helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

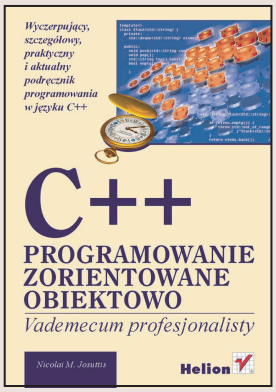
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++. Projektowanie zorientowane obiektowo. Vademecum profesjonalisty



Autor: Nicolai M. Josuttis
Tłumaczenie: Jaromir Senczyk
ISBN: 83-7361-195-9
Tytuł oryginału: Object-Oriented Programming in C++
Format: B5, stron: 560

C++ jest obecnie wiodącym językiem programowania obiektowego. Jego podstawowymi zaletami w stosunku do innych języków obiektowych jest wysoka efektywność i uniwersalność. Stosowany jest do tworzenia komercyjnego oprogramowania oraz efektywnych rozwiązań złożonych problemów.

Książka krok po kroku omawia wszystkie właściwości języka i wyjaśnia sposoby ich praktycznego użycia. Przedstawione przykłady programów nie są zbyt skomplikowane, by nie odrywać Twojej uwagi od omawianych zagadnień, ale nie są też sztucznie uproszczone. Kluczowym założeniem języka C++ jest programowanie z wykorzystaniem szablonów, które umożliwiają tworzenie rozwiązań o wysokim poziomie ogólności — na przykład implementację polimorfizmu. Nicolai Josuttis omawia możliwość łączenia szablonów z programowaniem obiektowym, która decyduje o potężnych możliwościach języka C++ jako narzędzia tworzenia wydajnych programów. W tym zakresie książka wykracza daleko poza podstawy.

- Wprowadzenie do C++ i programowania obiektowego
- Podstawowe pojęcia języka C++
- Programowanie klas
- Dziedziczenie i polimorfizm
- Składowe dynamiczne i statyczne
- Szablony języka C++
- Szczegółowe omówienie standardowej biblioteki wejścia-wyjścia

Książka ta jest idealnym podręcznikiem umożliwiającym studiowanie języka C++ w domowym zaciszu. Prezentuje ona zagadnienia podstawowe, ale w wielu przypadkach przekracza je dostarczając prawdziwie profesjonalnej wiedzy.

Wyczerpujący, szczegółowy, praktyczny i aktualny podręcznik programowania w języku C++.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa **11**

Rozdział 1. O książce **13**

1.1. Dlaczego napisałem tę książkę? 13
1.2. Wymagania 14
1.3. Organizacja książki 14
1.4. W jaki sposób należy czytać książkę? 15
1.5. Przykłady programów i dodatkowe informacje 15

Rozdział 2. Wprowadzenie: język C++ i programowanie obiektowe **19**

2.1. Język C++ 19
2.1.1. Kryteria projektowania 19
2.1.2. Historia języka 20
2.2. C++ jako język programowania obiektowego 20
2.2.1. Obiekty, klasy i instancje 21
2.2.2. Klasy w języku C++ 23
2.2.3. Hermetyzacja danych 25
2.2.4. Dziedziczenie 27
2.2.5. Polimorfizm 28
2.3. Inne koncepcje języka C++ 29
2.3.1. Obsługa wyjątków 30
2.3.2. Szablony 30
2.3.3. Przestrzenie nazw 32
2.4. Terminologia 32

Rozdział 3. Podstawowe pojęcia języka C++ **35**

3.1. Pierwszy program 35
3.1.1. „Hello, World!” 35
3.1.2. Komentarze w języku C++ 37
3.1.3. Funkcja main() 37
3.1.4. Wejście i wyjście 39
3.1.5. Przestrzenie nazw 40
3.1.6. Podsumowanie 41
3.2. Typy, operatory i instrukcje sterujące 41
3.2.1. Pierwszy program, który przeprowadza obliczenia 41
3.2.2. Typy podstawowe 44

3.2.3. Operatory 48
3.2.4. Instrukcje sterujące 54
3.2.5. Podsumowanie 57
3.3. Funkcje i moduły 58
3.3.1. Pliki nagłówkowe 58
3.3.2. Plik źródłowy zawierający definicję 60
3.3.3. Plik źródłowy zawierający wywołanie funkcji 60
3.3.4. Kompilacja i konsolidacja 61
3.3.5. Rozszerzenia nazw plików 62
3.3.6. Systemowe pliki nagłówkowe i biblioteki 63
3.3.7. Preprocesor 63
3.3.8. Przestrzenie nazw 66
3.3.9. Słowo kluczowe static 67
3.3.10. Podsumowanie 69
3.4. Łańcuchy znaków 70
3.4.1. Pierwszy przykład programu wykorzystującego łańcuchy znaków 70
3.4.2. Kolejny przykładowy program wykorzystujący łańcuchy znaków 74
3.4.3. Przegląd operacji na łańcuchach znaków 78
3.4.4. Łańcuchy znaków i C-łańcuchy 79
3.4.5. Podsumowanie 80
3.5. Kolekcje 80
3.5.1. Program wykorzystujący klasę vector 81
3.5.2. Program wykorzystujący klasę deque 82
3.5.3. Wektory i kolejki 83
3.5.4. Iteratory 84
3.5.5. Przykładowy program wykorzystujący listy 87
3.5.6. Przykłady programów wykorzystujących kontenery asocjacyjne 88
3.5.7. Algorytmy 92
3.5.8. Algorytmy wielozakresowe 96
3.5.9. Iteratory strumieni 98
3.5.10. Uwagi końcowe 100
3.5.11. Podsumowanie 101
3.6. Obsługa wyjątków 102
3.6.1. Powody wprowadzenia obsługi wyjątków 102
3.6.2. Koncepcja obsługi wyjątków 104
3.6.3. Standardowe klasy wyjątków 105
3.6.4. Przykład obsługi wyjątku 106
3.6.5. Obsługa nieoczekiwanych wyjątków 109
3.6.6. Funkcje pomocnicze obsługi błędów 110
3.6.7. Podsumowanie 111
3.7. Wskaźniki, tablice i C-łańcuchy 112
3.7.1. Wskaźniki 112
3.7.2. Tablice 115
3.7.3. C-łańcuchy 117
3.7.4. Podsumowanie 121
3.8. Zarządzanie pamięcią za pomocą operatorów new i delete 121
3.8.1. Operator new 123
3.8.2. Operator delete 123

3.8.3. Dynamiczne zarządzanie pamięcią tablic 124
3.8.4. Obsługa błędów związanych z operatorem new 126
3.8.5. Podsumowanie 126
3.9. Komunikacja ze światem zewnętrznym 126
3.9.1. Parametry wywołania programu 126
3.9.2. Dostęp do zmiennych środowiska 128
3.9.3. Przerwanie działania programu 128
3.9.4. Wywoływanie innych programów 129
3.9.5. Podsumowanie 130

Rozdział 4. Programowanie klas 131

4.1. Pierwsza klasa: Fraction 131
4.1.1. Zanim rozpoczniemy implementację 131
4.1.2. Deklaracja klasy Fraction 134
4.1.3. Struktura klasy 135
4.1.4. Funkcje składowe 138
4.1.5. Konstruktory 138
4.1.6. Przeciążenie funkcji 140
4.1.7. Implementacja klasy Fraction 141
4.1.8. Wykorzystanie klasy Fraction 146
4.1.9. Tworzenie obiektów tymczasowych 151
4.1.10. Notacja UML 152
4.1.11. Podsumowanie 152
4.2. Operatory klas 153
4.2.1. Deklaracje operatorów 153
4.2.2. Implementacja operatorów 156
4.2.3. Posługiwanie się operatorami 163
4.2.4. Operatory globalne 164
4.2.5. Ograniczenia w definiowaniu operatorów 165
4.2.6. Specjalne właściwości niektórych operatorów 166
4.2.7. Podsumowanie 170
4.3. Optymalizacja efektywności kodu 170
4.3.1. Wstępna optymalizacja klasy Fraction 171
4.3.2. Domyślne parametry funkcji 174
4.3.3. Funkcje rozwijane w miejscu wywołania 175
4.3.4. Optymalizacja z perspektywy użytkownika 177
4.3.5. Instrukcja using 178
4.3.6. Deklaracje pomiędzy instrukcjami 179
4.3.7. Konstruktory kopiujące 181
4.3.8. Podsumowanie 182
4.4. Referencje i stałe 183
4.4.1. Konstruktory kopiujące i przekazywanie parametrów 183
4.4.2. Referencje 184
4.4.3. Stałe 187
4.4.4. Stałe funkcje składowe 189
4.4.5. Klasa Fraction wykorzystująca referencje 190
4.4.6. Wskaźniki stałych i stałe wskaźniki 193
4.4.7. Podsumowanie 195

4.5. Strumienie wejścia i wyjścia....196
4.5.1. Strumienie....196
4.5.2. Korzystanie ze strumieni....197
4.5.3. Stan strumienia....203
4.5.4. Operatory wejścia i wyjścia dla typów definiowanych przez użytkownika....205
4.5.5. Podsumowanie....214
4.6. Klasy zaprzyjaźnione i inne typy....214
4.6.1. Automatyczna konwersja typów....215
4.6.2. Słowo kluczowe explicit....217
4.6.3. Funkcje zaprzyjaźnione....217
4.6.4. Funkcje konwersji....223
4.6.5. Problemy automatycznej konwersji typu....225
4.6.6. Inne zastosowania słowa kluczowego friend....227
4.6.7. Słowo kluczowe friend i programowanie obiektowe....227
4.6.8. Podsumowanie....228
4.7. Obsługa wyjątków w klasach....229
4.7.1. Powody zastosowania obsługi wyjątków w klasie Fraction....229
4.7.2. Obsługa wyjątków w klasie Fraction....230
4.7.3. Klasy wyjątków....237
4.7.4. Ponowne wyrzucenie wyjątku....237
4.7.5. Wyjątki w destruktorach....238
4.7.6. Wyjątki i deklaracje interfejsów....238
4.7.7. Hierarchie klas wyjątków....239
4.7.8. Projektowanie klas wyjątków....242
4.7.9. Wyrzucanie standardowych wyjątków....244
4.7.10. Bezpieczeństwo wyjątków....244
4.7.11. Podsumowanie....245

Rozdział 5. Dziedziczenie i polimorfizm....247

5.1. Dziedziczenie pojedyncze....249
5.1.1. Klasa Fraction jako klasa bazowa....249
5.1.2. Klasa pochodna RFraction....251
5.1.3. Deklaracja klasy pochodnej RFraction....253
5.1.4. Dziedziczenie i konstruktory....255
5.1.5. Implementacja klas pochodnych....258
5.1.6. Wykorzystanie klasy pochodnej....260
5.1.7. Konstruktory obiektów klasy bazowej....262
5.1.8. Podsumowanie....264
5.2. Funkcje wirtualne....264
5.2.1. Problemy z przesłanianiem funkcji....265
5.2.2. Statyczne i dynamiczne wiązanie funkcji....267
5.2.3. Przeciążenie i przesłanianie....271
5.2.4. Dostęp do parametrów klasy bazowej....273
5.2.5. Destruktory wirtualne....274
5.2.6. Właściwe sposoby stosowania dziedziczenia....275
5.2.7. Inne pułapki przesłaniania funkcji....279
5.2.8. Dziedziczenie prywatne i deklaracje dostępu....281
5.2.9. Podsumowanie....284

5.3. Polimorfizm 285
5.3.1. Czym jest polimorfizm? 285
5.3.2. Polimorfizm w języku C++ 287
5.3.3. Przykład polimorfizmu w języku C++ 288
5.3.4. Abstrakcyjna klasa bazowa GeoObj 291
5.3.5. Zastosowanie polimorfizmu wewnątrz klas 298
5.3.6. Polimorfizm nie wymaga instrukcji wyboru 304
5.3.7. Przywracanie obiektowi jego rzeczywistej klasy 304
5.3.8. Projektowanie przez kontrakt 308
5.3.9. Podsumowanie 309
5.4. Dziedziczenie wielokrotne 310
5.4.1. Przykład dziedziczenia wielokrotnego 310
5.4.2. Wirtualne klasy bazowe 315
5.4.3. Identyczność i adresy 318
5.4.4. Wielokrotne dziedziczenie tej samej klasy bazowej 321
5.4.5. Podsumowanie 321
5.5. Pułapki projektowania z użyciem dziedziczenia 322
5.5.1. Dziedziczenie kontra zawieranie 322
5.5.2. Błędy projektowania: ograniczanie dziedziczenia 323
5.5.3. Błędy projektowania: dziedziczenie zmieniające wartość 324
5.5.4. Błędy projektowania: dziedziczenie zmieniające interpretację wartości 326
5.5.5. Unikajmy dziedziczenia! 327
5.5.6. Podsumowanie 327

