Wisnu Anggoro

C++

Struktury danych i algorytmy

Helion

Packt›

Tytuł oryginału: C++ Data Structures and Algorithms

Tłumaczenie: Maksymilian Gutowski

ISBN: 978-83-283-5185-1

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cppstr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Struktury danych i algorytmy w C++

W pierwszym rozdziale położymy solidny fundament, który pozwoli Ci z łatwością przejść przez kolejne rozdziały. Oto tematy, które omówimy w tym rozdziale:

- tworzenie, kompilowanie i uruchamianie prostego programu C++;
- konstruowanie abstrakcyjnego typu danych w celu utworzenia typu danych zdefiniowanego przez użytkownika;
- wykorzystanie kodu za pomocą szablonów C++ i **Standard Template Library** (**STL**);
- analizowanie złożoności algorytmów w celu zmierzenia wydajności kodu.

Wymagania techniczne

Do przeczytania tego rozdziału i posłużenia się przedstawionym w nim kodem źródłowym konieczne są następujące zasoby:

- komputer stacjonarny lub laptop z systemem Windows, Linux albo macOS;
- GNU GCC v5.4.0 lub nowsza;
- Code::Block IDE v17.12 (dla systemów Windows i Linux) lub Code::Block IDE v13.12 (dla macOS);
- pliki z kodem, które znajdziesz w archiwum pobranym ze strony *ftp://ftp.helion.pl/przyklady/cppstr.zip*.

Podstawy C++

Przed przejściem do omówienia struktur danych i algorytmów w C++ musimy poświęcić nieco miejsca podstawom samego języka C++. W tym podrozdziale napiszemy prosty program, skompilujemy go, a następnie uruchomimy. Omówimy także podstawowe oraz zaawansowane typy danych i kontrolę przepływu, zanim zajmiemy się przepływem sterowania.

Pierwszy kod w C++

W C++ wykonywanie kodu rozpoczyna się od funkcji `main()`. Ta funkcja jest zbiorem instrukcji określających konkretne zadania do wykonania. W związku z tym program w C++ musi składać się z przynajmniej jednej funkcji o nazwie `main()`. Poniższy kod jest najprostszym programem C++, jaki można z powodzeniem skompilować i uruchomić:

```
int main()
{
   return 0;
}
```

Załóżmy, że powyższy kod został zapisany jako plik *simple.cpp*. Możemy skompilować kod przy użyciu kompilatora g++, wprowadzając poniższą komendę kompilacji w konsoli w katalogu, w którym znajduje się plik *simple.cpp*:

```
g++ simple.cpp
```

Jeśli nie zobaczymy żadnego komunikatu o błędzie, oznacza to, że plik wyjściowy został automatycznie wygenerowany. Po wprowadzeniu komendy kompilacji w konsoli systemu Windows otrzymamy plik o nazwie *a.exe*, a wprowadzenie jej w powłoce Bash, na przykład w systemach Linux i macOS, spowoduje utworzenie pliku o nazwie *a.out*.

Możemy określić nazwę pliku wejściowego, używając opcji `-o` i podając pożądaną nazwę pliku. Poniższa komenda kompilacji spowodowałaby zatem zwrócenie pliku wyjściowego o nazwie *simple.out*:

```
g++ simple.cpp -o simple.out
```

Po uruchomieniu pliku wyjściowego (poprzez wprowadzenie nazwy `a` i naciśnięcie klawisza *Enter* w konsoli systemu Windows lub wpisanie `./a.out` i naciśnięcie klawisza *Enter* w powłoce Bash) w oknie konsoli nic się nie pojawi. Jest tak, ponieważ nie każemy jeszcze niczego w niej wyświetlać. Aby nasz plik *simple.cpp* nabrał sensu, zrefaktoryzujmy kod tak, by otrzymywał dane wejściowe od użytkownika i je wyświetlał. Kod powinien wyglądać następująco:

```
// in_out.cpp
#include <iostream>

int main ()
{
   int i;
```

```
    std::cout << "Podaj liczbę całkowitą: ";
    std::cin >> i;
    std::cout << "Podana wartość to " << i;
    std::cout << "\n";
return 0;
}
```

Jak widać, w powyższym kodzie dodaliśmy kilka linijek, tak aby program wyświetlał informacje w konsoli i umożliwiał użytkownikowi wprowadzenie danych wejściowych. Program wstępnie wyświetla tekst i prosi użytkownika o podanie liczby całkowitej. Po wpisaniu przez użytkownika wybranej przez niego liczby i naciśnięciu *Enter* program wyświetla podaną liczbę. Zdefiniowaliśmy też nową zmienną `i` należącą do typu danych `int`. Służy ona do przechowywania danych w formacie liczby całkowitej (temat zmiennych i typów danych zostanie poruszony w jednym z kolejnych punktów).

Załóżmy, że zapisaliśmy powyższy kod w pliku *in_out.cpp*; możemy go skompilować, używając następującego polecenia:

```
g++ in_out.cpp
```

Jeśli uruchomimy następnie program, otrzymamy w konsoli poniższy wynik (na potrzeby przykładu wprowadziłem liczbę 3):

```
Podaj liczbę całkowitą: 3
Podana wartość to 3
```

Wiesz już, że do wyświetlenia tekstu w konsoli używamy instrukcji `std::cout`, a do wprowadzania danych wejściowych do programu instrukcji `std::cin`. Na początku pliku *in_out.cpp* znajduje się instrukcja `#include <iostream>`, która wskazuje kompilatorowi, gdzie szukać implementacji poleceń `std::cout` i `std::cin`, jako że są one określone w pliku nagłówkowym `iostream`.

Na samym początku pliku znajduje się jeszcze wiersz zaczynający się od dwóch ukośników (//). Wskazują one, że dany wiersz nie jest traktowany jako kod, ma on więc zostać zignorowany przez kompilator. Wiersz ten służy do zapisania komentarza lub oznaczenia określonego działania w kodzie, tak aby ułatwić innym programistom jego zrozumienie.

Usprawnianie pracy nad kodem przy użyciu IDE

Udało nam się dotąd napisać kod C++, skompilować i uruchomić go. Ciągłe kompilowanie kodu i wykonywanie go w linii poleceń byłoby jednak nudne. Aby ułatwić sobie pracę z kodem, skorzystamy ze **zintegrowanego środowiska programistycznego** (ang. *integrated development environment* — **IDE**), w którym będziemy mogli kompilować i uruchamiać kod za jednym kliknięciem. Możesz skorzystać z dowolnego dostępnego na rynku IDE C++, zarówno płatnego,

jak i darmowego. Sam preferuję IDE Code::Blocks, ponieważ jest darmowe, otwarte i wieloplatformowe, wobec czego działa w systemach Windows, Linux i macOS. Więcej informacji na temat tego IDE, w tym instrukcje dotyczące pobierania go, instalacji oraz użytkowania, znajdziesz na jego oficjalnej stronie internetowej pod adresem *http://www.codeblocks.org/*.

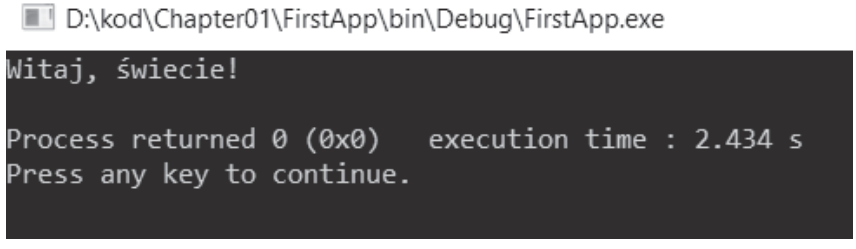
Moglibyśmy zautomatyzować proces kompilacji przy użyciu toolchaina takiego jak Make lub CMake, ale wymagałoby to dodatkowych wyjaśnień. Ponieważ tematem tej książki są struktury danych i algorytmy, podjęcie tego wątku zwiększyłoby długość tekstu — z tego względu nie omówimy tego zagadnienia szerzej. Na nasze potrzeby IDE oferuje najlepszą możliwość automatyzacji procesu kompilacyjnego, ponieważ używa do tego toolchaina.

