helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

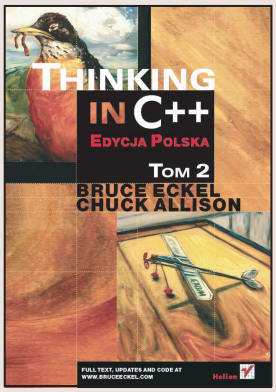
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Thinking in C++. Edycja polska. Tom II



Autor: Bruce Eckel, Chuck Allison
Tłumaczenie: Przemysław Szeremiota, Tomasz Żmijewski
ISBN: 83-7361-409-5
Tytuł oryginału: Thinking in C++, Vol. 2: Practical Programming, Second Edition
Format: B5, stron: 688

Nauka języka C++ i szczegółowe poznanie jego możliwości to poważne wyzwanie nie tylko dla początkującego, ale również dla zaawansowanego programisty. W książce Thinking in C++. Edycja polska. Bruce Eckel w doskonały sposób przedstawił podstawowe zagadnienia związane z tym językiem. Jeśli opanowałeś materiał z tej książki, możesz rozpocząć lekturę drugiego tomu.

Książka, którą trzymasz w ręce – Thinking in C++. Edycja polska. Tom II to kolejny bestseller Bruce'a Eckela poświęcony językowi C++. Tym razem Bruce w typowy dla siebie, prosty i zrozumiały sposób opisuje zaawansowane aspekty programowania w C++. Dowiesz się, jak korzystać z referencji, przeciążania operatorów, dziedziczenia i obiektów dynamicznych, a także poznasz zagadnienia zaawansowane – prawidłowe użycie szablonów, wyjątków i wielokrotnego dziedziczenia. Wszystkie tematy opatrzone są ćwiczeniami.

- obsługa wyjątków
- programowanie defensywne
- standardowa biblioteka C++
- strumienie wejścia-wyjścia
- wzorce projektowe
- zaawansowane metody programowania obiektowego
- współbieżność

Kody źródłowe znajdujące się w książce są zgodne z wieloma kompilatorami C++.

Bruce Eckel jest prezesem MindView, Inc., firmy prowadzącej kursy i szkolenia zarówno otwarte, jak i zamknięte, a także zajmującej się doradztwem i nadzorem nad projektami związanymi z technologiami obiektowymi i wzorcami projektowymi. Jest autorem książek Thinking in Java oraz pierwszego tomu Thinking in C++, a także współautorem książki Thinking in C#. Napisał także kilka innych książek i ponad 150 artykułów. Od ponad 20 lat prowadzi wykłady i seminaria na całym świecie. Był członkiem Komitetu Standardów C++. Zdobył tytuł naukowy w dziedzinie fizyki stosowanej i inżynierii oprogramowania.

Chuck Allison jest matematykiem, pełniącym obecnie funkcję wykładowcy na wydziale informatyki uniwersytetu stanowego Utah Valley. Do niedawna pełnił funkcję redaktora w magazynie C/C++ Users Journal. Jest także autorem książki C&C++ Code Capsules: A Guide for Practitioners i kilkudziesięciu artykułów poświęconych programowaniu w C i C++. Brał udział w tworzeniu standardu C++. Jego artykuły i omówienia napisanych przez niego książek można znaleźć w witrynie WWW www.freshsources.com.

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

Spis treści

Wstęp **11**
Cele 11
Rozdziały 12
Ćwiczenia 14
Rozwiązania do ćwiczeń 14
Kod źródłowy 15
Obsługiwane wersje języka 16
Standardy językowe 17
O okładce 17

Część I Tworzenie niezawodnych systemów **19**

Rozdział 1. Obsługa wyjątków **21**
Tradycyjna obsługa błędów 22
Wyrzucanie wyjątku 24
Przechwytywanie wyjątku 25
Blok try 25
Obsługa wyjątków 25
Zakończenie i kontynuacja 26
Dopasowywanie wyjątków 27
Przechwytywanie dowolnych wyjątków 29
Ponowne wyrzucanie wyjątku 29
Wyjątki nieprzechwycone 30
Czyszczenie 32
Zarządzanie zasobami 33
Wszystko jest obiektem 34
auto_ptr 36
Bloki try na poziomie funkcji 38
Wyjątki standardowe 39
Specyfikacje wyjątków 41
Jakieś lepsze specyfikacje wyjątków? 45
Specyfikacja wyjątków i dziedziczenie 45
Kiedy nie używać specyfikacji wyjątków? 46
Bezpieczeństwo wyjątków 47
Programowanie z użyciem wyjątków 50
Kiedy unikać wyjątków? 51
Typowe zastosowania wyjątków 52
Narzuty 55
Podsumowanie 57
Ćwiczenia 57

Rozdział 2. Programowanie defensywne 59
Asercje 61
Najprostszy system testów jednostkowych, który ma szansę zadziałać 65
Automatyczne testowanie 66
Szkielet TestSuite 70
Zestawy testowe 73
Kod szkieletu testowego 74
Techniki usuwania błędów 79
Makra śledzące 79
Plik śladu 80
Znajdowanie wycieków pamięci 81
Podsumowanie 86
Ćwiczenia 86

Część II Standardowa biblioteka C++ 91

Rozdział 3. Wszystko o łańcuchach 93
Czym jest łańcuch? 94
Tworzenie i inicjalizacja łańcuchów C++ 95
Operacje na łańcuchach 98
Dodawanie, wstawianie i łączenie łańcuchów 98
Podmiana znaków łańcucha 100
Sklejanie za pomocą przeciążonych operatorów spoza klasy 103
Szukanie w łańcuchach 104
Znajdowanie od końca 107
Znajdowanie pierwszego i ostatniego ze zbioru znaków 109
Usuwanie znaków z łańcuchów 111
Porównywanie łańcuchów 112
Łańcuchy a znaki 116
Przykład zastosowania łańcuchów 121
Podsumowanie 125
Ćwiczenia 126

Rozdział 4. Strumienie wejścia-wyjścia 129
Po co nowa biblioteka? 129
Iostream przybywa z odsieczą 133
Wstawianie i pobieranie 134
Typowe zastosowania 137
Dane wejściowe pobierane wierszami 139
Obsługa błędów strumieni 140
Strumienie związane z plikami 143
Przykład przetwarzania pliku 143
Tryby otwarcia 145
Buforowanie strumieni 146
Przeszukiwanie strumieni wejścia-wyjścia 148
Strumienie powiązane z łańcuchami 151
Łańcuchowe strumienie wejściowe 152
Łańcuchowe strumienie wyjściowe 153
Formatowanie strumieni wyjściowych 156
Flagi formatujące 156
Pola formatujące 158
Szerokość, wypełnienie, dokładność 159
Kompletny przykład 160

Manipulatory ... 162
Manipulatory z argumentami ... 163
Tworzenie manipulatorów ... 166
Efektory ... 167
Przykłady wykorzystujące iostream ... 169
Zarządzanie kodem źródłowym biblioteki klas ... 169
Wykrywanie błędów kompilatora ... 173
Prosty rejestrator danych ... 175
Obsługa wielu języków ... 179
Strumienie znaków szerokich ... 179
Ustawienia lokalne ... 181
Podsumowanie ... 183
Ćwiczenia ... 183

Rozdział 5. Wszystko o szablonach ... 187

Parametry szablonów ... 187
Parametry szablonów niebędące typami ... 188
Domyślne argumenty szablonów ... 190
Szablony jako parametry szablonów ... 191
Słowo kluczowe typename ... 196
Użycie słowa kluczowego template jako wskazówki ... 198
Szablony składowe ... 199
Szablony funkcji ... 201
Dedukowanie typu argumentów szablonu funkcji ... 202
Przeciążanie szablonów funkcji ... 205
Pobieranie adresu wygenerowanej z szablonu funkcji ... 206
Stosowanie funkcji do sekwencji STL ... 209
Częściowe uporządkowanie szablonów funkcji ... 212
Specjalizacja szablonów ... 213
Specjalizacja jawna ... 214
Specjalizacja częściowa ... 215
Przykład praktyczny ... 217
Unikanie nadmiarowego kodu ... 220
Odszukiwanie nazw ... 224
Nazwy w szablonach ... 224
Szablony i funkcje zaprzyjaźnione ... 228
Idiomy programowania za pomocą szablonów ... 233
Cechy charakterystyczne ... 233
Reguły ... 238
Tajemniczo powtarzający się wzorzec szablonów ... 240
Szablony i metaprogramowanie ... 242
Programowanie na poziomie kompilacji ... 243
Szablony wyrażeń ... 251
Modele kompilacji szablonów ... 256
Model włączania ... 256
Ukonkretnianie jawne ... 257
Model separacji ... 259
Podsumowanie ... 260
Ćwiczenia ... 261

Rozdział 6. Algorytmy uogólnione ... 265

Algorytmy uogólnione — wprowadzenie ... 265
Predykaty ... 268
Iteratory strumieni ... 270
Złożoność algorytmu ... 272

Obiekty funkcyjne ... 274
Klasyfikacja obiektów funkcyjnych ... 275
Automatyczne tworzenie obiektów funkcyjnych ... 276
Adaptowalność obiektów funkcyjnych ... 279
Więcej przykładów wykorzystania obiektów funkcyjnych ... 281
Adaptery wskaźników do funkcji ... 287
Pisanie własnych adapterów obiektów funkcyjnych ... 293
Katalog algorytmów STL ... 297
Narzędzia przydatne w tworzeniu przykładów ... 299
Wypełnianie i generowanie sekwencji ... 303
Zliczanie ... 304
Manipulowanie sekwencjami ... 305
Wyszukiwanie i zastępowanie elementów ... 310
Porównywanie sekwencji ... 316
Usuwanie elementów sekwencji ... 319
Sortowanie i operacje na sekwencjach posortowanych ... 322
Operacje na stertach ... 331
Wykonywanie operacji na wszystkich elementach sekwencji ... 332
Algorytmy numeryczne ... 339
Narzędzia ... 342
Tworzenie własnych algorytmów uogólnionych ... 343
Podsumowanie ... 345
Ćwiczenia ... 345

Rozdział 7. Kontenery ... 351

Kontenery i iteratory ... 351
Dokumentacja biblioteki STL ... 353
Wprowadzenie ... 353
Kontenery przechowujące ciągi znakowe ... 358
Dziedziczenie po kontenerach STL ... 360
Kraina iteratorów ... 362
Iterator w kontenerach dwukierunkowych ... 364
Kategorie iteratorów ... 365
Iteratory predefiniowane ... 367
Kontenery sekwencyjne: vector, list i deque ... 373
Podstawowe operacje na kontenerach sekwencyjnych ... 373
Kontener typu vector ... 376
Kontener typu deque ... 383
Konwersja sekwencji ... 385
Kontrolowany dostęp swobodny ... 387
Kontener typu list ... 388
Wymienianie całych sekwencji ... 393
Kontener typu set ... 394
Klasa iteratora słów ... 397
Szablon stack ... 402
Szablon queue ... 405
Kolejki priorytetowe ... 410
Przechowywanie bitów ... 418
Typ bitset<n> ... 419
Typ vector<bool> ... 422
Kontenery asocjacyjne ... 424
Generowanie elementów i wypełnianie kontenerów asocjacyjnych ... 428
Magia kontenerów typu map ... 431
Kontener typu multimap ... 433
Kontener typu multiset ... 436

Korzystanie z wielu kontenerów STL ... 439
Czyszczenie kontenera wskaźników ... 442
Tworzenie własnych kontenerów ... 444
Rozszerzenia biblioteki STL ... 446
Kontenery spoza STL ... 448
Podsumowanie ... 452
Ćwiczenia ... 453

Część III Zagadnienia zaawansowane ... 457

Rozdział 8. Rozpoznawanie typu w czasie wykonania programu ... 459
Rzutowanie w czasie wykonania ... 459
Operator typeid ... 464
Rzutowanie na pośrednie poziomy hierarchii klas ... 466
Wskaźniki na typ void ... 467
RTTI a szablony ... 468
Wielodziedziczenie ... 469
Zastosowania mechanizmu RTTI ... 470
Sortownia odpadków ... 471
Implementacja i narzuty mechanizmu RTTI ... 475
Podsumowanie ... 475
Ćwiczenia ... 476

Rozdział 9. Wielodziedziczenie ... 479
Wprowadzenie do wielodziedziczenia ... 479
Dziedziczenie interfejsu ... 481
Dziedziczenie implementacji ... 484
Duplikaty podobiektów ... 489
Wirtualne klasy bazowe ... 493
Wyszukiwanie nazw ... 502
Unikanie wielodziedziczenia ... 504
Rozszerzanie interfejsu ... 506
Podsumowanie ... 509
Ćwiczenia ... 510

Rozdział 10. Wzorce projektowe ... 513
Pojęcie wzorca ... 513
Wyższość kompozycji nad dziedziczeniem ... 515
Klasyfikacja wzorców ... 515
Właściwości, pojęcia, wzorce ... 516
Upraszczanie pojęć ... 516
Posłaniec ... 517
Parametr zbiorczy ... 518
Singleton ... 519
Odmiany Singletona ... 520
Polecenie — wybór operacji ... 524
Polecenia izolujące obsługę zdarzeń ... 526
Izolowanie obiektów ... 529
Pośrednik — dostęp do innego obiektu ... 529
Stan — modyfikowanie czynności obiektu ... 531
Adapter ... 532
Metoda szablonowa ... 535
Strategia — dynamiczny wybór algorytmu ... 536
Łańcuch odpowiedzialności — wypróbowywanie sekwencji strategii ... 537

Fabryki — hermetyzacja procesu tworzenia obiektu ... 540
Fabryki polimorficzne ... 542
Fabryka abstrakcyjna ... 545
Konstruktory wirtualne ... 547
Builder — tworzenie złożonych obiektów ... 552
Obserwator ... 558
Pojęcie klasy wewnętrznej ... 561
Przykład zastosowania Obserwatora ... 564
Dyspozycja wielokrotna ... 567
Wielokrotna dyspozycja z wzorcem Wizytator ... 571
Podsumowanie ... 575
Ćwiczenia ... 575

Rozdział 11. Współbieżność ... 579
Motywacja ... 580
Współbieżność w języku C++ ... 582
Instalacja biblioteki ZThread ... 582
Definiowanie zadań ... 584
Klasa Thread ... 585
Tworzenie interfejsu użytkownika o krótkim czasie odpowiedzi ... 587
Uproszczenie kodu z wykorzystaniem Wykonawców ... 589
Tymczasowe zawieszanie działania wątków ... 592
Usypianie wątków ... 594
Priorytety wątków ... 595
Współdzielenie ograniczonych zasobów ... 597
Gwarantowanie istnienia obiektów ... 597
Niewłaściwe odwołania do zasobów ... 601
Kontrola dostępu ... 603
Uproszczenie kodu z wykorzystaniem Strażników ... 605
Pamięć lokalna wątku ... 608
Zatrzymywanie zadań ... 610
Zapobieganie kolizji odwołań do strumieni wejścia-wyjścia ... 610
Ogród botaniczny ... 611
Zatrzymywanie zadań zablokowanych ... 616
Przerwanie ... 617
Współpraca wątków ... 623
Oczekiwanie i nadawanie sygnałów ... 623
Zależności typu producent-konsument ... 627
Rozwiązywanie problemów wielowątkowości przez kolejkowanie ... 630
Sygnalizacja rozgłoszeniowa ... 635
Zakleszczenie ... 641
Podsumowanie ... 647
Ćwiczenia ... 649

Dodatki ... 653

Dodatek A Zalecana lektura ... 655
Język C++ ... 655
Książki Bruce'a Eckela ... 656
Książki Chucka Allisona ... 657
Zaawansowane omówienia języka C++ ... 658
Książki o wzorcach projektowych ... 660

Dodatek B Uzupełnienie ... 661

Skorowidz ... 667

Rozdział 5.

Wszystko o szablonach

*Szablony C++ to zdecydowanie więcej niż „kontenery zmiennych typu **T**". Wprawdzie początkowo chodziło o stworzenie bezpiecznych ze względu na typy, ogólnych kontenerów, ale we współczesnym C++ szablony służą też do generowania specjalizowanego kodu i optymalizowania wykonania programu już na etapie kompilacji poprzez użycie specyficznych konstrukcji programistycznych.*

W niniejszym rozdziale Czytelnik będzie mógł się praktycznie zapoznać z możliwościami (i pułapkami) programowania za pomocą szablonów C++. Obszerniejszą analizę odpowiednich zagadnień języka oraz pułapek znaleźć można w doskonałej książce Davida Vandevoorde'a i Nico Josuttisa[1].

Parametry szablonów

Jak już powiedzieliśmy w pierwszym tomie, szablony mają dwie odmiany i występują jako: szablony funkcji i szablony klas. Jedne i drugie są w całości opisane swoimi parametrami. Każdy parametr szablonu sam może być:

1. Typem (wbudowanym lub zdefiniowanym przez użytkownika).
2. Stałą w chwili kompilacji (na przykład liczby całkowite, wskaźniki i referencje danych statycznych; często określa się takie stałe jako parametry niebędące typami).
3. Innym szablonem.

Wszystkie przykłady z pierwszego tomu należą do pierwszej kategorii i takie parametry występują najczęściej. Klasycznym przykładem prostego szablonu będącego kontenerem jest klasa **Stack** (Stos). Jako kontener obiekt **Stack** nie dba o to, jakiego typu obiekty zawiera; sposób przechowywania tych obiektów nie zmienia się zależnie od ich typu. Z tego powodu można użyć parametru będącego typem do opisania typu obiektów przechowywanych na stosie:

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

```
template<class T> class Stack {
  T* data;
  size_t count;
public:
  void push(const T& t);
  // itd.
};
```

Konkretny typ, który w danym stosie ma być użyty, podaje się jako argument odpowiadający parametrowi **T**:

```
Stack<int> myStack; // stos liczb całkowitych int
```

Kompilator używa obiektu **Stack** dostosowanego do typu **int** przez podstawienie **int** pod **T** i wygenerowanie odpowiedniego kodu. Nazwa instancji klasy wygenerowanej na podstawie tego szablonu to **Stack<int>**.

Parametry szablonów niebędące typami

Można też używać parametrów szablonów niebędących typami, o ile tylko odpowiadają one liczbom całkowitym znanym na etapie kompilacji. Można, na przykład, stworzyć stos o ustalonej wielkości, przekazując parametr niebędący typem, odpowiadający wielkości tablicy użytej do implementacji tego stosu:

```
template<class T, size_t N>
class Stack {
  T data[N]; // N to ustalona pojemność
  size_t count;
public:
  void push(const T& t);
  // itd.
};
```

Podczas tworzenia instancji takiego szablonu musisz podać wartość stałą znaną podczas kompilacji, która będzie użyta jako wartość parametru **N**:

```
Stack<int, 100> myFixedStack;
```

Wartość **N** jest znana już na etapie kompilacji, więc tablica **data** może zostać umieszczona na stosie, a nie w pamięci wymagającej alokowania, co może przyspieszyć działanie programu dzięki uniknięciu opóźnień związanych z dynamiczną alokacją pamięci. Zgodnie z przedstawioną wcześniej zasadą, nazwą uzyskanej w ten sposób klasy będzie **Stack<int, 100>**. Oznacza to, że każda kolejna wartość **N** powoduje utworzenie innego typu klasy. Na przykład klasa **Stack<int, 99>** to klasa inna niż **Stack<int, 100>**.

Szablon klasy **bitset**, szczegółowo omawiany w rozdziale 7., to jedyna klasa standardowej biblioteki C++, w której używany jest parametr szablonu niebędący typem; parametr ten określa liczbę bitów, jakie może przechowywać obiekt **bitset**. Poniższy przykładowy generator liczb losowych używa obiektu **bitset** do śledzenia liczb tak, aby wszystkie liczby w danym zakresie były zwracane w losowej kolejności bez powtórzeń tak długo, aż użyte zostaną wszystkie liczby. Poniższy przykład pokazuje także przeciążenie funkcji **operator()** w celu uzyskania składni podobnej do wywołania funkcji:

```
//: C05:Urand.h
//{-bor}
// Losowe generowanie liczb bez powtórzeń
#ifndef URAND_H
#define URAND_H
#include <bitset>
#include <cstddef>
#include <cstdlib>
#include <ctime>
using std::size_t;
using std::bitset;

template<size_t UpperBound>
class Urand {
  bitset<UpperBound> used;
public:
  Urand() {
    srand(time(0));  // Losowo
  }
  size_t operator()(); // Funkcja "generator"
};

template<size_t UpperBound>
inline size_t Urand<UpperBound>::operator()() {
  if(used.count() == UpperBound)
    used.reset();  // i od nowa (wyczyść bitset)
  size_t newval;
  while(used[newval = rand() % UpperBound])
    ; // aż do znalezienia niepowtarzalnej wartości
  used[newval] = true;
  return newval;
}
#endif // URAND_H ///:~
```

Niepowtarzalność w klasie **Urand** uzyskujemy dzięki śledzeniu w zbiorze **bitset** wszystkich liczb, jakie występują w zakresie (górne ograniczenie tego zakresu podawane jest jako argument szablonu) i rejestrując użycie każdej liczby przez ustawienie odpowiedniego bitu w strukturze **used**. Kiedy wszystkie liczby zostaną użyte, zbiór **bitset** jest czyszczony i losowanie odbywa się od nowa. Oto prosty przykład użycia obiektu **Urand**:

```
//: C05:UrandTest.cpp
//{-bor}
#include <iostream>
#include "Urand.h"
using namespace std;

int main() {
  Urand<10> u;
  for(int i = 0; i < 20; ++i)
    cout << u() << ' ';
} ///:~
```

Jak jeszcze powiemy dalej, argumenty szablonów niebędące typami są ważne w przypadku optymalizowania obliczeń numerycznych.

Domyślne argumenty szablonów

Parametrom szablonu klasy można przypisać argumenty domyślne (niedopuszczalne jest ich stosowanie w przypadku szablonów funkcji). Tak jak w przypadku argumentów domyślnych funkcji powinno się je definiować tylko raz, w pierwszej deklaracji lub definicji widzianej przez kompilator. Kiedy jakiś argument domyślny zostanie już zdefiniowany, wszystkie dalsze szablony też muszą mieć wartości domyślne. Aby pokazana powyżej klasa stosu o ustalonej wielkości była łatwiejsza w użyciu, można dodać do niej argument domyślny:

```
template<class T, size_t N = 100>
class Stack {
  T data[N]; // N to ustalona pojemność
  size_t count;
public:
  void push(const T& t);
  // itd.
};
```

Teraz, jeśli podczas deklarowania obiektu **Stack** pominięty zostanie drugi argument szablonu, wartość **N** domyślnie będzie równa 100.