Rozdział 6. Składowe dynamiczne i statyczne 329

6.1. Składowe dynamiczne 329
6.1.1. Implementacja klasy String 329
6.1.2. Konstruktory i składowe dynamiczne 334
6.1.3. Implementacja konstruktora kopiującego 336
6.1.4. Destruktory 337
6.1.5. Implementacja operatora przypisania 337
6.1.6. Pozostałe operatory 339
6.1.7. Wczytywanie łańcucha klasy String 341
6.1.8. Komercyjne implementacje klasy String 344
6.1.9. Inne zastosowania składowych dynamicznych 346
6.1.10. Podsumowanie 347
6.2. Inne aspekty składowych dynamicznych 348
6.2.1. Składowe dynamiczne w obiektach stałych 348
6.2.2. Funkcje konwersji dla składowych dynamicznych 351
6.2.3. Funkcje konwersji i instrukcje warunkowe 353
6.2.4. Stałe jako zmienne 355
6.2.5. Zapobieganie wywołaniu domyślnych operacji 357
6.2.6. Klasy zastępcze 358
6.2.7. Obsługa wyjątków z użyciem parametrów 361
6.2.8. Podsumowanie 365
6.3. Dziedziczenie i klasy o składowych dynamicznych 365
6.3.1. Klasa CPPBook::String jak klasa bazowa 365
6.3.2 Klasa pochodna ColString 368
6.3.3. Dziedziczenie funkcji zaprzyjaźnionych 371

6.3.4. Plik źródłowy klasy pochodnej ColString 373
6.3.5. Aplikacja klasy ColString 374
6.3.6. Dziedziczenie specjalnych funkcji dla składowych dynamicznych 375
6.3.7. Podsumowanie 376
6.4. Klasy zawierające klasy 376
6.4.1. Obiekty jako składowe innych klas 377
6.4.2. Implementacja klasy Person 377
6.4.3. Podsumowanie 383
6.5. Składowe statyczne i typy pomocnicze 383
6.5.1. Statyczne składowe klas 384
6.5.2. Deklaracje typu wewnątrz klasy 389
6.5.3. Typy wyliczeniowe jako statyczne stałe klasy 391
6.5.4. Klasy zagnieżdżone i klasy lokalne 392
6.5.5. Podsumowanie 393

Rozdział 7. Szablony 395

7.1. Dlaczego szablony? 395
7.1.1. Terminologia 396
7.2. Szablony funkcji 396
7.2.1. Definiowanie szablonów funkcji 397
7.2.2. Wywoływanie szablonów funkcji 398
7.2.3. Praktyczne wskazówki dotyczące używania szablonów 399
7.2.4. Szablony i automatyczna konwersja typu 399
7.2.5. Przeciążanie szablonów 400
7.2.6. Zmienne lokalne 402
7.2.7. Podsumowanie 402
7.3. Szablony klas 403
7.3.1. Implementacja szablonu klasy Stack 403
7.3.2. Zastosowanie szablonu klasy Stack 406
7.3.3. Specjalizacja szablonów klas 408
7.3.4. Domyślne parametry szablonu 410
7.3.5. Podsumowanie 412
7.4. Inne parametry szablonów 412
7.4.1. Przykład zastosowania innych parametrów szablonów 412
7.4.2. Ograniczenia parametrów szablonów 415
7.4.3. Podsumowanie 416
7.5. Inne zagadnienia związane z szablonami 416
7.5.1. Słowo kluczowe typename 416
7.5.2. Składowe jako szablony 417
7.5.3. Polimorfizm statyczny z użyciem szablonów 420
7.5.4. Podsumowanie 424
7.6. Szablony w praktyce 424
7.6.1. Kompilacja kodu szablonu 424
7.6.2. Obsługa błędów 429
7.6.3. Podsumowanie 430

Rozdział 8. Standardowa biblioteka wejścia i wyjścia w szczegółach 433

8.1. Standardowe klasy strumieni 433
8.1.1. Klasy strumieni i obiekty strumieni 434
8.1.2. Stan strumienia 436

8.1.3. Operatory standardowe 439
8.1.4. Funkcje standardowe 440
8.1.5. Manipulatory 443
8.1.6. Definicje formatu 445
8.1.7. Internacjonalizacja 455
8.1.8. Podsumowanie 458
8.2. Dostęp do plików 458
8.2.1. Klasy strumieni dla plików 458
8.2.2. Wykorzystanie klas strumieni plików 459
8.2.3. Znaczniki plików 461
8.2.4. Jawne otwieranie i zamykanie plików 462
8.2.5. Swobodny dostęp do plików 463
8.2.6. Przekierowanie standardowych kanałów do plików 466
8.2.7. Podsumowanie 467
8.3. Klasy strumieni łańcuchów 467
8.3.1. Klasy strumieni łańcuchów 468
8.3.2. Operator rzutowania leksykalnego 470
8.3.3. Strumienie C-łańcuchów 472
8.3.4. Podsumowanie 474

Rozdział 9. Inne właściwości języka 475

9.1. Dodatkowe informacje o bibliotece standardowej 475
9.1.1. Operacje na wektorach 475
9.1.2. Operacje wspólne dla wszystkich kontenerów STL 482
9.1.3. Algorytmy STL 482
9.1.4. Ograniczenia wartości numerycznych 488
9.1.5. Podsumowanie 492
9.2. Definiowanie specjalnych operatorów 493
9.2.1. Inteligentne wskaźniki 493
9.2.2. Obiekty funkcji 496
9.2.3. Podsumowanie 500
9.3. Inne aspekty operatorów new i delete 500
9.3.1. Operatory new i delete, które nie wyrzucają wyjątków 500
9.3.2. Określanie położenia obiektu 501
9.3.3. Funkcje obsługi operatora new 501
9.3.4. Przeciążanie operatorów new i delete 506
9.3.5. Dodatkowe parametry operatora new 509
9.3.6. Podsumowanie 510
9.4. Wskaźniki funkcji i wskaźniki składowych 510
9.4.1. Wskaźniki funkcji 510
9.4.2. Wskaźniki składowych 511
9.4.3. Wskaźniki składowych i zewnętrzne interfejsy 514
9.4.4. Podsumowanie 515
9.5. Łączenie programów w językach C i C++ 516
9.5.1. Łączenie zewnętrzne 516
9.5.2. Pliki nagłówkowe w językach C i C++ 517
9.5.3. Kompilacja funkcji main() 517
9.5.4. Podsumowanie 518

9.6. Dodatkowe słowa kluczowe 518
9.6.1. Unie 518
9.6.2. Typy wyliczeniowe 519
9.6.3. Słowo kluczowe volatile 520
9.6.4. Podsumowanie 520

Rozdział 10. Podsumowanie 521
10.1. Hierarchia operatorów języka C++ 521
10.2. Właściwości operacji klas 523
10.3. Zasady automatycznej konwersji typów 524
10.4. Przydatne zasady programowania i projektowania 525

Dodatek A Bibliografia 529

Dodatek B Słownik 533

Skorowidz 539

Rozdział 7.
Szablony

W rozdziale tym przedstawiona zostanie koncepcja szablonów. Szablony umożliwiają parametryzację kodu dla różnych typów. Dzięki temu na przykład funkcja wyznaczająca najmniejszą wartość lub klasa kolekcji może zostać zaimplementowana, zanim ustalony zostanie typ parametru funkcji lub elementu kolekcji. Kod generowany na podstawie szablonu nie przetwarza jednak dowolnych typów: w momencie, gdy ustalony zostanie typ parametru. Obowiązuje również zwykła kontrola zgodności typów.

Najpierw przedstawione zostaną szablony funkcji, a następnie szablony klas. Omówienie szablonów zakończymy przedstawieniem sposobów ich wykorzystania, w tym specjalnych technik projektowania.

Więcej informacji na temat szablonów można znaleźć w książce *C++ Templates — The Complete Guide*[1], której autorami są Nicolai M. Josuttis i David Vandevoorde (patrz *VandevoordeJosuttisTemplate*). Książka ta zawiera podobne wprowadzenie do problematyki szablonów, wyczerpujący opis szablonów, szeregu technik kodowania i zaawansowanych zastosowań szablonów.

7.1. Dlaczego szablony?

W językach programowania, które wiążą zmienne z określonym typem danych, często pojawia się sytuacja wymagająca wielokrotnego zdefiniowania tej samej funkcji dla różnych typów parametrów. Typowym przykładem jest funkcja zwracająca większą z przekazanych jej dwóch wartości. Musi ona zostać zaimplementowana osobno dla każdego typu, dla którego chcemy ja wykorzystywać. W języku C można tego uniknąć, posługując się makrodefinicją. Ponieważ działanie makrodefinicji polega na mechanicznym zastępowaniu przez preprocesor jednego tekstu programu innym, rozwiązanie takie nie jest zalecane (ze względu na brak kontroli zgodności typów, możliwość wystąpienia efektów ubocznych, etc.).

Nie tylko wielokrotna implementacja funkcji wymaga dodatkowej pracy. Podobna sytuacja występuje w przypadku implementacji zaawansowanych typów, takich jak kontenery. Zarządzanie kolekcją elementów kontenera wymaga implementacji szeregu funkcji,

[1] *C++. Szablony. Vademecum profesjonalisty*, Helion 2003.

w przypadku których istotny jest przede wszystkim typ elementu kolekcji. Gdyby nie możliwość wykorzystania specjalnych właściwości języka C++, wszystkie te funkcje musiałyby być implementowane za każdym razem, gdy pojawia się nowy typ elementów kolekcji.

Stos jest typowym przykładem, w którym pojawia się wiele implementacji służących zarządzaniu różnymi obiektami. Stos umożliwia przechowywanie i usuwanie elementów konkretnego typu. Jeśli stosowalibyśmy poznane dotychczas konstrukcje języka C++, zmuszeni bylibyśmy implementować jego operacje osobno dla każdego typu elementów, które mogą być umieszczone na stosie, ponieważ deklaracja stosu wymaga podania typu jego elementów. Te same operacje musielibyśmy więc implementować wielokrotnie, mimo, że rzeczywisty algorytm nie ulega zmianie. Nie tylko wymaga to dodatkowej pracy, ale może stać się także źródłem błędów.

Dlatego też w języku C++ wprowadzono *szablony*. Szablony reprezentują funkcje bądź klasy, które nie zostały zaimplementowane dla konkretnego typu, ponieważ typ ten zostanie dopiero zdefiniowany. Aby użyć szablonu funkcji lub klasy, programista aplikacji musi określić typ, dla którego dany szablom zostanie zrealizowany. Wystarczy więc, że funkcja wyznaczająca większą z dwóch wartości zostanie zaimplementowana tylko raz — jako szablon. Podobnie kontenery, takie jak stosy, listy itp., wymagają tylko jednokrotnego zaimplementowania i przetestowania, a następnie mogą być wykorzystywane dla dowolnego typu elementów, który umożliwia przeprowadzanie odpowiednich operacji.

Sposoby definiowania i posługiwania się szablonami zostaną wyjaśnione poniżej, najpierw dla funkcji na przykładzie funkcji `max()`, a następnie dla klas, na przykładzie stosu.

7.1.1. Terminologia

Terminologia związana z szablonami nie jest ściśle zdefiniowana. Na przykład funkcja, której typ został sparametryzowany, nazywana jest czasami *szablonem funkcji*, a innym razem *funkcją szablonu*. Ponieważ drugie z tych określeń jest nieco mylące (może określać szablon funkcji, ale także funkcję nie związaną z szablonem), powinno się raczej używać określenia *szablon funkcji*. Podobnie klasa sparametryzowana ze względu na typ powinna być nazywana *szablonem klasy*, a nie *klasą szablonu*.