Po zainstalowaniu IDE Code::Blocks możemy utworzyć nowy projekt, klikając menu *File/New/*
↪Project (plik/nowy/projekt). Na ekranie pojawi się nowe okno, w którym będziemy mogli wybrać pożądany typ projektu. Dla większości przykładów w tej książce będziemy używać typu *Console Application* (aplikacja konsolowa). Naciśnij przycisk *Go* (dalej), aby kontynuować.

W kolejnym oknie wskazujemy język, czyli C++, a następnie określamy nazwę projektu i lokalizację docelową (swojemu projektowi nadałem nazwę *FirstApp*). Po zakończeniu pracy z kreatorem otrzymujemy nowy projekt z plikiem *main.cpp* zawierającym następujący kod:

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Witaj, świecie!" << endl;
   return 0;
}
```

Możemy teraz skompilować i uruchomić powyższy kod, klikając opcję *Build and run* (skompiluj i uruchom) w menu *Build* (kompilacja). Na ekranie pojawi się następujące okno konsoli:



Na powyższym rysunku widzimy, że konsola wykorzystuje `namespace std` z wiersza po wierszu `#include <iostream>`. Wiersz ten wskazuje kompilatorowi, że kod używa `namespace` (przestrzeni nazw) o nazwie `std`. Dzięki temu nie musimy wskazywać `std::` przy każdym wywołaniu instrukcji `cin` i `cout`. Kod jest prostszy niż wcześniej.

Definiowanie zmiennych przy użyciu podstawowych typów danych

W poprzednich przykładowych kodach skorzystaliśmy ze zmiennej (służącej do przechowywania elementu danych), aby móc przetwarzać zawarte w niej dane w ramach różnych działań. W C++ zmiennym przypisujemy konkretne typy danych; zmienna może przechowywać wyłącznie dane tego typu, który został jej uprzednio przypisany. Oto lista fundamentalnych typów danych w C++; niektórych z nich użyliśmy już w poprzednim przykładzie:

- Typ logiczny (`bool`), który służy do przechowywania wyłącznie dwóch elementów danych warunkowych — `true` i `false`.
- Typ znakowy (`char`, `wchar_t`, `char16_t` i `char32_t`), który służy do przechowywania pojedynczych znaków ASCII.
- Typ zmiennoprzecinkowy (`float`, `double` i `long double`), który służy do przechowywania ułamków dziesiętnych.
- Typ całkowity (`short`, `int`, `long` i `long long`), który służy do przechowywania liczb całkowitych.
- Typ pusty (`void`), który jest w zasadzie słowem kluczowym, umieszczanym tam, gdzie w innym przypadku znalazłby się typ danych, aby wskazać *brak danych*.

Zmienne można tworzyć na dwa sposoby: w drodze definiowania lub inicjalizacji. Zdefiniowanie zmiennej powoduje utworzenie zmiennej bez określenia jej początkowej wartości. Inicjalizacja zmiennej powoduje utworzenie zmiennej i przypisanie jej początkowej wartości. Oto przykłady definiowania zmiennych:

```
int iVar;
char32_t cVar;
long long llVar;
bool boVar;
```

Oto przykłady inicjalizacji zmiennych:

```
int iVar = 100;
char32_t cVar = 'a';
long long llVar = 9223372036854775805;
bool boVar = true;
```

Powyższy kod ukazuje inicjalizowanie zmiennych poprzez użycie techniki **inicjalizacji kopią**. Polega ona na przypisaniu wartości zmiennej za pomocą znaku równości (=). Innym sposobem jest **inicjalizacja bezpośrednia**, która polega na użyciu nawiasów do przypisania wartości zmiennej. Oto przykłady inicjalizacji zmiennych z wykorzystaniem tej drugiej techniki:

```
int iVar(100);
char32_t cVar('a');
long long llVar(9223372036854775805);
bool boVar(true);
```

Poza technikami inicjalizacji kopią i inicjalizacji bezpośredniej możemy posłużyć się inicjalizacją jednolitą, wykorzystującą nawiasy klamrowe. Poniższy kod ukazuje technikę **inicjalizacji nawiasowej**:

```
int iVar{100};
char32_t cVar{'a'};
long long llVar{9223372036854775805};
bool boVar{true};
```

Nie możemy zdefiniować zmiennej przy użyciu typu danych `void`, na przykład `void vVar`, ponieważ przy definiowaniu zmiennej musimy wybrać taki typ danych, abyśmy mogli przechować dane w zmiennej. Kiedy definiujemy zmienną jako `void`, oznacza to, że niczego nie zamierzamy w niej przechowywać.

Sterowanie przepływem kodu

Jak już wspomniałem, wykonywanie programu C++ zaczyna się od funkcji `main()`, w której instrukcje wykonywane są po kolei. Można jednak zmienić tę kolejność, używając instrukcji sterowania przepływem. W C++ istnieją różne instrukcje sterowania, ale omówimy jedynie kilka wybranych, które są najczęściej wykorzystywane w projektowaniu algorytmów.

Instrukcja warunkowa

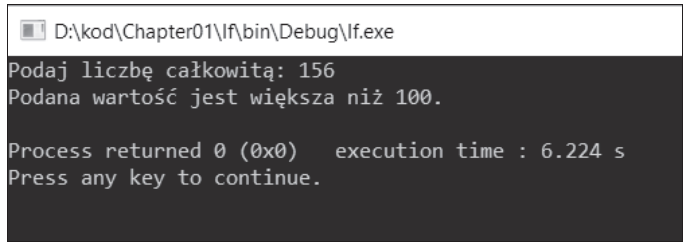
Instrukcja warunkowa jest jednym z elementów, które mogą zmienić przepływ programu. Kiedy taka instrukcja zostaje użyta, wykonywane są jedynie te wiersze, które przypisane są do warunku o wartości `true`. Do wprowadzenia takiej instrukcji używamy słów kluczowych `if` i `else`.

Przekształćmy kod pliku *in_out.cpp* tak, aby wykorzystywał instrukcję warunkową. Program będzie musiał jedynie określić, czy wprowadzona liczba jest większa od `100`. Kod ten wygląda następująco:

```
// If.cbp
#include <iostream>

using namespace std;

int main ()
{
   int i;
   cout << " Podaj liczbę całkowitą: ";
   cin >> i;
   cout << "Podana wartość jest ";

   if(i > 100)
      cout << "większa niż 100.";
   else
      cout << "równa lub mniejsza niż 100.";

   cout << endl;
   return 0;
}
```

Jak widzimy, para słów kluczowych `if` i `else` określa, czy wprowadzona liczba jest większa od 100. Z powyższego kodu wykonana zostanie tylko jedna z instrukcji zawartych w instrukcji warunkowej — albo przypisana słowu kluczowemu `if`, albo słowu kluczowemu `else`.

Po skompilowaniu i uruchomieniu powyższego kodu otrzymamy następujący wynik w oknie konsoli:



Z powyższego rysunku widzimy, że wiersz `std::cout << "równa lub mniejsza niż 100.";` nie został wykonany, ponieważ wprowadzono liczbę większą niż 100.

Ponadto instrukcja `if…else` może składać się z więcej niż dwóch instrukcji warunkowych. Możemy zrefaktoryzować powyższy kod, tak aby znalazło się w nim więcej instrukcji warunkowych, w sposób następujący:

```
// If_ElseIf.cbp
#include <iostream>

using namespace std;

int main ()
{
   int i;
   cout << " Podaj liczbę całkowitą: ";
   cin >> i;
   cout << "Podana wartość jest ";

   if(i > 100)
      cout << "większa niż 100.";
   else if(i < 100)
      cout << "mniejsza niż 100.";
   else
      cout << "równa 100.";

   cout << endl;
   return 0;
}
```

Innym słowem kluczowym wykorzystywanym w instrukcjach warunkowych jest `switch`. Zanim je omówimy, napiszmy prosty kalkulator pozwalający na dodawanie, odejmowanie, mnożenie i dzielenie. Najpierw użyjemy słowa kluczowego `if…else`. Kod powinien wyglądać następująco:

