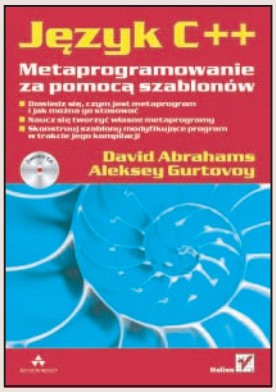
Język C++. Metaprogramowanie za pomocą szablonów



Autorzy: David Abrahams, Aleksey Gurtovoy
Tłumaczenie: Rafał Jońca
ISBN: 83-7361-935-6
Tytuł oryginału: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond
Format: B5, stron: 336

Metaprogramowanie to jedna z nowości, które pojawiły się ostatnio w świecie języka C++. Metaprogram to program będący w stanie modyfikować lub generować kod innego programu. Wykorzystanie zasad metaprogramowania pozwala na przykład na dynamiczną modyfikację programu podczas jego kompilacji. Pierwszym językiem pozwalającym na korzystanie z możliwości metaprogramowania jest C++ biblioteką STL.

„C++. Metaprogramowanie za pomocą szablonów" to książka przeznaczona dla tych programistów, którzy korzystają już z biblioteki STL i chcą zastosować ją do tworzenia metaprogramów. Opisano w niej zasady metaprogramowania, typy możliwe do wykorzystania w szablonach przeznaczonych do implementacji funkcji związanych z metaprogramowaniem oraz sposoby tworzenia szablonów modyfikujących programy podczas kompilacji.

- Typy i metafunkcje
- Operacje, sekwencje i iteratory
- Algorytmy biblioteki MPL i tworzenie własnych algorytmów
- Usuwanie błędów w szablonach
- Modyfikowanie programu w czasie kompilacji
- Język DSEL

Metaprogramowanie to nowość. Poznaj je już teraz, aby być przygotowanym na dzień, w którym stanie się standardem.

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

Spis treści

Rozdział 1.

Wprowadzenie

Warto potraktować ten rozdział jako rozgrzewkę przed pozostałą częścią książki. Przećwiczymy tutaj najważniejsze narzędzia, a także zapoznamy się z podstawowymi pojęciami i terminologią. Pod koniec rozdziału każdy powinien już mniej więcej wiedzieć, o czym jest niniejsza książka, i być głodnym kolejnych informacji.

1.1. Zaczynamy

Jedną z przyjemnych kwestii związanych z metaprogramowaniem szablonami jest współdzielenie pewnej właściwości z tradycyjnymi, starymi systemami. Po napisaniu metaprogramu można go używać bez zastanawiania się nad jego szczegółami — oczywiście o ile wszystko działa prawidłowo.

Aby uświadomić każdemu, że przedstawiona kwestia to nie tylko wymyślna teoria, prezentujemy prosty program C++, który po prostu używa elementu zaimplementowanego jako metaprogram szablonu.

```
#include "libs/mpl/book/chapter1/binary.hpp"
#include <iostream>

int main()
{
    std::cout << binary<101010>::value << std::endl;
    return 0;
}
```

Nawet jeśli jest się dobrym w arytmetyce binarnej i od razu można odgadnąć wynik działania programu bez jego uruchamiania, warto zadać sobie ten trud i go skompilować oraz uruchomić. Poza upewnieniem się w kwestii samej koncepcji, jest to dobry test sprawdzający, czy wykorzystywany kompilator potrafi obsłużyć kod przedstawiany w książce. Wynikiem działania programu powinno być wyświetlenie na standardowym wyjściu wartości dziesiętnej liczby binarnej 101010:

1.2. Czym jest metaprogram?

Gdy potraktować słowo **metaprogram** dosłownie, oznacza ono „program o programie”[1]. Od strony bardziej praktycznej metaprogram to program modyfikujący kod. Choć sama koncepcja brzmi nieco dziwacznie, zapewne nieraz nieświadomie korzystamy z takich rozwiązań. Przykładem może być kompilator C++, który modyfikuje kod C++ w taki sposób, by uzyskać kod w asemblerze lub kod maszynowy.