Proces, w którym przez podstawienie do szablonu wartości parametru powstaje zwykła klasa, funkcja lub funkcja składowa, nazywany jest *tworzeniem instancji szablonu*. Niestety, termin „tworzenie instancji” używany jest także w terminologii obiektowej dla określenia tworzenia obiektu danej klasy. Dlatego też znaczenie terminu „tworzenie instancji” zależy w przypadku języka C++ od kontekstu, w którym zostaje on użyty.

7.2. Szablony funkcji

Jak już wspomnieliśmy, szablony funkcji umożliwiają definiowanie grup funkcji, które mogą następnie zostać użyte dla różnych typów.

W przeciwieństwie do działania makrodefinicji, działanie szablonów nie polega na mechanicznym zastępowaniu tekstów. Semantyka funkcji zostaje sprawdzona przez kompilator, co pozwala zapobiec niepożądanym efektom ubocznym. Nie istnieje na przykład (tak jak w przypadku makrodefinicji) niebezpieczeństwo wielokrotnego zastąpienia parametru `n++`, na skutek którego pojedyncza instrukcja inkrementacji staje się wielokrotną inkrementacją.

7.2.1. Definiowanie szablonów funkcji

Szablony funkcji definiowane są jak zwykłe funkcje, a parametryzowany typ poprzedza ich deklarację. Na przykład szablon funkcji wyznaczającej większą z dwóch wartości zostanie zadeklarowany w następujący sposób:

```
template <typename T>
const T& max(const T&, const T&);
```

W pierwszym wierszu zadeklarowany został parametr typu `T`. Słowo kluczowe `typename` oznacza, że następujący po nim symbol reprezentuje typ. Słowo to zostało wprowadzone w języku C++ stosunkowo późno. Wcześniej używano zamiast niego słowa kluczowego `class`:

```
template <class T>
```

Pomiędzy słowami tymi nie ma różnic semantycznych. Użycie słowa kluczowego `class` nie wymaga, by parametryzowany typ był klasą. Używając obu słów kluczowych możemy korzystać z dowolnych typów (podstawowych, klas etc.) pod warunkiem, że umożliwiają one wykonywanie operacji używanych przez szablon. W naszym przykładzie dla typu `T` musi być zdefiniowany operator porównania (`<`), którego używa implementacja szablonu funkcji `max()`.

Zastosowanie symbolu `T` dla typu szablonu nie jest wymagane, ale w praktyce bywa często stosowane. W poniższej deklaracji symbol `T` używany jest w celu określenie typu funkcji oraz typu jej parametrów:

```
// tmpl/max1.hpp
template <typename T>
const T& max(const T& a, const T& b)
{
    return  a > b ? a : b;
}
```

Instrukcje umieszczone w ciele szablonu nie różnią się niczym od instrukcji zwykłej implementacji funkcji. W powyższym przykładzie porównywane są dwie wartości typu `T` i zwracana jest większa z nich. Zastosowanie referencji stałych (`const T&`) zapobiega tworzeniu kopii parametrów funkcji i wartości zwracanych przez funkcję (patrz podrozdział 4.4).

7.2.2. Wywoływanie szablonów funkcji

Szablonu funkcji używamy w taki sam sposób, jak zwykłej funkcji. Demonstruje to poniższy program[2]:

```
// tmpl/max1.cpp
#include <iostream>
#include <string>
#include "max1.hpp"

int main()
{
    int         a, b;  // dwie zmienne typu int
    std::string s, t;  // dwie zmienne typu std::string
    //...
    std::cout << max(a,b) << std::endl;    // max() dla dwóch wartości całkowitych
    std::cout << ::max(s,t) << std::endl;  // max() dla dwóch łańcuchów
}
```

Dopiero w momencie, gdy funkcja `max()` zostaje wywołana dla dwóch obiektów takiego samego typu, szablon staje się rzeczywistym kodem. Kompilator wykorzystuje definicję szablonu i tworzy jego instancję, zastępując typ `T` typem `int` lub `std::string`. W ten sposób tworzony jest rzeczywisty kod implementacji dla typu `int` bądź typu `std::string`. Szablony nie są więc kompilowane jako kod, który może działać z dowolnym typem danych, lecz jedynie wykorzystywane w celu wygenerowania kodu dla konkretnego typu. Jeśli szablon `max()` zostanie wywołany dla siedmiu różnych typów, to skompilowanych zostanie siedem funkcji.

Proces, w którym na podstawie szablonu generowany jest kod, który zostanie skompilowany, nazywany jest *tworzeniem instancji* lub, precyzyjniej (ze względu na zastosowanie określenia „tworzenie instancji" w terminologii obiektowej), *tworzeniem instancji szablonu*.

Utworzenie instancji szablonu jest oczywiście możliwe tylko wtedy, gdy dla danego typu zdefiniowane są wszystkie operacje używane przez szablon. Aby możliwe było utworzenie instancji szablonu `max()` dla typu `std::string`, w klasie `std::string` musi być zdefiniowany operator porównania (`<`).

Zauważmy, że w przeciwieństwie do działania makrodefinicji, działanie szablonów nie polega na zastępowaniu tekstów. Wywołanie

```
max(x++, z *= 2);
```

nie jest może szczególnie użyteczne, ale będzie działać poprawnie. Zwróci większą wartość jednego z przekazanych jej wyrażeń, a każde z wyrażeń wartościowane będzie tylko raz (czyli zmienna `x` będzie inkrementowana jeden raz).

---

[2] Wywołanie szablonu `max()` dla typu string zostało jawnie poprzedzone operatorem globalnego zakresu, ponieważ w standardowej bibliotece zdefiniowana jest funkcja `std::max()`. Ponieważ typ `string` również znajduje się w przestrzeni nazw `std`, to funkcja `max()` nie poprzedzona operatorem zakresu zostałaby odnaleziona właśnie w tej przestrzeni (patrz „Wyszukiwanie Koeniga", str. 179).

7.2.3. Praktyczne wskazówki dotyczące używania szablonów

Koncepcja szablonów wykracza poza zwykły model kompilacji (konsolidacji), wykorzystujący odrębne jednostki translacji. Nie można na przykład umieścić szablonów w osobnym module i skompilować go, a następnie osobno skompilować aplikacji używającej tych szablonów i skonsolidować oba uzyskane pliki wynikowe. Nie jest to możliwe, ponieważ typ, dla którego ma zostać użyty szablon, zostaje określony dopiero w momencie użycia szablonu.

Istnieją różne sposoby rozwiązania tego problemu. Najprostszy i najbardziej uniwersalny polega na umieszczeniu całego kodu szablonu w pliku nagłówkowym. Dołączając następnie zawartości pliku nagłówkowego do kodu aplikacji umożliwiamy generację i kompilację kodu dla konkretnych typów.

Nieprzypadkowo więc definicja szablonu `max()` z poprzedniego przykładu została umieszczona w pliku nagłówkowym. Należy przy tym zauważyć, że słowo kluczowe `inline` (patrz podrozdział 4.3.3) nie musi być zastosowane. W przypadku szablonów dopuszczalne jest istnienie wielu definicji w różnych jednostkach translacji. Jeśli jednak preferujemy rozwijanie szablonu funkcji w miejscu jego wywołania, powinniśmy zasygnalizować to kompilatorowi właśnie za pomocą słowa kluczowego `inline`.

Więcej zagadnień związanych z posługiwaniem się szablonami zostanie omówionych w podrozdziale 7.6.

7.2.4. Szablony i automatyczna konwersja typu

Podczas tworzenia instancji szablonu nie jest brana pod uwagę automatyczna konwersja typu. Jeśli szablon posiada wiele parametrów typu `T`, przekazane mu argumenty muszą być tego samego typu. Wywołanie szablonu `max()` dla obiektów różnych typów nie jest więc możliwe:

```
template <typename T>
const T& max(const T&, const&); // oba parametry mają typ T
...
int i;
long l;
...
max(i,l) // BŁĄD: zmienne i oraz l posiadają różne typy
```

Jawna kwalifikacja

Wywołując szablon możemy zastosować *jawną kwalifikację* typu, dla którego zostanie on użyty:

```
max<long>(i, l) // wywołanie szablonu max() dla typu long
```

W tym przypadku tworzona jest instancja szablonu funkcji `max()` dla typu `long` jako parametru `T` szablonu. Następnie, podobnie jak w przypadku zwykłych funkcji, kompilator sprawdza, czy przekazane parametry mogą być użyte jako wartości typu `long`, co jest możliwe w naszym przykładzie ze względu na istnienie domyślnej konwersji typu `int` do typu `long`.

Szablony o wielu parametrach

Szablon może zostać zdefiniowany także dla różnych typów:

```
template <typename T1, typename T2>
inline T1 max(const T1&, const T2&)
{
    return a < b ? b : a;
}
...
int i;
long l;
...
max(i,l) // zwraca wartość typu int
```

Problemem w tym przypadku jest typ wartości zwracanej przez funkcję, ponieważ w momencie definiowania szablonu nie wiemy, który z parametrów zostanie zwrócony. Dodatkowo, jeśli zwrócony będzie drugi parametr, dla wartości zwracanej utworzony zostanie lokalny obiekt tymczasowy, ponieważ posiada ona inny typ. Obiekt tymczasowy nie może zostać zwrócony przez referencję, wobec czego typ zwracany przez szablon został zmieniony z `const T&` na `T`.

W takim przypadku lepszym rozwiązaniem jest możliwość jawnej kwalifikacji.

7.2.5. Przeciążanie szablonów

Szablony mogą być przeciążane dla pewnych typów. W ten sposób ogólna implementacja szablonu może zostać zastąpiona inną implementacją dla konkretnych typów. Rozwiązanie takie posiada szereg zalet:

- Szablony funkcji mogą zostać zdefiniowane dla dodatkowych typów oraz ich kombinacji (na przykład może zostać zdefiniowana funkcja `max()` o parametrach typu `float` i `CPPBook::Fraction`).
- Implementacje mogą zostać zoptymalizowane dla konkretnych typów.
- Typy, dla których implementacja szablonu nie jest odpowiednia, mogą zostać właściwie obsłużone.

Wywołanie szablonu `max()` dla C-łańcuchów (typ `const char*`) spowoduje błąd:

```
const char* s1;
const char* s2;
...
const char* maxstring = max(s1,s2); // BŁĄD: porównuje adresy
```

Implementacja szablonu porówna w tym przypadku adresy C-łańcuchów zamiast ich zawartości (patrz podrozdział 3.7.3).

Problem ten możemy rozwiązać poprzez przeciążenie szablonu dla C-łańcuchów:

```
inline const char* max (const char* a, const char* b)
{
    return std::strcmp(a,b) > 0 ? a : b;
}
```

Przeciążenie szablonu może także dotyczyć wskaźników. Możemy w ten sposób sprawić, że jeśli szablon `max()` zostanie wywołany dla wskaźników, porównane zostaną wskazywane przez nie obiekty, a nie ich adresy. Na przykład:

```
template <typename T>
T* const& max (T* const& a, T* const& b)
{
    return *a > *b ? a : b;
}
```

Zwróćmy uwagę, że jeśli wskaźnik ma zostać przekazany jako stała referencja, słowo kluczowe `const` musi zostać umieszczone po znaku gwiazdki. W przeciwnym razie zadeklarowany zostanie wskaźnik do stałej (patrz także podrozdział 4.4.6).