```
// If_ElseIf_2.cbp
#include <iostream>
```

```
using namespace std;

int main ()
{
   int i, a, b;

   cout << "Działanie: " << endl;
   cout << "1. Dodawanie" << endl;
   cout << "2. Odejmowanie" << endl;
   cout << "3. Mnożenie" << endl;
   cout << "4. Dzielenie" << endl;
   cout << "Podaj numer działania: ";
   cin >> i;

   cout << "Podaj pierwszą liczbę: ";
   cin >> a;
   cout << "Podaj drugą liczbę: ";
   cin >> b;

   cout << "Wynik ";

   if(i == 1)
      cout << a << " + " << b << " to " << a + b;
   else if(i == 2)
      cout << a << " - " << b << " to " << a - b;
   else if(i == 3)
      cout << a << " * " << b << " to " << a * b;
   else if(i == 4)
      cout << a << " / " << b << " to " << a / b;

   cout << endl;
   return 0;
}
```

Jak widać w powyższym kodzie, mamy do wyboru cztery opcje, do których obsługi używamy instrukcji warunkowej `if…else`. Po wykonaniu kodu otrzymamy następujący wynik:

```
D:\kod\Chapter01\If_ElseIf_2\bin\Debug\If_ElseIf_2.exe
Działanie:
1. Dodawanie
2. Odejmowanie
3. Mnożenie
4. Dzielenie
Podaj numer działania: 2
Podaj pierwszą liczbę: 12
Podaj drugą liczbę: 4
Wynik 12 - 4 to 8

Process returned 0 (0x0)   execution time : 20.627 s
Press any key to continue.
```

Możemy też skorzystać ze słowa kluczowego switch. Po refaktoryzacji kod powinien wyglądać następująco:

```
// Switch.cbp
#include <iostream>

using namespace std;

int main ()
{
   int i, a, b;

   cout << "Działanie: " << endl;
   cout << "1. Dodawanie" << endl;
   cout << "2. Odejmowanie" << endl;
   cout << "3. Mnożenie" << endl;
   cout << "4. Dzielenie" << endl;
   cout << "Podaj numer działania: ";
   cin >> i;

   cout << "Podaj pierwszą liczbę: ";
   cin >> a;
   cout << "Podaj drugą liczbę: ";
   cin >> b;

   cout << "Wynik ";

   switch(i)
   {
   case 1:
      cout << a << " + " << b << " to " << a + b;
      break;
   case 2:
      cout << a << " - " << b << " to " << a - b;
      break;
   case 3:
      cout << a << " * " << b << " to " << a * b;
      break;
   case 4:
      cout << a << " / " << b << " to " << a / b;
      break;
   }

   cout << endl;
   return 0;
}
```

Po uruchomieniu powyższego kodu otrzymamy taki sam wynik jak w przypadku *If_ElseIf_2.cbp*.

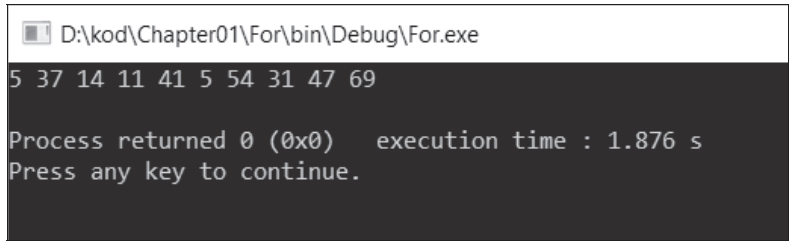
Pętle

W C++ istnieje kilka instrukcji pętli: `for`, `while` i `do…while`. Pętli `for` zazwyczaj używa się wtedy, kiedy wiadomo, ile iteracji ma nastąpić, podczas gdy `while` i `do…while` powtarzają instrukcje aż do momentu, w którym pożądany warunek zostanie spełniony.

Załóżmy, że chcemy wygenerować dziesięć losowych liczb między `0` a `100`; wykorzystanie pętli `for` jest w tym przypadku najlepszym rozwiązaniem, ponieważ wiemy dokładnie, ile liczb chcemy uzyskać. W tym celu możemy napisać poniższy kod:

```
// For.cbp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int GenerateRandomNumber(int min, int max)
{
   // Używa zmiennej statycznej ze względu na wydajność,
   // tak aby obliczyć tę wartość tylko jednokrotnie.
   static const double fraction =
      1.0 / (static_cast<double>(RAND_MAX) + 1.0);

   // Równo rozkłada losowe liczby w wybranym zakresie.
   return min + static_cast<int>(
      (max - min + 1) * (rand() * fraction));
}

int main()
{
   // Określa wstępny seed (ziarno) na podstawie zegara systemowego.
   srand(static_cast<unsigned int>(time(0)));

   // Zapętla dziesięciokrotnie.
   for (int i=0; i < 10; ++i)
   {
      cout << GenerateRandomNumber(0, 100) << " ";
   }
   cout << "\n";

   return 0;
}
```

W powyższym kodzie tworzymy inną funkcję poza `main()`, czyli `GenerateRandomNumber()`. Kod wywołuje funkcję dziesięciokrotnie przy użyciu pętli `for`, tak jak widać w kodzie powyżej. Powinniśmy uzyskać następujący wynik:



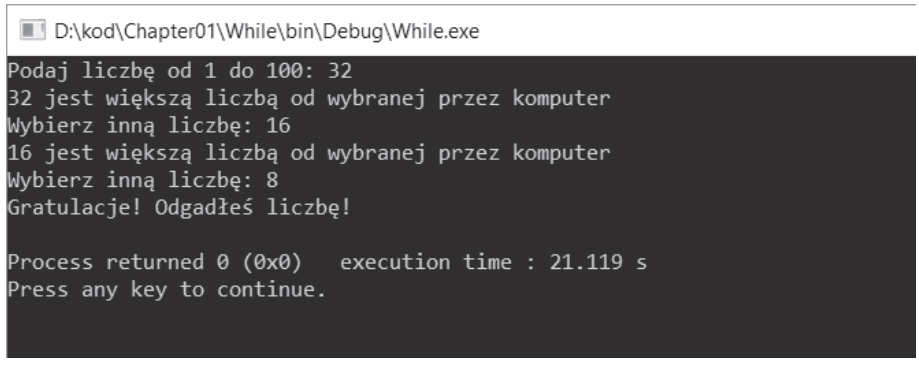
Wróćmy do pozostałych instrukcji pętli, o których wspomniałem, czyli `while` i `do…while`. Ich działanie jest dość podobne. Różnica polega na tym, że kiedy używamy pętli `while`, istnieje możliwość, że instrukcja zawarta w pętli w ogóle nie zostanie wykonana, podczas gdy instrukcja z pętli `do…while` musi zostać wykonana przynajmniej raz.

Stwórzmy teraz prostą grę opartą na pętlach. Komputer będzie generował liczbę od 1 do 100, a zadaniem użytkownika będzie jej odgadnięcie. Program będzie podawał użytkownikowi wskazówki tuż po wprowadzeniu zgadywanej liczby, informując go o tym, czy podana liczba jest większa, czy mniejsza od wylosowanej przez komputer. Gra będzie się kończyć, kiedy liczba podana przez użytkownika będzie równa tej, którą wygenerował komputer. Kod wygląda następująco:

```
// While.cbp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int GenerateRandomNumber(int min, int max)
{
   // Używa zmiennej statycznej ze względu na wydajność,
   // tak aby obliczyć tę wartość tylko jednokrotnie.
   static const double fraction =
      1.0 / (static_cast<double>(RAND_MAX) + 1.0);

   // Równo rozkłada losowe liczby w wybranym zakresie.
   return min + static_cast<int>(
      (max - min + 1) * (rand() * fraction));
}

int main()
{
   // Określa wstępny seed (ziarno) na podstawie zegara systemowego.
   srand(static_cast<unsigned int>(time(0)));

   // Komputer generuje losową liczbę od 1 do 100.
   int computerNumber = GenerateRandomNumber(1, 100);

   // Użytkownik zgaduje liczbę i ją wprowadza.
   int userNumber;
   cout << "Podaj liczbę od 1 do 100: ";
   cin >> userNumber;
```