Można też podać wartości domyślne dla wszystkich argumentów, ale wtedy i tak podczas deklarowania instancji szablonu trzeba będzie użyć pustej pary nawiasów, aby kompilator „wiedział", że chodzi o szablon klasy:

```
template<class T = int, size_t N = 100> // oba argumenty domyślne
class Stack {
  T data[N]; // N to ustalona pojemność
  size_t count;
public:
  void push(const T& t);
  // itd.
};

Stack<> myStack; // równoważne Stack<int, 100>
```

Argumenty domyślne są intensywnie wykorzystywane w standardowej bibliotece C++. Przykładowo szablon klasy **vector** zadeklarowany został następująco:

```
template <class T, class Allocator = allocator<T> >
class vector;
```

Zwróć uwagę na spację między dwoma zamykającymi nawiasami kątowymi. Dzięki niej kompilator nie zinterpretuje tych dwóch znaków (>>) jako operatora przesunięcia w prawo.

Z powyższej deklaracji wynika, że szablon **vector** tak naprawdę ma dwa argumenty: typ przechowywanych obiektów oraz typ odpowiadający alokatorowi używanemu w wektorze (więcej o alokatorach powiemy w rozdziale 7.) Jeśli drugi argument zostanie pominięty, użyty zostanie standardowy szablon **allocator** sparametryzowany pierwszym parametrem szablonu. Z deklaracji wynika też, że w kolejnych parametrach szablonów można używać parametrów poprzednich — tutaj tak użyto parametru **T**.

Wprawdzie w szablonach funkcji nie można używać domyślnych argumentów szablonu, ale parametrów szablonów można używać jako argumentów domyślnych zwykłych funkcji. Poniższy szablon funkcji sumuje elementy sekwencji:

```
//: C05:FuncDef.cpp
#include <iostream>
using namespace std;

template<class T>
T sum(T* b, T* e, T init = T()) {
  while(b != e)
    init += *b++;
  return init;
}

int main() {
  int a[] = {1,2,3};
  cout << sum(a, a + sizeof a / sizeof a[0]) << endl; // 6
} ///:~
```

Trzeci argument **sum()** jest wartością początkową dla sumy elementów. Argument ten jest pominięty, więc domyślnie wynosi **T()**; w przypadku typu **int** i innych typów wbudowanych pseudokonstruktor realizuje inicjalizację zerem.

Szablony jako parametry szablonów

Trzeci rodzaj parametrów, jakich można użyć w szablonie, to inny szablon klasy. Może to wydawać się dziwne, gdyż szablony są typami, a parametry będące typami są dozwolone; jeśli jednak w kodzie szablonu jako parametr ma być użyty inny szablon, kompilator musi najpierw „wiedzieć", że parametrem jest właśnie szablon. Poniższy przykład pokazuje użycie szablonu jako parametru innego szablonu:

```
//: C05:TempTemp.cpp
// Pokazuje użycie szablonu jako parametru szablonu
#include <cstddef>
#include <iostream>
using namespace std;

template<class T>
class Array { // Zwykły ciąg dający się przedłużać
  enum {INIT = 10};
  T *data;
  size_t capacity;
  size_t count;
public:
  Array() {
    count = 0;
    data = new T[capacity = INIT];
  }
  ~Array() { delete [] data; }
  void push_back(const T& t) {
    if(count == capacity) {
      // Powiększenie tablicy bazowej
      size_t newCap = 2 * capacity;
```

```
      T* newData = new T[newCap];
      for (size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete data;
      data = newData;
      capacity = newCap;
    }
    data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() {
    return data;
  }
  T* end() {
    return data + count;
  }
};

template<class T, template<class> class Seq>
class Container {
  Seq<T> seq;
public:
  void append(const T& t) {
    seq.push_back(t);
  }
  T* begin() {
    return seq.begin();
  }
  T* end() {
    return seq.end();
  }
};

int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

Szablon klasy **Array** to najzwyklejszy kontener zawierający sekwencję elementów. Szablon **Container** ma dwa parametry: typ przechowywanych obiektów oraz strukturę danych służącą do przechowywania danych. Poniższy wiersz implementacji klasy **Container** wymusza poinformowanie kompilatora, że parametr **Seq** jest szablonem:

```
Seq<T> seq;
```

Gdybyśmy nie zadeklarowali **Seq** jako parametru będącego szablonem, kompilator zgłosi błąd, twierdząc, że **Seq** nie jest szablonem, a jako takiego go używamy. W funkcji **main()** tworzona jest instancja szablonu **Container** w ten sposób, w klasie **Array** przechowywane były liczby całkowite — zatem w tym przykładzie **Seq** oznacza **Array**.

Zauważ, że w tym wypadku nie jest konieczne nazywanie parametru **Seq** w deklaracji **Container**. Odpowiedni wiersz to:

```
template<class T, template<class> class Seq>
```

Wprawdzie moglibyśmy zapisać:

```
template<class T, template<class U> class Seq>
```

lecz parametr **U** nie jest w ogóle potrzebny. Wszystko, co istotne, to to, że **Seq** jest szablonem klasy mającym pojedynczy parametr będący typem. Jest to analogia do pomijania nazw parametrów funkcji, kiedy nie są potrzebne, na przykład przy przeciążaniu operatora postinkrementacji:

```
T operator++(int);
```

Słowo kluczowe **int** jest potrzebne jedynie ze względów formalnych, więc odpowiadający mu argument nie musi mieć nazwy.

Poniższy program korzysta z tablicy o ustalonej wielkości mającej dodatkowy parametr szablonu odpowiadający wielkości tej tablicy:

```
//: C05:TempTemp2.cpp
// Wielozmienny szablon jako parametr szablonu
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N>
class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T,size_t N,template<class,size_t> class Seq>
class Container {
  Seq<T,N> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  const size_t N = 10;
  Container<int, N, Array> container;
```

```
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

Znowu nazwy parametrów w deklaracji **Seq** wewnątrz deklaracji **Containter** są zbędne, ale potrzebne są dwa dodatkowe parametry do zadeklarowania pola **seq**, stąd na najwyższym poziomie pojawia się parametr **N** niebędący typem.

Łączenie argumentów domyślnych z parametrami szablonów będących szablonami jest nieco bardziej problematyczne. Kiedy kompilator sprawdza parametr wewnętrzny parametru szablonu, nie są uwzględniane argumenty domyślne, wobec czego w celu uzyskania dokładnego dopasowania konieczne jest powtórzenie wartości domyślnych. Poniższy przykład wykorzystuje argument domyślny do szablonu **Array** o stałej wielkości, pokazuje też, jak sobie z tym dziwactwem języka radzić:

```
//: C05:TempTemp3.cpp
//{-bor}
//{-msc}
// Łączenie parametrów szablonów będących szablonami
// z argumentami domyślnymi
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N = 10>  // Argument domyślny
class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T, template<class, size_t = 10> class Seq>
class Container {
  Seq<T> seq;  // Użyto argumentu domyślnego
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  Container<int, Array> container;
  container.append(1);
```

```
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

W poniższym wierszu trzeba włączyć domyślną wielkość równą 10:

```
template<class T, template<class, size_t = 10> class Seq>
```

Obie definicje, **seq** w **Container** i **container**, w **main()** wykorzystują wartości domyślne. Jedynym sposobem użycia czegokolwiek innego niż wartość domyślna jest skorzystanie z metody z poprzedniego programu, **TempTemp2.cpp**. Jest to jedyny wyjątek od podanej wcześniej zasady mówiącej, że argumenty domyślne powinny pojawiać się w danej jednostce kompilacji tylko raz.

Wobec faktu, że wszystkie standardowe kontenery przechowujące elementy w formie sekwencji (**vector**, **list** oraz **deque** omówiono dokładnie w rozdziale 7.) mają domyślny argument **allocator**, powyższa technika jest przydatna, jeśli trzeba jedną z tych trzech metod uporządkowania przekazać jako parametr szablonu. W poniższym programie **vector** i **list** są przekazywane do dwóch instancji szablonu **Container**.

```
//: C05:TempTemp4.cpp
//{-bor}
//{-msc}
// Przekazanie standardowych sekwencji jako argumentów szablonu
#include <iostream>
#include <list>
#include <memory>  // Deklaracja allocator<T>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
                    class Seq>
class Container {
  Seq<T> seq; // Niejawnie stosowane jest domyślne allocator<T>
public:
  void push_back(const T& t) { seq.push_back(t); }
  typename Seq<T>::iterator begin() { return seq.begin(); }
  typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
  // Użyj wektora
  Container<int, vector> vContainer;
  vContainer.push_back(1);
  vContainer.push_back(2);
  for(vector<int>::iterator p = vContainer.begin();
      p != vContainer.end(); ++p) {
    cout << *p << endl;
  }
  // Użyj listy
  Container<int, list> lContainer;
  lContainer.push_back(3);
  lContainer.push_back(4);
```

```
  for(list<int>::iterator p2 = lContainer.begin();
      p2 != lContainer.end(); ++p2) {
    cout << *p2 << endl;
  }
} ///:~
```

W tym wypadku nazwaliśmy pierwszy parametr szablonu wewnętrznego **Seq** (ta nazwa to **U**), gdyż alokatory w standardowych sekwencjach muszą być sparametryzowane tym samym typem, co obiekt ich używający. Poza tym, skoro domyślna wartość parametru **allocator** jest znana, możemy ją pominąć w dalszych odwołaniach do **Seq<T>**, jak w poprzednim programie. Jednak dokładne omówienie tego przykładu wymaga objaśnienia znaczenia słowa kluczowego **typename**.

Słowo kluczowe typename

Niech będzie dany program:

```
//: C05:TypenamedID.cpp
//{-bor}
// Słowo 'typename' używane jest jako prefiks typów zagnieżdżonych

template<class T> class X {
  // Gdyby nie było typename, powstałby tu błąd
  typename T::id i;
public:
  void f() { i.g(); }
};

class Y {
public:
  class id {
  public:
    void g() {}
  };
};

int main() {
  X<Y> xy;
  xy.f();
} ///:~
```

W definicji szablonu zakłada się, że klasa **T** musi mieć pewnego rodzaju zagnieżdżony identyfikator **id**. Jednak **id** może być też statycznym polem **T** (a wtedy można byłoby na **id** wykonywać operacje bezpośrednio), natomiast nie można „tworzyć obiektu typu **id**". Jednak właśnie to się tu dzieje — identyfikator **id** jest traktowany tak, jakby był typem zagnieżdżonym w **T**. W przypadku klasy **Y id** jest faktycznie typem zagnieżdżonym, ale bez słowa kluczowego **typename** kompilator nie wiedziałby o tym w chwili kompilowania **X**.

Jeśli kompilator umożliwia traktowanie identyfikatora występującego w szablonie jako typu lub jako czegoś innego niż typ, przyjmie, że identyfikator jest czymś innym niż typ, czyli założy, że identyfikator odnosi się do obiektu (przy czym dane typów prostych też są traktowane jako obiekty), wyliczenia lub czegoś podobnego. Jednak kompilator nie założy, i nie może założyć, że jest to typ.

Wobec tego, że domyślne działanie kompilatora powoduje, że we wskazanych wyżej dwóch miejscach nie chodzi o typ, trzeba użyć słowa kluczowego **typename** dla nazw zagnieżdżonych (wyjątkiem są listy inicjalizujące konstruktora, gdzie nie jest to ani potrzebne, ani dozwolone). W powyższym przykładzie kompilator widząc **typename T::id**, „wie" (dzięki słowu kluczowemu **typename**), że **id** odnosi się do typu zagnieżdżonego, więc może stworzyć obiekt tego typu.

Cała ta zasada w formie skróconej brzmi: jeśli typ odnoszący się do kodu wewnątrz szablonu jest kwalifikowany parametrem szablonu będącym typem, powinien być poprzedzony słowem kluczowym **typename**, chyba że występuje w specyfikacji klasy bazowej lub na liście inicjalizacyjnej w tym samym zakresie (w obu tych przypadkach słowa tego nie można użyć).

Powyższa dyskusja całkowicie objaśnia użycie słowa kluczowego **typename** w programie **TempTemp4.cpp**. Bez tego słowa kluczowego kompilator założyłby, że wyrażenie **Seq<T>::iterator** nie jest typem, podczas gdy dalej używane jest ono do zdefiniowania wartości zwracanej przez funkcje **begin()** i **end()**.

W następnym przykładzie, zawierającym szablon funkcji pozwalający wydrukować dowolną standardową sekwencję C++, pokazano podobny sposób użycia **typename**:

```
//: C05:PrintSeq.cpp
//{-msc}
// Funkcja drukująca typowe sekwencje C++
#include <iostream>
#include <list>
#include <memory>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
                  class Seq>
void printSeq(Seq<T>& seq) {
  for (typename Seq<T>::iterator b = seq.begin();
       b != seq.end();)
    cout << *b++ << endl;
}

int main() {
  // Przetwarzanie wektora
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  printSeq(v);
  // Przetwarzanie listy
  list<int> lst;
  lst.push_back(3);
  lst.push_back(4);
  printSeq(lst);
} ///:~
```

Znowu, gdyby nie słowo kluczowe **typename**, kompilator zinterpretowałby **iterator** jako statyczne pole należące do **Seq<T>**, czyli byłby to błąd składniowy — przecież typ jest wymagany.

Typedef i typename

Nie wolno zakładać, że słowo kluczowe **typename** utworzy nową nazwę typu. Celem stosowania tego słowa jest wskazanie kompilatorowi, że tak zakwalifikowany identyfikator ma być zinterpretowany jako typ. Wiersz:

```
typename Seq<T>::iterator It;
```

mówi, że zmienna **It** jest zmienną typu **Seq<T>::iterator**. Jeśli należałoby stworzyć nazwę nowego typu, jak zwykle trzeba użyć **typedef**:

```
typedef typename Seq<It>::iterator It;
```

Użycie typename zamiast class

Innym zastosowaniem słowa kluczowego **typename** jest możliwość użycia **typename** zamiast **class** na liście argumentów szablonu. Czasami powstający kod jest dzięki temu czytelniejszy:

```
//: C05:UsingTypename.cpp
// Użycie 'typename' na liście argumentów szablonu

template<typename T> class X { };

int main() {
  X<int> x;
} ///:~
```

Prawdopodobnie nieczęsto spotkasz się z takim użyciem słowa kluczowego **typename**, gdyż słowo to zostało dodane do języka dość długo po zdefiniowaniu szablonów.

Użycie słowa kluczowego template jako wskazówki

Tak jak słowo kluczowe **typename** pomaga kompilatorowi w sytuacjach, kiedy nie spodziewa się on identyfikatora typu, istnieje też potencjalna trudność związana z leksemami niebędącymi identyfikatorami, jak znaki < i >; czasami oznaczają one symbole mniejszości i większości, a czasami ograniczają listy parametrów szablonu. W ramach przykładu skorzystajmy jeszcze raz z klasy **bitset**:

```
//: C05:DotTemplate.cpp
// Pokazuje użycie konstrukcji .template
#include <bitset>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

template<class charT, size_t N>
basic_string<charT> bitsetToString(const bitset<N>& bs) {
  return bs. template to_string<charT, char_traits<charT>,
                                allocator<charT> >();
}
```

```
int main() {
  bitset<10> bs;
  bs.set(1);
  bs.set(5);
  cout << bs << endl; // 0000100010
  string s = bitsetToString<char>(bs);
  cout << s << endl;  // 0000100010
} ///:~
```

Klasa **bitset** realizuje konwersję na obiekt-łańcuch za pomocą funkcji składowej **to_string**. Aby obsłużyć różne klasy łańcuchów, **to_string** sama jest szablonem, zgodnie z definicją szablonu **basic_string** omawianą w rozdziale 3. Deklaracja **to_string** w ramach **bitset** wygląda następująco:

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

Nasz pokazany powyższej szablon funkcji **bitsetToString()** pozwala zwracać obiekty **bitset** jako różnego rodzaju łańcuchy znaków. Aby uzyskać, na przykład, łańcuch z szerokimi znakami, należy wywołanie zapisać następująco:

```
wstring s = bitsetToString<wchar_t>(bs);
```

Zauważ, że **basic_string** korzysta z domyślnych argumentów szablonu, więc nie musimy powtarzać argumentów **char_traits** i **allocator** w wartości zwracanej. Niestety, **bitset::to_string** nie używa argumentów domyślnych. Użycie **bitsetToString<char>(bs)** jest wygodniejsze niż pisanie za każdym razem całego **bs.template to_string<char, char_traits, allocator<char> >()**.

Instrukcja **return** w funkcji **bitsetToString()** zawiera słowo kluczowe **template** w dość dziwnym miejscu — zaraz po operatorze „kropka" zastosowanym do obiektu **bs** typu **bitset**. Wynika to stąd, że kiedy analizowany jest szablon, znak **<** znajdujący się za napisem **to_string** zostałby interpretowany jako operator mniejszości, a nie początek lub lista argumentów szablonu. W punkcie „Odszukiwanie nazw" w dalszej części rozdziału słowo kluczowe **template** użyte w tym kontekście informuje kompilator, że dalej następuje nazwa szablonu; dzięki temu znak **<** zostanie zinterpretowany prawidłowo. Takie samo rozumowanie dotyczy operatorów **->** i **::** stosowanych do szablonów. Tak jak w przypadku słowa kluczowego **template** opisana technika unikania wieloznaczności może być używana jedynie w szablonach[2].

Szablony składowe

Szablon funkcji **bitset::to_string()** jest przykładem *szablonu składowego* czyli szablonu zadeklarowanego w innej klasie lub szablonie klasy. Dzięki temu można tworzyć rozmaite kombinacje niezależnych argumentów szablonów. Praktyczny przykład znaleźć można w szablonie klas **complex** w standardowej bibliotece C++. Szablon **complex** ma parametr będący typem, który pozwala zadeklarować typ zmiennoprzecinkowy służący

[2] Komitet standaryzacyjny C++ rozważa rezygnację z klauzuli „jedynie w szablonach", jeśli chodzi o unikanie wieloznaczności. Niektóre kompilatory już teraz pozwalają na stosowanie takich podpowiedzi poza szablonami.

do przechowywania części rzeczywistej i urojonej liczby zespolonej. Poniższy fragment kodu z biblioteki standardowej pokazuje konstruktor szablonu składowego szablonu klasy **complex**:

```
template<typename T>
class complex {
public:
  template<class X> complex(const complex<X>&);
```

Standardowy szablon **complex** ma już gotowe specjalizacje z parametrem **T** o wartościach **float**, **double** oraz **long double**. Pokazany powyżej konstruktor szablonu składowego pozwala stworzyć nową liczbę zespoloną wykorzystującą inny typ zmiennoprzecinkowy jako typ bazowy, jak poniżej:

```
complex<float> z(1, 2);
complex<double> w(z);
```

W deklaracji **w** parametr **T** szablonu **complex** ma wartość **double**, zaś **X** to **float**. Szablony składowe bardzo ułatwiają tego typu konwersje.

Wobec tego, że definiowanie szablonu w szablonie jest operacją zagnieżdżenia, przedrostki oznaczające szablony muszą to zagnieżdżanie odzwierciedlać, jeśli tylko szablon składowy zostanie zdefiniowany poza definicją klasy zewnętrznej. Jeśli, na przykład, mamy zaimplementować szablon klasy **complex** i należy zdefiniować konstruktor szablonu składowego poza definicją klasy opartej na szablonie **complex**, trzeba byłoby zrobić to tak:

```
template<typename T>
template<typename X>
complex<T>::complex(const complex<X>& c) {/* tutaj treść... */}
```

Inne zastosowanie szablonów funkcji składowych w bibliotece standardowej to inicjalizacja kontenerów takich jak **vector**. Załóżmy, że mamy **vector** zawierający dane typu **int** i chcemy tym wektorem zainicjalizować nowy wektor z liczbami **double**:

```
int data[5] = {1, 2, 3, 4, 5};
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

O ile tylko elementy **v1** są zgodne co do przypisania z elementami **v2** (a **double** i **int** w tym sensie są zgodne), wszystko działa prawidłowo. Szablon klasy **vector** ma następujący konstruktor szablonu składowego:

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
    const Allocator& = Allocator());
```

Taka konstrukcja tak naprawdę w deklaracjach wektora jest użyta dwukrotnie. Kiedy **v1** jest inicjalizowane na podstawie tablicy liczb **int**, **InputIterator** jest typu **int***. Kiedy **v2** jest inicjalizowane na podstawie **v1**, **InputIterator** w instancji konstruktora szablonu składowego jest typu **vector<int>::iterator**.

Szablony składowe mogą być także klasami (nie muszą być funkcjami, choć zwykle to właśnie jest potrzebne). W następnym przykładzie pokazano szablon klasy składowej znajdujący się wewnątrz szablonu klasy zewnętrznej:

```
//: C05:MemberClass.cpp
// Szablon klasy składowej
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T>
class Outer {
public:
  template<class R>
  class Inner {
  public:
    void f();
  };
};

template<class T> template <class R>
void Outer<T>::Inner<R>::f() {
  cout << "Outer == " << typeid(T).name() << endl;
  cout << "Inner == " << typeid(R).name() << endl;
  cout << "Pełna Inner == " << typeid(*this).name() << endl;
}

int main() {
  Outer<int>::Inner<bool> inner;
  inner.f();
} ///:~
```

Omawiany w rozdziale 8. operator **typeid** zwraca obiekt, którego funkcja składowa **name()** podaje zapisany jako łańcuch jego typ lub typ zmiennej. Wprawdzie dokładna postać tego łańcucha jest różna w różnych kompilatorach, ale wynik działania powyższego programu będzie podobny do:

```
Outer == int
Inner == bool
Pełna Inner == Outer<int>::Inner<bool>
```

Deklaracja zmiennej **inner** w programie głównym powoduje ukonkretnienie **Inner<bool>** i **Outer<int>**.

Szablony funkcji składowych nie mogą być deklarowane jako wirtualne. Stosowane obecnie kompilatory „spodziewają się", że będą w stanie określić wielkość tablicy funkcji wirtualnych klasy w chwili analizowania tej klasy. Umożliwienie stosowania wirtualnych szablonów funkcji składowych wymagałoby znajomości wszystkich wywołań takich funkcji składowych w programie, co jest niemożliwe, szczególnie w przypadku projektów zawierających wiele plików.

Szablony funkcji

Tak jak szablon klasy opisuje rodzinę klas, tak szablon funkcji opisuje rodzinę funkcji. Składnia tworzenia szablonów jednych i drugich jest w zasadzie identyczna, a różnice dotyczą głównie sposobu użycia. Podczas tworzenia instancji szablonu klas zawsze

trzeba stosować nawiasy kątowe i podawać wszystkie niedomyślne argumenty szablonu. Z kolei w przypadku szablonów funkcji często można pomijać argumenty szablonu, zaś domyślne argumenty szablonu są niedopuszczalne. Przyjrzyjmy się typowej implementacji szablonu funkcji **min()** zadeklarowanego w pliku nagłówkowym **<algorithm>**:

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

Szablon taki można wywołać, podając typ argumentów w nawiasach kątowych, tak jak w przypadku szablonów klas:

```
int z = min<int>(i, j);
```

Użycie takiej składni informuje kompilator, że potrzebny jest szablon funkcji **min** z typem **int** użytym zamiast parametru **T**, więc kompilator wygeneruje odpowiedni kod. Zgodnie z konwencją nazewniczą dotyczącą klas generowanych przez szablony klas można się spodziewać, że nazwą tak wywołanej funkcji będzie **min<int>**.

Dedukowanie typu argumentów szablonu funkcji

Zawsze można użyć jawnej specyfikacji szablonu funkcji jak powyżej, ale często wygodnie byłoby pominąć argumenty szablonu i pozwolić kompilatorowi wywnioskować ich wartości na podstawie argumentów funkcji, na przykład tak:

```
int z = min(i, j);
```

Jeśli **i** i **j** są liczbami **int**, kompilator „wie", że potrzebna jest funkcja **min<int>()**, i dokonuje automatycznie odpowiedniego zastąpienia. Typy muszą być identyczne, gdyż szablon pierwotnie zdefiniowany jest z jednym tylko argumentem określającym typ, używanym dla obu parametrów funkcji. W przypadku argumentów funkcji, których typ wynika z parametru szablonu, nie są wykonywane żadne standardowe konwersje. Jeśli, na przykład, trzeba byłoby znaleźć mniejszą z liczb **int** i **double**, poniższa próba wywołania nie powiodłaby się:

```
int z = min(x, j); // x jest typu double
```

Zmienne **x** i **j** mają różne typy, więc żaden pojedynczy parametr **T** w szablonie **min** nie może zostać użyty, a wobec tego wywołanie jest niezgodne z deklaracją szablonu. Można tego problemu uniknąć, rzutując jeden argument na typ argumentu drugiego albo stosując wywołanie z pełną specyfikacją, na przykład:

```
int z = min<double>(x, j);
```

W ten sposób kompilator „wie", że ma wygenerować funkcję **min()** korzystającą z typu **double**, a wtedy już zmienna **j** może w sposób standardowy przekształcona na typ **double** (gdyż funkcja **min<double>(const double&, const double&)** może zostać wygenerowana).

Kuszące byłoby zażądanie użycia w przypadku szablonu **min** użycia dwóch parametrów, dzięki czemu typy argumentów mogłyby być dowolne:

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

Często jest to dobre rozwiązanie, choć w tym wypadku jego zastosowanie jest dyskusyjne: **min()** musi zwrócić wartość, a nie ma jednoznacznie dobrej metody określenia typu tej wartości — czy powinien to być typ **T** albo **U**.

Jeśli typ wartości zwracanej przez funkcję jest niezależnym parametrem szablonu, zawsze w chwili wywołania funkcji trzeba podać ten typ jawnie, gdyż nie ma żadnych podstaw do wywnioskowania tego typu. Taka sytuacja występuje w przypadku szablonu **fromString** pokazanego poniżej:

```
//: C05:StringConv.h
#ifndef STRINGCONV_H
#define STRINGCONV_H
// Szablony funkcji przekształcających na łańcuchy i z łańcuchów
#include <string>
#include <sstream>

template<typename T>
T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
}

template<typename T>
std::string toString(const T& t) {
  std::ostringstream s;
  s << t;
  return s.str();
}
#endif // STRINGCONV_H ///:~
```

Podane szablony funkcji obsługują konwersje na typ **std::string** oraz z tego typu dla dowolnych typów mających odpowiednio operatory wstawiania i pobierania danych ze strumienia. Oto program testowy pokazujący użycie typu **complex** z biblioteki standardowej:

```
//: C05:StringConvTest.cpp
#include "StringConv.h"
#include <iostream>
#include <complex>
using namespace std;

int main() {
  int i = 1234;
  cout << "i == \"" << toString(i) << "\"\n";
  float x = 567.89;
  cout << "x == \"" << toString(x) << "\"\n";
```

```
  complex<float> c(1.0, 2.0);
  cout << "c == \"" << toString(c) << "\"\n";
  cout << endl;

  i = fromString<int>(string("1234"));
  cout << "i == " << i << endl;
  x = fromString<float>(string("567.89"));
  cout << "x == " << x << endl;
  c = fromString< complex<float> >(string("(1.0,2.0)"));
  cout << "c == " << c << endl;
} ///:~
```

Wynik działania tego programu będzie następujący:

```
i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
c == (1,2)
```

Zauważ, że w każdej instancji **fromString** parametr szablonu jest podawany w wywołaniu. Jeśli masz szablon funkcji z parametrami szablonu zarówno dla parametrów funkcji jak i wartości zwracanej, konieczne jest zadeklarowanie najpierw parametru odpowiadającego wartości zwracanej; w przeciwnym przypadku niemożliwe będzie pomijanie parametrów szablonu odpowiadających parametrom funkcji. W ramach przykładu przyjrzyjmy się takiemu oto dobrze znanemu szablonowi funkcji[3]:

```
//: C05:ImplicitCast.cpp
template<typename R, typename P>
R implicit_cast(const P& p) {
  return p;
}

int main() {
  int i = 1;
  float x = implicit_cast<float>(i);
  int j = implicit_cast<int>(x);
  // char* p = implicit_cast<char*>(i);
} ///:~
```

Jeśli zamienisz znajdujące się na początku pliku parametry **R** i **P** miejscami, niemożliwe będzie skompilowanie takiego programu, gdyż typ wartości zwracanej nie będzie podany (po takiej zmianie pierwszy parametr odpowiadałby typowi parametru funkcji). Ostatni wiersz (zakomentowany) jest niedopuszczalny, gdyż nie istnieje standardowa konwersja typu **int** na **char***; **implicit_cast** służy do realizacji tych konwersji, które są dopuszczalne.

Zachowując pewną ostrożność, można nawet wydedukować wielkość tablicy. Poniższy przykład zawiera wykonujący to szablon funkcji inicjalizującej tablicę, **init2**:

[3] Zobacz: Stroustrup, *The C++ Programming Language, 3rd Edition*, Addison Wesley, strony 335 – 336.

```
//: C05:ArraySize.cpp
#include <cstddef>
using std::size_t;

template<size_t R, size_t C, typename T>
void init1(T a[R][C]) {
  for (size_t i = 0; i < R; ++i)
    for (size_t j = 0; j < C; ++j)
      a[i][j] = T();
}

template<size_t R, size_t C, class T>
void init2(T (&a)[R][C]) {  // parametr będący referencją
  for (size_t i = 0; i < R; ++i)
    for (size_t j = 0; j < C; ++j)
      a[i][j] = T();
}

int main() {
  int a[10][20];
  init1<10,20>(a);  // podanie konieczne
  init2(a);         // rozmiar zostanie określony przez kompilator
} ///:~
```

Wymiary tablicy nie są przekazywane jako część typu parametru funkcji, chyba że parametr ten jest przekazywany przez wskaźnik lub referencję. Szablon funkcji **init2** zawiera deklarację **a** jako referencji tablicy dwuwymiarowej, więc wymiary **R** i **C** zostaną wyznaczone przez mechanizm szablonów, przez co **init2** staje się wygodnym narzędziem do inicjalizowania dwuwymiarowych tablic dowolnej wielkości. Szablon **init1** nie przekazuje tablicy przez referencję, wobec czego tutaj wielkość tablicy musi być jawnie podawana, choć typ parametru może być nadal wywnioskowany.

Przeciążanie szablonów funkcji

Tak jak w przypadku zwykłych funkcji, można przeciążać szablony funkcji mające takie same nazwy. Kiedy kompilator przetwarza w programie wywołanie funkcji, musi zdecydować, który szablon lub zwykła funkcje będzie „najlepiej pasować" do danego wywołania.

Jeśli istnieje opisany wcześniej szablon funkcji **min()**, dodajmy jeszcze zwykłe funkcje:

```
//: C05:MinTest.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
```

```
double min(double x, double y) {
  return (x < y) ? x : y;
}

int main() {
  const char *s2 = "mówią \"Ni-!\"", *s1 = "rycerze którzy";
  cout << min(1, 2) << endl;      // 1: 1 (szablon)
  cout << min(1.0, 2.0) << endl;  // 2: 1 (double)
  cout << min(1, 2.0) << endl;    // 3: 1 (double)
  cout << min(s1, s2) << endl;    // 4: rycerze którzy (const
                                  //                  char*)
  cout << min<>(s1, s2) << endl;  // 5: mówią "Ni-!"
                                  //    (szablon)
} ///:~
```

Teraz oprócz szablonu funkcji mamy zdefiniowane dwie zwykłe funkcje: wersję **min()** używającą łańcuchów języka C oraz wersję z **double**. Gdyby nie istniał szablon, wywołanie w wierszu 1. wywoływałoby wersję **double** funkcji **min()**, a to z uwagi na konwersję standardową typu **int** na **double**. Jednak istniejący szablon pozwala wygenerować wersję **int**, która jest uważana za lepiej dopasowaną, więc ten szablon zostanie użyty. Wywołanie w wierszu 2. odpowiada wersji z **double**, zaś wywołanie z wiersza 3. wywołuje tę samą funkcję, niejawnie konwertując 1 na 1.0. W wierszu 4. bezpośrednio wywoływana wersja const **char*** funkcji **min()**. W wierszu 5. wymuszamy na kompilatorze użycie szablonu, gdyż do nazwy funkcji dodajemy pustą parę nawiasów kątowych; wobec tego z szablonu generowana jest wersja **const char*** i jest ona używana (widać to po nieprawidłowej odpowiedzi — nastąpiło najzwyklejsze porównanie adresów!)[4]. Jeśli zastanawiasz się, czemu zamiast dyrektywy **using namespace std;** użyliśmy deklaracji **using**, wiedz, że niektóre kompilatory niejawnie dołączają pliki nagłówkowe deklarujące **std::min()**, które mogłyby spowodować konflikt z naszymi deklaracjami nazw **min()**.

Jak powiedziano wyżej, można przeciążać szablony mające tę samą nazwę, o ile tylko kompilator będzie w stanie je od siebie odróżnić. Można byłoby, na przykład, zadeklarować szablon funkcji **min()** mający trzy argumenty:

```
template<typename T>
const T& min(const T& a, const T& b, const T& c);
```

Wersje tego szablonu będą generowane tylko dla trójargumentowych wywołań **min()**, gdzie wszystkie argumenty będą tego samego typu.

Pobieranie adresu wygenerowanej z szablonu funkcji

Często potrzebny bywa adres funkcji — na przykład mamy funkcję pobierającą jako argument wskaźnik innej funkcji. Oczywiście można przekazać także funkcję wygenerowaną z szablonu, a zatem potrzebny będzie jej adres[5]:

[4] Formalnie rzecz biorąc, porównywanie dwóch wskaźników nieznajdujących się w jednej tablicy daje wynik nieokreślony, ale stosowane obecnie kompilatory nie traktują tego poważnie. Co więcej, zachowują się one w takiej sytuacji poprawnie.

[5] Za ten przykład jesteśmy wdzięczni Nathanowi Myersowi.

```
//: C05:TemplateFunctionAddress.cpp
// Pobieranie adresu funkcji wygenerowanej z szablonu

template <typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template <typename T>
  void g(void (*pf)(T*)) {}

int main() {
  // Pełna specyfikacja typu
  h(&f<int>);
  // Odgadywanie typu:
  h(&f);
  // Pełna specyfikacja typu
  g<int>(&f<int>);
  // Odgadywanie typu:
  g(&f<int>);
  // Specyfikacja częściowa, ale wystarczająca
  g<int>(&f);
} ///:~
```

Przykład ten porusza kilka kwestii. Po pierwsze, mimo że używamy szablonów, pasować muszą sygnatury. Funkcja **h()** pobiera wskaźnik funkcji typu **void** mającej argument **int***, właśnie taką funkcję wygeneruje szablon **f()**. Po drugie, funkcja, która jako argument pobiera wskaźnik funkcji, sama może być szablonem, jak w przypadku szablonu **g()**.

W funkcji **main()** widać, że działa też dedukowanie typu. W pierwszym jawnym wywołaniu **h()** podawany jest argument szablonu, ale wobec tego, że **h()** przyjmuje jedynie adres funkcji mającej parametr **int***, mógłby się tego domyślić kompilator. Jeszcze ciekawiej jest w przypadku funkcji **g()**, gdyż tutaj mamy do czynienia z dwoma szablonami. Kompilator nie może „wywnioskować" typu, nie mając żadnych danych, ale jeśli w przypadku **f()** lub **g()** podany zostanie typ **int**, resztę już można będzie odgadnąć.

Problem pojawia się, kiedy próbujemy przekazać jako parametry funkcje **tolower** czy **toupper** zadeklarowane w pliku nagłówkowym **<cctype>**. Można ich używać, na przykład, wraz z algorytmem **transform** (szczegółowo omawianym w następnym rozdziale), aby łańcuchy przekształcać na wielkie lub na małe litery. Jednak trzeba zachować tu ostrożność, gdyż funkcje te mają wiele deklaracji. Najprościej byłoby zapisać coś takiego:

```
// Zmienna s to std::string
transform(s.begin(), s.end(), s.begin(), tolower);
```

Algorytm **transform** stosuje swój czwarty parametr (tutaj **tolower()**) do każdego znaku łańcucha **s**, wynik umieszcza w **s**, przez co nadpisuje każdy znak **s** odpowiadającą mu małą literą. Jednak tak zapisana instrukcja może zadziałać lub nie! W poniższym kontekście nie zadziała:

```
//: C05:FailedTransform.cpp {-xo}
#include <algorithm>
#include <cctype>
#include <iostream>
```

```
#include <string>
using namespace std;

int main() {
 string s("LOWER");
 transform(s.begin(), s.end(), s.begin(), tolower);
 cout << s << endl;
} ///:~
```

Nawet jeśli Twój kompilator sobie z tym programem poradzi, to program jest i tak niepoprawny. Wynika to stąd, że plik nagłówkowy **<iostream>** udostępnia także dwuargumentowe wersje **tolower()** i **toupper()**:

```
template <class charT> charT toupper(charT c, const locale& loc);
template <class charT> charT tolower(charT c, const locale& loc);
```

Te szablony funkcji mają drugi argument typu **locale**. Kompilator nie jest w stanie stwierdzić, czy powinien użyć jednoargumentowej wersji **tolower()** z **<cctype>** czy powyższej. Problem ten można rozwiązać (prawie), za pomocą rzutowania w czasie wywołania **transform**:

```
transform(s.begin(), s.end(), s.begin(), static_cast<int (*)(int)>(tolower));
```

(przypomnijmy, że **tolower()** i **toupper()** zamiast **char** korzystają z **int**). Powyższe rzutowanie jasno informuje, że chodzi o jednoargumentową wersję **tolower()**. Znowu, kod ten może zadziałać w niektórych kompilatorach, ale nie musi. Powód tym razem jest mniej oczywisty — w implementacji bibliotek dopuszczalne jest użycie w przypadku funkcji odziedziczonych po języku C „wiązania C" (czyli nazwa funkcji nie zawiera wszystkich informacji pomocniczych[6], jak to ma miejsce normalnie w C++). W takim wypadku rzutowanie nie powiedzie się, gdyż **transform** jest szablonem funkcji C++ i oczekuje czwartego argumentu zgodnego z językiem C++; rzutowanie nie może natomiast zmieniać sposobu wiązania. Fatalnie!

Rozwiązaniem jest umieszczenie wywołań **tolower()** w kontekście wykluczającym wieloznaczność. Można, na przykład, napisać funkcję, którą nazwiemy **strTolower()**, następnie zamieścimy ją we własnym pliku bez włączania **<iostream>**:

```
//: C05:StrTolower.cpp {O} {-mwcc}
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;

string strTolower(string s) {
  transform(s.begin(), s.end(), s.begin(), tolower);
  return s;
} ///:~
```

Plik nagłówkowy **<iostream>** nie jest używany i stosowane przez autorów kompilatory nie włączyły w takiej sytuacji dwuargumentowej wersji **tolower()**[7], więc już wszystko jest w porządku. Teraz już nowej funkcji można używać normalnie:

---

[6] Chodzi o informacje takie jak typy zakodowane w nazwie funkcji.

[7] Kompilatory C++ mogą jednak dowolnie wstawiać nazwy. Na szczęście większość z nich nie deklaruje nazw, których nie potrzebuje.

```
//: C05:Tolower.cpp {-mwcc}
//{L} StrTolower
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
string strTolower(string);

int main() {
  string s("LOWER");
  cout << strTolower(s) << endl;
} ///:~
```

Inne rozwiązanie to napisanie szablonu funkcji opakowującej, który jawnie wywoła odpowiednią wersję **tolower()**:

```
//: C05:ToLower2.cpp
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

template<class charT>
charT strTolower(charT c) {
  return tolower(c);  // wywołanie wersji jednoargumentowej
}

int main() {
  string s("LOWER");
  transform(s.begin(),s.end(),s.begin(),&strTolower<char>);
  cout << s << endl;
} ///:~
```

Pokazana wersja ma tę zaletę, że może przetwarzać zarówno zwykłe, jak i szerokie znaki, gdyż typ znaku jest parametrem szablonu. Komitet standaryzacyjny C++ pracuje nad taką modyfikacją języka, która pozwoliłaby na zadziałanie także pierwszego przykładu (bez rzutowania) i w końcu nadejdzie dzień, kiedy o wszystkich tych sztuczkach można będzie zapomnieć.[8]

Stosowanie funkcji do sekwencji STL

Załóżmy, że chcemy wziąć kontener sekwencyjny STL (więcej o takich kontenerach dowiesz się w następnych rozdziałach; jak na razie wystarczy użyć znajomego **vector**) i zastosować do wszystkich jego obiektów pewną funkcję. Kontener **vector** może zawierać obiekty dowolnego typu, więc potrzebna jest funkcja działająca z kontenerem **vector** dowolnego typu:

```
//: C05:ApplySequence.h
// Stosuje funkcję do kontenera sekwencyjnego STL
```

[8] Dla osób zainteresowanych dodajmy, że zagadnienie to oznaczono jako „Core Issue 352”.

```
// const, 0 argumentów, wartość zwracana dowolnego typu
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)() const) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)();
}

// const, 1 argumentów, wartość zwracana dowolnego typu
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A) const, A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a);
}

// const, 2 argumenty, wartość zwracana dowolnego typu
template<class Seq, class T, class R,
         class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2) const,
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
     ((*it++)->*f)(a1, a2);
}
// nie const, 0 argumentów, wartość zwracana dowolnego typu
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)()) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)();
}

// nie const, 1 argument, wartość zwracana dowolnego typu
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A), A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a);
}

// nie const, 2 argumenty, wartość zwracana dowolnego typu
template<class Seq, class T, class R,
         class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end())
    ((*it++)->*f)(a1, a2);
}
// Itd., aby obsłużyć najbardziej prawdopodobne argumenty ///:~
```

Szablon funkcji **apply()** pobiera referencję do kontenera klasy i wskaźnik funkcji składowej obiektu umieszczonego w tej klasie. Do przechodzenia po sekwencji i stosowania funkcji dla każdego jej elementu służy iterator. Mamy przeciążone funkcje dla parametrów normalnych oraz z **const**, więc możemy stosować zarówno funkcje stałe, jak i nie-stałe.

Zwróć uwagę, że w **ApplySequence.h** nie korzystamy z plików nagłówkowych STL (a właściwie to z żadnych plików nagłówkowych), więc tak naprawdę nie jesteśmy ograniczeni jedynie do kontenerów STL. Jednak robimy pewne założenia dotyczące sekwencji STL (przede wszystkim co do nazwy i zachowania obiektu **iterator**).

Jak widać, istnieje tu więcej niż jedna wersja **apply()**, co znowu pokazuje przeciążanie szablonów funkcji. Wprawdzie szablony te pozwalają na stosowanie dowolnego typu wartości zwracanej (jest ona pomijana, choć informacja o jej typie jest potrzebna, aby dobrać wskaźnik funkcji składowej), to każda wersja ma inną liczbę argumentów; jest to szablon, więc argumenty te mogą być dowolnych typów. Jedyne ograniczenie polega na tym, że nie istnieje „super-szablon" pozwalający automatycznie tworzyć poszczególne szablony. To Ty musisz zdecydować, ile argumentów przewidzieć.

W celu przetestowania różnych przeciążonych wersji **apply()** stworzono klasę **Gromit**[9] mającą funkcje z różną liczbą argumentów:

```
//: C05:Gromit.h
// Sztuczny pies. Zawiera funkcje składowe
// o różnej liczbie argumentów
#include <iostream>

class Gromit {
  int arf;
  int totalBarks;
public:
  Gromit(int arf = 1) : arf(arf + 1), totalBarks(0) {}
  void speak(int) {
    for(int i = 0; i < arf; i++) {
      std::cout << "arf! ";
      ++totalBarks;
    }
    std::cout << std::endl;
  }
  char eat(float) const {
    std::cout << "chomp!" << std::endl;
    return 'z';
  }
  int sleep(char, double) const {
    std::cout << "zzz..." << std::endl;
    return 0;
  }
  void sit() {
    std::cout << "Siedzi..." << std::endl;
  }
}; ///:~
```

Teraz można użyć szablonu funkcji **apply()**, aby zastosować funkcje składowe klasy **Gromit** do **vector<Gromit*>**, na przykład:

```
//: C05:ApplyGromit.cpp
// Testowanie ApplySequence.h
#include <cstddef>
#include <iostream>
```

[9] Nawiązanie do angielskich animowanych filmów krótkometrażowych Nicka Parka, *Wallace i Gromit*.