Przeciążając szablony funkcji powinniśmy wprowadzać jedynie niezbędne modyfikacje, takie jak zmiany liczby parametrów czy jawne ich określenie. W przeciwnym razie wprowadzone zmiany mogą stać się powodem powstawania nieoczekiwanych efektów. Dlatego też w naszym przykładzie argumenty wszystkich przeciążonych implementacji powinny być przekazywane poprzez stałe referencje:

```
// tmpl/max2.cpp
#include <iostream>
#include <cstring>

// zwraca większą z dwóch wartości dowolnego typu
template <typename T>
inline const T& max (const T& a, const T& b)
{
    std::cout << "max<>() dla T" << std::endl;
    return  a < b  ?  b : a;
}

// dla dwóch wskaźników
template <typename T>
inline T* const& max (T* const& a, T* const& b)
{
    std::cout << "max<>() dla T*" << std::endl;
    return  *a < *b  ?  b : a;
}

// dla dwóch C-łańcuchów
inline const char* const& max (const char* const& a,
                               const char* const& b)
{
    std::cout << "max<>() dla char*" << std::endl;
    return  std::strcmp(a,b) < 0  ?  b : a;
}
```

Wykonanie przedstawionego poniżej programu:

```
// tmpl/max2.cpp
#include <iostream>
#include <string>
#include "max2.hpp"

int main()
```

```
{
    int a=7;                                // dwie zmienne typu int
    int b=11;
    std::cout << max(a,b) << std::endl;     // max() dla dwóch argumentów typu int

    std::string s="hello";                  // dwa łańcuchy
    std::string t="holla";
    std::cout << ::max(s,t) << std::endl;  // max() dla dwóch argumentów typu string

    int* p1 = &b;                           // dwa wskaźniki
    int* p2 = &a;
    std::cout << *max(p1,p2) << std::endl; // max() dla dwóch wskaźników

    const char* s1 = "hello";               // dwa C-łańcuchy
    const char* s2 = "otto";
    std::cout << max(s1,s2) << std::endl;  // max() dla dwóch C-łańcuchów
}
```

spowoduje wyświetlenie następujących napisów:

```
max<>() dla T
11
max<>() dla T
holla
max<>() dla T*
11
max<>() dla char*
otto
```

7.2.6. Zmienne lokalne

Szablony funkcji mogą posiadać zmienne lokalne typu będącego parametrem szablonu. Na przykład szablon funkcji zamieniającej wartości dwóch parametrów może zostać zaimplementowany w następujący sposób (proszę porównać z implementacją funkcji `swap()` przedstawioną na stronie 185).

```
template <typename T>
void swap (T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}
```

Zmienne lokalne mogą być również statyczne. W takim przypadku tworzone są zmienne statyczne wszystkich typów, dla których wywoływany jest szablon funkcji.

7.2.7. Podsumowanie

- Szablony są schematami kodu kompilowanego po wybraniu określonego typu danych.
- Tworzenie kodu w języku C++ na podstawie szablonu nazywamy *tworzeniem instancji szablonu*.

- Szablony mogą posiadać wiele parametrów.
- Szablony funkcji mogą być przeciążane.

7.3. Szablony klas

W takim sam sposób, jak parametryzowane są typy funkcji, mogą również być parametryzowane typy w klasach. Możliwość taka jest szczególnie przydatna w przypadku kontenerów używanych do zarządzania obiektami pewnego typu. Szablony klas możemy wykorzystać do implementacji kontenerów, dla których typ elementów nie jest jeszcze znany. W terminologii obiektowej szablony klas nazywane są *klasami parametryzowanymi*.

Implementacja szablonów klas zostanie omówiona na przykładzie klasy stosu. Implementacja ta wykorzystywać będzie szablon klasy `vector<>`, dostępny w bibliotece standardowej (patrz podrozdziały 3.5.1 i 9.1.1).

7.3.1. Implementacja szablonu klasy Stack

Podobnie, jak w przypadku szablonów funkcji, także deklaracja i definicja szablonu klasy umieszczana jest zwykle w pliku nagłówkowym. Zawartość pliku nagłówkowego klasy `Stack` jest następująca:

```
// tmpl/stack1.hpp
#include <vector>
#include <stdexcept>

// **** Początek przestrzeni nazw CPPBook ********************************
namespace CPPBook {

template <typename T>
class Stack {
  private:
    std::vector<T> elems;  // elementy

  public:
    Stack();               // konstruktor
    void push(const T&);   // umieszcza nowy element na szczycie
    void pop();            // usuwa element ze szczytu
    T top() const;         // zwraca element znajdujący się na szczycie
};

// konstruktor
template <typename T>
Stack<T>::Stack()
{
    // nie wykonuje żadnych instrukcji
}

template <typename T>
void Stack<T>::push(const T& elem)
```

```
{
    elems.push_back(elem);    // umieszcza kopię na szczycie stosu
}

template<typename T>
void Stack<T>::pop()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): pusty stos");
    }
    elems.pop_back();         // usuwa element ze szczytu stosu
}

template <typename T>
T Stack<T>::top() const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): pusty stos");
    }
    return elems.back();      // zwraca kopię elementu znajdującego się na szczycie
}

} // **** Koniec przestrzeni nazw CPPBook ********************************
```

Deklaracja szablonu klasy

Podobnie, jak w przypadku szablonu funkcji, deklaracja szablonu klasy poprzedzona jest określeniem parametru typu `T` (parametrów szablonu może być oczywiście więcej):

```
template <typename T>
class Stack {
  ...
};
```

Zamiast słowa kluczowego `typename` może być też użyte słowo `class`:

```
template <class T>
class Stack {
  ...
};
```

Wewnątrz klasy typ `T` może być używany w deklaracjach składowych klasy i funkcji składowych w taki sam sposób, jak każdy zwykły typ. W naszym przykładzie elementy stosu zarządzane są wewnątrz klasy za pomocą wektora o elementach typu `T` (szablon jest zaimplementowany z użyciem innego szablonu), funkcja `push()` używa referencji stałej typu `T` jako parametru, a funkcja `top()` zwraca obiekt typu `T`.

Klasa stosu posiada typ `Stack<T>`, gdzie `T` jest parametrem szablonu. Typ ten musi zostać użyty za każdym razem, gdy posługujemy się klasą stosu. Nazwy `Stack` używamy jedynie podczas definiowania klasy oraz jej konstruktorów i destruktorów:

```
class Stack {
  ...
};
```

Poniższy przykład przedstawia sposób użycia szablonu klasy jako typu parametrów funkcji lub wartości przez nie zwracanych (w deklaracjach konstruktora kopiującego i operatora przypisania)[3]:

```
template <typename T>
class Stack {
     ...
     Stack (const Stack<T>&); // konstruktor kopiujący
     Stack<T>& operator= (const Stack<T>&); // operator przypisania
     ...
};
```

Implementacja funkcji składowych

Definiując funkcję składową szablonu klasy musimy określić jej przynależność do szablonu. Przykład implementacji funkcji `push()` pokazuje, że nazwa funkcji musi zostać poprzedzona pełnym typem szablonu `Stack<T>`:

```
template <typename T>
void Stack<T>::push(const T& elem)
{
     elems.push_back(elem); // umieszcza kopię na stosie
}
```

Implementacja ta w rzeczywistości deleguje operację do odpowiedniej funkcji szablonu klasy `vector` używanego wewnętrznie do zarządzania elementami stosu. Szczyt stosu jest w tym przypadku równoważny elementowi znajdującemu się na końcu wektora.

Zauważmy, że funkcja `pop()` usuwa element ze szczytu stosu, ale go nie zwraca. Operacji tej odpowiada funkcja `pop_back()` wektora. Powodem takiego zachowania obu funkcji jest niebezpieczeństwo związane z możliwością wyrzucenia wyjątku (patrz podrozdział 4.7.10). Implementacja funkcji `pop()`, która zwraca usunięty element i charakteryzuje się wysokim poziomem bezpieczeństwa, nie jest możliwa. Przeanalizujmy działanie wersji funkcji `pop()` zwracającej element usunięty ze szczytu stosu:

```
template <typename T>
T Stack<T>::pop()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): pusty stos");
    }
    T elem = elems.back(); // tworzy kopię elementu na szczycie
    elems.pop_back(); // usuwa element ze szczytu
    return elem; // zwraca kopię elementu, który znajdował się na szczycie
}
```

Niebezpieczeństwo polega na możliwości wyrzucenia wyjątku przez konstruktor kopiujący tworzący wartość zwracaną przez funkcję. Ponieważ wcześniej element został już usunięty ze stosu, nie ma możliwości przywrócenie wyjściowego stanu stosu, gdy

[3] Standard języka C++ definiuje pewne zasady, które pozwalają określić kiedy użycie typu `Stack` zamiast typu `Stack<T>` jest wystarczające wewnątrz deklaracji klasy. W praktyce łatwiej jednak jest używać zawsze typu `Stack<T>`, gdy wymagany jest typ klasy.

pojawi się wyjątek. Należy podjąć decyzję, czy bardziej interesuje nas bezpieczne wykonanie funkcji, czy możliwość uzyskania zwróconego obiektu[4].

Zwróćmy także uwagę, że funkcje składowe `pop_back()` i `back()` (ta druga zwraca ostatni element wektora) zachowują się w nieokreślony sposób, gdy wektor jest pusty (patrz strony 479 i 481). Dlatego też funkcje szablonu klasy `Stack` sprawdzają najpierw, czy stos jest pusty, i wyrzucają wyjątek `std::out_of_range` (patrz podrozdział 4.7.9):

```
template<typename T>
T Stack<T>::top() const
{
    if (elems.empty()) {
         throw std::out_of_range("Stack<>::top(): pusty stos");
    }
    return elems.back(); // zwraca kopię szczytu stosu
}
```

Funkcje składowe szablonu klasy mogą być też implementowane wewnątrz deklaracji szablonu:

```
template <typename T>
class Stack {
     ...
     void push(const T& elem) {
        elems.push_back(elem); // umieszcza na stosie kopię obiektu
    }
    ...
};
```

7.3.2. Zastosowanie szablonu klasy Stack

Deklarując obiekt szablonu klasy musimy zawsze określić typ, który będzie parametrem szablonu:

```
// tmpl/stest1.cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include "stack1.hpp"

int main()
{
    try {
        CPPBook::Stack<int>         intStack;       // stos liczb całkowitych
        CPPBook::Stack<std::string> stringStack;    // stos łańcuchów

        // operacje na stosie liczb całkowitych
        intStack.push(7);
        std::cout << intStack.top() << std::endl;
        intStack.pop();

        // operacje na stosie łańcuchów
```

[4] Zagadnienie to zostało omówione po raz pierwszy przez Toma Cargilla w *CargillExceptionSafety* oraz przedstawione zostało w *SutterExceptional*.

```
        std::string s = "hello";
        stringStack.push(s);
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
    }
    catch (const char* msg) {
        std::cerr << "Wyjątek: " << msg << std::endl;
        return EXIT_FAILURE;
    }
}
```

Instancja szablonu klasy tworzona jest dla podanego typu. Deklaracja stosu `stringStack` powoduje wygenerowanie kodu klasy `Stack` dla typu `std::string` oraz dla wszystkich funkcji składowych wywoływanych w programie.

Zwróćmy uwagę, że w przypadku tworzenia instancji szablonów klas generowany jest kod jedynie dla tych funkcji składowych, które rzeczywiście są wywoływane. Jest to istotne nie tylko z punktu widzenia efektywności. Umożliwia także stosowanie szablonu klasy dla typów, które nie dysponują operacjami wymaganymi przez wszystkie funkcje składowe szablonu pod warunkiem, że wywoływane są tylko te funkcje, dla których dostępne są odpowiednie operacje. Przykładem może być szablon klasy, którego niektóre funkcje składowe używają operatora porównania (<). Dopóty, dopóki funkcje te nie są wywoływane, szablon może być wykorzystywany dla typu, który nie posiada zdefiniowanego operatora <.

W naszym przykładzie kod został wygenerowany na podstawie szablonu dla dwóch klas, `int` oraz `std::string`. Jeśli szablon klasy posiada składowe statyczne, zostaną utworzone dwa ich zestawy.

Dla każdego użytego typu szablon klasy określa typ, który może być używany w programie jak każdy zwykły typ:

```
void foo(const CPPBook::Stack<int>& s) // parametr s jest stosem liczb całkowitych
{
    CPPBook::Stack<int> istack[10]; // istack jest tablicą dziesięciu stosów liczb
    całkowitych
    ...
}
```

W praktyce często wykorzystuje się polecenie `typedef`, aby łatwiej posługiwać się szablonami klas w programie:

```
typedef CPPBook::Stack<int> IntStack;

void foo(const IntStack& s) // parametr s jest stosem liczb całkowitych
{
    IntStack istack[10]; // istack jest tablicą dziesięciu stosów liczb całkowitych
    ...
}
```

Parametry szablonów mogą być dowolnymi typami. Na przykład wskaźnikami do wartości typu `float` lub nawet stosami liczb całkowitych:

```
CPPBook::Stack<float*> floatPtrStack; // stos wskaźników do wartości typu float
CPPBook::Stack<CPPBook::Stack<int> > intStackStack; // stos stosów liczb całkowitych
```

Istotne jest jedynie, by typy dysponowały odpowiednimi operacjami.