```
    // Uruchamia pętlę WHILE.
    while(userNumber != computerNumber)
    {
       cout << userNumber << " jest ";
       if(userNumber > computerNumber)
          cout << "większą";
       else
          cout << "mniejszą";
       cout << " liczbą od wybranej przez komputer" << endl;
       cout << "Wybierz inną liczbę: ";
       cin >> userNumber;
    }

    cout << "Gratulacje! Odgadłeś liczbę!" << endl;
    return 0;
}
```

W powyższym kodzie widzimy, że dwie zmienne — `computerNumber` i `userNumber` — obsługują porównywane liczby. Istnieje pewne prawdopodobieństwo, że wartość `computerNumber` od razu okaże się równa wartości `userNumber`. Jeśli tak będzie, instrukcja zawarta w pętli `while` w ogóle nie zostanie wykonana. Przepływ programu widać na poniższym zrzucie z konsoli:

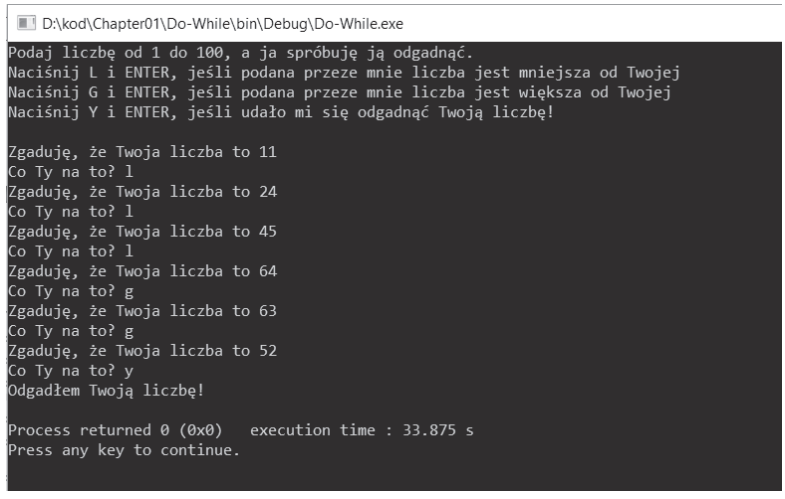


W powyższym kodzie udało nam się zaimplementować pętlę `while`. Choć przypomina ona pętlę `do…while`, nie możemy zrefaktoryzować tego kodu, zastępując `while` pętlą `do…while`. Możemy jednak napisać inną grę, która taką pętlę wykorzystuje. W tym przypadku to użytkownik będzie wybierał liczbę, a zadaniem komputera będzie odgadnięcie jej. Kod powinien wyglądać następująco:

```
// Do-While.cbp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int GenerateRandomNumber(int min, int max)
{
    // Używa zmiennej statycznej ze względu na wydajność,
    // tak aby obliczyć tę wartość tylko jednokrotnie.
```

```
    static const double fraction =
       1.0 / (static_cast<double>(RAND_MAX) + 1.0);

    // Równo rozkłada losowe liczby w wybranym zakresie.
    return min + static_cast<int>(
       (max - min + 1) * (rand() * fraction));
}

int main()
{
    // Określa wstępny seed (ziarno) na podstawie zegara systemowego.
    srand(static_cast<unsigned int>(time(0)));

    char userChar;

    int iMin = 1;
    int iMax = 100;
    int iGuess;

    // Interfejs gry.
    cout << "Podaj liczbę od 1 do 100, ";
    cout << "a ja spróbuję ją odgadnąć." << endl;
    cout << "Naciśnij L i ENTER, jeśli podana przeze mnie liczba jest mniejsza od
    ↪Twojej";
    cout << endl;
    cout << "Naciśnij G i ENTER, jeśli podana przeze mnie liczba jest większa od
    ↪Twojej ";
    cout << endl;
    cout << "Naciśnij Y i ENTER, jeśli udało mi się odgadnąć Twoją liczbę!";
    cout << endl << endl;

    // Uruchamia pętlę DO-WHILE.
    do
    {
       iGuess = GenerateRandomNumber(iMin, iMax);
       cout << "Zgaduję, że Twoja liczba to " << iGuess << endl;
       cout << "Co Ty na to? ";
       cin >> userChar;
       if(userChar == 'L' || userChar == 'l')
          iMin = iGuess + 1;
       else if(userChar == 'G' || userChar == 'g')
          iMax = iGuess - 1;
    }
    while(userChar != 'Y' && userChar != 'y');

    cout << "Odgadłem Twoją liczbę!" << endl;
    return 0;
}
```

Jak widzimy powyżej, program musi odgadnąć liczbę wybraną przez użytkownika przynajmniej raz, przy czym może mu się udać trafić za pierwszym podejściem. Możemy wobec tego użyć tutaj pętli `do…while`. Po skompilowaniu i uruchomieniu kodu otrzymamy rezultat przypominający to, co widać na poniższym rysunku:



Na powyższym zrzucie widać, że wybrałem liczbę 52. Program podał liczbę 11. Ponieważ była ona mniejsza od mojej, program podał kolejną, czyli 24. Podał on następnie kolejną liczbę w oparciu o wskazówkę, aż wreszcie wskazał właściwą liczbę. Program zakończy wykonywanie pętli `do…while`, jeśli użytkownik naciśnie klawisz *y*, co widzimy w kodzie.

Wykorzystanie zmiennych za pośrednictwem zaawansowanych typów danych

W poprzednim punkcie omówiliśmy podstawowe typy danych. Służą one do definiowania lub inicjalizowania zmiennych, aby sprawić, by zmienne mogły przechowywać określone typy danych. Do definiowania zmiennych można jednak używać także innych typów danych: `enum` (wyliczeniowego) i `struct` (strukturalnego).

Typ wyliczeniowy może mieć różne wartości, które są zdefiniowane jako stałe zwane **enumeratorami**. Służy on do tworzenia kolekcji stałych. Przyjmijmy, że chcemy napisać grę karcianą w C++. Jak wiadomo, talia kart do gry składa się z 52 kart, dzielących się na cztery kolory (trefl, karo, kier i pik), z czego każdy kolor ma 13 kart. Taką talię możemy przedstawić następująco:

```
enum CardSuits
{
   Club,
   Diamond,
   Heart,
   Spade
};

enum CardElements
{
   Ace,
   Two,
   Three,
```

```
    Four,
    Five,
    Six,
    Seven,
    Eight,
    Nine,
    Ten,
    Jack,
    Queen,
    King
};
```

Aby skorzystać z powyższych typów danych `enum` (`CardSuits` i `CardElements`), możemy dokonać inicjalizacji zmiennych w sposób następujący:

```
CardSuits suit = Club;
CardElements element = Ace;
```

W rzeczywistości zmienne wyliczeniowe zawsze zawierają stałe liczby całkowite. Łańcuch znaków podany dla elementu `enum` jest jedynie nazwą stałej. Pierwszy element ma wartość `0`, z tym że sami wprost definiujemy inną wartość. Kolejne elementy mają przyrostowe wartości liczbowe względem pierwszego. Co za tym idzie, w powyższym przykładzie `CardSuits` element `Club` ma wartość `0`, a `Diamond`, `Heart` i `Spade` mają odpowiednio wartości `1`, `2` i `3`.

Napiszmy teraz program, który będzie generował losową kartę. Możemy wykorzystać funkcję `GenerateRandomNumber()` z poprzedniego kodu. Oto gotowy kod:

```
// Enum.cbp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

enum CardSuits
{
   Club,
   Diamond,
   Heart,
   Spade
};

enum CardElements
{
   Ace,
   Two,
   Three,
   Four,
   Five,
   Six,
   Seven,
   Eight,
   Nine,
   Ten,
```

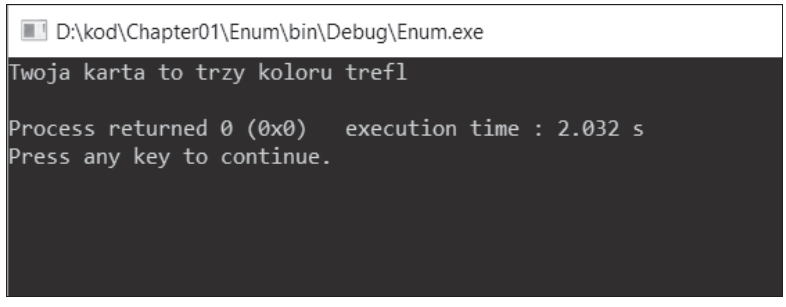
```
   Jack,
   Queen,
   King
};

string GetSuitString(CardSuits suit)
{
   string s;
   switch(suit)
   {
      case Club:
         s = "trefl";
         break;
      case Diamond:
         s = "karo";
         break;
      case Heart:
         s = "kier";
         break;
      case Spade:
         s = "pik";
         break;
   }

   return s;
}

string GetElementString(CardElements element)
{
   string e;

   switch(element)
   {
      case Ace:
         e = "as";
         break;
      case Two:
         e = "dwa";
         break;
      case Three:
         e = "trzy";
         break;
      case Four:
         e = "cztery";
         break;
      case Five:
         e = "pięć";
         break;
      case Six:
         e = "sześć";
         break;
      case Seven:
         e = "siedem";
         break;
```