```
#include <vector>
#include "ApplySequence.h"
#include "Gromit.h"
#include "../purge.h"
using namespace std;

int main() {
  vector<Gromit*> dogs;
  for(size_t i = 0; i < 5; i++)
    dogs.push_back(new Gromit(i));
  apply(dogs, &Gromit::speak, 1);
  apply(dogs, &Gromit::eat, 2.0f);
  apply(dogs, &Gromit::sleep, 'z', 3.0);
  apply(dogs, &Gromit::sit);
  purge(dogs);
} ///:~
```

Funkcja **purge()** to drobny programik pomocniczy wywołujący **delete** dla każdego elementu sekwencji. Jej definicja znajduje się w rozdziale 7.; funkcja ta jest używana w książce wielokrotnie.

Wprawdzie definicja **apply()** jest dość skomplikowana i raczej zbyt trudna dla nowicjusza, jej stosowanie jest proste i oczywiste, więc nowicjusz sobie świetnie poradzi z odpowiedzą na pytanie, *co* ta funkcja robi, o ile nie będzie zmuszany do zrozumienia, *jak* to robi. Tego typu zasady programowania warto stosować wszędzie w programach — trudne szczegóły wydziela się tak, aby znał je tylko ich projektant. Użytkowników interesuje jedynie to, co można osiągnąć, zaś użyta implementacja nie jest potrzebna. W następnym rozdziale zajmiemy się stosowaniem funkcji do sekwencji w sposób jeszcze bardziej elastyczny.

Częściowe uporządkowanie szablonów funkcji

Wspomnieliśmy wcześniej, że zwykłe przeciążanie funkcji **min()** jest lepszym rozwiązaniem od używania szablonów. Jeśli już istnieje funkcja pasująca do wywołania, to po co generować nową? Jeśli jednak zwykłej funkcji nie ma, przeciążanie szablonów funkcji może powodować wieloznaczności. Aby zminimalizować szansę wystąpienia tego typu problemów, zdefiniowano uporządkowanie szablonów funkcji, które pozwala wybrać szablon *najbardziej wyspecjalizowany*, o ile tylko istnieje. Jeden szablon funkcji uważa się za bardziej wyspecjalizowany od innego, jeśli wszystkie możliwe listy jego argumentów pasują do tego innego szablonu, ale nieprawdziwe jest stwierdzenie odwrotne. Spójrzmy na następujące deklaracje szablonów funkcji pochodzące z dokumentacji standardu C++:

```
template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);
```

Pierwszy szablon pasuje do dowolnego typu. Drugi szablon jest bardziej wyspecjalizowany, gdyż pasuje jedynie do typów wskaźnikowych. Innymi słowy, zbiór możliwych wywołań drugiego szablonu jest podzbiorem wywołań szablonu pierwszego.

Analogiczny związek istnieje między drugim a trzecim szablonem — trzeci może być wywołany jedynie ze wskaźnikami na wartości stałe, zaś drugi — z dowolnymi wskaźnikami. Poniższy przykład stanowi ilustrację opisanych tu zasad:

```
//: C05:PartialOrder.cpp
// Pokazuje uporządkowanie szablonów funkcji
#include <iostream>
using namespace std;

template<class T>
void f(T) {
  cout << "T\n";
}

template<class T>
void f(T*) {
  cout << "T*\n";
}

template<class T>
void f(const T*) {
  cout << "const T*\n";
}

int main() {
  f(0);            // T
  int i = 0;
  f(&i);           // T*
  const int j = 0;
  f(&j);           // const T*
} ///:~
```

Wywołanie **f(&i)** oczywiście pasuje do pierwszego szablonu, ale wywoływany jest szablon drugi jako bardziej wyspecjalizowany. Trzeci szablon nie może być w tym przypadku wywołany, gdyż wskaźnik z argumentu nie wskazuje wartości stałej. Wywołanie **f(&j)** pasuje do wszystkich trzech szablonów (na przykład w drugim szablonie **T** przyjęłoby wartość **const int**), ale użyty zostanie szablon trzeci jako najbardziej wyspecjalizowany.

Jeśli wśród dostępnych szablonów nie można określić tego „najbardziej wyspecjalizowanego", powstaje niejednoznaczność i kompilator zgłasza błąd. Stąd nazwa „uporządkowanie częściowe" — nie zawsze możliwe jest udzielenie jednoznacznej odpowiedzi. Analogiczne reguły dotyczą szablonów klas (więcej informacji na ten temat znajduje się dalej, w punkcie „Specjalizacja częściowa").

Specjalizacja szablonów

W języku C++ pojęcie *specjalizacja* ma ściśle określone znaczenie i wiąże się z szablonami. Definicja szablonu jest z samej swej natury pewnym *uogólnieniem* — szablon opisuje grupę funkcji lub klas w ramach pewnych pojęć ogólnych. Kiedy podawane

są argumenty szablonu, wynikiem jest jego specjalizacja, gdyż wybierana jest jedna konkretna funkcja lub klasa spośród wielu możliwych. Szablon funkcji **min()** pokazany na początku tego rozdziału jest uogólnieniem funkcji określającej minimum, gdyż nie jest podany typ parametrów takiej funkcji. Kiedy jako parametr szablonu podany zostanie typ, czy to jawnie, czy niejawnie (przez dedukcję na podstawie argumentów), kompilator generuje odpowiedni kod, na przykład **min<int>()**, będący specjalizacją szablonu. Kod wygenerowany jest uważany za *instancję* szablonu, tak samo jak każdy inny fragment kodu generowany według szablonów.

Specjalizacja jawna

Kod danej specjalizacji kodu można podać samemu, jeśli tylko zachodzi taka potrzeba. Podawanie własnych specjalizacji szablonu bywa potrzebne w przypadku szablonów klas, ale zaczniemy od szablonu funkcji **min()** — na jego przykładzie pokażemy obowiązującą składnię.

Przypomnijmy, że w pliku **MinTest.cpp** prezentowanym wcześniej w tym rozdziale pokazaliśmy następującą zwykłą funkcję:

```
const char* min(const char* a, const char* b) {
 return(strcmp(a, b) < 0) ? a : b;
}
```

Funkcja ta była potrzebna po to, aby w **min()** porównywane były łańcuchy, a nie ich adresy. Wprawdzie w tym wypadku niczego nie zyskujemy, ale jednak zdefiniujmy specjalizację szablonu **min()** dla typu **const char***, jak poniżej:

```
//: C05:MinTest2.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<class T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
// Jawna specjalizacja szablonu min
template<>
const char* const& min<const char*>(const char* const& a,
                                    const char* const& b) {
  return (strcmp(a, b) < 0) ? a : b;
}

int main() {
  const char *s2 = "mówią \"Ni-!\"", *s1 = "rycerze którzy";
  cout << min(s1, s2) << endl;
  cout << min<>(s1, s2) << endl;
} ///:~
```

Przedrostek **template<>** informuje kompilator, że dalej znajduje się specjalizacja szablonu. Typ specjalizacji musi pojawiać się w nawiasach kątowych zaraz za nazwą funkcji, jak miałoby to miejsce w wywołaniu jawnej specjalizacji. Zwróć uwagę na

dokładne podstawianie **const char*** zamiast **T**. Kiedy pierwotny szablon odnosi się do **T**, słowo kluczowe **const** oznacza modyfikację *całego* typu **T**. Jest to stały wskaźnik na daną typu **const char***. Wobec tego w specjalizacji zamiast **const T** trzeba zapisać **const char* const**. Kiedy kompilator znajdzie wywołanie **min()** z argumentem **const char***, użyje naszej wersji **const char*** funkcji **min()**. Dwa wywołania funkcji **min()** w powyższym przykładzie odwołują się do tej samej specjalizacji.

Jawne specjalizacje są bardziej przydatne w przypadku szablonów klas, a nie szablonów funkcji. Kiedy jednak podajesz pełną specjalizację szablonu klasy, konieczne może być zaimplementowanie wszystkich funkcji składowych. Wynika to stąd, że podajesz osobną klasę, zaś program klienta spodziewa się, że zaimplementowany będzie pełny interfejs.

Biblioteka standardowa zawiera jawną specjalizację klasy **vector** w przypadku przechowywania obiektów typu **bool**. Celem stosowania **vector<bool>** jest umożliwienie zaoszczędzenia miejsca w implementacjach biblioteki przez upakowanie bitów w liczby całkowite.[10]

Jak zauważyłeś wcześniej w tym rozdziale, deklaracja głównego szablonu klasy **vector** ma postać:

```
template <class T, class Allocator = allocator<T> >
class vector {...};
```

Aby stworzyć specjalizację szablonu dla obiektów typu **bool**, można byłoby jawną specjalizację zapisać następująco:

```
template <>
class vector<bool, allocator<bool> > {...};
```

Definicja ta zostanie natychmiast rozpoznana jako jawna specjalizacja, gdyż ma przedrostek **template<>**, zaś wszystkie podstawowe parametry szablonu są ustawione w liście argumentów dołączonej do nazwy klasy. Celem stosowania **vector<bool>** jest umożliwienie zaoszczędzenia w implementacji biblioteki pamięci przez spakowanie bitów w liczby całkowite.

Okazuje się, że specjalizacja **vector<bool>** jest elastyczniejsza niżby to wynikało z dotychczasowego opisu; zajmiemy się tym nieco dalej.

Specjalizacja częściowa

Szablony mogą być też częściowo specjalizowane, to znaczy przynajmniej jeden parametr szablonu pozostaje w takiej specjalizacji tak czy inaczej „otwarty”. Tak właśnie zachowuje się specjalizacja **vector<bool>** — określany jest typ obiektu (**bool**), ale alokator nie jest podawany. Oto faktyczna deklaracja specyfikacji **vector<bool>**:

```
template <class Allocator>
class vector<bool, Allocator>;
```

[10] **vector<bool>** szczegółowo omówimy w rozdziale 7.

Specjalizację częściową można rozpoznać po niepustej liście parametrów w nawiasach kątowych zarówno za słowem kluczowym **template** (są to parametry niepodane), jak i po nazwie klasy (parametry podane). Z uwagi na sposób zdefiniowania specjalizacji **vector<bool>** użytkownik może podać własny alokator mimo to, że typ **bool** jest już ustalony. Innymi słowy specjalizacja, a w szczególności specjalizacja częściowa, zapewniają pewnego rodzaju „przeciążanie" szablonów klas.

Częściowe uporządkowanie szablonów klas

Zasady decydujące, które szablony klas będą wybierane w poszczególnych sytuacjach, są podobne do częściowego uporządkowania szablonów funkcji — wybierany jest szablon „najbardziej wyspecjalizowany". Oto przykład — łańcuch w poszczególnych funkcjach składowych **f()** podaje znaczenie poszczególnych definicji szablonów:

```
//: C05:PartialOrder2.cpp
// Pokazuje częściowe uporządkowanie szablonów klas
#include <iostream>
using namespace std;

template<class T, class U> class C {
public:
  void f() {
    cout << "Szablon główny\n";
  }
};

template<class U> class C<int, U> {
public:
  void f() {
    cout << "T == int\n";
  }
};

template<class T> class C<T, double> {
public:
  void f() {
    cout << "U == double\n";
  }
};

template<class T, class U> class C<T*, U> {
public:
  void f() {
    cout << "użyto T*\n";
  }
};

template<class T, class U> class C<T, U*> {
public:
  void f() {
    cout << "użyto U*\n";
  }
};
```

```
template<class T, class U> class C<T*, U*> {
public:
  void f() {
    cout << "użyto T* i U*\n";
  }
};

template<class T> class C<T, T> {
public:
  void f() {
    cout << "T == U\n";
  }
};

int main() {
  C<float, int>().f();    // 1: Szablon główny
  C<int, float>().f();    // 2: T == int
  C<float, double>().f(); // 3: U == double
  C<float, float>().f();  // 4: T == U
  C<float*, float>().f(); // 5: użyto T* [T to float]
  C<float, float*>().f(); // 6: użyto U* [U to float]
  C<float*, int*>().f();  // 7: użyto T* i U* [float,int]

  // Poniższe wywołania są niejednoznaczne
//   8: C<int, int>().f();
//   9: C<double, double>().f();
//  10: C<float*, float*>().f();
//  11: C<int, int*>().f();
//  12: C<int*, int*>().f();
} ///:~
```

Jak widać, można częściowo wyspecyfikować parametry szablonu — mogą być np. wskaźnikami lub być takie same. Kiedy użyta zostanie specjalizacja **T***, jak w wierszu 5., **T** nie jest przekazywanym wskaźnikiem najwyższego rzędu; jest nim typ, do którego wskaźnik się odnosi (tutaj **float**). Specyfikacja **T*** to wzorzec dający się dopasować do typów wskaźnikowych. Gdyby trzeba było użyć **int****, jak w argumencie pierwszego szablonu, **T** byłoby typem **int***. Wiersz 8. jest niejednoznaczny, gdyż ma pierwszy parametr **int**, zaś dwa parametry szablonu są niezależne, a zatem jeden parametr nie jest bardziej wyspecyfikowany niż drugi. Analogiczne rozumowanie można przeprowadzić dla wierszy 9. do 12.

Przykład praktyczny

Łatwo jest dziedziczyć po szablonie klasy, można stworzyć nowy szablon będący instancją innego i po tym innym dziedziczący. Jeśli szablon **vector** zaspokaja prawie wszystkie Twoje potrzeby, ale w jakiejś aplikacji wskazane byłoby użycie jego wersji mogącej samodzielnie się posortować, w łatwy sposób można ponownie wykorzystać kod szablonu **vector**. Poniższy przykład pokazuje dziedziczenie po **vector<T>** oraz dodanie możliwości sortowania. Zauważ, że dziedziczenie po klasie **vector** niemającej destruktora wirtualnego mogłoby być niebezpieczne, gdyby trzeba było w destruktorze przeprowadzić jakieś porządki:

```
//: C05:Sortable.h
// Specjalizacja szablonu
#ifndef SORTABLE_H
#define SORTABLE_H
#include <string>
#include <vector>
using std::size_t;

template<class T>
class Sortable : public std::vector<T> {
public:
  void sort();
};

template<class T>
void Sortable<T>::sort() { // proste sortowanie
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(this->at(j-1) > this->at(j)) {
        T t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;;
      }
}

// Specjalizacja częściowa dla wskaźników
template<class T>
class Sortable<T*> : public std::vector<T*> {
public:
  void sort();
};

template<class T>
void Sortable<T*>::sort() {
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(*this->at(j-1) > *this->at(j)) {
        T* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}

// Pełna specjalizacja dla char*
// (jako funkcja inline dla wygody — normalnie treść funkcji
//  byłaby w osobnym pliku, tutaj byłaby tylko jej deklaracja)
template<> inline void Sortable<char*>::sort() {
  for(size_t i = this->size(); i > 0; --i)
    for(size_t j = 1; j < i; ++j)
      if(std::strcmp(this->at(j-1), this->at(j)) > 0) {
        char* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}
#endif // SORTABLE_H ///:~
```

Szablon **Sortable** nakłada ograniczenie na wszystkie wykorzystujące go klasy — muszą one obsługiwać operator >. Działa to prawidłowo tylko w przypadku obiektów niebędących wskaźnikami (w tym obiektów typów wbudowanych). W pełnej specjalizacji dla typu **char*** elementy porównywane są za pomocą funkcji **strcmp()**, dzięki czemu można sortować wektory zawierające dane typu **char*** zgodnie z kolejnością odpowiadających elementom tych wektorów łańcuchów. Użycie zapisu **this->** jest konieczne[11], a dlaczego, wyjaśnimy w dalszej części rozdziału.[12]

Oto plik wykorzystujący **Sortable.h**; dodatkowo wykorzystywany jest generator liczb losowych przedstawiony wcześniej:

```
//: C05:Sortable.cpp
//{bor} (z uwagi na bitset w Urand.h)
// Testowanie specjalizacji szablonu
#include <cstddef>
#include <iostream>
#include "Sortable.h"
#include "Urand.h"
using namespace std;

#define asz(a) (sizeof a / sizeof a[0])

char* words[] = {
  "is", "running", "big", "dog", "a",
};
char* words2[] = {
  "this", "that", "theother",
};

int main() {
  Sortable<int> is;
  Urand<47> rnd;
  for(size_t i = 0; i < 15; i++)
    is.push_back(rnd());
  for(size_t i = 0; i < is.size(); i++)
    cout << is[i] << ' ';
  cout << endl;
  is.sort();
  for(size_t i = 0; i < is.size(); i++)
    cout << is[i] << ' ';
  cout << endl;

  // Wykorzystywana jest częściowa specjalizacja szablonu
  Sortable<string*> ss;
  for(size_t i = 0; i < asz(words); i++)
    ss.push_back(new string(words[i]));
  for(size_t i = 0; i < ss.size(); i++)
    cout << *ss[i] << ' ';
  cout << endl;
  ss.sort();
```

[11] Zamiast **this->** można byłoby użyć dowolnej innej poprawnej kwalifikacji, na przykład **Sortable::at()** czy **vector<T>::at()**. Jakaś kwalifikacja jednak być musi.

[12] Zobacz też objaśnienia do programu **PriorityQueue6.cpp** w rozdziale 7.

```
  for(size_t i = 0; i < ss.size(); i++) {
    cout << *ss[i] << ' ';
    delete ss[i];
  }
  cout << endl;

  // Wykorzystywana jest pełna specjalizacja char*
  Sortable<char*> scp;
  for(size_t i = 0; i < asz(words2); i++)
    scp.push_back(words2[i]);
  for(size_t i = 0; i < scp.size(); i++)
    cout << scp[i] << ' ';
  cout << endl;
  scp.sort();
  for(size_t i = 0; i < scp.size(); i++)
    cout << scp[i] << ' ';
  cout << endl;
} ///:~
```

Każda z powyższych instancji szablonu korzysta z innej wersji tego szablonu. **Sortable<int>** korzysta z szablonu głównego. **Sortable<string*>** korzysta ze specjalizacji częściowej dla wskaźników. W końcu **Sortable<char*>** wykorzystuje pełną specjalizację dla **char***. Bez tej specjalizacji mogłoby się wydawać, że wszystko jest w porządku, gdyż słowa z tablicy **words** byłyby posortowane prawidłowo, a to dzięki częściowej specjalizacji porównującej pierwsze litery każdego słowa. Jednak już słowa z **words2** byłyby posortowane nieprawidłowo.

Unikanie nadmiarowego kodu

Zawsze, kiedy ukonkretniany jest szablon klasy, kod definicji takiej klasy dla danej specjalizacji jest generowany wraz ze wszystkimi funkcjami składowymi wywoływanymi w programie. Generowane są jedynie funkcje składowe, które są naprawdę wywoływane. To dobra wiadomość, co pokazuje poniższy program:

```
//: C05:DelayedInstantiation.cpp
// Funkcje składowe szablonu klasy nie są ukonkretniane,
// jeśli nie są potrzebne

class X {
public:
  void f() {}
};

class Y {
public:
  void g() {}
};

template <typename T> class Z {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
};
```

```
int main() {
  Z<X> zx;
  zx.a(); // Z<X>::b() nie zostanie utworzona
  Z<Y> zy;
  zy.b(); // Z<Y>::a() nie zostanie utworzona
} ///:~
```

Tutaj, mimo że szablon **Z** przewiduje użycie funkcji składowych **f()** i **g()** dla **T**, to fakt, że program można skompilować sugeruje, że generowane są jedynie **Z<X>::a()** wywoływane jawnie dla **zx**. Gdyby jednocześnie wygenerowana została **Z<X>::b()**, pokazany zostałby komunikat błędu z powodu próby wywołania nieistniejącej funkcji **X::g()**. Analogicznie, wywołanie **zy.b()** nie powoduje wygenerowania **Z<Y>::a()**. Wobec tego szablon **Z** może być użyty z **X** i **Y**; gdyby jednak wszystkie funkcje składowe były generowane już przy pierwszym tworzeniu klasy, zastosowanie wielu szablonów zostałoby znacząco ograniczone.

Załóżmy, że mamy pewien kontener — niech to będzie **Stack**. Użyjmy specjalizacji dla typów **int**, **int*** oraz **char***. Wygenerowane zostaną trzy wersje kodu **Stack** i zostaną one włączone do programu. Jednym z głównych powodów stosowania szablonów jest unikanie ręcznego powielania kodu. Kod jednak nadal jest powielany, tyle że robi to kompilator. Ograniczyć można narzut związany z implementacją dla typów wskaźnikowych, stosując kombinację specjalizacji pełnej i częściowej tak, aby uzyskać pojedynczą klasę. Kluczem do tego rozwiązania jest całkowita specjalizacja typu **void***, a następnie wyprowadzenie wszystkich innych typów wskaźnikowych z implementacji **void*** tak, aby wspólny kod mógł być wspólnie używany. Poniższy program ilustruje tę technikę:

```
//: C05:Nobloat.h
// Współdzielenie kodu zapisującego wskaźniki na stosie (Stack)
#ifndef NOBLOAT_H
#define NOBLOAT_H
#include <cassert>
#include <cstddef>
#include <cstring>

// Szablon główny
template<class T>
class Stack {
  T* data;
  std::size_t count;
  std::size_t capacity;
  enum {INIT = 5};
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new T[INIT];
  }
  void push(const T& t) {
    if (count == capacity) {
      // Zwiększ tablicę
      std::size_t newCapacity = 2*capacity;
      T* newData = new T[newCapacity];
      for (size_t i = 0; i < count; ++i)
        newData[i] = data[i];
```

```
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
    --count;
  }
  T top() const {
    assert(count > 0);
    return data[count-1];
  }
  size_t size() const {return count;}
};

// Pełna specjalizacja dla typu void*
template<>
class Stack<void *> {
  void** data;
  std::size_t count;
  std::size_t capacity;
  enum {INIT = 5};
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new void*[INIT];
  }
  void push(void* const & t) {
    if (count == capacity) {
      std::size_t newCapacity = 2*capacity;
      void** newData = new void*[newCapacity];
      std::memcpy(newData, data, count*sizeof(void*));
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
    --count;
  }
  void* top() const {
    assert(count > 0);
    return data[count-1];
  }
  std::size_t size() const {return count;}
};

// Częściowa specjalizacja dla innych typów wskaźnikowych
template<class T>
class Stack<T*> : private Stack<void *> {
  typedef Stack<void *> Base;
```