Generatory analizatorów składniowych takie jak YACC [Joh79] to kolejny przykład programów manipulujących programem. Wejściem dla YACC jest wysokopoziomowy opis analizatora składniowego zawierający zasady gramatyczne i odpowiednie polecenia umieszczone w nawiasach klamrowych. Aby na przykład przetworzyć i wykonać działania arytmetyczne zgodnie z przyjętą kolejnością wykonywania działań, można zastosować następujący opis gramatyki dla programu YACC.

```
expression : term
           | expression '+' term { $$ = $1 + $3; }
           | expression '-' term { $$ = $1 - $3; }
           ;
term : factor
     | term '*' factor { $$ = $1 * $3; }
     | term '/' factor { $$ = $1 / $3; }
     ;
factor : INTEGER
       | group
       ;
group : '(' expression ')'
      ;
```

Program YACC wygeneruje plik źródłowy języka C++ zawierający (poza wieloma innymi elementami) funkcję `yyparse()`, którą wywołuje się w celu przeanalizowania tekstu zgodnie z podaną gramatyką i wykonania określonych działań[2].

```
int main()
{
  extern int yyparse();
  return yyparse();
}
```

Użytkownicy programu YACC działają przede wszystkim w dziedzinie projektowania analizatorów składniowych, więc język YACC można nazwać **językiem specjalistycznym** (**dziedzinowym**). Ponieważ pozostała część głównego programu wymaga zastosowania ogólnego języka programowania i musi się komunikować z analizatorem składniowym, YACC konwertuje język specjalistyczny na **język macierzysty**, C, który użytkownik kompiluje i konsoliduje z pozostałym kodem. Język specjalistyczny przechodzi więc przez dwa kroki przekształceń, a użytkownik bardzo dobrze zna granicę między nim a pozostałą częścią programu głównego.

---

[1] W filozofii, podobnie jak w programowaniu, przedrostek „meta” oznacza „o” lub „o jeden poziom opisowy wyżej”. Wynika to z oryginalnego greckiego znaczenia „ponad” lub „poza”.

[2] Oczywiście trzeba jeszcze zaimplementować odpowiednią funkcję `yylex()` dokonującą rozbioru tekstu. W rozdziale 10. znajduje się pełny przykład. Ewentualnie warto zajrzeć do dokumentacji programu YACC.

1.3. Metaprogramowanie w języku macierzystym

YACC to przykład translatora — metaprogramu, którego język specjalistyczny różni się od języka macierzystego. Bardziej interesująca postać metaprogramowania jest dostępna w językach takich jak Scheme [SS75]. Programista metaprogramu Scheme definiuje własny język specjalistyczny jako podzbiór dopuszczalnych programów samego języka Scheme. Metaprogram wykonuje się w tym samym kroku przekształceń co pozostała część programu użytkownika. Programiści często przemieszczają się między typowym programowaniem, metaprogramowaniem i pisaniem języków specjalistycznych, nawet tego nie dostrzegając. Co więcej, potrafią w sposób niemalże nierozróżnialny scalić w tym samym systemie wiele dziedzin.

Co ciekawe, kompilator C++ zapewnia niemalże dokładnie taką samą użyteczność metaprogramowania jak przedstawiony wcześniej przykład. Pozostała część książki omawia odblokowywanie siły tkwiącej w szablonach i opisuje sposoby jej użycia.

1.4. Metaprogramowanie w języku C++

W języku C++ metaprogramowanie odkryto niemalże przypadkowo ([Unruh94], [Veld95b]), gdy udowodniono, iż szablony zapewniają bardzo elastyczny język metaprogramowania. W niniejszym podrozdziale omówimy podstawowe mechanizmy i typowe rozwiązania używane w metaprogramowaniu w języku C++.

1.4.1. Obliczenia numeryczne

Najprostsze metaprogramy C++ wykonują obliczenia na liczbach całkowitych w trakcie kompilacji. Jeden z pierwszych metaprogramów został przedstawiony na spotkaniu komitetu C++ przez Erwina Unruha — w zasadzie był to niedozwolony fragment kodu, którego komunikat o błędzie zawierał ciąg wyliczonych liczb pierwszych!