```
            case Eight:
               e = "osiem";
               break;
            case Nine:
               e = "dziewięć";
               break;
            case Ten:
               e = "dziesięć";
               break;
            case Jack:
               e = "walet";
               break;
            case Queen:
               e = "dama";
               break;
            case King:
               e = "król";
               break;
         }

         return e;
      }

      int GenerateRandomNumber(int min, int max)
      {
         // Używa zmiennej statycznej ze względu na wydajność,
         // tak aby obliczyć tę wartość tylko jednokrotnie.
         static const double fraction =
            1.0 / (static_cast<double>(RAND_MAX) + 1.0);

         // Równo rozkłada losowe liczby w wybranym zakresie.
         return min + static_cast<int>(
            (max - min + 1) * (rand() * fraction));
      }

      int main()
      {
         // Określa wstępny seed (ziarno) na podstawie zegara systemowego.
         srand(static_cast<unsigned int>(time(0)));

         // Generuje losowy kolor i kartę.
         int iSuit = GenerateRandomNumber(0, 3);
         int iElement = GenerateRandomNumber(0, 12);

         CardSuits suit = static_cast<CardSuits>(iSuit);
         CardElements element = static_cast<CardElements>(iElement);

         cout << "Twoja karta to ";
         cout << GetElementString(element);
         cout << " koloru " << GetSuitString(suit) << endl;

         return 0;
      }
```

W powyższym kodzie widać, że możemy wykorzystać dane `enum` przy użyciu wartości liczby całkowitej. Musimy jednak rzutować wartość `int`, tak aby była kompatybilna z `enum`, używając `static_cast<>`, tak jak poniżej:

```
int iSuit = GenerateRandomNumber(0, 3);
int iElement = GenerateRandomNumber(0, 12);
CardSuits suit = static_cast<CardSuits>(iSuit);
CardElements element = static_cast<CardElements>(iElement);
```

Po skompilowaniu i uruchomieniu kodu otrzymamy następujący wynik:



Inny zaawansowany typ danych w C++ to `struct`. Jest to zagregowany typ danych, który grupuje wiele indywidualnych zmiennych. Zmienne `suit` i `element` z powyższego kodu można zgrupować następująco:

```
struct Cards
{
   CardSuits suit;
   CardElements element;
};
```

Gdybyśmy dodali powyższą zmienną `struct` do kodu *Enum.cbp*, musielibyśmy tak zrefaktoryzować funkcję `main()`:

```
int main()
{
   // Określa wstępny seed (ziarno) na podstawie zegara systemowego.
   srand(static_cast<unsigned int>(time(0)));

   Cards card;
   card.suit = static_cast<CardSuits>(
      GenerateRandomNumber(0, 3));
   card.element = static_cast<CardElements>(
      GenerateRandomNumber(0, 12));

   cout << "Twoja karta to ";
   cout << GetElementString(element);
   cout << " koloru " << GetSuitString(suit) << endl;

   return 0;
}
```

Po uruchomieniu powyższego kodu (który znajdziesz w repozytorium pod nazwą *Struct.cbp*) uzyskalibyśmy taki sam wynik jak w przypadku *Enum.cbp*.

Tworzenie abstrakcyjnych typów danych

Abstrakcyjny typ danych (ang. *abstract data type* — **ADT**) składa się z kolekcji danych i wykonywanych na nich operacji. ADT odnosi się jedynie do listy operacji, które można wykonać, ale nie do implementacji. Sama implementacja jest ukryta i właśnie z tego względu mówi się o **abstrakcyjności**.

Wyobraźmy sobie, że mamy odtwarzacz DVD, który jest dla nas źródłem rozrywki w wolnym czasie. Odtwarzacz ma też pilota. Na pilocie znajdują się różne przyciski, służące do wysuwania płyty, rozpoczynania i wstrzymywania odtwarzania, regulowania poziomu głośności i tak dalej. Podobnie jak w przypadku ADT, nie mamy najmniejszego pojęcia, w jaki sposób odtwarzacz podgłaśnia dźwięk, kiedy naciskamy przycisk podgłaśniania. Wykonujemy jedynie operację podgłaśniania, a odtwarzacz robi wszystko za nas — nie musimy znać implementacji tej operacji.

W odniesieniu do przepływu procesów musimy uwzględnić abstrakcję implementacji ADT, ukrywanie informacji i enkapsulację. Oto objaśnienia tych trzech technik:

- **Abstrakcja** polega na ukrywaniu szczegółów implementacji operacji dostępnych w ADT.
- **Ukrywanie informacji** to ukrywanie danych, na które implementacja ma wpływ.
- **Enkapsulacja** polega na grupowaniu wszystkich podobnych danych i funkcji.

Wykorzystanie klas C++ przy tworzeniu ADT zdefiniowanych przez użytkownika

Klasy są kontenerami zmiennych i operacji (metod), które wpływają na te zmienne. Jak już wspomniałem, skoro ADT wykorzystują techniki enkapsulacji do grupowania wszystkich podobnych do siebie danych i funkcji, do grupowania można także użyć klas. Dane i metody klas znajdują się w trzech sekcjach o różnej kontroli dostępu:

- **Publiczna.** Dowolny użytkownik klasy ma dostęp do danych i metod.
- **Chroniona.** Dostęp do danych i metod mają jedynie metody klasy, klasy pochodne i zaprzyjaźnione.
- **Prywatna.** Dostęp do danych i metod mają jedynie metody klasy i klasy zaprzyjaźnione.

Powróćmy teraz do definicji abstrakcji i ukrywania informacji. Abstrakcję możemy zaimplementować przy użyciu słów kluczowych `protected` lub `private`, aby ukryć metody poza klasą, a ukrywanie informacji za pomocą tych samych słów kluczowych, by ukryć dane poza klasą.

Napiszmy teraz prostą klasę `Animal`:

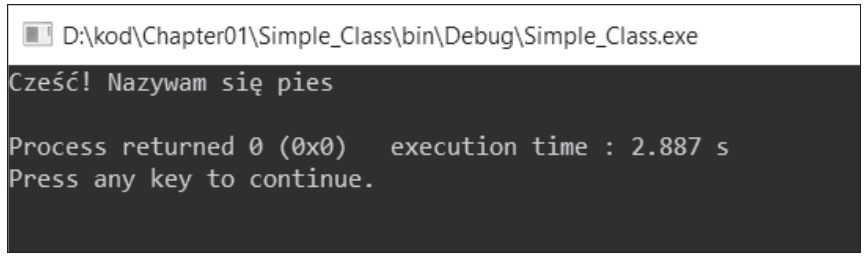
```
class Animal
{
private:
   string m_name;

public:
   void GiveName(string name)
   {
      m_name = name;
   }

   string GetName()
   {
      return m_name;
   }
};
```

Jak widać na przykładzie powyższego kodu, nie możemy bezpośrednio uzyskać dostępu do zmiennej `m_name`, ponieważ określiliśmy ją jako `private`. Mamy jednak dwie metody `public`, przy użyciu których możemy skorzystać ze zmiennej znajdującej się wewnątrz klasy. Metoda `GiveName()` modyfikuje wartość `m_name`, a `GetName()` zwraca wartość tej zmiennej. Poniżej widnieje kod, który wykorzystuje klasę `Animal` w taki sposób.