```
public:
  void push(T* const & t) {Base::push(t);}
  void pop() {Base::pop();}
  T* top() const {return static_cast<T*>(Base::top());}
  std::size_t size() {return Base::size();}
};
#endif // NOBLOAT_H ///:~
```

Pokazany prosty stos powiększa swoją pojemność w miarę wypełniania się danymi. Specjalizacja **void*** jest traktowana jako specjalizacja pełna dzięki przedrostkowi **template<>** (czyli lista parametrów szablonu jest pusta). Jak wspomniano wcześniej, konieczne jest zaimplementowanie w specjalizacji szablonu klasy wszystkich funkcji składowych. Oszczędności pojawiają się w przypadku innych typów wskaźnikowych. Częściowa specjalizacja innych typów wskaźnikowych dziedziczy prywatnie po **Stack <void*>**, gdyż ze **Stack<void*>** korzystamy po prostu jako z implementacji i nie chcemy ujawniać jej interfejsu innym użytkownikom. Funkcje składowe każdego ukonkretnienia wskaźnikiem to niewielkie funkcje pośredniczące w wywoływaniu odpowiednich funkcji **Stack<void*>**. Zatem zawsze, gdy użyty zostanie wskaźnik inny niż **void***, klasa będzie stanowiła niewielki procent jej wielkości w razie użycia szablonu głównego.[13] Oto program testujący:

```
//: C05:NobloatTest.cpp
#include <iostream>
#include <string>
#include "Nobloat.h"
using namespace std;

template<class StackType>
void emptyTheStack(StackType& stk) {
  while (stk.size() > 0) {
    cout << stk.top() << endl;
    stk.pop();
  }
}
// przeciążona emptyTheStack (to nie jest specjalizacja!)
template<class T>
void emptyTheStack(Stack<T*>& stk) {
  while (stk.size() > 0) {
    cout << *stk.top() << endl;
    stk.pop();
  }
}

int main() {
  Stack<int> s1;
  s1.push(1);
  s1.push(2);
  emptyTheStack(s1);
  Stack<int *> s2;
  int i = 3;
  int j = 4;
  s2.push(&i);
  s2.push(&j);
  emptyTheStack(s2);
} ///:~
```

[13] Funkcje wyrzedzające są typu inline, więc dla **Stack<void*>** nie zostanie wygenerowany żaden kod!

Dla wygody stworzyliśmy dwa szablony funkcji **emptyStack**. Szablony funkcji nie pozwalają dokonywać częściowej specjalizacji, więc podaliśmy szablony przeciążone. Druga wersja **emptyStack** jest bardziej wyspecjalizowana od pierwszej, więc będzie używana przy stosowaniu typów wskaźnikowych. W programie ukonkretniane są trzy szablony funkcji: **Stack<int>**, **Stack<void*>** oraz **Stack<int*>**. **Stack<void*>** jest ukonkretniany niejawnie, gdyż dziedziczy po nim **Stack<int*>**. Jeśli program wykorzystuje ukonkretnienia szablonów dla wielu rozmaitych typów wskaźnikowych, oszczędność związana z wielkością kodu wynikająca z użycia pojedynczego szablonu **Stack** może być znaczna.

Odszukiwanie nazw

Kiedy kompilator natyka się na identyfikator, musi określić jego typ i zasięg (a w przypadku zmiennych także czas życia). Wie o tym większość programistów, ale kiedy w grę wchodzą szablony, grono wszechwiedzących zmniejsza się. Kiedy kompilator po raz pierwszy natyka się na szablon, nie wszystko jest wiadome, więc kompilator, zanim będzie mógł szablonu poprawnie użyć, musi odnaleźć jego ukonkretnienie. Z tego wynikają dwie fazy kompilacji szablonów.

Nazwy w szablonach

W pierwszej fazie kompilator analizuje definicję szablonu, szukając w niej oczywistych błędów składniowych oraz interpretując wszystkie nazwy, które zinterpretować może. Nazwy, które już mogą być zinterpretowane, to te, które nie zależą od parametrów szablonu; nazwami tymi kompilator zajmuje się podczas normalnego odszukiwania nazw (i w razie potrzeby podczas odszukiwania nazw zależnych od argumentów, zobacz dalej). Nazwy, które jeszcze nie mogą być zinterpretowane, to tak zwane *nazwy zależne*, które zależą od argumentów szablonu. Wobec tego ukonkretnianie to druga faza kompilacji szablonu. Podczas tej fazy kompilator sprawdza, czy zamiast szablonu głównego trzeba użyć jego specjalizacji jawnej.

Zanim przejdziemy do przykładu, musimy zdefiniować jeszcze dwa pojęcia. ***Nazwa kwalifikowana*** to nazwa z nazwą klasy jako przedrostkiem, nazwa z nazwą obiektu i operatorem „kropka” lub nazwa ze wskaźnikiem obiektu i operatorem „strzałka”. Oto przykłady nazw kwalifikowanych:

```
MojaKlasa::f();
x.f();
p->f();
```

W książce tej nieraz już używaliśmy nazw kwalifikowanych; ostatnio w związku ze słowem kluczowym **typename**. Nazwy te są nazwami kwalifikowanymi, gdyż nazwy docelowe (jak **f** powyżej) są jawnie powiązane z klasą, dzięki czemu kompilator „wie”, gdzie szukać deklaracji tych nazw.

Inne pojęcie, które trzeba omówić, to ***odszukiwanie nazw zależne od parametrów***[14] (ang. *argument-dependent lookup — ADL*), technika służąca początkowo do uproszczenia wywołań funkcji niebędących składowymi (w tym operatorów) zadeklarowanych w przestrzeniach nazw. Spójrz na poniższy fragment kodu:

```
#include <iostream>
#include <string>
// ...
  std::string s("hello");
  std::cout << s << std::endl;
```

Zwróć uwagę na brak dyrektywy **using namespace std;**; jest to praktyka typowa między innymi dla plików nagłówkowych. Jeśli brak tej dyrektywy, w przypadku elementów z przestrzeni nazw **std** konieczne jest stosowanie kwalifikatora **std::**. Jednak nie wszędzie umieściliśmy deklaracje tej przestrzeni; widzisz, gdzie ich brak?

Nie podaliśmy, których funkcji operatorów należy użyć. Chcemy, aby zadziałało to jak poniżej, ale nie napisaliśmy tego!

```
std::operator<<(std::operator<<(std::cout, s),std::endl);
```

Aby oryginalna instrukcja przekazywania danych na standardowe wyjście zadziałała zgodnie z oczekiwaniami, ADL mówi, że kiedy pojawia się wywołanie funkcji bez kwalifikatora i deklaracja nie występuje w (normalnym) zasięgu, przeszukiwane są przestrzenie nazw każdego z jej argumentów i tam szuka się stosownej deklaracji. W oryginalnej instrukcji pierwsze wywołanie funkcji to:

```
operator<<(std::cout, s);
```

W pierwszym naszym fragmencie kodu takiej funkcji nie ma, więc kompilator stwierdza, że pierwszy argument tej funkcji, **std::cout**, pochodzi z przestrzeni nazw **std**. Wobec tego dodaje tę przestrzeń nazw do listy przeszukiwanych zakresów, w których szukana będzie funkcja pasująca do sygnatury **operator<<(std::ostream&, std::string)**. W pliku nagłówkowym **<string>** znajduje się zadeklarowana w przestrzeni nazw **std** funkcja, więc jest ona wywoływana.

Użycie przestrzeni nazw bez istnienia ADL byłoby niedogodne (ale z drugiej strony zauważ, że ADL powoduje, że pobierane są *wszystkie* deklaracje szukanej nazwy ze wszystkich odpowiednich przestrzeni nazw; jeśli brak najlepszego dopasowania, pojawią się niejednoznaczności).

Aby wyłączyć działanie ADL, można zamknąć nazwę funkcji w nawiasach:

```
(f)(x, y); // wyłączone ADL
```

Teraz przyjrzyjmy się programowi pochodzącemu z prezentacji Herba Suttera:

```
// Lookup.cpp
// Działa tylko z EDG i Metrowerks (opcja specjalna)
#include <iostream>
```

[14] Zwane też *odszukiwaniem Koeniga* od nazwiska Andrew Koeniga, który jako pierwszy zaproponował tę technikę komitetowi standaryzacyjnemu C++. ADL może być stosowane zawsze, niezależnie od tego, czy użyto szablonów czy też nie.

```
using std::cout;
using std::endl;

void f(double) { cout << "f(double)\n"; }

template<class T>
class X {
public:
 void g() { f(1); }
 };

void f(int) { cout << "f(int)\n"; }

int main() {
  X<int>().g();
} ///:~
```

Jedyny kompilator z badanych przez nas, który bezproblemowo przetworzy ten kod, to interfejs Edison Design Group[15], choć niektóre kompilatory, na przykład Metrowerks, mają opcję umożliwiającą poprawienie sposobu analizowania nazw. Wynikiem powinno być:

```
f(double)
```

gdyż **f** jest nazwą niezależną, która może zostać zinterpretowana na wczesnym etapie kompilacji przez sprawdzenie kontekstu, w którym szablon jest zdefiniowany, kiedy jedynie **f(double)** jest w zasięgu. Niestety, istnieje już mnóstwo kodu, którego działanie opiera się na niestandardowym zachowaniu: powiązaniu wywołania **f(1)** w **g()** z dalszym **f(int)**, tak że projektanci kompilatorów nie są skłonni do zmiany działania swoich produktów.

Oto przykład bardziej szczegółowy[16]:

```
//: C05:Lookup2.cpp {-bor} {-g++}{-dmc}
// Microsoft: użyj opcji -Za (tryb ANSI)
#include <algorithm>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

void g() { cout << "globalna g()\n"; }

template <class T>
class Y {
public:
  void g() { cout << "Y<" << typeid(T).name()
                  << ">::g()\n"; }
  void h() { cout << "Y<" << typeid(T).name()
                  << ">::h()\n"; }
  typedef int E;
};
```

[15] Interfejs ten wykorzystuje szereg kompilatorów, w tym Comeau C++.

[16] Także ten fragment oparty jest na przykładach Herba Suttera.

```
typedef double E;

template<class T>
void swap(T& t1, T& t2) {
  cout << "globalna swap()\n";
  T temp = t1;
  t1 = t2;
  t2 = temp;
}

template<class T>
class X : public Y<T> {
public:
  E f() {
    g();
    this->h();
    T t1 = T(), t2 = T(1);
    cout << t1 << endl;
    swap(t1, t2);
    std::swap(t1, t2);
    cout << typeid(E).name() << endl;
    return E(t2);
  }
};

int main() {
  X<int> x;
  cout << x.f() << endl;
} ///:~
```

Wynik działania tego programu powinien być następujący:

```
globalna g()
Y<int>::h()
0
globalna swap()
double
1
```

Przyjrzyjmy się teraz deklaracjom w **X::f()**:

- **E**, typ zwracany z **X::f()** nie jest nazwą zależną, więc jego analiza odbyła się już podczas analizowania szablonu; odnajdywana jest instrukcja **typedef** wiążąca **E** z **double**. Może wydawać się to dziwne, gdyż w przypadku klas niebędących szablonami najpierw znaleziona zostałaby deklaracja **E** w klasie bazowej, ale jest to zachowanie zgodne z zasadami C++ (klasa bazowa, **Y**, jest *zależną klasą bazową*, więc nie będzie przeszukiwania podczas definiowania szablonu).
- Wywołanie **g()** także jest niezależne, gdyż nie odwołuje się do **T**. Gdyby funkcja **g** miała parametry będące nazwami klasy zdefiniowanej w innej przestrzeni nazw, o dalszym działaniu zdecydowałby mechanizm ADL, gdyż w zasięgu nie ma **g** z parametrami. Tak jak jest to zapisane obecnie, wywołanie pasuje do globalnej deklaracji **g()**.

- Wywołanie **this->h()** to nazwa kwalifikowana, zaś kwalifikujący ją obiekt (tu **this**) odnosi się do obiektu aktualnego, typu **X**, który wskutek dziedziczenia zależy z kolei od nazwy **Y<T>**. W **X** nie ma funkcji **h()**, więc analiza spowoduje odszukanie zasięgu klasy bazowej **X**, czyli w **Y<T>**. Jest to nazwa zależna, więc jest odszukiwana podczas ukonkretniania, kiedy już znane jest **Y<T>** (włącznie z ewentualnymi specjalizacjami, które mogą być zapisane po definicji **X**), więc wywołana zostanie **Y<int>::h()**.
- Deklaracje **t1** i **t2** są zależne.
- Wywołanie **operator<<(cout, t1)** jest zależne, gdyż **t1** jest typu **T**. Jest przeszukiwane później, kiedy **T** jest utożsamione z **int**, zaś odpowiedni operator dla **int** znaleziony zostanie w **std**.
- Niekwalifikowane wywołanie **swap()** jest zależne, gdyż jego argumenty są typu **T**. Powoduje to ostatecznie ukonkretnienie globalnego **swap(int&, int&)**.
- Kwalifikowane wywołanie **std::swap()** *nie jest* zależne, gdyż **std** to ustalona przestrzeń nazw. Kompilator „wie", że ma szukać odpowiedniej deklaracji w **std** (aby nazwa kwalifikowana była traktowana jako zależna, kwalifikator na lewo od „**::**" musi odnosić się do parametru szablonu). Później szablon funkcji **std::swap()** generuje w chwili ukonkretniania **std::swap(int&, int&)**. W **X<T>::f()** nie pozostają już żadne nazwy zależne.

Podsumujmy zatem: odszukiwanie nazw ma miejsce podczas ukonkretniania, jeśli nazwa jest zależna, wyjątkiem są niekwalifikowane nazwy zależne, w przypadku których próba odszukania nazwy odbywa się już na etapie analizy definicji. Wszystkie niezależne nazwy występujące w szablonie też są wcześnie odszukiwane, w chwili analizy definicji szablonu (w razie potrzeby kolejne odszukiwanie ma miejsce podczas ukonkretniania, kiedy znany jest typ aktualnych argumentów).

Jeśli przeanalizowałeś ten przykład na tyle dokładnie, aby go zrozumieć, przygotuj się na jeszcze jedną niespodziankę związaną z deklarowaniem funkcji zaprzyjaźnionych.

Szablony i funkcje zaprzyjaźnione

Deklaracja funkcji zaprzyjaźnionej w klasie pozwala funkcjom niebędącym składowymi klasy sięgać do niepublicznych pól tej klasy. Jeśli nazwa funkcji zaprzyjaźnionej jest kwalifikowana, zostanie znaleziona w odpowiedniej przestrzeni nazw lub klasie. Jeśli jest jednak niekwalifikowana, kompilator musi założyć jakieś miejsce zdefiniowania tej funkcji, gdyż wszystkie identyfikatory muszą mieć jednoznacznie określony zasięg. Oczekuje się, że funkcja ta będzie zdefiniowana w najbliższej przestrzeni nazw zawierającej klasę, z którą funkcja jest zaprzyjaźniona. Często jest to po prostu zasięg globalny. Zagadnienie to wyjaśnia poniższy przykład (bez szablonów):

```
//: C05:FriendScope.cpp
#include <iostream>
using namespace std;

class Friendly {
  int i;
```

```
public:
  Friendly(int theInt) { i = theInt; }
  friend void f(const Friendly&); // wymaga definicji globalnej
  void g() { f(*this); }
};

void h() {
  f(Friendly(1));  // korzysta z ADL
}

void f(const Friendly& fo) {  // definicja funkcji zaprzyjaźnionej
  cout << fo.i << endl;
}

int main() {
  h();                // pokazuje 1
  Friendly(2).g();    // pokazuje 2
} ///:~
```

Deklaracja **f()** w klasie **Friendly** nie jest kwalifikowana, więc kompilator spodziewa się znaleźć definicję w zasięgu obejmującym plik (w tym wypadku przestrzeń nazw zawierająca klasę **Friendly**). Definicja ta pojawia się przed definicją funkcji **h()**. Powiązanie wywołania **f()** w **h()** z tą samą funkcją to już inna sprawa; tym zajmuje się ADL. Argumentem funkcji **f()** w **h()** jest obiekt **Friendly**, więc deklaracja **f()** jest szukana w klasie **Friendly** — z powodzeniem. Gdyby było tam wywołanie **f(1)** (co też ma sens, gdyż 1 można niejawnie przekształcić w **Friendly(1)**), wywołanie by się nie powiodło, gdyż kompilator „nie wie", gdzie ma szukać deklaracji **f()**. Kompilator EDG prawidłowo zgłosi, że w takim wypadku funkcja **f** nie jest zdefiniowana.

Załóżmy teraz, że **Friendly** i **f** są szablonami, jak poniżej:

```
//: C05:FriendScope2.cpp
#include <iostream>
using namespace std;

// Konieczne deklaracje uprzedzające
template<class T>
class Friendly;
template<class T>
void f(const Friendly<T>&);

template<class T>
class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f<>(const Friendly<T>&);
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

template<class T>
void f(const Friendly<T>& fo) {
  cout << fo.t << endl;
```

```
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

Najpierw zwróć uwagę na nawiasy kątowe w deklaracji **f** wewnątrz klasy **Friendly**. Są one konieczne, aby poinformować kompilator, że **f** jest szablonem. W przeciwnym przypadku kompilator szukałby bezskutecznie zwykłej funkcji **f**. W nawiasach moglibyśmy wstawić parametr szablonu (**<T>**), ale łatwo go wywnioskować z deklaracji.

Deklaracja wyprzedzająca szablonu funkcji **f** przed definicją klasy jest niezbędna, mimo że w poprzednim przykładzie nie była potrzebna. Standard języka mówi, że szablony funkcji zaprzyjaźnionych muszą być wcześniej deklarowane. Aby prawidłowo zadeklarować **f**, zadeklarowana musi być też klasa **Friendly**, gdyż **f** pobiera właśnie argument typu **Friendly**; stąd deklaracja wyprzedzająca tej klasy na początku. Pełną definicję **f** moglibyśmy umieścić zaraz za początkową deklaracją **Friendly**, bez oddzielania deklaracji od definicji, ale zdecydowaliśmy się zostawić kod w takiej postaci, bardziej podobnej do poprzedniego przykładu.

Pozostaje do rozpatrzenia jeszcze jeden przypadek użycia funkcji zaprzyjaźnionych w szablonach — pełne ich zdefiniowanie wewnątrz samej definicji szablonu klasy. Oto odpowiednia modyfikacja poprzedniego przykładu:

```
// C05:FriendScope3.cpp  {-bor}
// Microsoft: użyj opcji -Za (zgodnej z ANSI)
#include <iostream>
using namespace std;

template<class T>
class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f(const Friendly<T>& fo) {
    cout << fo.t << endl;
}
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

Między tym przykładem a poprzednim jest istotna różnica — **f** nie jest tu szablonem, ale zwykłą funkcją (pamiętaj, że nawiasy kątowe były niezbędne do zaznaczenia, że **f()** jest szablonem). Przy każdym ukonkretnieniu szablonu klasy **Friendly** tworzona jest nowa funkcja przeciążona, która jako argument ma aktualną specjalizację klasy

Friendly. To właśnie Dan Saks nazwał „poznawaniem nowych przyjaciół”.[17] Jest to najwygodniejszy sposób definiowania nowych funkcji zaprzyjaźnionych dla szablonów.

Aby rzecz wyjaśnić, załóżmy, że chcesz do szablonu klasy dodać zaprzyjaźnione operatory niebędące składowymi. Oto szablon klasy zawierającej po prostu ogólną wartość:

```
template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
};
```

Nowicjusze nierozumiejący poprzedniego przykładu denerwują się, gdyż nie potrafią zmusić do pracy prostego operatora wstawiającego dane do strumienia wyjściowego. Jeśli nie zdefiniujesz swoich operatorów w ramach definicji klasy **Box**, jak pokazano wcześniej, musisz dodać deklaracje wyprzedzające:

```
//: C05:Box1.cpp
// Definiowanie operatorów szablonu
#include <iostream>
using namespace std;

// Deklaracje uprzedzające
template<class T>
class Box;
template<class T>
Box<T> operator+(const Box<T>&, const Box<T>&);
template<class T>
ostream& operator<<(ostream&, const Box<T>&);

template<class T>
class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
  friend Box operator+<>(const Box<T>&, const Box<T>&);
  friend ostream& operator<< <>(ostream&, const Box<T>&);
};

template<class T>
Box<T> operator+(const Box<T>& b1, const Box<T>& b2) {
  return Box<T>(b1.t + b2.t);
}

template<class T>
ostream& operator<<(ostream& os, const Box<T>& b) {
  return os << '[' << b.t << ']';
}

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl;  // [3]
//  cout << b1 + 2 << endl; // brak konwersji niejawnych
} ///:~
```

[17] W przemówieniu mającym miejsce na seminarium *The C++ Seminar* w Portland w Oregonie we wrześniu 2001 roku.

Definiujemy tu i operator dodawania, i operator pisania do strumienia wyjściowego. Program główny ujawnia wady takiego rozwiązania — nie można polegać na konwersjach niejawnych (wyrażenie **b1 + 2**), gdyż szablony ich nie obsługują. Korzystanie z definicji wbudowanych w klasę, a nie w szablon, jest krótsze i bardziej niezawodne:

```
//: C05:Box2.cpp
// Definiowanie operatorów poza szablonami
#include <iostream>
using namespace std;

template<class T>
class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
  friend Box operator+(const Box<T>& b1,
                       const Box<T>& b2) {
    return Box<T>(b1.t + b2.t);
  }
  friend ostream& operator<<(ostream& os,
                             const Box<T>& b) {
    return os << '[' << b.t << ']';
  }
};

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl;  // [3]
  cout << b1 + 2 << endl;   // [3]
} ///:~
```

Operatory to zwykłe funkcje (przeciążane dla każdej specjalizacji **Box**, w tym wypadku **int**), normalnie stosowane są niejawne konwersje. Wobec tego wyrażenie **b1 + 2** jest poprawne.

Zwróć uwagę, że jest jeden typ, który nie może być zdefiniowany jako zaprzyjaźniony z **Box** ani żadnym innym szablonem klas. Typem tym jest **T**, a właściwie typ parametryzujący daną klasę. Nic nam nie wiadomo o jakichkolwiek istotnych powodach, dla których nie powinno to być dozwolone, tym niemniej deklaracja **friend class T** jest niepoprawna i nie powinna dać się skompilować.

Szablony zaprzyjaźnione

Można dokładnie opisać, które specjalizacje szablonu są zaprzyjaźnione z daną klasą. W poprzednich przykładach zaprzyjaźnione były jedynie specjalizacje szablonu funkcji **f** dla tego samego typu, dla którego specjalizowana była klasa **Friendly**. Na przykład z klasą **Friendly<int>** zaprzyjaźniona była jedynie funkcja **f<int>(const Friendly<int>&)**. Działo się tak wskutek użycia parametru szablonu **Friendly** w specjalizacji **f**, w deklaracji zaprzyjaźnienia. Gdybyśmy chcieli, moglibyśmy, na przykład, uczynić daną, ustaloną specjalizację **f**, przyjacielem wszystkich klas z rodziny **Friendly**:

```
// wewnątrz Friendly:
friend void f<>(const Friendly<double>&);
```

Jeśli zamiast **T** użyjemy **double**, specjalizacja **f** z typem **double** będzie miała dostęp do wszystkich niepublicznych składowych dowolnej specjalizacji szablonu **Friendly**. Specjalizacja **f<double>()** nadal nie będzie ukonkretniona, póki nie zostanie wywołana jawnie.

Analogicznie, jeśli zdefiniujesz funkcję niebędącą szablonem bez parametrów zależnych od **T**, dana pojedyncza funkcja będzie zaprzyjaźniona ze wszystkimi wystąpieniami **Friendly**:

```
// wewnątrz Friendly:
friend void g(int);  // g(int) zaprzyjaźnia się ze wszystkimi klasami Friendly
```

Jak zwykle, wobec tego, że **g(int)** nie jest kwalifikowana, musi być zdefiniowana w pliku (zasięgu przestrzeni nazw zawierającej klasę **Friendly**).

Możliwe jest też takie zakodowanie, aby wszystkie specjalizacje **f** były zaprzyjaźnione ze wszystkimi specjalizacjami **Friendly**; jest to tak zwany *szablon zaprzyjaźnienia*:

```
template<class T> class Friendly {
  template<class U> friend void f<>(const Friendly<U>&);
```

Deklaracja argumentu szablonu dla deklaracji zaprzyjaźnienia jest niezależna od **T**, więc dozwolona jest dowolna kombinacja **T** i **U**, a więc to, o co chodziło. Tak jak szablony funkcji składowych, szablony zaprzyjaźnienia mogą pojawiać się też w klasach niebędących szablonami.

Idiomy programowania za pomocą szablonów

Język to owoc myśli, więc nowe cechy języka powodują powstawanie nowych technik programowania. W tym podrozdziale zajmiemy się niektórymi typowymi idiomami programowania za pomocą szablonów, które pojawiły się od chwili dołączenia szablonów do definicji języka C++.[18]

Cechy charakterystyczne

Cechy charakterystyczne (ang. *traits*) to technika stworzona przez Nathana Myersa polegająca na grupowaniu deklaracji związanych z danym typem. Mówiąc najkrócej, cechy charakterystyczne pozwalają „mieszać i dobierać" pewne typy i wartości w zależności od bieżącego kontekstu pozwalającego używać ich w elastyczny sposób, dzięki czemu poprawia się czytelność i łatwość utrzymania kodu.

Najprostszym przykładem cech charakterystycznych jest szablon klasy **numeric_limits** zdefiniowany w pliku nagłówkowym **<limits>**. Główny szablon zdefiniowany się następująco:

[18] Jeszcze jeden idiom związany z szablonami będzie omówiony w rozdziale 9.