Ponieważ niedozwolonego kodu nie da się wydajnie stosować w dużych systemach, przyjrzyjmy się bardziej praktycznym aplikacjom. Kolejny metaprogram (który leży u podstaw przedstawionego wcześniej testu kompilatora) zamienia liczby dziesiętne bez znaku na ich odpowiedniki binarne, co umożliwia wyrażanie stałych binarnych w przyjaznej formie.

```
template <unsigned long N>
struct binary
{
    static unsigned const value
       = binary<N/10>::value * 2       // dodaje bardziej znaczący bit
         + N%10;                       // przejście do mniej znaczącego bitu
};
```

```
template <>                              // specjalizacja
struct binary<0>                         // przerywa rekurencję
{
    static unsigned const value = 0;
};

unsigned const one   =    binary<1>::value;
unsigned const three =   binary<11>::value;
unsigned const five  =  binary<101>::value;
unsigned const seven =  binary<111>::value;
unsigned const nine  = binary<1001>::value;
```

Jeżeli ktoś zastanawia się, gdzie jest program, proponujemy rozważyć, co się stanie w momencie próby dostępu do zagnieżdżonej składowej `::value` z `binary<N>`. Tworzy się egzemplarze szablonu `binary` z coraz to mniejszymi `N` aż do osiągnięcia przez `N` zera. Warunkiem końca jest specjalizacja. Innymi słowy, przypomina to działanie funkcji rekurencyjnej. Czy jest to program czy może funkcja? Ogólnie rzecz biorąc, kompilator **zinterpretuje** ten krótki metaprogram.

Sprawdzanie błędów

Nic nie stoi na przeszkodzie, aby użytkownik przekazał do `binary` wartość 678, która nie jest poprawną wartością binarną. Wynik na pewno nie będzie sensowny (zostanie wykonane działanie $6 \cdot 2^2 + 7 \cdot 2^1 + 8 \cdot 2^0$), a przekazanie wartości 678 na pewno wskazuje błąd użytkownika. W rozdziale 3. przedstawimy rozwiązanie zapewniające, iż `binary<N>::value` skompiluje się tylko wtedy, gdy reprezentacja dziesiętna `N` będzie się składała tylko z samych zer i jedynek.

Ponieważ język C++ wprowadza rozróżnienie między wyrażeniami obliczanymi w trakcie kompilacji i w trakcie działania programu, metaprogramy wyglądają inaczej niż ich tradycyjne odpowiedniki. Podobnie jak w Scheme programista metaprogramów C++ pisze kod w tym samym języku co tradycyjne programy, ale w C++ ma dostęp tylko do podzbioru elementów języka związanych z etapem kompilacji. Porównajmy poprzedni program z prostą wersją `binary` wykonaną jako tradycyjny program.

```
unsigned binary(unsigned long N)
{
    return N == 0 ? 0 : N%10 + 2 * binary(N/10);
}
```

Podstawowa różnica między przedstawionymi wersjami polega na sposobie obsługi warunku zakończenia: metaprogram używa **specjalizacji szablonu** do opisu tego, co dzieje się dla `N` równego zero. Przerywanie za pomocą specjalizacji to element wspólny niemal dla wszystkich metaprogamów C++, choć czasem wszystko ukryte jest za interfejsem biblioteki metaprogramowania.

Inną bardzo ważną różnicę między językiem C++ tradycyjnym i wykonywanym w trakcie kompilacji obrazuje poniższa wersja `binary`, która korzysta z pętli `for` zamiast z rekurencji.

```
unsigned binary(unsigned long N)
{
    unsigned result = 0;
    for (unsigned bit = 0x1; N; N /= 10, bit <<= 1)
```

```
    {
        if (N%10)
            result += bit;
    }
    return result;
}
```

Choć ta wersja jest dłuższa od rozwiązania rekurencyjnego, zapewne zastosuje ją większość programistów C++, gdyż jest na ogół wydajniejsza od rekurencji.

Część języka C++ związana z czasem kompilacji nazywana jest często „językiem czysto funkcyjnym", a to z powodu właściwości, jakie współdzieli z językami takimi jak Haskell: (meta)dane są niezmienne, a (meta)funkcje nie mogą mieć efektów ubocznych. Wynika z tego, iż C++ czasu kompilacji nie posiada tradycyjnych zmiennych używanych w typowym języku C++. Ponieważ nie można napisać pętli (poza pętlą nieskończoną) bez sprawdzania pewnego zmiennego stanu zakończenia, iteracje po prostu nie są dostępne w trakcie kompilacji. Z tego względu w metaprogramach C++ wszechobecna jest rekurencja.