Zauważmy, że dwa następujące po sobie znaki zamykające szablonu muszą być oddzielone odstępem. W przeciwnym razie kompilator zinterpretuje je jako operator >> i zasygnalizuje błąd składni:

```
// BŁĄD: niedozwolone użycie operatora >>
CPPBook::Stack<CPPBook::Stack<int>> intStackStack;
```

7.3.3. Specjalizacja szablonów klas

Przez *specjalizację* szablonów klas rozumiemy ich osobną implementację dla różnych typów. Podobnie, jak w przypadku przeciążania szablonów klas (patrz podrozdział 7.2.5), umożliwia to optymalizację implementacji dla pewnych typów lub uniknięcie niepożądanego zachowania na skutek utworzenia instancji szablonu dla pewnego typu. Tworząc specjalizację szablonu klasy musimy pamiętać o utworzeniu specjalizacji wszystkich jego funkcji składowych. Możliwe jest stworzenie specjalizacji pojedynczej funkcji składowej, ale uniemożliwia ono stworzenie specjalizacji całej klasy.

Jawna specjalizacja wymaga dodania słowa `template<>` przed deklaracją klasy oraz typu szablonu po nazwie klasy:

```
template<>
class Stack<std::string> {
  ...
};
```

Definicja każdej funkcji składowej musi rozpoczynać się słowem `template<>`, a typ `T` musi zostać zastąpiony określonym typem szablonu:

```
template<>
void Stack<std::string>::push(const std::string& elem)
{
    elems.push_back(elem); // umieszcza kopię na stosie
}
```

A oto kompletny przykład specjalizacji szablonu klasy `Stack<>` dla typu `std::string`:

```
// tmpl/stack2.hpp
#include <deque>
#include <string>
#include <stdexcept>

// **** Początek przestrzeni nazw CPPBook *********************************
namespace CPPBook {

template<>
class Stack<std::string> {
  private:
    std::deque<std::string> elems;  // elementy

  public:
```

```
    Stack() {                       // konstruktor
    }
    void push(const std::string&);  // umieszcza nowy element na szczycie
    void pop();                     // usuwa element ze szczytu
    std::string top() const;        // zwraca element znajdujący się na szczycie
};

void Stack<std::string>::push(const std::string& elem)
{
    elems.push_back(elem);          // umieszcza element na szczycie
}

void Stack<std::string>::pop()
{
    if (elems.empty()) {
        throw std::out_of_range
                ("Stack<std::string>::pop(): pusty stos");
    }
    elems.pop_back();               // usuwa element ze szczytu
}

std::string Stack<std::string>::top() const
{
    if (elems.empty()) {
        throw std::out_of_range
                ("Stack<std::string>::top(): pusty stos");
    }
    return elems.back();            // zwraca kopię elementu znajdującego się na szczycie
}

} // **** Koniec przestrzeni nazw CPPBook ********************************
```

Specjalizacja szablonu dla łańcuchów zastąpiła używany wewnętrznie wektor kolejką. Nie jest to jakaś przełomowa zmiana, ale ilustruje ona możliwość zupełnie innej implementacji szablonu klasy dla określonego typu.

Zakresy wartości numerycznych zdefiniowane w bibliotece standardowej stanowią kolejny przykład zastosowania specjalizacji szablonów (patrz podrozdział 9.1.4).

Częściowa specjalizacja

Możliwe jest też tworzenie *częściowych specjalizacji* szablonów. Na przykład dla szablonu klasy:

```
template <typename T1, typename T2>
class MyClass {
  ...
};
```

możemy utworzyć następującą specjalizację częściową:

```
// specjalizacja częściowa: takie same parametry typu
template <typename T>
class MyClass<T,T> {
  ...
};
```

```
// specjalizacja częściowa: drugim parametrem jest typ int
template <typename T>
class MyClass<T,int> {
  ...
};

// specjalizacja częściowa: oba parametry są wskaźnikami
template <typename T1, typename T2>
class MyClass<T1*,T2*> {
  ...
};
```

Powyższych szablonów używamy w następujący sposób:

```
MyClass<int, float> mif; // MyClass<T1,T2>
MyClass<float, float> mff; // MyClass<T,T>
MyClass<float, int> mfi; // MyClass<T,int>
MyClass<int*, float*> mp; // MyClass<T1*,T2*>
```

Jeśli do deklaracji pasuje kilka specjalizacji częściowych, nie jest ona jednoznaczna:

```
MyClass<int,int> m; //BŁĄD: pasują MyClass<T,T>
//i MyClass<T,int>
MyClass<int*,int*> m; //BŁĄD: pasują MyClass<T,T>
//i MyClass<T1*,T2*>
```

Druga z powyższych deklaracji może zostać rozstrzygnięta, jeśli zdefiniowana zostanie specjalizacja dla wskaźników tego samego typu:

```
template <typename T>
class MyClass<T*,T*> {
  ...
};
```

7.3.4. Domyślne parametry szablonu

W przypadku szablonów klas możemy zdefiniować domyślne wartości parametrów (nie jest to możliwe dla szablonów funkcji). Domyślne wartości parametrów szablonu mogą odnosić się do pozostałych jego parametrów.

Na przykład możemy sparametryzować kontener używany do zarządzania elementami stosu i zdefiniować wektor jako domyślny typ kontenera:

```
// tmpl/stack3.hpp
#include <vector>
#include <stdexcept>

// **** Początek przestrzeni nazw CPPBook ********************************
namespace CPPBook {

template <typename T, typename CONT = std::vector<T> >
class Stack {
  private:
    CONT elems;    // elementy

  public:
```

```
    Stack();              // konstruktor
    void push(const T&);  // umieszcza nowy element na szczycie
    void pop();           // usuwa element ze szczytu
    T top() const;        // zwraca element znajdujący się na szczycie
};

// konstruktor
template <typename T, typename CONT>
Stack<T,CONT>::Stack()
{
    // nie wykonuje żadnych instrukcji
}

template <typename T, typename CONT>
void Stack<T,CONT>::push(const T& elem)
{
    elems.push_back(elem);    // umieszcza kopię na szczycie stosu
}

template <typename T, typename CONT>
void Stack<T,CONT>::pop()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): pusty stos");
    }
    elems.pop_back();         // usuwa element ze szczytu stosu
}

template <typename T, typename CONT>
T Stack<T,CONT>::top() const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): pusty stos");
    }
    return elems.back();      // zwraca kopię elementu znajdującego się na szczycie
}

} // **** Koniec przestrzeni nazw CPPBook ********************************
```

Tak zdefiniowanego stosu możemy używać w ten sam sposób, co poprzednich wersji szablonu, ale z dodatkową możliwością określenia innego typu kontenera elementów:

```
// tmpl/stest3.cpp
#include <iostream>
#include <deque>
#include <cstdlib>
#include "stack3.hpp"

int main()
{
    try {
        // stos liczb całkowitych
        CPPBook::Stack<int> intStack;
        // stos liczb zmiennoprzecinkowych
        CPPBook::Stack<double,std::deque<double> > dblStack;

        // operacje na stosie liczb całkowitych
```

```
        intStack.push(7);
        std::cout << intStack.top() << std::endl;
        intStack.pop();

        // operacje na stosie liczb zmiennoprzecinkowych
        dblStack.push(42.42);
        std::cout << dblStack.top() << std::endl;
        dblStack.pop();
        std::cout << dblStack.top() << std::endl;
        dblStack.pop();
    }
    catch (const char* msg) {
        std::cerr << "Wyjątek: " << msg << std::endl;
        return EXIT_FAILURE;
    }
}
```

Za pomocą deklaracji:

```
CPPBook::Stack<double,std::deque<double> >
```

utworzony został stos wartości zmiennoprzecinkowych, wykorzystujący kolejkę jako wewnętrzny kontener elementów.

7.3.5. Podsumowanie

- Za pomocą szablonów klasy mogą być implementowane dla typów, które nie zostały jeszcze zdefiniowane.
- Zastosowanie szablonów klas umożliwia parametryzację kontenerów ze względu na typ ich elementów.
- W przypadku użycia szablonu klasy generowany jest kod tylko dla tych funkcji składowych, które rzeczywiście są wywoływane.
- Implementacja szablonów klas może być wyspecjalizowana dla pewnych typów. Możliwa jest także częściowa specjalizacja szablonu klasy.
- Parametry szablonów klas mogą posiadać wartości domyślne.

7.4. Inne parametry szablonów

Parametry szablonów nie muszą być typami. Mogą być elementarnymi wartościami, podobnie jak parametry funkcji. W ten sposób możemy zdefiniować grupę funkcji lub klas sparametryzowaną względem pewnych wartości.

7.4.1. Przykład zastosowania innych parametrów szablonów

W poniższym przykładzie zdefiniujemy kolejną wersję szablonu stosu, która będzie zarządzać elementami stosu za pomocą zwykłej tablicy o stałym rozmiarze. Unikniemy w ten sposób kosztów związanych z dynamicznym zarządzaniem pamięcią.

Deklaracja tej wersji szablonu przedstawia się następująco:

```
// tmpl/stack4.hpp
#include <stdexcept>

// **** Początek przestrzeni nazw CPPBook ********************************
namespace CPPBook {

template <typename T, int MAXSIZE>
class Stack {
  private:
    T elems[MAXSIZE];     // elementy
    int numElems;         // bieżąca liczba elementów na stosie

  public:
    Stack();              // konstruktor
    void push(const T&);  // umieszcza nowy element na szczycie
    void pop();           // usuwa element ze szczytu
    T top() const;        // zwraca element znajdujący się na szczycie
};

// konstruktor
template <typename T, int MAXSIZE>
Stack<T,MAXSIZE>::Stack()
  : numElems(0)    // brak elementów
{
    // nie wykonuje żadnych instrukcji
}

template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push(const T& elem)
{
    if (numElems == MAXSIZE) {
        throw std::out_of_range("Stack<>::push(): pełen stos");
    }
    elems[numElems] = elem;   // umieszcza element w tablicy
    ++numElems;               // zwiększa liczbę elementów na stosie
}

template<typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::pop()
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::pop(): pusty stos");
    }
    --numElems;               // zmniejsza liczbę elementów
}

template <typename T, int MAXSIZE>
T Stack<T,MAXSIZE>::top() const
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::top(): pusty stos");
    }
    return elems[numElems-1];  // zwraca kopię elementu znajdującego się na szczycie
}

} // **** Koniec przestrzeni nazw CPPBook ********************************
```

Drugi z parametrów szablonu stosu, `MAXSIZE`, określa rozmiar stosu. Używany jest nie tylko w celu zadeklarowania odpowiedniej wielkości tablicy, ale także przez funkcję `push()` w celu sprawdzenia, czy stos jest pełen.

Korzystając z tej wersji szablonu stosu musimy wyspecyfikować typ elementów stosu oraz jego wielkość:

```
// tmpl/stest4.cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include "stack4.hpp"

int main()
{
    try {
        CPPBook::Stack<int,20>         int20Stack;   // stos 20 wartości całkowitych
        CPPBook::Stack<int,40>         int40Stack;   // stos 40 wartości całkowitych
        CPPBook::Stack<std::string,40> stringStack;  // stos 40 łańcuchów

        // operacje na stosie liczb całkowitych
        int20Stack.push(7);
        std::cout << int20Stack.top() << std::endl;
        int20Stack.pop();

        // operacje na stosie łańcuchów
        std::string s = "hello";
        stringStack.push(s);
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
    }
    catch (const char* msg) {
        std::cerr << "Wyjątek: " << msg << std::endl;
        return EXIT_FAILURE;
    }
}
```

Warto zauważyć, że w powyższym przykładzie stosy `int20Stack` i `int40Stack` posiadają różne typy i nie mogą być przypisywane bądź używane jeden zamiast drugiego.

Parametry szablonu mogą posiadać wartości domyślne:

```
template <typename T = int, int MAXSIZE = 100 >
class Stack {
  ...
};
```

Powyższy przykład nie jest zbyt użyteczny, Domyślne wartości powinny być zgodne z intuicyjnym oczekiwaniem użytkownika. Ani typ `int`, ani wielkość stosu równa 100 nie są intuicyjne. W takim przypadku lepiej pozostawić specyfikacje wartości parametrów programiście aplikacji.

7.4.2. Ograniczenia parametrów szablonów

Z użyciem innych parametrów szablonów związane są pewne ograniczenia. Parametry szablonów mogą być, oprócz typów, stałymi wyrażeniami całkowitymi, adresami obiektów lub funkcji, które są globalnie dostępne w programie.

