . . .	146
Uaktualnij kod w celu użycia semantyki przenoszenia. . . . .	148
Subtelności semantyki przenoszenia. . . . .	148
Spłaszczanie struktur danych. . . . .	151
Podsumowanie. . . . .	152
<b>7 Optymalizowanie aktywnych instrukcji. . . . .</b>	<b>153</b>
Usuwanie kodu z pętli. . . . .	154
Buforuj wartość końcową pętli. . . . .	155
Użyj bardziej efektywnych instrukcji pętli . . . . .	155
Odliczaj w dół zamiast w górę . . . . .	156
Usuń z pętli niezależny kod . . . . .	157
Usuń z pętli niepotrzebne wywołania funkcji . . . . .	158
Usuń z pętli ukryte wywołania funkcji . . . . .	161
Usuń z pętli kosztowne, wolno zmieniające się wywołania . . . . .	163
Przesuń pętle do funkcji, aby zredukować dodatkowy koszt wywołań . . . . .	164
Rzadziej wykonuj niektóre akcje . . . . .	165
A co z całą resztą?. . . . .	166
Usuwanie kodu z funkcji . . . . .	167
Koszt wywołań funkcji . . . . .	167
Deklaruj krótkie funkcje jako <code>inline</code> . . . . .	171

Definiuj funkcje przed pierwszym użyciem. . . . .	172
Eliminuj niepotrzebny polimorfizm . . . . .	172
Usuń nieużywane interfejsy . . . . .	173
Wybieraj implementację w czasie kompilacji przy użyciu szablonów . . .	177
Eliminuj zastosowania idiomu PIMPL . . . . .	178
Eliminuj wywołania bibliotek DLL . . . . .	180
Użyj statycznych funkcji składowych zamiast funkcji składowych. . . . .	180
Przenieś wirtualny destruktor do klasy podstawowej. . . . .	181
Optymalizowanie wyrażeń. . . . .	182
Uprość wyrażenia. . . . .	182
Grupowanie stałych . . . . .	184
Użyj mniej kosztownych operatorów . . . . .	184
Używaj arytmetyki liczb całkowitych zamiast arytmetyki liczb zmiennoprzecinkowych . . . . .	185
Typ double może być szybszy niż float . . . . .	187
Zastąp obliczenia iteracyjne wzorami . . . . .	187
Optymalizacja idiomów przepływu sterowania . . . . .	189
Użyj switch zamiast if – else if – else . . . . .	189
Użyj funkcji wirtualnych zamiast switch lub if . . . . .	190
Korzystaj z bezkosztowej obsługi wyjątków . . . . .	191
Podsumowanie. . . . .	193
<b>8 Zastosowanie lepszych bibliotek. . . . .</b>	<b>195</b>
Optymalizacja użycia biblioteki standardowej . . . . .	195
Filozofia standardowej biblioteki C++ . . . . .	196
Problemy ze stosowaniem standardowej biblioteki C++ . . . . .	197
Optymalizowanie istniejących bibliotek. . . . .	199
Zmieniaj tak mało, jak to tylko możliwe. . . . .	200
Dodawaj funkcje zamiast zmieniać funkcjonalność. . . . .	200
Projektowanie zoptymalizowanych bibliotek . . . . .	201
Koduj szybko, ubolewaj w czasie wolnym . . . . .	201
W projektowaniu bibliotek ascetyzm to zaleta. . . . .	202
Podejmuj decyzje o alokacji pamięci poza biblioteką. . . . .	203
Programuj szybkie biblioteki . . . . .	203
Łatwiej jest optymalizować funkcje niż framework . . . . .	204
Spłaszcz hierarchie dziedziczenia . . . . .	204
Spłaszcz łańcuchy wywołań . . . . .	205
Spłaszcz wielowarstwowe projekty . . . . .	205
Unikaj dynamicznego wyszukiwania . . . . .	206
Wystrzegaj się „wszechmocnych funkcji” . . . . .	207
Podsumowanie. . . . .	209