1.4.2. Obliczenia typu

Dużo ważniejsza od obliczania wartości liczbowych w trakcie kompilacji jest zdolność języka C++ do obliczania **typów**. W zasadzie w pozostałej części książki dominuje obliczanie typu — pierwszy przykład przedstawiamy już na początku kolejnego rozdziału. Choć jesteśmy tutaj bardzo bezpośredni, zapewne większość osób traktuje metaprogramowanie szablonami jako „obliczenia związane z typami".

Choć dla dobrego zrozumienia obliczeń typów warto przeczytać rozdział 2., już teraz zamierzamy przedstawić przedsmak ich siły. Pamiętasz kalkulator wyrażeń wykonany w YACC? Wychodzi na to, iż nie potrzebujemy translatora, aby osiągnąć podobne działanie. Po odpowiednim otoczeniu kodu przez bibliotekę Boost Spirit poniższy w pełni poprawny kod C++ działa w zasadzie identycznie.

```
expr =
      ( term[expr.val = _1] >> '+' >> expr[expr.val += _1] )
    | ( term[expr.val = _1] >> '-' >> expr[expr.val -= _1] )
    | term[expr.val = _1]
    ;

term =
      ( factor[term.val = _1] >> '*' >> term[term.val *= _1] )
    | ( factor[term.val = _1] >> '/' >> term[term.val /= _1] )
    | factor[term.val = _1]
    ;

factor =
      integer[factor.val = _1]
    | ( '(' >> expr[factor.val = _1] >> ')' )
    ;
```

Każde przypisanie zapamiętuje obiekt funkcji, który analizuje i oblicza element gramatyki podany po prawej stronie. Zachowanie każdego zapamiętanego obiektu funkcyjnego

w momencie wywołania jest w pełni określone jedynie przez **typ** wyrażenia użytego do jego wykonania. Typ każdego wyrażenia jest **obliczany przez metaprogram** związany z poszczególnymi operatorami.

Podobnie jak YACC biblioteka Spirit jest metaprogramem generującym analizatory składniowe dla podanej gramatyki. Jednak w odróżnieniu od YACC Spirit definiuje swój język specjalistyczny jako podzbiór samego języka C++. Jeśli jeszcze nie dostrzegasz tego powiązania, nic nie szkodzi. Po przeczytaniu niniejszej książki na pewno wszystko stanie się oczywiste.

1.5. Dlaczego metaprogramowanie?

Jakie są zalety metaprogramowania? Z pewnością istnieją prostsze sposoby rozwiązania przedstawionych tutaj problemów. Przyjrzyjmy się dwóm innym podejściom i przeanalizujmy ich zastosowanie pod kątem interpretacji wartości binarnych i konstrukcji analizatorów składniowych.

1.5.1. Pierwsza alternatywa — obliczenia w trakcie działania programu

Chyba najprostsze podejście związane jest z wykonywaniem obliczeń w trakcie działania programu zamiast na etapie kompilacji. Można w tym celu wykorzystać jedną z implementacji **funkcji** `binary()` z poprzedniej części rozdziału. System analizy składniowej mógłby dokonywać interpretacji gramatyki w trakcie działania programu, na przykład przy pierwszym zadaniu analizy składniowej.

Oczywistym powodem wykorzystania metaprogramowania jest możliwość wykonania jak największej liczby zadań jeszcze przed uruchomieniem programu wynikowego — w ten sposób uzyskuje się **szybsze programy**. W trakcie kompilacji gramatyki YACC dokonuje analizy i optymalizacji tabeli generacji, co w przypadku wykonywania tych zadań w trakcie działania programu zmniejszyłoby ogólną wydajność. Podobnie, ponieważ `binary` wykonuje swoje zadanie w trakcie kompilacji, `::value` jest dostępna jako stała w trakcie kompilacji, a tym samym kompilator może ją bezpośrednio zamienić na kod obiektu, zaoszczędzając wyszukania w pamięci, gdy zostanie użyta.