Liczby zmiennoprzecinkowe i obiekty, których typem są klasy, nie mogą być parametrami szablonów klas:

```
template <double VAT> // BŁĄD: wartości zmiennoprzecinkowe
double process(double v) // nie są dozwolone jako parametry szablonów
{
     return v * VAT;
}

template <std::string name> // BŁĄD: obiekty klas
class MyClass { // nie są dozwolone jako parametry szablonów
  ...
};
```

Literały znakowe w roli parametrów szablonów również mogą być przyczyną problemów:

```
template <typename T, const char* name>
class MyClass {
  ...
};

MyClass<int, "hello"> x; // BŁĄD: literał "hello" nie jest dozwolony
```

Literały znakowe nie są bowiem globalnymi obiektami dostępnymi w dowolnym punkcie programu. Jeśli literał `"hello"` zostanie zdefiniowany w dwóch różnych modułach, to powstaną dwa różne łańcuchy.

W takiej sytuacji nie pomoże nawet użycie globalnego wskaźnika:

```
template <typename T, const char* name>
class MyClass {
  ...
};

const char* s = "hello";
MyClass<int,s> x; // BŁĄD: "hello" jest zawsze statyczne
```

Chociaż wskaźnik `s` jest globalnie dostępny, to wskazywany przez niego łańcuch nadal nie jest globalnie dostępny.

Rozwiązanie tego problemu jest następujące:

```
template <typename T, const char* name>
class MyClass {
  ...
};

extern const char s[] = "hello";
```

```
void foo() {
    MyClass<int,s> x; // poprawne
    ...
}
```

Globalna tablica `s` została zainicjowana łańcuchem `"hello"` i dlatego `s` reprezentuje globalnie dostępny łańcuch `"hello"`.

7.4.3. Podsumowanie

- Szablony mogą posiadać parametry, które nie są typami.
- Parametry te nie mogą być wartościami zmiennoprzecinkowymi lub obiektami. Nie mogą być też lokalne.
- Zastosowanie literałów znakowych jako parametrów szablonów możliwe jest w ograniczonym zakresie.

7.5. Inne zagadnienia związane z szablonami

W podrozdziale tym przedstawione zostaną inne aspekty szablonów, które wymagają omówienia. Przedstawione zostanie zastosowanie słowa kluczowego `typename` oraz możliwość definiowania składowych jako szablonów. Następnie przyjrzymy się sposobom implementacji polimorfizmu za pomocą szablonów.

7.5.1. Słowo kluczowe typename

Słowo kluczowe `typename` zostało wprowadzone podczas standaryzacji języka C++, aby umożliwić określenie, że dany symbol jest typem szablonu klasy. Demonstruje to poniższy przykład:

```
template <typename T>
class MyClass {
    typename T::SubType * ptr;
    ...
};
```

W powyższym przypadku słowo kluczowe `typename` zostało zastosowane w celu określenia, że `SubType` jest typem zdefiniowanym w klasie `T`. W ten sposób `ptr` został zadeklarowany jako wskaźnik do typu `SubType`.

Gdyby pominąć słowo kluczowe `typename`, kompilator przyjąłby, że symbol `SubType` reprezentuje statyczną wartość (zmienną lub obiekt) klasy `T`. Wtedy wyrażenie:

```
T::SubType * ptr
```

zostałoby zinterpretowane jako operacja mnożenia tej wartości przez wartość `ptr`.

Typowym przykładem zastosowania słowa kluczowego `typename` jest szablon funkcji, która używa iteratorów w celu dostępu do elementów kontenera STL (patrz podrozdział 3.5.4):

```
// tmpl/printcoll.hpp
#include <iostream>

// wyświetla elementy kontenera STL
template <typename T>
void printcoll(const T& coll)
{
    // wyświetla liczbę elementów kontenera
    std::cout << "liczba elementów: " << coll.size() << std::endl;

    // wyświetla elementy
    std::cout << "elementy: ";
    typename T::const_iterator pos;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

Parametrem powyższego szablonu funkcji może być kontener STL typu `T`. Szablon funkcji wyświetla elementy tego kontenera, posługując się lokalnym iteratorem. Dla iteratora tego określony został pomocniczy typ zdefiniowany przez kontener. Jego deklaracja wymaga użycia słowa kluczowego `typename`:

```
typename T::const_iterator pos; // const_iterator jest pomocniczym typem T
```

7.5.2. Składowe jako szablony

Składowe klas mogą być także szablonami. Dotyczy to tak wewnętrznych klas pomocniczych, jak i funkcji składowych.

Zastosowanie tej możliwości wyjaśnimy na przykładzie klasy `Stack<>`. Stosy mogą być sobie przypisywane tylko wtedy, gdy posiadają elementy takiego samego typu. Przypisywanie stosów o różnych typach elementów nie jest możliwe, ponieważ domyślny operator przypisania wymaga, by oba jego argumenty były tego samego typu.

Możliwość przypisywania stosów o różnych typach elementów możemy uzyskać poprzez zdefiniowanie operatora przypisania za pomocą szablonu. Deklaracja klasy `Stack<>` wyglądać będzie wtedy następująco:

```
// tmpl/stack5decl.hpp
template <typename T>
class Stack {
  private:
    std::deque<T> elems;   // elementy

  public:
    void push(const T&);   // umieszcza nowy element na szczycie
    void pop();            // usuwa element ze szczytu
    T top() const;         // zwraca element znajdujący się na szczycie
    bool empty() const {   // sprawdza, czy stos jest pusty
        return elems.empty();
    }
```

```
    // przypisanie stosu o elementach typu T2
    template <typename T2>
    Stack<T>& operator= (Stack<T2> const&);
};
```

W deklaracji klasy zaszły następujące zmiany:

- Pojawiła się deklaracja operatora przypisania stosu o innym typie elementów `T2`.
- Szablon wykorzystuje kolejkę ze względu na sposób implementacji operatora przypisania.

Operator przypisania stosu o elementach innego typu został zdefiniowany w następujący sposób:

```
// tmpl/stack5assign.hpp
template <typename T>
 template <typename T2>
Stack<T>& Stack<T>::operator= (const Stack<T2>& op2)
{
    if ((void*)this == (void*)&op2) {    // przypisanie do samego siebie?
        return *this;
    }

    Stack<T2> tmp(op2);              // tworzy kopię przypisywanego stosu

    elems.clear();                   // usuwa istniejące elementy
    while (!tmp.empty()) {           // kopiuje elementy
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
```

Przyjrzyjmy się najpierw składni definicji szablonu funkcji należącej do szablonu klasy. Szablon o parametrze `T2` został zdefiniowany wewnątrz szablonu o parametrze `T`:

```
template <typename T>
 template <typename T2>
...
```

Mogłoby się wydawać, że implementacja takiej funkcji będzie polegać na bezpośrednim dostępie do elementów stosu `op2` i skopiowaniu ich. Jednak zauważmy, że instancje szablonów utworzone dla różnych typów same są różnymi typami. Dlatego też `Stack<T2>` posiada inny typ niż stos, dla którego wywoływany jest operator. Dostęp do elementów stosu `op2` możliwy jest jedynie za pośrednictwem publicznego interfejsu. W tym celu musi nam wystarczyć funkcja `top()`. Jednak, aby kolejne elementy kopiowanego stosu mogły pojawić się na jego szczycie, konieczne jest także użycie funkcji `pop()` usuwającej elementy stosu. Dlatego najpierw należy utworzyć kopię stosu `op2`. Ponieważ funkcja `top()` zwraca elementy stosu w odwrotnej kolejności do porządku, w jakim zostały na nim umieszczone, zmuszeni jesteśmy użyć kontenera, który umożliwia wstawianie elementów na początek. Skorzystamy w tym celu z kolejki dysponującej funkcją `push_front()`.

Zauważmy również, że funkcja operatora zachowuje kontrolę typów. Stosy nie mogą być przypisywane, gdy nie jest możliwe przypisanie ich elementów. Jeśli spróbujemy przypisać stos wartości całkowitych stosowi łańcuchów, w poniższym wierszu pojawi się błąd:

```
elems.push_front(tmp.top()); // wstawia kopię na początek kolejki
```

ponieważ funkcja `tmp.top()` zwraca łańcuch, który nie może zostać użyty jako typ `int`.

Zwróćmy uwagę, że szablon operatora przypisania nie ukrywa domyślnego operatora przypisania. Operator ten jest nadal dostępny i wywoływany dla operacji przypisania dwóch stosów tego samego typu.

Implementację szablonu stosu możemy zmodyfikować tak, by wykorzystywała wektor. Deklaracja szablonu stosu będzie wyglądać następująco:

```
template <typename T, typename CONT = std::deque<T> >
class Stack {
  private:
    CONT elems;   // elementy

  public:
    Stack(); // konstruktor
    void push(const T&);   // umieszcza nowy element na szczycie
    void pop();            // usuwa element ze szczytu
    T top() const;         // zwraca element znajdujący się na szczycie
    bool empty() const {   // sprawdza, czy stos jest pusty
        return elems.empty();
    }

    // przypisanie stosu o elementach typu T2
    template <typename T2, typename CONT2>
    Stack<T,CONT>& operator= (const Stack<T2,CONT2> &);
};
```

Ponieważ kompilator tworzy kod jedynie dla tych funkcji, które są rzeczywiście używane, możemy utworzyć stos wykorzystujący wektor do przechowywania elementów:

```
// stos o elementach typu int wykorzystujący wektor
CPPBook::Stack<int,std::vector<int> > vStack;
...
vStack.push(42);
vStack.push(7);
std::cout << vStack.top() << std::endl;
vStack.pop();
```

Dopóty, dopóki nie próbujemy przypisać stosowi drugiego stosu o innym typie elementów, program będzie działać poprawnie.

Kompletny kod tego przykładu znajduje się w plikach *tmpl/stack6.hpp* i *tmpl/stest6.cpp*. Nie należy się zrażać, jeśli kompilator zgłosi dla nich błędy. Ponieważ przykłady te wykorzystują praktycznie wszystkie najważniejsze konstrukcję języka C++ związane z szablonami, niektóre niestandardowe kompilatory nie potrafią ich poprawnie skompilować.

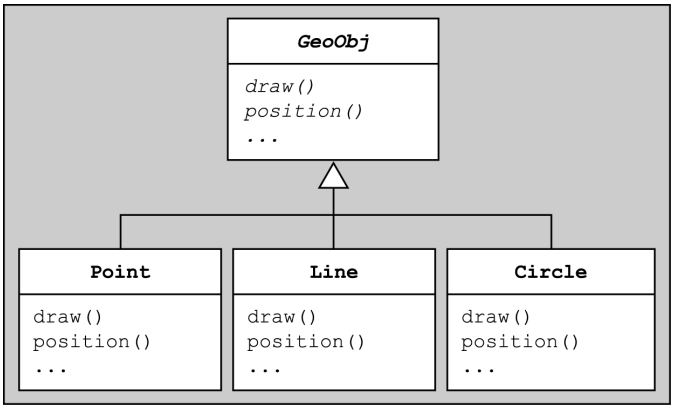
7.5.3. Polimorfizm statyczny z użyciem szablonów

Polimorfizm implementowany jest zwykle za pomocą dziedziczenia (patrz podrozdział 5.3). Możliwa jest również implementacja polimorfizmu za pomocą szablonów. Zostanie ona przedstawiona w niniejszym podrozdziale.

Polimorfizm dynamiczny

W przypadku zastosowania dziedziczenia do implementacji polimorfizmu klasa bazowa (zwykle abstrakcyjna) definiuje interfejs *uogólnienia*, który używany jest przez szereg klas konkretnych (patrz podr5ozdział 5.3).

Na przykład uogólnieniem obiektów geometrycznych może być klasa `GeoObj` (wprowadzona w podrozdziale 5.3.3), dla której tworzone są konkretne klasy pochodne (patrz rysunek 7.1).



Rysunek 7.1. *Polimorfizm dynamiczny zaimplementowany z użyciem dziedziczenia*

Program wykorzystujący uogólnienie musi używać wskaźników do obiektów klasy bazowej, co może wyglądać następująco:

```
// tmpl/dynapoly.cpp
// rysuje obiekt geometryczny
void myDraw (const GeoObj& obj)
{
    obj.draw();
}