```
template<class T> class numeric_limits {
public:
  static const bool is_specialized = false;
  static inline T min() throw ();
  static inline T max() throw ();
  static const int digits = 0;
  static const int digits10 = 0;
  static const bool is_signed = false;
  static const bool is_integer = false;
  static const bool is_exact = false;
  static const int radix = 0;
  static T epsilon() throw();
  static T round_error() throw();
  static const int min_exponent = 0;
  static const int min_exponent10 = 0;
  static const int max_exponent = 0;
  static const int max_exponent10 = 0;
  static const bool has_infinity = false;
  static const bool has_quiet_NaN = false;
  static const bool has_signaling_NaN = false;
  static const float_denorm_style has_denorm =
                                  denorm_absent;
  static const bool has_denorm_loss = false;
  static T infinity() throw();
  static T quiet_NaN() throw();
  static T signaling_NaN() throw();
  static T denorm_min() throw();
  static const bool is_iec559 = false;
  static const bool is_bounded = false;
  static const bool is_modulo = false;
  static const bool traps = false;
  static const bool tinyness_before = false;
  static const float_round_style round_style =
                                  round_toward_zero;
};
```

Plik nagłówkowy **<limits>** zawiera specjalizacje dla wszystkich najważniejszych typów numerycznych (wtedy pole **is_specialized** ma wartość **true**). Aby uzyskać, na przykład, podstawę dla liczb **double** dla systemu zmiennoprzecinkowego, można użyć wyrażenia **numeric_limits<double>::radix**. Aby z kolei znaleźć najmniejszą możliwą liczbę całkowitą, używa się **numeric_limits<int>::min()**. Nie wszystkie składniki klasy **numeric_limits** mają zastosowanie do wszystkich typów podstawowych — na przykład **epsilon()** odnosi się jedynie do typów zmiennoprzecinkowych.

Wartości, które zawsze będą całkowite, definiowane są jako składowe pola statyczne. Te, które mogą nie być całkowite (na przykład najmniejsza możliwa wartość typu **float**) są implementowane jako statyczne funkcje *inline*. Wynika to z faktu, że w C++ tylko *całkowite* składowe pola statyczne mogą być inicjalizowane w definicji klasy.

W rozdziale 3. pokazywaliśmy użycie cech charakterystycznych znaków do kontrolowania sposobu przetwarzania znaków w klasach obsługujących łańcuchy. Klasy **std::string** oraz **std::wstring** są specjalizacjami szablonu **std::basic_string** zdefiniowanego następująco:

```
template<class charT,
  class traits = char_traits<charT>,
  class allocator = allocator<charT> >
  class basic_string;
```

Parametr szablonu **charT** oznacza używany typ znaków: **char** lub **wchar_t**. Główny szablon **char_traits** jest zwykle pusty, zaś w bibliotece standardowej definiowane są jego specjalizacje dla typów **char** i **wchar_t**. Oto specjalizacja **char_traits<char>** zgodna ze standardem C++:

```
template<> struct char_traits<char> {
  typedef char char_type;
  typedef int int_type;
  typedef streamoff off_type;
  typedef streampos pos_type;
  typedef mbstate_t state_type;
static void assign(char_type& c1, const char_type& c2);
static bool eq(const char_type& c1, const char_type& c2);
static bool lt(const char_type& c1, const char_type& c2);
static int compare(const char_type* s1,
                   const char_type* s2, size_t n);
static size_t length(const char_type* s);
static const char_type* find(const char_type* s,
                             size_t n,
                             const char_type& a);
static char_type* move(char_type* s1,
                       const char_type* s2, size_t n);
static char_type* copy(char_type* s1,
                       const char_type* s2, size_t n);
static char_type* assign(char_type* s, size_t n,
                         char_type a);
static int_type not_eof(const int_type& c);
static char_type to_char_type(const int_type& c);
static int_type to_int_type(const char_type& c);
static bool eq_int_type(const int_type& c1,
                        const int_type& c2);
static int_type eof();
};
```

Funkcje te są wykorzystywane przez szablon klasy **basic_string** do operacji na znakach związanych z przetwarzaniem łańcuchów. Kiedy deklarujesz zmienną typu **string**, na przykład:

```
std::string s;
```

tak naprawdę deklarujesz następujące **s** (z uwagi na istnienie domyślnych argumentów szablonu w specyfikacji **basic_string**):

```
std::basic_string<char, std::char_traits<char>,
                  std::allocator<char> > s;
```

Z uwagi na to, że cechy charakterystyczne znaków zostały oddzielone od szablonu klasy **basic_string**, można podać specjalne klasy opisujące znaki, zastępując nimi **std::char_traits**. Ilustruje to poniższy przykład:

```
//: C05:BearCorner.h
#ifndef BEARCORNER_H
#define BEARCORNER_H
```

```
#include <iostream>
using std::ostream;

// Klasy przedmiotów (podawanych gościom)
class Milk {
public:
  friend ostream& operator<<(ostream& os, const Milk&) {
    return os << "Mleko";
  }
};

class CondensedMilk {
public:
  friend ostream&
  operator<<(ostream& os, const CondensedMilk &) {
    return os << "Mleko skondensowane";
  }
};

class Honey {
public:
  friend ostream& operator<<(ostream& os, const Honey&) {
    return os << "Miód";
  }
};

class Cookies {
public:
  friend ostream& operator<<(ostream& os, const Cookies&) {
    return os << "Słodycze";
  }
};

// Klasy gości:
class Bear {
public:
  friend ostream& operator<<(ostream& os, const Bear&) {
    return os << "Teodor";
  }
};

class Boy {
public:
  friend ostream& operator<<(ostream& os, const Boy&) {
    return os << "Patryk";
  }
};

// Główny szablon cech (pusty może opisywać cechy wspólne)
template<class Guest> class GuestTraits;

// Cechy - specjalizacje typu Guest
template<> class GuestTraits<Bear> {
public:
  typedef CondensedMilk beverage_type;
  typedef Honey snack_type;
};
```

```
template<> class GuestTraits<Boy> {
public:
  typedef Milk beverage_type;
  typedef Cookies snack_type;
};
#endif // BEARCORNER_H ///:~

//: C05:BearCorner.cpp
// Pokazuje użycie klas cech charakterystycznych
#include <iostream>
#include "BearCorner.h"
using namespace std;

// Specjalizowane klasy cech
class MixedUpTraits {
public:
  typedef Milk beverage_type;
  typedef Honey snack_type;
};

// Szablon Guest (korzysta z klasy cech)
template<class Guest, class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << "Wchodzi " << theGuest
         << " podajemy " << bev
         << " i " << snack << endl;
  }
};

int main() {
  Boy cr;
  BearCorner<Boy> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear> pc2(pb);
  pc2.entertain();
  BearCorner<Bear, MixedUpTraits> pc3(pb);
  pc3.entertain();
} ///:~
```

W powyższym programie instancje klas gości, **Boy** i **Bear** (chłopiec i niedźwiedź), otrzymują poczęstunek zgodny ze swoimi gustami. Chłopcy lubią mleko i słodycze, zaś niedźwiedzie lubią mleko skondensowane i miód. To powiązanie gości z przedmiotami wykonywane jest przez specjalizację podstawowego (pustego) szablonu klasy z cechami charakterystycznymi. Domyślne argumenty **BearCorner** gwarantują, że goście otrzymają odpowiednie smakołyki, ale można to przypisanie zmienić, podając po prostu klasę, która opisze żądane zachowanie, jak w przypadku powyższej klasy **MixedUpTraits**. Wynikiem działania programu jest:

```
Wchodzi Patryk podajemy Mleko i Słodycze
Wchodzi Teodor podajemy Mleko skondensowane i Miód
Wchodzi Teodor podajemy Mleko i Miód
```

Użycie cech charakterystycznych ma dwie zasadnicze zalety: (1) zapewnia elastyczność i możliwość rozszerzania parowanych obiektów z odpowiednimi atrybutami lub działaniami oraz (2) zapewnia, że lista parametrów szablonu będzie krótka i czytelna. Gdyby z gościem powiązanych byłoby 30 typów, niewygodne byłoby podawanie w każdej deklaracji **BearCorner** wszystkich 30 argumentów bezpośrednio. Grupowanie typów w odrębne klasy z cechami znacząco upraszcza programowanie.

Technika cech charakterystycznych jest używana także przy implementowaniu strumieni i ustawień lokalnych, jak to widzieliśmy w rozdziale 4. Przykład cech charakterystycznych iteratora znaleźć można w pliku nagłówkowym **PrintSequence.h** w rozdziale 6.

Reguły

Jeśli przyjrzysz się specjalizacji **char_traits** dla typu **wchar_t**, zauważysz, że jest on praktycznie identyczny jak odpowiednik dla typu **char**:

```
template<> struct char_traits<wchar_t> {
  typedef wchar_t char_type;
  typedef win_t int_type;
  typedef streamoff off_type;
  typedef wstreampos pos_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char_type& c1, const char_type& c2);
  static int compare(const char_type* s1,
                     const char_type* s2, size_t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s,
                               size_t n,
                               const char_type& a);
  static char_type* move(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* copy(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n,
                           char_type a);
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,
                          const int_type& c2);
  static int_type eof();
};
```

Właściwie jedyna różnica to zestaw używanych typów: **char** i **int** lub **wchar_t** i **wint_t**. Funkcjonalność obu specjalizacji jest taka sama.[19] Stanowi to podkreślenie faktu, że klasy cech charakterystycznych mają zawierać właśnie *cechy charakterystyczne*, a to,

[19] Fakt, że **char_traits<>::compare()** może wywołać **strcmp()** w jednym wypadku, a w drugim **wcscmp()**, jest z naszego punktu widzenia nieistotny; wynik działania **compare()** jest taki sam.

co zmienia się między tymi klasami, to zwykle typy i wartości stałe, ewentualnie algorytmy stałe korzystające z parametrów szablonów związanych z typami. Klasy cech charakterystycznych zwykle same są szablonami, gdyż zawarte w nich typy i stałe mogą być rozpatrywane jako cechy charakterystyczne parametrów głównego szablonu (na przykład **char** i **wchar_t**).

Dobrze byłoby móc też powiązać pewną *funkcjonalność* z argumentami szablonu, aby program kliencki mógł łatwo dostosowywać je do swoich potrzeb. Na przykład poniższa wersja programu **BearCorner** obsługuje różne rodzaje rozrywek:

```
//: C05:BearCorner2.cpp
// Pokazuje użycie klas polityki
#include <iostream>
#include "BearCorner.h"
using namespace std;

// Klasy polityki (wymagają funkcji statycznej doAction()):
class Feed {
public:
  static const char* doAction() { return "je"; }
};

class Stuff {
public:
  static const char* doAction() { return "opycha się"; }
};

// Szablon Guest (korzysta z klas polityki i cech charakterystycznych)
template<class Guest, class Action,
         class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << theGuest << " " << Action::doAction()
         << ": " << bev
         << " i " << snack << endl;
  }
};

int main() {
  Boy cr;
  BearCorner<Boy, Feed> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear, Stuff> pc2(pb);
  pc2.entertain();
} ///:~
```

Parametr szablonu **Action** w klasie **BearCorner** powinien mieć statyczną funkcję składową **doAction()** używaną w **BearCorner<>::entertain()**. Użytkownicy mogą wybrać **Feed** lub **Stuff**; obie te klasy zawierają odpowiednią funkcję. Klasy zawierające tak

zakodowane funkcje nazywamy *klasami reguł*. „Reguły" (ang. *policy*) przyjmowania gości w powyższym programie są realizowane przez **Feed::doAction()** i **Stuff::doAction()**. Tutaj są to akurat zwykle klasy, ale równie dobrze mogłyby być szablonami; możliwe byłoby też łączenie ich z dziedziczeniem. Więcej o projektowaniu przy użyciu klas reguł znajdzie Czytelnik w książce Andrei Alexandrescu[20]; pozycja ta jest podstawowym źródłem wiedzy na ten temat.

Tajemniczo powtarzający się wzorzec szablonów

Każdy początkujący programista C++ jest w stanie zmodyfikować klasę tak, aby śledziła liczbę istniejących aktualnie obiektów tej klasy. Wystarczy dodać pola statyczne i zmodyfikować konstruktor i destruktor:

```
//: C05:CountedClass.cpp
// Zliczanie obiektów klasy przez pola statyczne
#include <iostream>
using namespace std;

class CountedClass {
  static int count;
public:
  CountedClass() { ++count; }
  CountedClass(const CountedClass&) { ++count; }
  ~CountedClass() { --count; }
  static int getCount() { return count; }
};

int CountedClass::count = 0;

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;   // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;   // 2
{ // dowolny zasięg:
    CountedClass c(b);
    cout << CountedClass::getCount() << endl; // 3
    a = c;
    cout << CountedClass::getCount() << endl; // 3
  }
  cout << CountedClass::getCount() << endl;   // 2
} ///:~
```

Wszystkie konstruktory klasy **CountedClass** zwiększają wartość pola statycznego **count**, zaś destruktory zmniejszają wartość tego pola. Statyczna funkcja składowa **getCount()** zwraca aktualną liczbę obiektów.

Ręczne dodawanie wszystkich tych pól w każdej klasie, której obiekty chcesz zliczać, jest zadaniem nudnym. Zwykle w językach obiektowych, jeśli kod się powtarza, korzysta się z dziedziczenia, ale w tym wypadku jest to półśrodek. Zauważ, co się stanie, kiedy wydzielisz logikę zliczania do klasy bazowej:

[20] *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley, 2001.

```
//: C05:CountedClass2.cpp
// Błędna próba zliczania obiektów
#include <iostream>
using namespace std;

class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};
int Counted::count = 0;

class CountedClass : public Counted {};
class CountedClass2 : public Counted {};

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 3 (błąd)
} ///:~
```

Wszystkie klasy pochodne **Counted** mają to samo statyczne pole, więc liczba obiektów jest faktycznie liczbą wszystkich obiektów klas pochodnych tej klasy. Potrzebna jest nam metoda automatycznego generowania *różnych* klas bazowych dla każdej klasy pochodnej. Służy do tego dziwna konstrukcja szablonu pokazana poniżej:

```
//: C05:CountedClass3.cpp
#include <iostream>
using namespace std;

template<class T>
class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted<T>&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};
template<class T>
int Counted<T>::count = 0;

// Dziwne definicje klas
class CountedClass : public Counted<CountedClass> {};
class CountedClass2 : public Counted<CountedClass2> {};

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
```

```
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 1 (!)
} ///:~
```

Każda klasa pochodna dziedziczy po innej klasie bazowej określanej przez użycie samej siebie (klasy pochodnej) jako parametru szablonu! Wydawać się może, że jest to definicja cykliczna; faktycznie, byłaby taką, gdyby jakiekolwiek pole klasy bazowej użyło w obliczeniach argumentu szablonu. Jednak żadne pola **Counted** nie zależą od **T**, wielkość **Counted** (wynosząca zero!) jest znana już podczas analizowania szablonu. Wobec tego nie ma znaczenia, który argument zostanie użyty do ukonkretnienia **Counted**, bo jej wielkość będzie zawsze taka sama. Wszelkie dziedziczenie po **Counted** może mieć miejsce zaraz po jej analizie, ponieważ nie występuje w ogóle rekurencja. Każda klasa bazowa jest niepowtarzalna, więc ma własne dane statyczne, dzięki czemu mamy wygodną metodę zliczania obiektów dowolnej klasy. Ten ciekawy idiom związany z dziedziczeniem jako pierwszy drukiem przedstawił Jim Coplien cytując go w artykule zatytułowanym „Curiously Recurring Template Patterns”[21].

Szablony i metaprogramowanie

W 1993 roku kompilatory zaczęły obsługiwać proste konstrukcje szablonów, dzięki czemu użytkownicy mogli definiować ogólne kontenery i funkcje. W tym samym czasie, kiedy zaczęto rozważać włączenie STL do standardu C++, między członkami komitetu standaryzacyjnego C++ krążyły błyskotliwe i zaskakujące przykłady kodowania, jak poniższy[22]:

```
//: C05:Factorial.cpp
// Obliczenia robione podczas kompilacji!
#include <iostream>
using namespace std;
template<int n>
struct Factorial {
   enum {val = Factorial<n-1>::val * n};
};
template<>
struct Factorial<0> {
   enum {val = 1};
};
int main() {
   cout << Factorial<12>::val << endl; // 479001600
} ///:~
```

To, że program ten pokaże prawidłową wartość **12!**, nie jest tak szokujące. To, co szokuje, to fakt, że wszystkie obliczenia są przeprowadzane, zanim program zostanie wykonany!

[21] *C++ Gems* pod redakcją Stana Lippmana, SIGS, 1996.

[22] Formalnie rzecz biorąc, występują tu stałe etapu kompilacji, więc zgodnie z powszechnie stosowaną konwencją ich nazwy powinny być zapisywane wielkimi literami. Tutaj zostawiliśmy je zapisane małymi literami, gdyż mają one być podobne do zmiennych.

Kiedy kompilator próbuje ukonkretnić **Factorial<12>**, okazuje się, że musi też ukonkretnić **Factorial<11>**, które z kolei wymaga **Factorial<10>** i tak dalej. W końcu rekurencja kończy się specjalizacją **Factorial<1>**, a wyniki obliczeń przekazywane są do pierwszego wywołania. W końcu wyrażenie **Factorial<12>::val** zastępowane jest stałą 479001600, po czym kompilacja się kończy. Wobec tego, że wszelkie obliczenia wykonuje kompilator, wartości tutaj użyte muszą być stałymi w chwili kompilacji, stąd użycie słowa kluczowego **enum**. Podczas wykonywania programu jedyne, co zostaje do zrobienia, to pokazanie stałej i znaku nowego wiersza. Aby przekonać się, że specjalizacja **Factorial** owocuje prawidłową wartością uzyskiwaną podczas kompilacji, można byłoby użyć tej wielkości jako rozmiaru tablicy:

```
double nums[Factorial<5>::val];
assert(sizeof nums == sizeof(double)*120);
```

Programowanie na poziomie kompilacji

To, co miało być wygodnym sposobem podstawiania parametrów będących typami, okazało się mechanizmem pozwalającym obsługiwać programowanie na poziomie kompilacji. Program tego typu nazywany jest **metaprogramem szablonowym** (gdyż tak naprawdę „programujesz program"), a jak się okazuje, metaprogramem o dużych możliwościach. Tak naprawdę metaprogramowanie za pomocą szablonów jest pełnym programowaniem w rozumieniu Turinga (ang. *Turing complete*), gdyż zawiera instrukcje warunkowe i pętle (realizowane za pomocą rekurencji). Zatem teoretycznie można w ten sposób zrealizować dowolne obliczenia.[23] Pokazany powyżej przykład liczenia silni pokazuje, jak należy implementować powtórzenia — należy napisać szablon rekurencyjny i w formie specjalizacji dopisać warunek końcowy. Poniższy przykład pokazuje sposób liczenia elementów ciągu Fibonacciego podczas kompilacji tą samą techniką:

```
//: C05:Fibonacci.cpp
#include <iostream>
using namespace std;

template<int n>
struct Fib {
   enum {val = Fib<n-1>::val + Fib<n-2>::val};
};
template<>
struct Fib<1> {
   enum {val = 1};
};
template<>
struct Fib<0> {
   enum {val = 0};
};

int main() {
   cout << Fib<5>::val << endl;   // 6
```

[23] W roku 1966 Böhm i Jacopini udowodnili, że dowolny język obsługujący instrukcje wyboru i pętle, pozwalający stosować dowolnie wiele zmiennych, jest równoważny maszynie Turinga, która jest uważana za zdolną do zapisania dowolnego algorytmu.