9	Optymalizacja wyszukiwania i sortowania . . . . .	211
	Tabele klucz/wartość wykorzystujące <code>std::map</code> i <code>std::string</code> . . . . .	212
	Zestaw narzędzi do podnoszenia wydajności wyszukiwania . . . . .	213
	Dokonywanie bazowego pomiaru . . . . .	214
	Zidentyfikuj aktywność do zoptymalizowania . . . . .	215
	Rozłóż aktywność do zoptymalizowania . . . . .	215
	Zmodyfikuj lub zastąp algorytmy i struktury danych. . . . .	216
	Przeprowadź proces optymalizacji na niestandardowych abstrakcjach. .218	
	Optymalizowanie wyszukiwania przy użyciu <code>std::map</code> . . . . .	219
	Użyj tablic znaków o stałym rozmiarze w roli kluczy <code>std::map</code> . . . . .	219
	Użyj <code>std::map</code> z kluczami ciągu w stylu języka C. . . . .	220
	Użyj <code>std::set</code> dla kluczy zawartych w wartościach . . . . .	223
	Optymalizowanie wyszukiwania przy użyciu nagłówka <code>&lt;algorithm&gt;</code> . . . . .	224
	Przeszukiwana tabela klucz/wartość w kontenerach sekwencji . . . . .	225
	<code>std::find()</code> : oczywista nazwa, koszt czasowy $O(n)$ . . . . .	226
	<code>std::binary_search()</code> : nie zwraca wartości. . . . .	227
	Wyszukiwanie binarne przy użyciu <code>std::equal_range()</code> . . . . .	228
	Wyszukiwanie binarne przy użyciu <code>std::lower_bound()</code> . . . . .	228
	Własna implementacja wyszukiwania binarnego . . . . .	229
	Własna implementacja wyszukiwania binarnego przy użyciu <code>strcmp()</code> .230	
	Optymalizowanie wyszukiwania w tabelach mieszania klucz/wartość . . . . .	231
	Mieszanie przy użyciu <code>std::unordered_map</code> . . . . .	232
	Mieszanie, gdy klucze są tablicami o stałej liczbie znaków . . . . .	233
	Mieszanie, gdy klucze są ciągami zakończonymi znakiem null . . . . .	234
	Mieszanie z niestandardową tabelą . . . . .	236
	Konsekwencje abstrakcji Stepanova . . . . .	237
	Optymalizuj sortowanie przy użyciu standardowej biblioteki C++. . . . .	239
	Podsumowanie. . . . .	240
10	Optymalizowanie struktur danych . . . . .	241
	Poznaj kontenery z biblioteki standardowej . . . . .	241
	Kontenery sekwencji . . . . .	242
	Kontenery asocjacyjne. . . . .	242
	Eksperymentowanie z kontenerami biblioteki standardowej . . . . .	243
	<code>std::vector</code> i <code>std::string</code> . . . . .	249
	Wpływ realokacji na wydajność. . . . .	250
	Wstawianie i usuwanie z <code>std::vector</code> . . . . .	251
	Iterowanie po kontenerze <code>std::vector</code> . . . . .	253
	Sortowanie kontenera <code>std::vector</code> . . . . .	254
	Przeszukiwanie kontenera <code>std::vector</code> . . . . .	254

std::deque . . . . .	255
Wstawianie i usuwanie z std::deque . . . . .	256
Iterowanie po kontenerze std::deque . . . . .	257
Sortowanie kontenera std::deque . . . . .	258
Przeszukiwanie kontenera std::deque . . . . .	258
std::list . . . . .	259
Wstawianie i usuwanie z std::list . . . . .	261
Iterowanie po kontenerze std::list . . . . .	261
Sortowanie kontenera std::list . . . . .	262
Przeszukiwanie kontenera std::list . . . . .	262
std::forward_list . . . . .	262
Wstawianie i usuwanie z std::forward_list . . . . .	264
Iterowanie po kontenerze std::forward_list . . . . .	264
Sortowanie kontenera std::forward_list . . . . .	264
Przeszukiwanie kontenera std::forward_list . . . . .	264
std::map i std::multimap . . . . .	264
Wstawianie i usuwanie z std::map . . . . .	266
Iterowanie po kontenerze std::map . . . . .	268
Sortowanie kontenera std::map . . . . .	268
Przeszukiwanie kontenera std::map . . . . .	268
std::set i std::multiset . . . . .	269
std::unordered_map i std::unordered_multimap . . . . .	270
Wstawianie i usuwanie z std::unordered_map . . . . .	274
Iterowanie po kontenerze std::unordered_map . . . . .	274
Przeszukiwanie kontenera std::unordered_map . . . . .	274
Inne struktury danych . . . . .	275
Podsumowanie . . . . .	277
<b>11 Optymalizowanie we/wy . . . . .</b>	<b>279</b>
Przepis na odczytywanie plików . . . . .	279
Tworzenie ascetycznej sygnatury funkcji . . . . .	281
Skracanie łańcucha wywołania . . . . .	283
Redukowanie realokacji . . . . .	283
Pobierz większe porcje – użyj większego bufora wejściowego . . . . .	286
Pobieraj większe porcje – odczytuj pojedyncze wiersze . . . . .	286
Ponowne skracanie łańcucha wywołań . . . . .	288
Zmiany, które nie pomogły . . . . .	290
Zapisywanie plików . . . . .	291
Odczytywanie z std::cin i zapisywanie w std::cout . . . . .	292
Podsumowanie . . . . .	292