// wyznacza odległość pomiędzy dwoma obiektami
Coord distance (const GeoObj& x1, const GeoObj& x2)
{
    Coord a = x1.position() - x2.position();
    return a.abs();
}

// rysuje heterogeniczną kolekcję obiektów geometrycznych
void drawElems (const std::vector<GeoObj*>& elems)
{
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i]->draw();
```

```
    }
}

int main()
{
    Line l;
    Circle c, c1, c2;

    myDraw(l);              // myDraw(GeoObj&) => Line::draw()
    myDraw(c);              // myDraw(GeoObj&) => Circle::draw()

    distance(c1,c2);        // distance(GeoObj&,GeoObj&)
    distance(l,c);          // distance(GeoObj&,GeoObj&)

    std::vector<GeoObj*> coll;  // heterogeniczna kolekcja
    coll.push_back(&l);         // wstawia odcinek
    coll.push_back(&c);         // wstawia okrąg
    drawElems(coll);            // rysuje kolekcję
}
```

Funkcje zostają skompilowane dla typu `GeoObj`. Decyzja o tym, która funkcja `draw()` wywołana zostanie wewnątrz funkcji `myDraw()`, zależy jednak od typu przekazanego obiektu i podejmowana jest w czasie wykonania programu. Jeśli funkcji `myDraw()` zostanie przekazany obiekt klasy `Circle`, to wywołana zostanie funkcja `Circle::draw()`. Jeśli obiekt reprezentujący odcinek, to funkcja `Line::draw()`. Podobnie wewnątrz funkcji `distance()` podejmowana jest decyzja, którą funkcję `position()` należy wywołać dla danego obiektu geometrycznego. Zastosowanie wskaźników obiektów typu `GeoObj` umożliwia także zadeklarowanie heterogenicznej kolekcji obiektów geometrycznych (bardziej zalecane jest użycie w tym celu inteligentnych wskaźników, patrz podrozdział 9.2.1).

Polimorfizm statyczny

Polimorfizm może zostać także zaimplementowany z wykorzystaniem szablonów zamiast dziedziczenia. W takim przypadku nie istnieje klasa bazowa definiująca wspólny interfejs. Właściwości obiektów są zdefiniowane niejawnie jako operacje typu będącego parametrem szablonu.

A oto przykład zastosowania szablonów do implementacji polimorfizmu z poprzedniego programu:

```
// tmpl/staticpoly.cpp
// rysuje obiekt geometryczny
template <typename GeoObj>
void myDraw (const GeoObj& obj)
{
    obj.draw();
}

// wyznacza odległość pomiędzy dwoma obiektami
template <typename GeoObj1, typename GeoObj2>
Coord distance (const GeoObj1& x1, const GeoObj2& x2)
{
    Coord a = x1.position() - x2.position();
    return a.abs();
```

```
}

// rysuje heterogeniczną kolekcję obiektów geometrycznych
template <typename GeoObj>
void drawElems (const std::vector<GeoObj>& elems)
{
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i].draw();
    }
}

int main()
{
    Line l;
    Circle c;
    Circle c1, c2;

    myDraw(l);          // myDraw<Line>(GeoObj&) => Line::draw()
    myDraw(c);          // myDraw<Circle>(GeoObj&) => Circle::draw()

    distance(c1,c2);    // distance<Circle,Circle>(GeoObj&,GeoObj&)
    distance(l,c);      // distance<Line,Circle>(GeoObj&,GeoObj&)

    // std::vector<GeoObj*> coll;  // BŁĄD: heterogeniczna kolekcja nie jest dozwolona
    std::vector<Line> coll;      // homogeniczna kolekcja
    coll.push_back(l);           // wstawia odcinek
    drawElems(coll);             // rysuje kolekcję
}
```

W przypadku dwóch pierwszych funkcji, `draw()` i `distance()`, typ `GeoObj` staje się parametrem szablonu. Stosując dwa różne parametry szablonu uzyskujemy możliwość przekazania dwóch różnych obiektów geometrycznych do funkcji `distance()`:

```
distance(l,c);      // distance<Line,Circle>(GeoObj&,GeoObj&)
```

W przypadku zastosowania szablonów nie jest już możliwe wykorzystanie heterogenicznej kolekcji obiektów. Natomiast typy elementów kolekcji nie muszą być już wskaźnikami:

```
std::vector<Line> coll; // kolekcja homogeniczna
```

W pewnych sytuacjach może okazać się to istotną zaletą.

Polimorfizm dynamiczny i statyczny

Dwie formy implementacji polimorfizmu w języku C++ możemy opisać w następujący sposób:

- Polimorfizm zaimplementowany przez zastosowanie dziedziczenia jest *powiązany* i *dynamiczny*:
  - *powiązanie* oznacza, że konkretne typy obiektów są zależne od innego typu (klasy bazowej);
  - *dynamika* polega na tym, że klasa wywoływanej funkcji zostaje ustalona dopiero podczas działania programu.

- Polimorfizm zaimplementowany z wykorzystaniem szablonów jest *niepowiązany* i *statyczny*:
  - *niepowiązanie* oznacza, że typy konkretne nie są zależne od innych typów;
  - *statyczny* polimorfizm polega na ustaleniu klasy wywoływanej funkcji podczas kompilacji programu.

Dynamiczny polimorfizm jest więc skróconą nazwą *powiązanego polimorfizmu dynamicznego*, podobnie termin *polimorfizm statyczny* stosowany jest jako skrócona wersja określenia *niepowiązany polimorfizm statyczny*. W innych językach programowania dostępne są jeszcze inne kombinacje (na przykład w języku Smalltalk występuje niepowiązany polimorfizm dynamiczny).

Zalety polimorfizmu dynamicznego są następujące:

- Umożliwia posługiwanie się heterogenicznymi kolekcjami obiektów.
- Wymaga mniej kodu (funkcje kompilowane są tylko raz dla typu `GeoObj`).
- Polimorficzne operacje mogą zostać dostarczone w postaci kodu wynikowego (kod szablonów musi zawsze być kodem źródłowym).
- Ma lepszą obsługę błędów przez kompilator (patrz podrozdział 7.6.2).

Zalety polimorfizmu statycznego wymienione zostały poniżej:

- Ma lepszą efektywność kodu (możliwa jest lepsza optymalizacja, ponieważ kod nie zawiera funkcji wirtualnych). W praktyce można uzyskać poprawę od 2 do nawet 10 razy.
- Jest niepowiązany (tworzone klasy nie są zależne od żadnego innego kodu). Dzięki temu możliwe jest użycie typów podstawowych.
- Nie wymaga stosowania wskaźników.
- Typy konkretne nie muszą implementować kompletnego interfejsu (ponieważ szablony wymagają tylko operacji, które są rzeczywiście wywoływane w programie).

Jeśli chodzi o kontrolę zgodności typów, obie formy polimorfizmu mają wady i zalety. Polimorfizm dynamiczny wymaga jawnego określenia typu uogólnienia dla konkretnego obiektu geometrycznego. W przypadku polimorfizmu statycznego każda klasa może zostać użyta jako rodzaj obiektu geometrycznego, jeśli tylko dysponuje odpowiednimi operacjami. Z drugiej jednak strony, polimorfizm dynamiczny nie gwarantuje, że homogeniczne kolekcje zawierają zawsze obiekty jednego typu. Aby sprawić, żeby kolekcja zawierała na przykład wyłącznie odcinki, należy samodzielnie zaprogramować kontrolę typów obiektów umieszczanych w kolekcji.

Powyższe uwagi skłaniają nas w praktyce do użycia polimorfizmu statycznego, przede wszystkim ze względu na jego wyższą efektywność. Natomiast gdy parametry szablonów nie są znane w momencie kompilacji lub program wymaga użycia kolekcji heterogenicznych, właściwym rozwiązaniem będzie zastosowanie polimorfizmu dynamicznego.

7.5.4. Podsumowanie

- Jeśli przy posługiwaniu się parametrem szablonu korzystamy z pomocniczego typu zdefiniowanego dla danego szablonu, to typ ten musi zostać określony za pomocą słowa kluczowego `typename`.
- Klasy wewnętrzne oraz funkcje składowe mogą być również szablonami. W ten sposób możemy uzyskać niejawną konwersję w przypadku operacji szablonów klas. Konwersja ta nie odbywa się jednak z pominięciem kontroli zgodności typów.
- Szablon operatora przypisania nie ukrywa domyślnego operatora przypisania.
- Polimorfizm może zostać zaimplementowany także za pomocą szablonów. Rozwiązanie takie posiada wady i zalety.

7.6. Szablony w praktyce

Szablony stanowią nową formę kodu źródłowego. Kompilator sprawdza składnię szablonów. Szablony stają się kodem wynikowym dopiero na skutek określenia typu będącego ich parametrem. Z koncepcją szablonów związane są pewne problemy, które zostaną omówione w bieżącym podrozdziale.

7.6.1. Kompilacja kodu szablonu

Szablony przetwarzane są przez kompilator dwukrotnie: za pierwszym razem sprawdzana jest składnia szablonu podczas kompilacji jego kodu, a kolejna kontrola ma miejsce podczas kompilacji kodu wygenerowanego dla konkretnego typu. W podrozdziale 7.2.3 wspomniane zostało już, że takie rozwiązanie wykracza poza tradycyjny model kompilacji i konsolidacji.

Dlatego też najprostszym rozwiązaniem jest umieszczanie szablonów w plikach nagłówkowych. Metoda ta posiada jednak istotne słabości:

- Ten sam kod kompilowany jest wielokrotnie. Na przykład każdy moduł wykorzystujący szablon `Stack` dla elementów typu `int` wymagać będzie ponownego skompilowania kodu szablonu. Nie tylko wydłuża to czas kompilacji, ale także umieszcza skompilowany kod szablonu w wielu plikach wynikowych. Jeśli pliki te używane są następnie do utworzenia pliku wykonywalnego, program konsolidujący powinien usunąć powtarzający się kod, ponieważ w przeciwnym razie niepotrzebnie powstanie plik wykonywalny o znacznym rozmiarze.
- Kod szablonu może być dostarczony użytkownikowi tylko w postaci kodu źródłowego. Rozwiązanie takie nie jest do przyjęcia, gdy kod szablonu zawiera istotne dla firmy rozwiązania technologiczne objęte prawem autorskim.

Istnieją dwie inne metody posługiwania się szablonami, które zostaną teraz omówione. Każda z nich posiada pewne wady. W przypadku szablonów, przynajmniej na razie, nie istnieje więc rozwiązanie doskonałe.

Dostępnych jest także szereg rozwiązań specyficznych dla poszczególny producentów. Na przykład rozwiązanie umożliwiające zastosowanie specjalnych poleceń preprocesora do przetwarzania szablonów. Rozwiązania takie nie będą przedmiotem naszego zainteresowania.

Jawne tworzenie instancji

Jednym ze sposobów zapobiegania wielokrotnej kompilacji tego samego szablonu jest technika jawnego tworzenia instancji.

Przy zastosowaniu tej techniki w plikach nagłówkowych umieszczamy jedynie deklaracje szablonów:

```
// tmpl/expl1.hpp
#ifndef EXPL_HPP
#define EXPL_HPP

// deklaracja szablonu funkcji max()
template <typename T>
const T& max(const T& a, const T& b);

// deklaracja szablonu klasy Stack<>
#include <vector>

// **** Początek przestrzeni nazw CPPBook ********************************
namespace CPPBook {

template <typename T>
class Stack {
  private:
    std::vector<T> elems;  // elementy
  public:
    Stack();               // konstruktor
    void push(const T&);   // umieszcza element na szczycie stosu
    void pop();            // usuwa element ze szczytu
    T top() const;         // zwraca element znajdujący się na szczycie
};

} // **** Koniec przestrzeni nazw CPPBook ********************************

#endif // EXPL_HPP
```

Definicje szablonów umieszczamy w osobnym pliku nagłówkowym, który dołącza plik nagłówkowy zawierający ich deklaracje:

```
// tmpl/expldef1.hpp
#ifndef EXPLDEF_HPP
#define EXPLDEF_HPP

#include "expl.hpp"
#include <stdexcept>

// definicja szablonu funkcji max()
template <typename T>
const T& max(const T& a, const T& b)
```

```
{
    return (a > b ? a : b);
}

// definicja funkcji szablonu klasy Stack<>

// **** Początek przestrzeni nazw CPPBook ********************************
namespace CPPBook {