```
  cout << Fib<20>::val << endl;  // 6765
} ///:~
```

Z matematycznego punktu widzenia liczby Fibonacciego definiuje się następująco:

$$f_n = \begin{cases} 0, n = 0 \\ 1, n = 1 \\ f_{n-2} + f_{n-1}, n > 1 \end{cases}$$

Pierwsze dwa przypadki prowadzą do powyższych specjalizacji, zaś wzór z trzeciego wiersza staje się szablonem głównym.

Pętle na etapie kompilacji

Aby zrealizować pętlę w metaprogramie szablonowym, trzeba najpierw zapisać tę pętlę jako rekurencję. Aby, na przykład, podnieść liczbę całkowitą **n** do potęgi **p**, zamiast używać pętli jak poniżej:

```
int val = 1;
while(p--)
  val *= n;
```

trzeba ją przekształcić w procedurę rekurencyjną:

```
int power(int n, int p) {
  return (p == 0) ? 1 : n*power(n, p - 1);
}
```

To można bez problemu zapisać już jako metaprogram:

```
//: C05:Power.cpp
#include <iostream>
using namespace std;

template<int N, int P>
struct Power {
  enum {val = N * Power<N, P-1>::val};
};
template<int N>
struct Power<N, 0> {
  enum {val = 1};
};
int main() {
  cout << Power<2, 5>::val << endl;  // 32
} ///:~
```

Jako warunku końcowego trzeba użyć częściowej specjalizacji, gdyż wartość **N** nadal jest swobodnym parametrem szablonu. Zauważ, że program ten zadziała tylko dla wykładników potęg nieujemnych.

Poniższy metaprogram oparty na pracy Czarneckiego i Eiseneckera[24] jest interesujący o tyle, że korzysta z parametru szablonu będącego szablonem oraz symuluje przekazanie funkcji jako parametru innej funkcji, co powoduje przejście przez liczby **0..n**:

[24] Czarnecki i Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison Wesley, 2000, str. 417.

```
//: C05:Accumulate.cpp
// Na etapie kompilacji przekazuje "funkcję" jako parametr
#include <iostream>
using namespace std;

// Zbiera wyniki z F(0)..F(n)
template<int n, template<int> class F>
struct Accumulate {
   enum {val = Accumulate<n-1, F>::val + F<n>::val};
};
// Warunek końcowy (zwraca wartość F(0))
template<template<int> class F>
struct Accumulate<0, F> {
   enum {val = F<0>::val};
};
// Różne "funkcje":
template<int n>
struct Identity {
   enum {val = n};
};
template<int n>
struct Square {
   enum {val = n*n};
};
template<int n>
struct Cube {
   enum {val = n*n*n};
};
int main() {
   cout << Accumulate<4, Identity>::val << endl; // 10
   cout << Accumulate<4, Square>::val << endl;   // 30
   cout << Accumulate<4, Cube>::val << endl;     // 100
} ///:~
```

Szablon główny **Accumulate** próbuje wyliczyć sumę **F(n)+F(n – 1)...F(0)**. Warunek końcowy uzyskujemy przez specjalizację częściową „zwracającą" **F(0)**. Sam parametr **F** jest szablonem i zachowuje się jak funkcja, jak w poprzednich przykładach przedstawionych w tym punkcie. Szablony **Identity**, **Square** i **Cube** wyliczają odpowiednie funkcji dla swoich parametrów szablonów zgodnie ze swoimi nazwami (tożsamość, kwadrat i sześcian). Pierwsze ukonkretnienie **Accumulate** w **main()** wylicza sumę 4 + 3 + 2 + 1 + 0, gdyż funkcja **Identity** po prostu „zwraca" swój parametr szablon. Drugi wiersz funkcji **main()** powoduje dodanie kwadratów liczb (16 + 9 + 4 + 1 + 0), zaś ostatni wylicza sumę sześcianów (64 + 27 + 8 + 1 + 0).

Zwijanie pętli

Projektanci algorytmów zawsze dążyli do optymalizacji swoich programów. Jedna z uznanych metod optymalizacji, szczególnie przydatnych w programowaniu numerycznym, to zwijanie pętli — technika minimalizująca narzut związany z pętlami. Kwintesencją tej techniki jest mnożenie macierzy. Poniższa funkcja mnoży macierz i wektor (załóżmy, że wcześniej zdefiniowano **ROWS** i **COLS**):

```
void mult(int a[ROWS][COLS], int x[COLS], int y[ROWS]) {
  for(int i = 0; i < ROWS; ++i) {
      y[i] = 0;
```

```
      for(int j = 0; j < COLS; ++j)
        y[i] += a[i][j]*x[j];
  }
}
```

Jeśli **COLS** jest liczbą parzystą, narzut związany z inkrementacją i porównywaniem zmiennej kontrolnej pętli **j** można zredukować o połowę przez „zwinięcie" obliczeń w pętli wewnętrznej w pary:

```
void mult(int a[ROWS][COLS], int x[COLS], int y[ROWS]) {
  for(int i = 0; i < ROWS; ++i) {
      y[i] = 0;
      for(int j = 0; j < COLS; j += 2)
        y[i] += a[i][j]*x[j] + a[i][j+1]*x[j+1];
  }
}
```

Ogólnie rzecz biorąc, jeśli **COLS** dzieli się przez **k**, w każdej iteracji pętli wewnętrznej można wykonać **k** operacji, przez co znakomicie redukuje się narzut. Oszczędności będą widoczne jedynie w przypadku dużych tablic, ale takie właśnie tablice występują w poważnych obliczeniach matematycznych.

Także definiowanie funkcji jako funkcji *inline* stanowi pewną formę zwijania pętli. Przyjrzyjmy się następującej metodzie obliczania potęg liczb całkowitych:

```
//: C05:Unroll.cpp
// Zwijanie niejawnej pętli przez funkcje inline
#include <iostream>
using namespace std;

template<int n>
inline int power(int m) {
   return power<n-1>(m) * m;
}
template<>
inline int power<1>(int m) {
   return m;
}
template<>
inline int power<0>(int m) {
   return 1;
}
int main()
{
   int m = 4;
   cout << power<3>(m) << endl;
} ///:~
```

Kompilator musi wygenerować trzy specjalizacje **power<>**: jedną dla każdego z parametrów szablonu 3, 2 i 1. Kod każdej z tych funkcji może być zdefiniowany jako funkcja *inline*, kod faktycznie wstawiany do funkcji **main()** to wyrażenie **m*m*m**. Wobec tego zwykła specjalizacja szablonu w połączeniu z funkcjami *inline* pozwala całkowicie uniknąć narzutu związanego z kontrolą pętli.[25]

[25] Istnieje znacznie lepszy sposób obliczania potęg liczb całkowitych — algorytm rosyjskiego kmiecia (ang. *Russian Peasant*).

Opisana metoda użycia zwijania pętli ograniczona jest przez możliwą głębokość definiowania funkcji *inline* w konkretnym kompilatorze.

Instrukcje warunkowe na etapie kompilacji

Aby zasymulować instrukcje warunkowe na etapie kompilacji, można użyć w deklaracji **enum** trójargumentowego operatora warunkowego. Poniższy program wykorzystuje tę technikę do wyliczenia maksimum dwóch liczb całkowitych na etapie kompilacji:

```
//: C05:Max.cpp
#include <iostream>
using namespace std;

template<int n1, int n2>
struct Max {
   enum {val = n1 > n2 ? n1 : n2};
};
int main() {
   cout << Max<10, 20>::val << endl;  // 20
} ///:~
```

Jeśli chcesz na etapie kompilacji użyć warunków do zdecydowania o sposobie generowania kodu, możesz wykorzystać specjalizację dla wartości **true** i **false**:

```
//: C05:Conditionals.cpp
// Warunki wybierające kod na etapie kompilacji
#include <iostream>
using namespace std;

template<bool cond> struct Select {};

template<> class Select<true> {
static void statement1() {
    cout << "Wykonywana jest instrukcja statement1\n";
  }
public:
  static void f() { statement1(); }
};

template<> class Select<false> {
  static void statement2() {
    cout << "Wykonywana jest instrukcja statement2\n";
  }
public:
  static void f() { statement2(); }
};

template<bool cond> void execute() {
  Select<cond>::f();
}

int main() {
  execute<sizeof(int) == 4>();
} ///:~
```

Program ten jest równoważny:

```
if(cond)
  statement1();
else
  statement2();
```

za wyjątkiem tego, że wyrażenie **cond** jest wyznaczane podczas kompilacji, poza tym to kompilator ukonkretnia odpowiednie wersje **execute<>()** i **Select<>**. Funkcja **Select<>::f()** jest wykonywana jako część programu. Instrukcję **switch** można symulować podobnie, ale zamiast **true** i **false** trzeba specjalizować wartości odpowiadające każdemu z możliwych przypadków.

Asercje na etapie kompilacji

W rozdziale 2. korzystaliśmy z asercji jako elementu programowania defensywnego. Asercja to w zasadzie obliczenie wyrażenia logicznego i wykonanie stosownej akcji — jeśli warunek jest prawdziwy, nierobienie niczego, natomiast jeśli warunek jest fałszywy, przerwanie wykonywania programu i pokazanie stosownego komunikatu. Jeśli możliwe jest wyznaczenie wartości wyrażenia już na etapie kompilacji, skorzystaj z asercji interpretowanej na tym etapie. Poniższy przykład pokazuje zastosowanie techniki odwzorowującej wyrażenie logiczne na deklarację tablicy:

```
//: C05:StaticAssert1.cpp {-xo}
// Proste narzędzie realizujące asercje na etapie kompilacji

#define STATIC_ASSERT(x) \
  do { typedef int a[(x) ? 1 : -1]; } while(0)

int main() {
  STATIC_ASSERT(sizeof(int) <= sizeof(long)); // powiedzie się
  STATIC_ASSERT(sizeof(double) <= sizeof(int)); // zawiedzie
} ///:~
```

Pętla **do** tworzy tymczasowy zakres definicji tablicy **a**, której wielkość jest określona badanym warunkiem. Nie można zdefiniować tablicy o wielkości –1, więc jeśli warunek nie jest spełniony, wykonanie instrukcji się nie powiedzie.

Pokazaliśmy wcześniej, jak wyznaczać na etapie kompilacji wyrażenia logiczne. Jeśli chodzi o symulację asercji na etapie kompilacji, pozostaje jeszcze tylko pokazanie komunikatu o błędzie i przerwanie wykonywania programu. Wszystko, co jest nam potrzebne, to błąd kompilacji — problemem jest natomiast wstawienie do komunikatu błędu sensownej podpowiedzi dla użytkownika. Poniższy przykład pochodzący od Alexandrescu[26] korzysta ze specjalizacji szablonu, klasy lokalnej i odrobiny magii w formie makr:

```
//: C05:StaticAssert2.cpp
//{-g++}
#include <iostream>
using namespace std;

// Szablon i specjalizacja
template<bool>
```

[26] *Modern C++ Design*, strony 23 – 26.

```
struct StaticCheck {
   StaticCheck(...);
};

template<>
struct StaticCheck<false>{};

// Makro (generuje klasę lokalną)
#define STATIC_CHECK(expr, msg) {              \
   class Error_##msg{};                        \
   sizeof((StaticCheck<expr>(Error_##msg()))); \
}

// Wykrywa konwersje zawężające
template<class To, class From>
To safe_cast(From from) {
   STATIC_CHECK(sizeof(From) <= sizeof(To),
                NarrowingConversion);
   return reinterpret_cast<To>(from);
}

int main() {
   void* p = 0;
   int i = safe_cast<int>(p);
   cout << "rzutowanie int OK\n";
//!   char c = safe_cast<char>(p);
} ///:~
```

W przykładzie tym definiowany jest szablon funkcji **safe_cast<>()** pozwalający sprawdzić, czy rzutowany obiekt nie jest większy od obiektu, na który następuje rzutowanie. Jeśli wielkość obiektu docelowego jest mniejsza, użytkownik zostanie poinformowany na etapie kompilacji, że nastąpiła próba realizacji konwersji zawężającej. Zauważ, że szablon klasy **StaticCheck** jest o tyle dziwny, że na **StaticCheck<true>** można rzutować *cokolwiek* (a to z powodu wielokropka w konstruktorze[27]), zaś na **StaticCheck<false>** nie można przekształcić niczego, gdyż dla tej specjalizacji nie podano żadnych konwersji. Chodzi o to, aby spróbować utworzyć instancję nowej klasy i spróbować przekonwertować ją na **StaticCheck<true>** *na etapie kompilacji*, jeśli warunek jest prawdziwy, lub na **StaticCheck<false>**, jeśli warunek jest niespełniony. Operator **sizeof** działa podczas kompilacji, więc użyto go przy próbie konwersji. Jeśli warunek nie będzie spełniony, kompilator zgłosi błąd mówiący, że nie potrafi przekonwertować nowej klasy na **StaticCheck<false>** (dodatkowe nawiasy w wywołaniu **sizeof** w **STATIC_CHECK()** są potrzebne, aby kompilator nie potraktował wywołania **sizeof** jako wywołania go na funkcji, co jest niedopuszczalne). Aby uzyskać jakieś użyteczne informacje o błędzie, odpowiedni tekst umieszczamy w nazwie nowej klasy.

Najlepszym sposobem zrozumienia tej techniki jest przeanalizowanie konkretnego przypadku jej użycia. Spójrzmy na następujący wiersz z powyższej funkcji **main()**:

```
int i = safe_cast<int>(p);
```

[27] Nie wolno przekazywać typów obiektowych innych niż wbudowane do specyfikacji parametru z trójkropkiem, ale wobec tego, że pytamy tutaj jedynie o rozmiar (operacja wykonywana podczas kompilacji), wyrażenie nigdy nie będzie wyliczane podczas wykonywania programu.

Wywołanie **safe_cast<int>(p)** obejmuje następujące rozwinięcie makra zastępujące pierwszy wiersz kodu:

```
{                                                    \
   class Error_NarrowingConversion {};               \
    sizeof(StaticCheck<sizeof(void*) <= sizeof(int)> \
           (Error_NarrowingConversion()));           \
}
```

Przypomnijmy, że operator preprocesora sklejający fragmenty tekstu, ##, łączy swoje operandy w pojedyncze wyrażenie, zatem **Error_##NarrowingConversion** staje się równe **Error_NarrowingConversion**. Klasa **Error_NarrowingConversion** jest *klasą lokalną* (czyli jest zadeklarowana w zakresie bez przestrzeni nazw), gdyż nie jest potrzebna nigdzie indziej w programie. Zastosowanie operatora **sizeof** powoduje próbę ustalenia wielkości obiektu **StaticCheck<true>** (gdyż warunek **sizeof(void*) <= sizeof(int)** w stosowanych przez nas systemach jest prawdziwy) utworzonego niejawnie z obiektu tymczasowego zwróconego przez wywołanie **Error_NarrowingConversion()**. Kompilator zna rozmiar nowej klasy **Error_NarrowingConversion** (jest ona pusta), więc użycie **sizeof** na zewnętrznym poziomie **STATIC_CHECK()** na etapie kompilacji jest dozwolone. Konwersja z tymczasowego obiektu **Error_NarrowingConversion** na **StaticCheck<true>** udaje się, więc udaje się też zewnętrzne wywołanie **sizeof** i wykonywanie programu jest kontynuowane.

Zastanówmy się teraz, co by się stało, gdybyśmy z ostatniego wiersza funkcji **main()** usunęli komentarz:

```
char c = safe_cast<char>(p);
```

Makro **STATIC_CHECK()** w **safe_cast<char>(p)** jest rozwijane jako:

```
{                                                    \
   class Error_NarrowingConversion {};               \
    sizeof(StaticCheck<sizeof(void*) <= sizeof(char)> \
           (Error_NarrowingConversion()));           \
}
```

Wyrażenie **sizeof(void*) <= sizeof(char)** jest fałszywe, więc próba konwersji z tymczasowego obiektu **Error_NarrowingConversion** na **StaticCheck<false>** wygląda następująco:

```
sizeof(StaticCheck<false>(Error_NarrowingConversion()));
```

Konwersja ta się nie udaje, więc kompilator przerywa swoje działanie pokazując komunikat:

```
Cannot cast from 'Error_NarrowingConversion' to
'StaticCheck<0>' in function
char safe_cast<char,void *>(void *)
```

Nazwa klasy **Error_NarrowingConversion** jest komunikatem informacyjnym starannie ułożonym przez programistę. Ogólnie rzecz biorąc, aby zrealizować opisaną metodą asercję statyczną, wystarczy wywołać makro **STATIC_CHECK** z warunkiem wyznaczanym podczas kompilacji i z nazwą tak dobraną, aby opisywała błąd, jaki wystąpił.

Szablony wyrażeń

Być może najbardziej zaawansowanym zastosowaniem szablonów jest technika wynaleziona niezależnie w 1994 roku przez Todda Veldhuizena[28] i Daveeda Vandevoorde — *szablony wyrażeń*. Szablony wyrażeń umożliwiają intensywną optymalizację pewnych obliczeń na etapie kompilacji, dzięki czemu kod wynikowy jest co najmniej równie szybki jak ręcznie optymalizowany kod Fortranu, a przy tym zachowuje naturalną notację matematyczną dzięki przeciążaniu operatorów. Wprawdzie na co dzień techniki tej się raczej nie używa, ale jest ona podstawą wielu zaawansowanych bibliotek matematycznych pisanych w C++.[29]

Zobaczmy przykład stosowania szablonów wyrażeń dla typowych działań algebry liniowej, takich jak dodawanie dwóch macierzy lub wektorów[30], jak poniżej:

```
D = A + B + C;
```

W implementacjach najprymitywniejszych wyrażenia dawałyby szereg wartości tymczasowych — jedno dla **A + B**, jedno dla **(A + B) + C**. Kiedy zmienne te odpowiadają ogromnym macierzom lub wektorom, związane z takimi obliczeniami zużycie zasobów jest niedopuszczalny. Szablony wyrażeń pozwalają zrealizować te same wyrażenia bez wartości tymczasowych.

Poniższy program zawiera definicję klasy **MyVector** symulującej wektory matematyczne dowolnego wymiaru. Jako wymiaru wektora używamy argumentu szablonu niebędącego typem. Definiujemy też klasę **MyVectorSum** działającą jako klasa zastępcza dla sumy obiektów **MyVector**. Pozwala to skorzystać z opóźnionego wyliczania, tak że dodawanie składników wektora przeprowadzane jest na życzenie, bez konieczności stosowania wartości tymczasowych.

```
//: C05:MyVector.cpp
// Wyrzucenie wartości tymczasowych dzięki użyciu szablonów
#include <cstddef>
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;

// Klasa zastępująca sumy wektorów
template<class, size_t> class MyVectorSum;
```

[28] Przedruk oryginalnego artykułu Todda można znaleźć w książce Lippmana *C++ Gems*, SIGS, 1996. Trzeba tu też zaznaczyć, że poza zapewnieniem notacji matematycznej i zoptymalizowaniem kodu szablony wyrażeń pozwalają w umieszczać bibliotekach C++ paradygmaty i mechanizmy znane z innych języków programowania, jak wyrażenia lambda. Innym przykładem jest fantastyczna biblioteka klas **Spirit**, która jest parserem intensywnie wykorzystującym szablony wyrażeń, pozwalającym na zapis bezpośrednio w C++ (odpowiedniej) notacji EBNF, w wyniku czego tworzone parsery są wyjątkowo szybkie. Więcej informacji na ten temat można znaleźć pod adresem *http://spirit.sourceforge.net/*.

[29] A ściśle rzecz biorąc, bibliotek Blitz++ (*http://www.oonumerics.org/blitz/*), Matrix Template Library (*http://www.osl.iu.edu/research/mtl/*) i POOMA (*http://www.acl.lanl.gov/pooma/*).

[30] Chodzi o „wektor" w znaczeniu matematycznym — jednowymiarową tablicę z liczbami o ustalonej długości.