Bardziej subtelny, ale i ważniejszy argument przemawiający za metaprogramowaniem wynika z faktu, iż wynik obliczeń może **wejść w znacznie głębszą interakcję z docelowym językiem**. Na przykład rozmiar tablicy można poprawnie określić na etapie kompilacji tylko jako stałą, na przykład `binary<N>::value` — nie można tego zrobić za pomocą wartości zwracanej przez funkcję. Akcje zawarte w nawiasach klamrowych gramatyki YACC mogą zawierać dowolny kod C lub C++, który zostanie wykonany jako część analizatora składniowego. Takie rozwiązanie jest możliwe tylko dlatego, że akcje są przetwarzane w trakcie **kompilacji** gramatyki i są przekazywane do docelowego kompilatora C++.

1.5.2. Druga alternatywa — analiza użytkownika

Zamiast wykonywać obliczenia w trakcie kompilacji lub działania programu, wykonujemy wszystko ręcznie. Przecież przekształcanie wartości binarnych na ich odpowiedniki szesnastkowe jest powszechnie stosowaną praktyką. Podobnie ma się sprawa z krokami przekształceń wykonywanymi przez program YACC lub bibliotekę Boost Spirit.

Jeśli alternatywą jest napisanie metaprogramu, który zostanie wykorzystany tylko raz, można argumentować, iż analiza użytkownika jest wygodniejsza — łatwiej skonwertować ręcznie wartość binarną na szesnastkową niż pisać wykonujący to samo zadanie metaprogram. Jeżeli jednak wystąpień takiej sytuacji będzie kilka, **wygoda** bardzo szybko przesuwa się w stronę przeciwną. Co więcej, po napisaniu metaprogramu można go rozpowszechnić, aby inni programiści również mogli wygodniej pisać programy.

Niezależnie od liczby użyć metaprogramu zapewnia on użytkownikowi **większą siłę wyrazu** kodu, gdyż można określić wynik w formie najlepiej tłumaczącej jego działanie. W kontekście, gdzie wartości poszczególnych bitów mają duże znaczenie, znacznie większy sens ma napisanie `binary<101010>::value` niż `42` lub tradycyjne `0x2a`. Podobnie kod w języku C dla ręcznie napisanego analizatora kodu na ogół zasłania logiczne związki między poszczególnymi elementami gramatyki.

Ponieważ ludzie są ułomni, a logikę metaprogramu wystarczy napisać tylko raz, wynikowy program ma większą szansę być **poprawnym i łatwiej modyfikowalnym**. Ręczna zamiana wartości binarnych zwiększa prawdopodobieństwo popełnienia błędu, bo jest bardzo nudna. Dla odróżnienia ręczne tworzenie tabel analizy składniowej jest tak niewdzięcznym zadaniem, iż unikanie w tej kwestii błędów jest poważnym argumentem za korzystaniem z generatorów takich jak YACC.

1.5.3. Dlaczego metaprogramowanie w C++?

W języku takim jak C++, gdzie język specjalistyczny stanowi po prostu podzbiór języka używanego w pozostałej części programu, metaprogramowanie jest szczególnie wygodne i użyteczne.

- Użytkownik może od razu korzystać z języka specjalistycznego bez uczenia się nowej składni lub przerywania układu pozostałej części programu.
- Powiązanie metaprogramów z pozostałym kodem, w szczególności innymi metaprogramami, staje się bardziej płynne.
- Nie jest wymagany żaden dodatkowy krok w trakcie kompilacji (jak ma to miejsce w przypadku YACC).

W tradycyjnym programowaniu powszechne są starania o odnalezienie złotego środka między wyrazistością, poprawnością a wydajnością kodu. Metaprogramowanie ułatwia przerwanie tej klasycznej szarady i przeniesienie obliczeń wymaganych do uzyskania wyrazistości i poprawności do etapu kompilacji.

1.6. Kiedy stosować metaprogramowanie?

Przedstawiliśmy kilka odpowiedzi na pytanie **dlaczego** metaprogramowanie i kilka przykładów wyjaśniających, **jak** działa metaprogramowanie. Warto jeszcze wyjaśnić, **kiedy** warto je stosować. W zasadzie już przedstawiliśmy większość najważniejszych kryteriów stosowania metaprogramowania szablonami. Jeśli spełnione są dowolne trzy z poniższych warunków, warto zastanowić się nad rozwiązaniem wykorzystującym metaprogramowanie.