12	Optymalizowanie równoległości.....	293
	Powtórzenie informacji o równoległości.....	294
	Wprowadzenie do świata równoległości.....	295
	Wykonanie z przeplotem.....	299
	Spójność sekwencyjna.....	300
	Wyścigi.....	301
	Synchronizacja.....	302
	Atomowość.....	303
	Powtórzenie informacji o obsłudze równoległości w języku C++.....	305
	Wątki.....	305
	Obietnice i przyszłości.....	307
	Zadania asynchroniczne.....	309
	Mutexy.....	311
	Blokady.....	312
	Zmienne warunkowe.....	313
	Atomowe operacje na współdzielonych zmiennych.....	316
	Plany: Przyszłość równoległości w języku C++.....	319
	Optymalizowanie wielowątkowych programów C++.....	321
	Wybieraj <code>std::async</code> zamiast <code>std::thread</code> .....	321
	Dopasuj liczbę wątków do liczby rdzeni.....	323
	Implementuj kolejkę zadań i pulę wątków.....	325
	Wykonywanie operacji we/wy w osobnym wątku.....	326
	Program bez synchronizacji.....	326
	Usuwanie kodu z uruchamiania i zamykania.....	329
	Zwiększenie wydajności synchronizacji.....	330
	Redukuj zakres sekcji krytycznych.....	330
	Ograniczaj liczbę równoległych wątków.....	331
	Unikaj problemu masowego pędu.....	332
	Unikaj konwoju blokady.....	333
	Zredukuj rywalizację.....	333
	Nie oczekuj aktywnie w jednordzeniowym systemie.....	335
	Nie czekaj w nieskończoność.....	335
	Tworzenie własnego mutexu może być nieefektywne.....	335
	Ogranicz długość kolejki wyjściowej producenta.....	336
	Biblioteki wspierające równoległość.....	336
	Podsumowanie.....	338
13	Optymalizowanie zarządzania pamięcią.....	339
	Powtórzenie wiadomości o API zarządzania pamięcią w języku C++.....	340
	Cykl życia zmiennych dynamicznych.....	340

Funkcje zarządzania pamięcią alokują i zwalniają pamięć. . . . .	341
Wyrażenia <code>new</code> konstruują zmienne dynamiczne. . . . .	344
Wyrażenia <code>delete</code> usuwają zmienne dynamiczne. . . . .	347
Jawne wywołania destruktora usuwają zmienne dynamiczne. . . . .	348
Wysoko wydajne menedżery pamięci. . . . .	349
Dostarczanie menedżera pamięci specyficznego dla klasy . . . . .	351
Menedżer pamięci o stałym rozmiarze bloku . . . . .	352
Arena bloków . . . . .	355
Dodawanie specyficznego dla klasy operator <code>new()</code> . . . . .	357
Wydajność menedżera pamięci o stałym rozmiarze bloku . . . . .	358
Różne warianty menedżera pamięci o stałym rozmiarze bloku . . . . .	359
Menedżery pamięci nieobsługujące wielowątkowości są wydajne . . . . .	360
Dostarczanie niestandardowych alokatorów. . . . .	360
Minimalny alokator C++11 . . . . .	363
Dodatkowe definicje dla alokatora C++98. . . . .	365
Alokator o stałym rozmiarze bloków . . . . .	369
Alokator o stałym rozmiarze bloków dla ciągów. . . . .	371
Podsumowanie. . . . .	373
<i>Indeks</i> . . . . .	375
<i>O autorze</i> . . . . .	389