```
template<class T, size_t N>
class MyVector {
  T data[N];
public:
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for (size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  MyVector<T,N>& operator=(const MyVectorSum<T,N>& right);
  const T& operator[](size_t i) const {
    return data[i];
  }
  T& operator[](size_t i) {
    return data[i];
  }
};
// Klasa zastępcza zawierająca referencje; wykonuje opóźnione dodawanie
template <class T, size_t N>
class MyVectorSum {
  const MyVector<T,N>& left;
  const MyVector<T,N>& right;
public:
  MyVectorSum(const MyVector<T,N>& lhs,
              const MyVector<T,N>& rhs)
      : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
  }
};
// Operator obsługujący v3 = v1 + v2
template<class T, size_t N>
MyVector<T,N>&
MyVector<T,N>::operator=(const MyVectorSum<T,N>& right) {
  for (size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}
// operator+ po prostu zapamiętuje referencje
template<class T, size_t N>
inline MyVectorSum<T,N>
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N>(left, right);
}
// Funkcje ułatwiające zapisanie poniższego programu testowego
template<class T, size_t N>
void init(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    v[i] = rand() % 100;
}
template<class T, size_t N>
void print(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}
```

```
int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  MyVector<int, 5> v4;
  // Jeszcze nie obsługiwane:
//!  v4 = v1 + v2 + v3;
} ///:~
```

Klasa **MyVectorSum** nie wykonuje obliczeń w chwili jej utworzenia; po prostu zapamiętuje referencje dodawanych wektorów. Obliczenia mają miejsce jedynie w chwili sięgania do sumy wektorów (zobacz **operator[]()**). Przeciążenie operatora przypisania dla klasy **MyVector** pobierające argument **MyVectorSum** jest potrzebne do wyrażeń typu:

```
v1 = v2 + v3;  // dodawanie dwóch wektorów
```

Kiedy wyznaczane jest wyrażenie **v1 + v2**, zwracany jest obiekt **MyVectorSum** (a właściwie wstawiane w miejsce wywołania, gdyż **operator+()** zadeklarowano jako *inline*). Jest to mały obiekt o stałej wielkości (zawiera tylko dwie referencje). Wtedy wywoływany jest wspomniany wcześniej operator przypisania:

```
v3.operator=<int,5>(MyVectorSum<int,5>(v2, v3));
```

W ten sposób każdemu elementowi **v3** przypisywana jest suma odpowiednich elementów **v1** i **v2**, wyliczana na bieżąco. Nie są tworzone żadne tymczasowe obiekty **MyVector**.

Pokazany program nie pozwala wyliczać wyrażeń mających więcej niż dwa operandy, na przykład:

```
v4 = v1 + v2 + v3;
```

Wynika to stąd, że po pierwszym dodawaniu następuje próba wykonania drugiego dodawania:

```
(v1 + v2) + v3;
```

co wymagałoby zdefiniowania funkcji **operator+()** mającej pierwszy argument typu **MyVectorSum**, a drugi typu **MyVector**. Można byłoby próbować zastosować szereg przeciążeń w celu obsługi wszystkich możliwych sytuacji, ale lepiej pracę tę zrzucić na szablony, jak w poniższej wersji naszego programu:

```
//: C05:MyVector2.cpp
// Obsługuje sumy dowolnej długości, wykorzystując szablony wyrażeń
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;
```

```
// Klasa zastępcza dla sum wektorów
template<class, size_t, class, class> class MyVectorSum;

template<class T, size_t N>
class MyVector {
  T data[N];
public:
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for (size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  template<class Left, class Right>
  MyVector<T,N>&
    operator=(const MyVectorSum<T,N,Left,Right>& right);
  const T& operator[](size_t i) const {
    return data[i];
  }
  T& operator[](size_t i) {
    return data[i];
  }
};
// Pozwala łączyć MyVector i MyVectorSum
template <class T, size_t N, class Left, class Right>
class MyVectorSum {
  const Left& left;
  const Right& right;
public:
  MyVectorSum(const Left& lhs, const Right& rhs)
      : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
  }
};
template<class T, size_t N>
template<class Left, class Right>
MyVector<T,N>&
MyVector<T,N>::
operator=(const MyVectorSum<T,N,Left,Right>& right) {
  for (size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}
// operator+ jedynie przechowuje referencje
template<class T, size_t N>
inline MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return
    MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
      (left,right);
}

template<class T, size_t N, class Left, class Right>
inline
MyVectorSum<T, N, MyVectorSum<T,N,Left,Right>,
            MyVector<T,N> >
```

```
operator+(const MyVectorSum<T,N,Left,Right>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N,MyVectorSum<T,N,Left,Right>,
                         MyVector<T,N> >
    (left, right);
}
// Funkcje upraszczające poniższy program testowy
template<class T, size_t N>
void init(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    v[i] = rand() % 100;
}
template<class T, size_t N>
void print(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}
int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  // Już obsługiwane:
  MyVector<int, 5> v4;
  v4 = v1 + v2 + v3;
  print(v4);
  MyVector<int, 5> v5;
  v5 = v1 + v2 + v3 + v4;
  print(v5);
} ///:~
```

Analizator szablonów „wnioskuje” o typie argumentów sumy, korzystając z argumentów szablonu **Left** i **Right**, zbędne jest jawne podawanie tych typów. Szablon **MyVectorSum** ma dwa dodatkowe parametry, tak że pozwala zapisać sumę dowolnej kombinacji par **MyVector** i **MyVectorSum**.

Operator przypisania jest teraz składowym szablonem funkcji. Dzięki temu dowolna para **<T, N>** może być łączona z dowolną parą **<Left, Right>**, tak że obiektowi **MyVector** można przypisać **MyVectorSum** z referencjami dowolnej pary **MyVector** i **MyVectorSum**.

Prześledźmy teraz, jak poprzednio, przykładowe przypisanie, aby zrozumieć dokładnie działanie programu. Zacznijmy od wyrażenia:

```
v4 = v1 + v2 + v3;
```

Wyrażenia wynikowe są nieczytelne, więc w dalszych objaśnieniach przez **MVS** będziemy oznaczać **MyVectorSum**, pomijać będziemy argumenty szablonów.

Pierwsza operacja to **v1 + v2** wywołująca funkcję *inline* **operator+()**, która z kolei wstawia do strumienia kompilacji **MVS(v1, v2)**. Wynik jest dodawany do **v3**, co powoduje powstanie obiektu tymczasowego skonstruowanego według wyrażenia **MVS(MVS(v1, v2), v3)**. Ostatecznie całe wyrażenie przybiera postać:

```
v4.operator+(MVS(MVS(v1, v2), v3));
```

Wszystkie te przekształcenia wykonywane są przez kompilator i dlatego technika ta nosi nazwę „szablonów wyrażeń". Szablon **MyVectorSum** odpowiada wyrażeniu (w tym wypadku dodawaniu), zaś powyższe zagnieżdżone wywołania są pozostałością po analizie drzewa lewostronnie wiązanego wyrażenia **v1 + v2 + v3**.

Doskonały artykuł Angeliki Langer i Klausa Krefta objaśnia, jak technika ta może być rozszerzana na bardziej złożone obliczenia.[31]

Modele kompilacji szablonów

Być może zauważyłeś, że wszystkie nasze przykładowe szablony mieściły się w pojedynczych jednostkach kompilacji (na przykład całe były umieszczane w jednoplikowych programach lub w plikach nagłówkowych programów wieloplikowych). Jest to niezgodne z powszechnie stosowaną praktyką rozdzielania zwykłych definicji funkcji od ich deklaracji przez umieszczanie tych ostatnich w plikach nagłówkowych, a także wydzielania implementacji funkcji do odrębnych plików (**.cpp**).

Przyczyny takiej praktyki rozdzielania kodu to:

- Wstawianie do plików nagłówkowych funkcji innych niż *inline* prowadzi do wielokrotnych definicji funkcji, co powoduje błędy konsolidatora.
- Ukrycie implementacji przed użytkownikami pomaga skrócić czas kompilacji.
- Producenci mogą dostarczać wstępnie skompilowany kod (na potrzeby danego kompilatora) wraz z nagłówkami, tak że użytkownik nie może podejrzeć implementacji funkcji.
- Kompilacja przebiega szybciej, gdyż pliki nagłówkowe są mniejsze.

Model włączania

Z kolei szablony nie są kodem jako takim, ale stanowią instrukcje dotyczące sposobu generowania kodu. Jedynie ukonkretnienia szablonów są prawdziwym kodem C++. Kiedy kompilator już przeanalizuje całą definicję szablonu podczas kompilacji i potem natknie się na odwołanie do tego szablonu w tej samej jednostce kompilacji, musi

[31] Langer i Kreft, „C++ Expression Templates", *C/C++ Users Journal*, Marzec 2003. Zobacz też artykuł Thomasa Beckera o szablonach wyrażeń opublikowany w tym samym piśmie w czerwcu 2003 roku (artykuł ten był inspiracją przy pisaniu tej części rozdziału).

jakoś obsłużyć sytuację związaną z tym, że równoważne miejsce ukonkretnienia może być w innej jednostce translacji. Najbardziej typowe podejście zakłada generowanie kodu dla ukonkretnienia w każdej jednostce translacji i pozwolenie konsolidatorowi na wyrzucenie duplikatów. Takie rozwiązanie zadziała dobrze w przypadku funkcji *inline*, które nie mogą być jako *inline* potraktowane, a także w przypadku tablic funkcji wirtualnych; to jeden z powodów popularności tego rozwiązania. Tym niemniej w niektórych kompilatorach lepiej jest polegać na bardziej skomplikowanych schematach, aby uniknąć generowania danego ukonkretnienia więcej niż raz. Tak czy inaczej, to system translacji C++ ma za zadanie uniknięcie błędów związanych z wieloma równoważnymi miejscami ukonkretnienia.

Wadą opisanego rozwiązania jest fakt, że użytkownik widzi cały kod źródłowy wszystkich szablonów, co stanowi problem dla producentów bibliotek. Inną wadą modelu włączania jest to, że pliki nagłówkowe są znacznie większe niż byłyby w przypadku odrębnego kompilowania treści funkcji. Może to dramatycznie wydłużyć czas kompilacji.

Aby uniknąć przynajmniej częściowo dużych nagłówków związanych z modelem włączania, C++ umożliwia dwa niewykluczające się mechanizmy organizacji kodu: można ręcznie ukonkretniać każdą specjalizację wykorzystując *ukonkretnienie jawne* lub korzystać z *szablonów eksportowanych*, które w dużym stopniu pozwalają na oddzielnie przeprowadzanie kompilacji.

Ukonkretnianie jawne

Można ręcznie wskazać kompilatorowi, aby ukonkretniał wybrane specjalizacje szablonów. W przypadku wykorzystywania tej techniki konieczne jest posiadanie jednej i tylko jednej takiej dyrektywy dla każdej specjalizacji. W przeciwnym razie powstałyby błędy związane z wielokrotną definicją, tak jak w przypadku zwykłych funkcji nie-*inline* mających identyczne sygnatury. Aby to zilustrować, najpierw rozdzielmy (błędnie) deklarację szablonu **min()** z tego rozdziału od jego definicji. Dalej znajdować się będą wzorce zwykłych funkcji, nie-*inline*. Poniższy przykład składa się z pięciu plików:

- **OurMin.h** — zawiera deklarację szablonu funkcji **min()**.
- **OurMin.cpp** — zawiera definicję funkcji **min()**.
- **UseMin1.cpp** — próbuje użyć ukonkretnienia **min()** typem **int**.
- **UseMin2.cpp** — próbuje użyć ukonkretnienia **min()** typem **double**.
- **MinMain.cpp** — wywołuje **usemin1()** i **usemin2()**.

```
//: C05:OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// Deklaracja min()
template<typename T> const T& min(const T&, const T&);
#endif // OURMIN_H ///:~

// OurMin.cpp
#include "OurMin.h"
// Definicja min()
```

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

//: C05:UseMin1.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin1() {
  std::cout << min(1,2) << std::endl;
} ///:~

//: C05:UseMin2.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin2() {
  std::cout << min(3.1,4.2) << std::endl;
} ///:~

//: C05:MinMain.cpp
//{L} UseMin1 UseMin2 MinInstances
void usemin1();
void usemin2();

int main() {
  usemin1();
  usemin2();
} ///:~
```

Przy próbie skompilowania i skonsolidowania tego programu konsolidator zgłasza nierozwiązane odwołania zewnętrzne do **min<int>()** i **min<double>()**. Powodem jest fakt, że kiedy kompilator natyka się na specjalizacje **min()** w **UseMin1** i **UseMin2**, widoczna jest jedynie deklaracja **min()**. Definicja nie jest widoczna, więc kompilator zakłada, że pochodzi ona z jakiejś innej jednostki translacji, więc potrzebne specjalizacje nie są na tym etapie ukonkretniane, a to z kolei powoduje błędy konsolidatora.

Aby rozwiązać ten problem, stwórzmy nowy plik, **MinInstances.cpp**, który jawnie ukonkretnia potrzebne specjalizacje **min()**:

```
//: C05:MinInstances.cpp {O}
#include "OurMin.cpp"

// Jawnie ukonkretnia Min dla int i double
template const int& min<int>(const int&, const int&);
template const double& min<double>(const double&,
                                   const double&);
///:~
```

Aby ręcznie ukonkretnić dane specjalizacje szablonu, trzeba poprzedzić deklarację specjalizacji słowem kluczowym **template**. Zauważ, że konieczne jest włączenie pliku **OurMin.cpp**, a nie **OurMin.h**, gdyż kompilator w celu ukonkretnienia musi mieć definicję szablonu. Jest to jedyne miejsce w programie, w którym musimy to zrobić,[32]

[32] Jak wyjaśniliśmy wcześniej, szablon trzeba jawnie ukonkretnić tylko jeden raz w całym programie.

gdyż daje to potrzebne nam ukonkretnienie **min()**; w innych plikach wystarczą same deklaracje. Plik **OurMin.cpp** włączamy makrem preprocesora, więc dodajemy dyrektywy chroniące nas przed błędami włączania:

```
//: C05:OurMin.cpp {O}
#ifndef OURMIN_CPP
#define OURMIN_CPP
#include "OurMin.h"

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
#endif // OURMIN_CPP ///:~
```

Kiedy teraz skompilujemy wszystkie pliki w pojedynczy program, odpowiednie ukonkretnienia **min()** zostaną znalezione i program zadziała prawidłowo, dając w wyniku:

```
1
3.1
```

Ręcznie można ukonkretniać klasy i statyczne pola danych. Kiedy jawnie ukonkretniamy klasę, ukonkretniane są wszystkie funkcje składowe danej specjalizacji poza tymi, które były jawnie ukonkretnione wcześniej. Jest to ważne, gdyż w przypadku użycia tego mechanizmu szereg szablonów staje się nieprzydatnych; w szczególności szablony realizujące różne funkcje w zależności od typu parametryzującego. Niejawne ukonkretnianie ma zaletę — ukonkretniane są jedynie te funkcje składowe, które są wywoływane. Jawne ukonkretnianie jest przydatne w dużych projektach, gdzie pozwala uniknąć dużej części czasu potrzebnego na kompilację. To, czy używa się ukonkretniania jawnego czy niejawnego, nie zależy od tego, jaka metoda kompilacji szablonów jest stosowana. Można korzystać z ukonkretniania ręcznego zarówno w modelu włączania, jak i w modelu separacji (omawianym dalej).

Model separacji

Model separacji przy kompilacji szablonów polega rozdzieleniu definicji szablonów funkcji lub definicji pól statycznych od ich deklaracji w ramach jednostek translacji, jak to się robi w przypadku zwykłych funkcji i danych, przez *eksportowanie* szablonów. Po przeczytaniu poprzednich dwóch punktów musi to brzmieć dziwnie. Po co męczyć się z modelem włączania, jeśli można zastosować starą i wypróbowaną metodę? Przyczyny są zarówno historyczne, jak i techniczne.

Historycznie rzecz biorąc, model włączania był pierwszym, który znalazł szerokie zastosowanie — wszystkie kompilatory C++ obsługują ten model. Jednym z powodów tego był fakt, że model separacji został zdefiniowany bardzo późno, ale też model włączania łatwiej jest zaimplementować. Zanim do końca opisano semantykę modelu separacji, powstało mnóstwo działającego kodu.

Model separacji jest na tyle trudny w implementacji, że aż do teraz (lato 2003) obsługuje go tylko jeden interfejs kompilatorów (EDG), a i tak obecnie nadal wymagana jest dostępność kodu źródłowego w chwili kompilacji, aby móc przeprowadzać ukonkretnienia

na żądanie. Planuje się skorzystanie z jakiejś formy kodu pośredniego zamiast kodu źródłowego, aby można było rozpowszechniać szablony „wstępnie skompilowane" bez kodu źródłowego. Z uwagi na złożoność analizy nazw (wiążącą się z odszukiwaniem nazw zależnych w kontekście definicji szablonu omawialiśmy to w tym rozdziale) pełna definicja szablonu nadal musi być dostępna w jakiejś formie, aby szablon mógł być ukonkretniony w chwili, kiedy jest to potrzebne.

Składnia rozdzielająca kod źródłowy definicji szablonu od jego deklaracji jest dość prosta; używa się w niej słowa kluczowego **export**:

```
// C05:OurMin2.h
// min deklarowany jest jako szablon eksportowany
// (działa tylko w kompilatorach zgodnych z EDG)
#ifndef OURMIN2_H
#define OURMIN2_H
export template<typename T> const T& min(const T&, const T&);
#endif // OURMIN2_H ///:~
```

Analogicznie jak **inline** czy **virtual**, tak i słowo kluczowe **export** może być wykorzystane tylko raz w strumieniu kompilacji, kiedy eskportowany szablon się pojawia. Z tego powodu nie trzeba go powtarzać w pliku implementacji, choć zrobienie tego uważane jest za dobrą praktykę:

```
// C05:OurMin2.cpp
// Definicja eksportowanego szablonu min
// (działa tylko w kompilatorach zgodnych z EDG)
#include "OurMin2.h"
export
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
} ///:~
```

Pokazane poprzednio pliki **UseMin** muszą zawierać tylko odwołania do odpowiedniego pliku nagłówkowego (**OurMin2.h**), zaś program główny nie ulega zmianie. Wprawdzie wydaje się, że mamy tu do czynienia z pełną separacją, ale plik z definicją szablonu (**OurMin2.cpp**) nadal musi być wysłany użytkownikom, gdyż jest analizowany przy każdym ukonkretnianiu **min()**, i będzie tak, aż pojawi się jakaś postać kodu pośredniego. O ile zatem standard przewiduje prawdziwy model separacji, nie wszystkie jego zalety można osiągnąć już dzisiaj. Tylko jedna rodzina kompilatorów — te oparte na interfejsie EDG — obsługuje słowo kluczowe **export**; nawet te kompilatory nie wykorzystują w pełni możliwości rozpowszechniania szablonów w wersji skompilowanej.

Podsumowanie

Szablony znacznie wykraczają poza zwykłą parametryzację typem. Dzięki możliwości połączenia dedukcji typów argumentów, specjalizacji oraz metaprogramowania, szablony C++ stają się potężnym mechanizmem generowania kodu.

Jedną ze słabości szablonów C++, o której nie wspomnieliśmy, jest trudność interpretowania komunikatów błędów kompilacji. Ilość nieprzyjaznego tekstu generowanego przez kompilator może wprawić w zakłopotanie. W kompilatorach C++ poprawiono już komunikaty błędów związane z szablonami, zaś Leor Zolman napisał narzędzie **STLFilt**, które poprawia czytelność tych komunikatów dzięki pobraniu informacji przydatnych w praktyce i odrzuceniu całej reszty.[33]

Inny ważny wniosek, który wypływa z tego rozdziału, to ten, że *szablony narzucają interfejs*. Zatem, choć słowo kluczowe **template** oznacza „przyjmę każdy typ", kod definicji szablonu wymaga obsłużenia pewnych operatorów i funkcji składowych — i to jest właśnie ten interfejs. Zatem w praktyce definicja szablonu mówi „Przyjmę każdy typ spełniający taki interfejs". Znacznie prościej byłoby, gdyby kompilator mógł powiedzieć „Ejże, typ, którym chcesz ukonkretnić szablon, nie spełnia wymogów interfejsu — nie mogę wygenerować kodu!". Użycie szablonów oznacza pewnego rodzaju „opóźnione sprawdzanie typów", znacznie elastyczniejsze od czysto obiektowej praktyki wymagania, aby wszystkie typy pochodziły od pewnych klas bazowych.

W rozdziałach 6. i 7. dokładnie zajmiemy się słynnym zastosowaniem szablonów — podzbiorem biblioteki standardowej C++ nazywanym zwykle Standardową biblioteką szablonów (STL). W rozdziałach 9. i 10. korzystać będziemy także z technik związanych z szablonami, których w tym rozdziale nie omawialiśmy.

Ćwiczenia

Rozwiązania wybranych ćwiczeń można znaleźć w dokumencie elektronicznym *The Thinking in C++ Volume 2 Annotated Solution Guide* dostępnym za niewielką opłatą na stronie *www.MindView.net*.

1. Napisz szablon funkcji jednoargumentowej mający pojedynczy parametr szablonu będący typem. Stwórz pełną specjalizację dla typu **int**. Utwórz też wersję przeciążoną tej funkcji (nie jako szablon) mającą pojedynczy parametr **int**. Niech Twój program główny wywołuje te trzy odmiany funkcji na różne sposoby.
2. Napisz szablon klasy wykorzystującej **vector** do zaimplementowania struktury stosu.
3. Zmodyfikuj swoje rozwiązanie poprzedniego ćwiczenia tak, aby typ kontenera używanego do implementacji stosu był parametrem szablonu.
4. W poniższym kodzie klasa **NonComparable** nie ma funkcji **operator=()**. Czemu obecność klasy **HardLogic** spowodowałaby błąd kompilacji, zaś **SoftLogic** nie?

```
//: C05:Exercise4.cpp {-xo}
class Noncomparable {};
```

[33] Odwiedź witrynę *http://www.bdsoft.com/tools/stlfilt.html*.

```
struct HardLogic {
  Noncomparable nc1, nc2;
  void compare() {
    return nc1 == nc2; // Błąd kompilacji
  }
};

template<class T> struct SoftLogic {
  Noncomparable nc1, nc2;
  void noOp() {}
  void compare() {
    nc1 == nc2;
  }
};

int main() {
  SoftLogic<Noncomparable> l;
  l.noOp();
} ///:~
```

5. Napisz szablon funkcji mający jeden parametr będący typem (**T**), mający cztery argumenty funkcji: tablicę obiektów **T**, indeks początkowy, indeks końcowy (włącznie) oraz opcjonalną wartość początkową. Funkcja zwraca sumę wszystkich elementów tablicy z podanego zakresu i wartości początkowej. Do zdefiniowania domyślnej wartości początkowej użyj konstruktora **T**.

6. Powtórz poprzednie ćwiczenie, ale użyj jawnego ukonkretnienia w celu ręcznego utworzenia specjalizacji dla typów **int** i **double**, korzystając z technik omówionych w tym rozdziale.

7. Czemu poniższy fragment kodu nie daje się skompilować? (Podpowiedź: do czego mają dostęp funkcje składowe klasy?).

```
//: C05:Exercise7.cpp {-xo}
class Buddy {};

template<class T> class My {
  int i;
public:
  void play(My<Buddy>& s) {
    s.i = 3;
  }
};

int main() {
  My<int> h;
  My<Buddy> me, bud;
  h.play(bud);
  me.play(bud);
} ///:~
```

8. Czemu poniższy kod nie daje się skompilować?

```
//: C05:Exercise8.cpp {-xo}
template<class T> double pythag(T a, T b, T c) {
  return (-b + sqrt(double(b*b - 4*a*c))) / 2*a;
}
```

```
int main() {
  pythag(1, 2, 3);
  pythag(1.0, 2.0, 3.0);
  pythag(1, 2.0, 3.0);
  pythag<double>(1, 2.0, 3.0);
} ///:~
```

9. Napisz szablony pobierające parametry inne niż typy w następujących odmianach: **int**, wskaźnik **int**, wskaźnik statycznego pola klasy typu **int**, wskaźnik statycznej funkcji składowej.

10. Napisz szablon klasy mający dwa parametry będące typami. Zdefiniuj częściową specjalizację pierwszego parametru, inną specjalizację częściową określającą drugi parametr. W każdej specjalizacji dodaj pola niewystępujące w szablonie głównym.

11. Zdefiniuj szablon klasy **Bob** pobierający pojedynczy parametr będący typem. Niech **Bob** będzie zaprzyjaźniona ze wszystkimi ukonkretnieniami szablonu klasy **Friendly** oraz zaprzyjaźniona z klasami szablonu **Picky** tylko wtedy, gdy typ parametru **Bob** i **Picky** jest identyczny. Dodaj do klasy **Bob** funkcje pokazujące zaprzyjaźnienie.