- Chcesz, aby kod został wyrażony w kategoriach dziedziny problemowej. Na przykład chcesz, by analizator składni przypominał gramatykę formalną, a nie zbiór tabel i podprocedur, lub działania na tablicach przypominały notację znaną z obiektów macierzy lub wektorów, zamiast stanowić zbiór pętli.
- Chcesz uniknąć pisania dużej ilości podobnego kodu implementacyjnego.
- Musisz wybrać implementację komponentu na podstawie właściwości jego parametrów typu.
- Chcesz skorzystać z zalet programowania generycznego w C++, na przykład statycznego sprawdzania typów i dostosowywania zachowań bez utraty wydajności.
- Chcesz to wszystko wykonać w języku C++ bez uciekania się do zewnętrznych narzędzi i generatorów kodu źródłowego.

1.7. Dlaczego biblioteka metaprogramowania?

Zamiast zajmować się metaprogramowaniem od podstaw, będziemy korzystali z wysokopoziomowej pomocy biblioteki MPL (*Boost Metaprogramming Library*). Nawet jeśli ktoś nie wybrał tej książki w celu zapoznania się ze szczegółami MPL, sądzimy, że czas poświęcony na jej naukę nie pójdzie na marne, gdyż łatwo ją wykorzystać w codziennej pracy.

1. **Jakość**. Większość programistów używających komponentów metaprogramowania szablonami traktuje je — całkiem słusznie — jako szczegóły implementacyjne wprowadzane w celu ułatwienia większych zadań. Dla odróżnienia, autorzy MPL skupili się na wykonaniu użytecznych narzędzi wysokiej jakości. Ogólnie komponenty z biblioteki są bardziej elastyczne i lepiej zaimplementowane niż te, które wykonałoby się samemu w celu przeprowadzenia innych zadań. Co więcej, przyszłe wydania z pewnością będą ulepszane i optymalizowane.

2. **Ponowne użycie**. Wszystkie biblioteki hermetyzują kod jako komponent wielokrotnego użytku. Co więcej, dobrze zaprojektowana biblioteka ogólna zapewnia szkielet pojęciowy mentalnego modelu rozwiązywania pewnych problemów. Podobnie jak standardowa biblioteka języka C++ dostarcza iteratory i protokół obiektów funkcyjnych, biblioteka MPL zapewnia iteratory typów i protokół metafunkcyjny. Dobrze wykonany szkielet pozwala programiście skupić się na decyzjach projektowych i szybkim wykonaniu właściwego zadania.
3. **Przenośność**. Dobra biblioteka potrafi gładko przejść przez niuanse różnic w platformach sprzętowych. Choć w teorii żaden z metaprogramów języka C++ nie powinien mieć problemów z przenośnością, rzeczywistość jest inna nawet po 6 latach od standaryzacji. Nie powinno to jednak dziwić — szablony C++ to najbardziej złożony aspekt tego języka programowania, który jednocześnie stanowi o sile metaprogramowania C++.
4. **Zabawa**. Wielokrotne pisanie tego samego kodu jest wyjątkowo nudne. Szybkie połączenie komponentów wysokiego poziomu w czytelne, eleganckie rozwiązanie to czysta zabawa! Biblioteka MPL redukuje nudę, eliminując potrzebę powtarzania najbardziej typowych wzorców metaprogramowania. Przede wszystkim elegancko unika się specjalizacji przerywających i jawnych rekurencji.
5. **Wydajność wytwarzania**. Poza satysfakcją personelu zdrowie projektów zależy również od przyjemności czerpanej z programowania. Gdy programista przestaje mieć frajdę z programowania, staje się zmęczony i powolny — błędny kod jest bardziej kosztowny od kodu pisanego dobrze, ale powoli.

Jak łatwo się przekonać, biblioteka MPL jest pisana zgodnie z tymi samymi zasadami, które przyświecają tworzeniu innych bibliotek. Wydaje nam się, że jej pojawienie się jest zwiastunem, iż metaprogramowanie szablonami jest gotowe opuścić pracownie badawcze i zacząć być stosowane przez programistów w codziennej pracy.

Chcielibyśmy zwrócić szczególną uwagę na czwarty z przedstawionych punktów. Biblioteka MPL nie tylko ułatwia korzystanie z metaprogramowania, ale również czyni je czystą przyjemnością. Mamy nadzieję, że innym osobom uczenie się jej przyniesie tyle radości, co nam przyniosło jej tworzenie i wykorzystywanie.