// konstruktor
template <typename T>
Stack<T>::Stack()
{
    // nie wykonuje żadnych instrukcji
}

template <typename T>
void Stack<T>::push(const T& elem)
{
    elems.push_back(elem);    // umieszcza kopię obiektu na szczycie stosu
}

template<typename T>
void Stack<T>::pop()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): pusty stos");
    }
    elems.pop_back();         // usuwa element ze szczytu stosu
}

template <typename T>
T Stack<T>::top() const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): pusty stos");
    }
    return elems.back();      // zwraca kopię elementu znajdującego się na szczycie
}

} // **** Koniec przestrzeni nazw CPPBook ********************************

#endif // EXPLDEF_HPP
```

Użycie szablonów wymaga teraz jedynie dołączenia plików nagłówkowych zawierających deklaracje:

```
// tmpl/expltest1.cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include "expl.hpp"

int main()
{
    try {
        CPPBook::Stack<int>         intStack;       // stos wartości całkowitych
        CPPBook::Stack<std::string> stringStack;    // stos łańcuchów
```

```
        // operacje na stosie wartości całkowitych
        intStack.push(7);
        intStack.push(max(intStack.top(),42));  // max() dla typu int
        std::cout << intStack.top() << std::endl;
        intStack.pop();

        // operacje na stosie łańcuchów
        std::string s = "hello";
        stringStack.push(s);
        stringStack.pop();
        stringStack.pop();
    }
    catch (const char* msg) {
        std::cerr << "Wyjątek: " << msg << std::endl;
        return EXIT_FAILURE;
    }
}
```

Niezbędne instancje szablonów mogą zostać utworzone jawnie w osobnym pliku:

```
// tmpl/expldef1.cpp
#include <string>
#include "expldef.hpp"

// jawnie tworzy niezbędną instancję szablonu funkcji
template const int& max(const int&, const int&);

// jawnie tworzy niezbędną instancję szablonu klasy
template CPPBook::Stack<int>;
template CPPBook::Stack<std::string>;
```

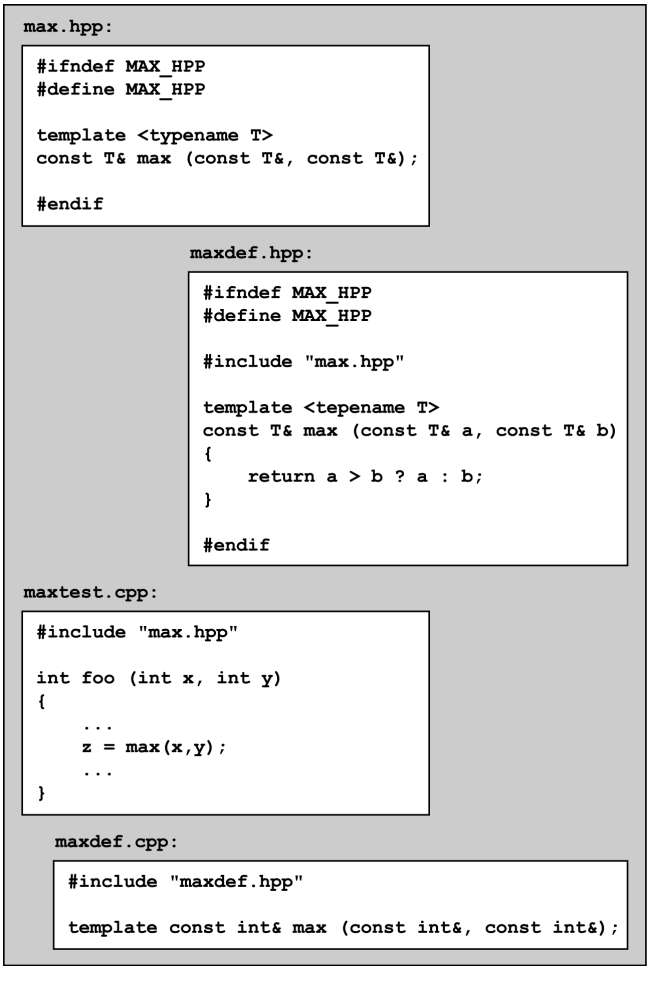
Jawne tworzenie instancji możemy zidentyfikować po tym, że nawiasy ostrokątne nie pojawiają się bezpośrednio po słowie kluczowym `template`. Jak pokazuje powyższy przykład, jawnie tworzone mogą być zarówno instancje szablonów funkcji, jak i szablonów całych klas. W tym drugim przypadku tworzone są instancje wszystkich funkcji składowych.

W przypadku szablonów klas możliwe jest także jawne tworzenie instancji poszczególnych funkcji składowych. Program może posłużyć się jawnym tworzeniem instancji w następujący sposób:

```
// tmpl/expldef2.cpp
#include <string>
#include "expldef.hpp"

// jawnie tworzy instancję szablonu funkcji
template const int& max(const int&, const int&);

// jawnie tworzy instancje niezbędnych funkcji szablonu klasy Stack<> dla typu int
template CPPBook::Stack<int>::Stack();
template void CPPBook::Stack<int>::push(const int&);
template int CPPBook::Stack<int>::top() const;
template void CPPBook::Stack<int>::pop();

// jawnie tworzy instancje niezbędnych funkcji szablonu klasy Stack<> dla typu
std::string
// - instancja funkcji top() nie jest wymagana
```

```
template CPPBook::Stack<std::string>::Stack();
template void CPPBook::Stack<std::string>::push(const std::string&);
template void CPPBook::Stack<std::string>::pop();
```

Rysunek 7.2 przedstawia organizację kodu źródłowego dla szablonu funkcji `max()`.



Rysunek 7.2. *Przykład organizacji kodu źródłowego szablonu*

Stosując procedurę jawnego tworzenia instancji szablonów możemy w pewnych sytuacjach zaoszczędzić sporo czasu. Dlatego warto umieszczać kod szablonu w dwóch osobnych plikach nagłówkowych (deklarację szablonu w jednym, a definicję w drugim). Jeśli nie chcemy stosować jawnego tworzenia instancji, wystarczy dołączyć do programu jedynie plik nagłówkowy zawierający definicję szablonu. Zaletą tej metody jest więc spora uniwersalność przy jednoczesnym braku istotnych wad.

Jeśli chcemy zachować możliwość stosowania funkcji rozwijanych w miejscu wywołania, muszą one zostać zaimplementowane w pliku zawierającym deklaracje w taki sam sposób, jak w przypadku zwykłych klas.

Model kompilacji szablonów

Jeszcze inna możliwość posługiwania się szablonami została zdefiniowana w standardzie języka C++ jako *model kompilacji szablonów*. Jeśli szablon zostanie opatrzony słowem kluczowym `export`, zostanie on automatycznie umieszczony w bazie szablonów lub w repozytorium szablonów.

Zastosowanie słowa kluczowego `export` do szablonów sprawia, że kompilator wykorzystuje repozytorium szablonów w celu sprawdzenia, czy dany szablon został już skompilowany dla danego typu lub czy jest automatycznie kompilowany i dołączany do działającego programu.

Działanie tej metody zależeć będzie w dużym stopniu od konkretnej implementacji. W chwili obecnej praktycznie nie są jeszcze dostępne kompilatory umożliwiające jej wykorzystanie.

7.6.2. Obsługa błędów

Kolejne problemy związane z użyciem szablonów ujawniają się, gdy programista popełnia błędy. Zasadniczym problemem jest to, że kompilator rozpoznaje typy błędów podczas kompilacji szablonu dla określonych typów. Następnie raportuje zwykle, gdzie został zidentyfikowany błąd. Taki komunikat o błędzie praktycznie uniemożliwia nam jakiekolwiek sensowne działanie. W komercyjnym kodzie podstawienia związane z użyciem szablonów są bowiem na tyle skomplikowane, że jakikolwiek komunikat o błędzie przestaje być czytelny.

Na przykład poniższy kod zawiera oczywisty błąd. Definiuje on kryterium wyszukiwania łańcuchów za pomocą szablonu `greater<>`, a następnie przekazuje mu parametr `int` zamiast `std::string`:

```
std::list<std::string> coll;
...
// wyszukuje pierwszy element większy od "A"
std::list<std::string>::iterator pos;
pos = find_if (coll.begin(), coll.end(),  // zakres
std::bind2nd(std::greater<int>(), "A")); // kryterium wyszukiwania
```

Kompilator wyświetli w tym przypadku następujący komunikat o błędzie:

```
/local/include/stl/_algo.h: In function 'struct _STL::_List_iterator<_STL::basic_
string<char,_STL::char_traits<char>,_STL::allocator<char> >,_STL::_Nonconst_traits
<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > > > _STL::
find_if<_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL::all
ocator<char> >,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,
_STL::allocator<char> > > >, _STL::binder2nd<_STL::greater<int> > >(_STL::_List_
iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> >,
_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator
<char> > > >, _STL:: _List_iterator<_STL::basic_string<char,_STL::char_traits<char>,
_STL::allocator<char> >,_STL::Nonconst_traits<_STL::basic_string<char,_STL::char
_traits<char>,_STL::allocator<char> > > >, _STL::binder2nd<_STL::greater<int> >,
STL::input_iterator_tag)':
```

```
/local/include/stl/_algo.h:115:     instantiated from ' ' '_STL::find_if<_STL::_List
_iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char>
>,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::
allocator<char> > > >, STL::binder2nd<_STL::greater<int> > >(_STL::_List_iterator<_STL
::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> >,_STL::_Nonconst
_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > > >,
_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator
<char> >,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::
allocator<char> > > >, _STL::binder2nd<_STL::greater<int> >)'
testprog.cpp:18:   instantiated from here
/local/include/stl/_algo.h:78: no match for call to '(_STL::binder2nd<_STL::greater
<int> >) (_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > &)'
/local/include/stl/_function.h:261: candidates are: bool _STL::binder2nd<_STL::greater
<int> >::operator ()(const int &) const
```

Powyższy tekst jest pojedynczym komunikatem o błędzie, informującym, że:

- funkcja zdefiniowana w pliku */local/include/stl/_algo.h*, której instancje utworzono w:
  - wierszu 115. pliku */local/include/stl/_algo.h*
  - wierszu 18. pliku *testprog.cpp*
- próbuje wywołać w wierszu 78. inną funkcję, której nie można odnaleźć.
- Kandydat do tego wywołania został odnaleziony w wierszu 261. pliku */local/include/stl/_function.h*

Kompilator raportuje co prawda o miejscu wystąpienia błędu, ale nawet jego odnalezienie w tak obszernym komunikacie może być trudne[5]. W naszym przykładzie jest to wiersz:

```
testprog.cpp:18:   instantiated from here
```

Należy teraz spróbować zrozumieć przyczynę błędu. Programista mający sporą praktykę może zauważyć w tekście komunikatu, że poszukiwany jest parametr typu `basic_string`, ale znaleziony zostaje jedynie parametr typu `int`. Jeśli podczas analizy tekstu komunikatu programista nie potrafi dojść do takiego wniosku, przynajmniej wie, w którym wierszu pojawił się błąd, i ma szansę zauważyć wywołanie dla niewłaściwego typu.

Przedstawiony przykład komunikatu o błędzie stanowi ilustrację jeszcze jednego problemu. Podczas stosowania szablonów kompilator wykonuje szereg podstawień. Ich efektem są niezwykle długie wewnętrzne nazwy symboli, nawet do 10 000 znaków. Nie każdy kompilator potrafi sobie z nimi poradzić.

7.6.3. Podsumowanie

- Szablony wykraczają poza zwykły model kompilacji i konsolidacji.
- Możliwe jest jawne tworzenie instancji szablonów.

[5] Niestety istnieją także kompilatory języka C++, które nie dostarczają nawet informacji o wierszu programu, który spowodował wystąpienie błędu.

- Większą elastyczność posługiwania się szablonami możemy uzyskać umieszczając deklarację i definicję szablonu w osobnych plikach nagłówkowych.
- Dla szablonów opracowany został specjalnym model kompilacji, ale w praktyce nie jest on jeszcze stosowany.
- Komunikaty o błędach związanych z szablonami mogą być trudne do zrozumienia. Najważniejsze jest, by odnaleźć fragment kodu, który spowodował wystąpienie błędu, i właściwie rozpoznać jego przyczynę.
- Podczas stosowania szablonów mogą być generowane wyjątkowo długie symbole.