```
// Simple_Class.cbp
#include <iostream>

using namespace std;

class Animal
{
private:
   string m_name;

public:
   void GiveName(string name)
   {
      m_name = name;
   }

   string GetName()
   {
      return m_name;
   }
};

int main()
{
   Animal dog = Animal();
   dog.GiveName("pies");
```

```
    cout << "Cześć! Nazywam się " << dog.GetName() << endl;

    return 0;
}
```

W powyższym kodzie utworzyliśmy zmienną o nazwie `dog` typu `Animal`. Zmienna `dog` ma wobec tego wszystkie opcje właściwe dla `Animal`, takie jak wywoływanie metod `GiveName()` i `GetName()`. Poniżej widnieje okno z wynikiem, jaki powinniśmy otrzymać po skompilowaniu i uruchomieniu kodu:



Możemy powiedzieć, że ADT `Animal` ma dwie funkcje: `GiveName(string name)` i `GetName()`.

Po omówieniu prostych klas być może dostrzegasz podobieństwa między typem `struct` a klasami. W rzeczy samej, mają one podobne zachowania. Różnice są jednak takie, że struktura ma domyślne składowe `public`, podczas gdy klasy mają domyślne składowe `private`. Sam zalecam korzystanie z typu `struct` wyłącznie do tworzenia struktur danych (jako że nie ma on żadnych metod) i używanie klas do tworzenia ADT.

Jak widać w powyższym kodzie, przypisujemy zmienną do instancji klasy `Animal` przy użyciu jej konstruktora:

```
Animal dog = Animal();
```

Możemy jednak inicjalizować dane składowe klasy przy użyciu konstruktora klasy. Nazwa konstruktora jest taka sama jak nazwa klasy. Zrefaktoryzujmy klasę `Animal` tak, aby miała własny konstruktor. Kod powinien docelowo wyglądać następująco:

```
// Constructor.cbp
#include <iostream>

using namespace std;

class Animal
{
private:
    string m_name;

public:
    Animal(string name) : m_name(name)
    {

    }
```

```
    string GetName()
    {
        return m_name;
    }
};

int main()
{
    Animal dog = Animal("pies");

    cout << "Cześć! Nazywam się " << dog.GetName() << endl;

    return 0;
}
```

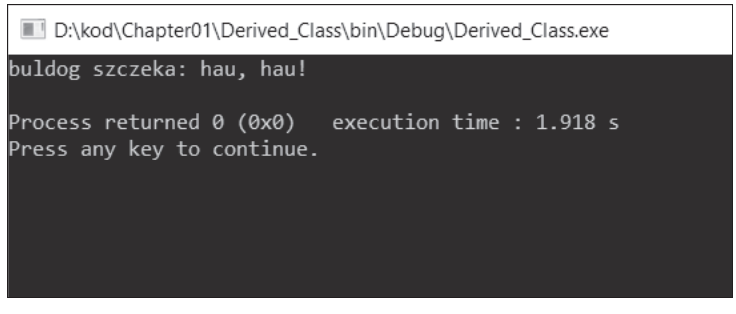
Jak widać, przy definiowaniu zmiennej `dog` inicjalizujemy także zmienną prywatną klasy `m_name`. Nie potrzebujemy już metody `GiveName()` do przypisywania zmiennej `private`. Po skompilowaniu i uruchomieniu powyższego kodu otrzymamy taki sam wynik jak wcześniej.

Zmiennej `dog` przypisaliśmy typ danych `Animal`. Możemy jednak również wywieść nową klasę z klasy bazowej. Uzyskana w ten sposób klasa pochodna będzie miała zachowania klasy bazowej. Zrefaktoryzujmy klasę `Animal` ponownie, dodając metodę wirtualną `MakeSound()`. Metoda wirtualna to metoda, która nie ma jeszcze implementacji, lecz samą definicję (znaną też jako **interfejs**). Klasa pochodna musi dodać implementację do metody wirtualnej przy użyciu słowa kluczowego `override` — w przeciwnym przypadku kompilator będzie marudził. Po uzyskaniu nowej klasy `Animal` utworzymy klasę o nazwie `Dog`, która będzie pochodną `Animal`. Kod powinien wyglądać następująco:

```
// Derived_Class.cbp
#include <iostream>

using namespace std;

class Animal
{
private:
    string m_name;

public:
    Animal(string name) : m_name(name)
    {

    }

    // Interfejs, który trzeba zaimplementować w klasie pochodnej.
    virtual string MakeSound() = 0;

    string GetName()
    {
        return m_name;
    }
};
```

```
class Dog : public Animal
{
public:
   // Przekazuje argumenty konstruktora.
   Dog(string name) : Animal(name) {}

   // Implementuje interfejs.
   string MakeSound() override
   {
      return "hau, hau!";
   }
};

int main()
{
   Dog dog = Dog("buldog");

   cout << dog.GetName() << " szczeka: ";
   cout << dog.MakeSound() << endl;

   return 0;
}
```

Mamy teraz dwie klasy: `Animal` (klasa bazowa) i `Dog` (klasa pochodna). Jak widać, klasa `Dog` musi zaimplementować metodę `MakeSound()`, ponieważ została ona zdefiniowana jako metoda wirtualna w klasie `Animal`. Instancja klasy `Dog` może też wywołać metodę `GetName()`, nawet jeśli nie została ona zaimplementowana w obrębie klasy `Dog`, jako że klasa pochodna dziedziczy wszystkie zachowania klasy bazowej. Po wykonaniu powyższego kodu powinniśmy otrzymać następujący wynik:



Tutaj znów możemy powiedzieć, że ADT `Dog` ma dwie funkcje: `GetName()` i `MakeSound()`.

Kolejnym wymogiem ADT jest zapewnienie możliwości kontrolowania wszystkich operacji przypisywania, aby możliwe było uniknięcie problemów z aliasingiem, wynikających z płytkiego kopiowania (niektóre składowe kopii mogą mieć referencje odnoszące się do tych samych obiektów co oryginał). W tym celu możemy zastosować technikę przeciążania operatora przypisania. Zrefaktoryzujemy teraz klasę `Dog`, tak aby uzyskała kopiujący operator przypisania. Kod powinien wyglądać następująco:

```
// Assignment_Operator_Overload.cbp
#include <iostream>

using namespace std;

class Animal
{
protected:
   string m_name;

public:
   Animal(string name) : m_name(name)
   {

   }

   // Interfejs, który trzeba zaimplementować w klasie pochodnej.
   virtual string MakeSound() = 0;

   string GetName()
   {
      return m_name;
   }
};

class Dog : public Animal
{
public:
   // Przekazuje argumenty konstruktora.
   Dog(string name) : Animal(name) {}

   // Przeciąża kopiujący operator przypisania.
   void operator = (const Dog &D)
   {
      m_name = D.m_name;
   }

   // Implementuje interfejs.
   string MakeSound() override
   {
      return "hau, hau!";
   }
};

int main()
{
   Dog dog = Dog("buldog");
   cout << dog.GetName() << " szczeka: ";
   cout << dog.MakeSound() << endl;

   Dog dog2 = dog;
   cout << dog2.GetName() << " szczeka: ";
   cout << dog2.MakeSound() << endl;

   return 0;
}
```

Dodaliśmy kopiujący operator przypisania, który przeciąża klasę `Dog`. Ponieważ jednak próbujemy uzyskać dostęp do zmiennej `m_name` w klasie bazowej z klasy pochodnej, musimy sprawić, aby zmienna `m_name` była `protected`, a nie `private`. Gdy kopiujemy `dog` na `dog2` w instrukcji `Dog dog2 = dog;` funkcji `main()`, możemy dopilnować, aby nie była to kopia płytka.

Posługiwanie się szablonami

Pobawmy się teraz szablonami. Szablony umożliwiają funkcjom i klasom współpracę z typami generycznymi. Dzięki nim funkcje i klasy mogą operować na wielu różnych typach danych i nie trzeba pisać ich ponownie na potrzeby każdego z typów. Przy użyciu szablonu możemy tworzyć różne typy danych, które omówimy w dalszej części tej książki.

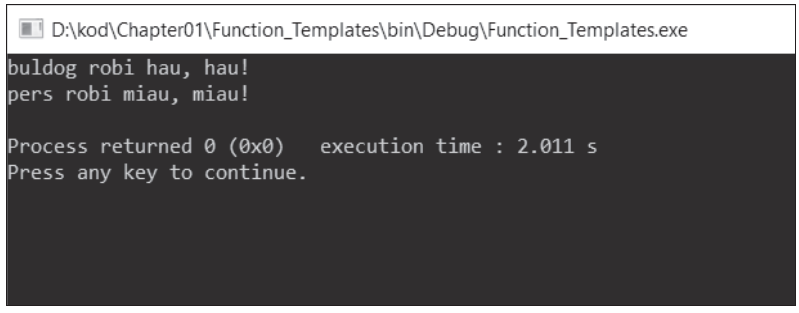
Szablony funkcji

Załóżmy, że mamy inną klasę pochodną od klasy `Animal` — na przykład `Cat`. Stworzymy teraz funkcję, która wywołuje metody `GetName()` i `MakeSound()` dla instancji `Dog` i `Cat`. Zamiast tworzyć dwie osobne funkcje, możemy skorzystać z szablonu następująco:

```
// Function_Templates.cbp
#include <iostream>

using namespace std;

class Animal
{
protected:
   string m_name;

public:
   Animal(string name) : m_name(name)
   {

   }

   // Interfejs, który trzeba zaimplementować w klasie pochodnej.
   virtual string MakeSound() = 0;

   string GetName()
   {
      return m_name;
   }
};

class Dog : public Animal
{
public:
   // Przekazuje argumenty konstruktora.
   Dog(string name) : Animal(name) {}

   // Przeciąża kopiujący operator przypisania.
   void operator = (const Dog &D)
```

```
    {
        m_name = D.m_name;
    }

    // Implementuje interfejs.
    string MakeSound() override
    {
        return "hau, hau!";
    }
};

class Cat : public Animal
{
public:
    // Przekazywanie argumentów konstruktora.
    Cat(string name) : Animal(name) {}

    // Przeciążenie kopiującego operatora przypisania.
    void operator = (const Cat &D)
    {
        m_name = D.m_name;
    }

    // Implementacja interfejsu.
    string MakeSound() override
    {
        return "miau, miau!";
    }
};

template<typename T>
void GetNameAndMakeSound(T& theAnimal)
{
    cout << theAnimal.GetName() << " robi ";
    cout << theAnimal.MakeSound() << endl;
}

int main()
{
    Dog dog = Dog("buldog");
    GetNameAndMakeSound(dog);

    Cat cat = Cat("pers");
    GetNameAndMakeSound(cat);

    return 0;
}
```

Widzimy, że funkcji `GetNameAndMakeSound()` możemy przekazać typy danych `Dog` i `Cat`, ponieważ zdefiniowaliśmy szablon `<typename T>` przed zdefiniowaniem funkcji. W C++ `typename` jest słowem kluczowym służącym do tworzenia szablonów. **Słowo kluczowe** wskazuje, że symbol w definicji lub deklaracji szablonu jest typem (w tym przykładzie symbol to `T`). W wyniku tego funkcja staje się generyczna i może przyjmować różne typy danych. Po skompilowaniu i wykonaniu powyższego kodu powinniśmy otrzymać następujący wynik w oknie konsoli:



Należy się koniecznie upewnić, czy typ danych przekazywany funkcji generycznej ma możliwość wywołania wszystkich jej operacji. Kompilator z powodzeniem skompiluje kod, nawet jeśli przekazany typ danych nie obsługuje oczekiwanej operacji. W powyższym przykładzie szablonu funkcji należy przekazać typ danych, który jest instancją klasy `Animal`, może to być więc albo instancja klasy `Animal`, albo instancja klasy pochodnej.

Szablony klas

Podobnie jak szablon funkcji, szablon klasy służy do tworzenia generycznych klas, które przyjmują różne typy danych. Zrefaktoryzujmy kod *Function_Template.cbp*, dodając do niego nowy szablon klasy. Kod powinien wyglądać następująco:

```
// Class_Templates.cbp
#include <iostream>

using namespace std;

class Animal
{
protected:
   string m_name;

public:
   Animal(string name) : m_name(name)
   {

   }

   // Interfejs, który trzeba zaimplementować w klasie pochodnej.
   virtual string MakeSound() = 0;

   string GetName()
   {
      return m_name;
   }
};

class Dog : public Animal
{
public:
   // Przekazuje argumenty konstruktora.
   Dog(string name) : Animal(name) {}
```

```
    // Przeciąża kopiujący operator przypisania.
    void operator = (const Dog &D)
    {
       m_name = D.m_name;
    }

    // Implementuje interfejs.
    string MakeSound() override
    {
       return "hau, hau!";
    }
};

class Cat : public Animal
{
public:
    // Przekazuje argumenty konstruktora.
    Cat(string name) : Animal(name) {}

    // Przeciąża kopiujący operator przypisania.
    void operator = (const Cat &D)
    {
       m_name = D.m_name;
    }

    // Implementuje interfejs.
    string MakeSound() override
    {
       return "miau, miau!";
    }
};

template<typename I>
void GetNameAndMakeSound(T& theAnimal)
{
   cout << theAnimal.GetName() << " robi ";
   cout << theAnimal.MakeSound() << endl;
}

template <typename T>
class AnimalTemplate
{
private:
   T m_animal;

public:
   AnimalTemplate(T animal) : m_animal(animal) {}
   void GetNameAndMakeSound(T& theAnimal)
   {
      cout << m_animal.GetName() << " robi ";
      cout << m_animal.MakeSound() << endl;
   }
};
```

```
int main()
{
   Dog dog = Dog("buldog");
   AnimalTemplate<Dog> dogTemplate(dog);
   dogTemplate.GetNameAndMakeSound(dog);

   Cat cat = Cat("pers");
   AnimalTemplate<Cat> catTemplate(cat);
   catTemplate.GetNameAndMakeSound(cat);

   return 0;
}
```

Utworzyliśmy nową klasę o nazwie `AnimalTemplate`. Jest to szablon klasy, którego można użyć z dowolnym typem danych. Musimy jednak zdefiniować typ danych w nawiasie ostrokątnym, kiedy korzystamy z instancji `dogTemplate` i `catTemplate`. Po skompilowaniu i uruchomieniu kodu otrzymamy taki sam wynik jak w przypadku kodu *Function_Template.cbp*.

Biblioteka standardowych szablonów

Programowanie w C++ wiąże się z korzystaniem z innej przydatnej funkcji, związanej ze stosowaniem szablonów klas — biblioteki standardowych szablonów (ang. *Standard Template Library* — **STL**). Jest to zbiór szablonów klas zapewniających wszystkie funkcje, których używa się powszechnie przy operowaniu na różnych strukturach danych. Na STL składają się cztery komponenty: algorytmy, kontenery, iteratory i funkcje. Przyjrzyjmy się im bliżej.

Algorytmy wykorzystywane są na zakresach elementów i służą m.in. do sortowania i wyszukiwania. Algorytm sortujący służy do porządkowania elementów w kolejności rosnącej lub malejącej. Algorytm wyszukiwania wykorzystuje się do odnajdowania konkretnych wartości w zbiorach elementów.

Kontenery służą do przechowywania obiektów i danych. Standardowym kontenerem, będącym w powszechnym użyciu, jest wektor. Wektor przypomina tablicę, ale ma możliwość automatycznej zmiany własnego rozmiaru, kiedy element zostaje do niego dodany lub z niego usunięty.

Iteratory służą do pracy z sekwencjami wartości. Każdy kontener ma własny iterator. Kontener wektora oferuje na przykład funkcje `begin()`, `end()`, `rbegin()` i `rend()`.

Funkcje biblioteki służą do przeciążania istniejących funkcji. Instancja tego komponentu to **funktor**, czy też **obiekt funkcji**. Funktor jest wskaźnikiem funkcji, parametryzuje istniejącą już funkcję.

W tym punkcie nie napiszemy żadnego kodu przykładowego, jako że chcę jedynie zwrócić Twoją uwagę na to, że STL — na szczęście — istnieje i jest przydatną biblioteką C++. Temat STL omówimy szerzej w kolejnych rozdziałach przy objaśnianiu tworzenia struktur danych.

Analiza algorytmów

Dobry algorytm musi mieć jak największą wydajność. W tym podrozdziale omówimy temat analizy złożoności czasowej podstawowych funkcji.

Analiza asymptotyczna

Zacznijmy od analizy asymptotycznej, pozwalającej na sprawdzenie złożoności czasowej algorytmów. Ta analiza pomija stałe oraz wyrażenia niższego rzędu. Załóżmy, że mamy funkcję, która wyświetla liczby od *0* do *n*. Oto jej kod:

```
void Looping(int n)
{
   int i = 0;

   while(i < n)
   {
      cout << i << endl;
      i = i + 1;
   }
}
```

Obliczmy teraz złożoność czasową powyższego algorytmu, licząc wszystkie instrukcje. Zacznijmy od pierwszej:

```
int i = 0;
```

Powyższa instrukcja wykonywana jest przez funkcję tylko raz, więc jej wartość to 1. Poniższy fragment kodu obejmuje pozostałe instrukcje z funkcji `Looping()`:

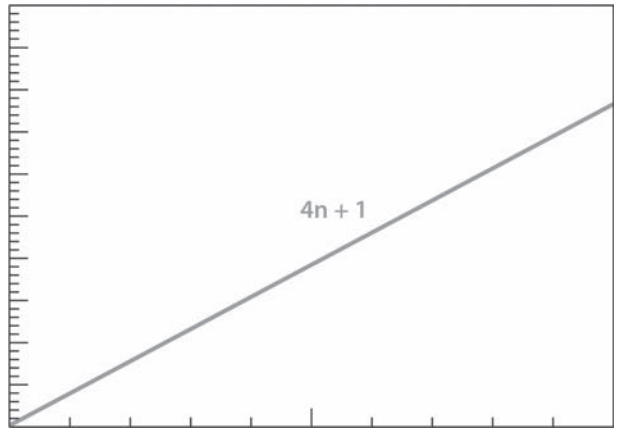
```
   while(i < n)
   {
      cout << i << endl;
      i = i + 1;
   }
```

Porównanie w pętli `while` ma wartość 1. Dla uproszczenia możemy powiedzieć, że wartość obydwu instrukcji w zakresie pętli `while` wynosi 3, ponieważ potrzebuje ona 1 do wyświetlenia zmiennej `i` oraz 2 do wykonania operacji przypisania (`=`) i dodawania (`+`).

Niemniej jednak liczba wykonanych instrukcji z powyższego fragmentu zależna jest od wartości *n*, czyli wynosi $(1 + 3) \cdot n$, czy też $4n$. Łączna liczba instrukcji, które muszą zostać wykonane w funkcji `Looping()`, to $1 + 4n$. Co za tym idzie, złożoność tej funkcji to:

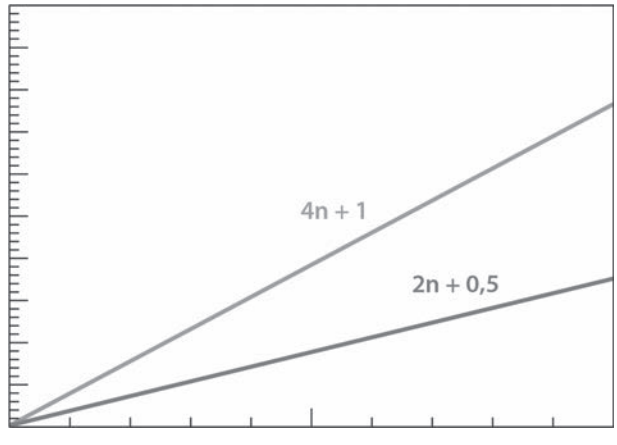
```
Złożoność czasowa(n) = 4n + 1
```

Oto wykres złożoności:



Na wszystkich przedstawionych w tej książce wykresach oś *x* odzwierciedla *liczbę danych wejściowych (n)*, a oś *y czas wykonania*.

Jak widać na powyższym wykresie, krzywa tej funkcji jest liniowa. Ponieważ jednak złożoność czasowa zależna jest także od innych parametrów, takich jak specyfikacja sprzętowa, dla funkcji `Looping()` możemy obliczyć inną złożoność, właściwą dla szybszego sprzętu. Przyjmijmy, że złożoność czasowa ma wartość *2n + 0,5*. W takim przypadku krzywa funkcji będzie wyglądać następująco:



Jak widać, krzywa jest liniowa dla obydwu złożoności. Z tego względu możemy pominąć stałą i wyrażenia niższego rzędu analizy asymptotycznej, a następnie wskazać, że złożoność wynosi *n*, tak jak w poniższym wzorze:

```
Złożoność czasowa(n) = n
```

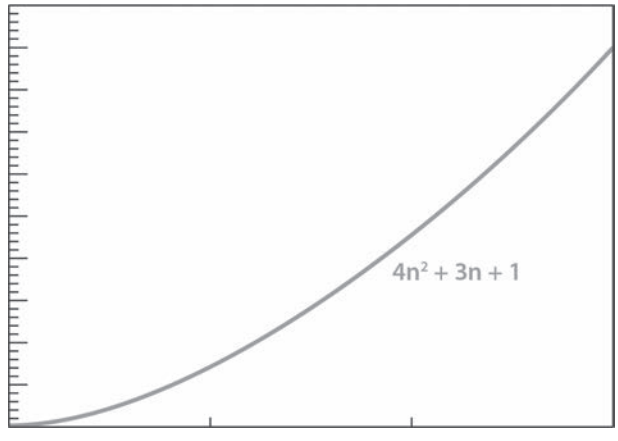
Przejdźmy do kolejnej funkcji. Gdy mamy do czynienia z zagnieżdżoną pętlą `while`, musimy uwzględnić drugą złożoność:

```
void Pairing(int n)
{
   int i = 0;

   while(i < n)
   {
      int j = 0;

      while(j < n)
      {
         cout << i << ", " << j << endl;
         j = j + 1;
      }
      i = i + 1;
   }
}
```

Na podstawie przypadku funkcji `Looping()` możemy stwierdzić, że złożoność wewnętrznej pętli `while` funkcji `Pairing()` wynosi $4n + 1$. Obliczamy następnie złożoność zewnętrznej pętli `while`, z czego uzyskujemy wynik $1 + (n \cdot (1 + (4n + 1) + 2))$, co równa się $1 + 3n + 4n^2$. Złożoność powyższego kodu jest zatem następująca:

```
Złożoność czasowa(n) = 4n² + 3n + 1
```

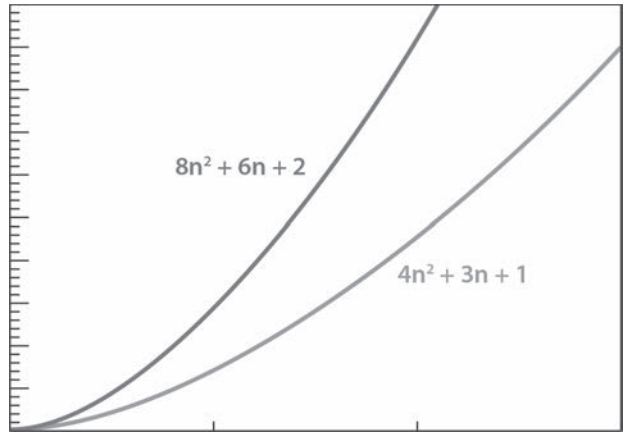
Krzywa tej złożoności wygląda tak:



Jeśli natomiast uruchomimy ten kod na wolniejszym sprzęcie, złożoność czasowa może stać się dwukrotnie większa. Zapisalibyśmy to następująco:

```
Złożoność czasowa(n) = 8n² + 6n + 2
```

Krzywa dla powyższego wzoru wyglądałaby tak jak na rysunku na następnej stronie.



Jak już wskazałem wcześniej, ponieważ analiza asymptotyczna pomija stałe i wyrażenia niższego rzędu, zapis złożoności wygląda następująco:

```
Złożoność czasowa(n) = n²
```

Najgorsze, średnie i najlepsze przypadki

W poprzednim punkcie obliczaliśmy złożoność czasową kodu przy użyciu algorytmu asymptotycznego. W tym punkcie zajmiemy się przypadkami implementacji algorytmu. Istnieją trzy przypadki implementacji złożoności czasowej algorytmu: najgorszy, średni i najlepszy. Zanim je omówimy, spójrzmy na poniższą implementację funkcji `Search()`:

```
int Search(int arr[], int arrSize, int x)
{
   // Iterowanie po elementach arr.
   for (int i = 0; i < arrSize; ++i)
   {
      // Jeśli x zostało znalezione, zwraca indeks x.
      if (arr[i] == x)
         return i;
   }

   // Jeśli x nie zostało znalezione, zwraca -1.
   return -1;
}
```

Jak widać, funkcja `Search()` znajduje indeks docelowego elementu (`x`) z tablicy `arr`, zawierającej `arrSize` elementów. Załóżmy, że mamy tablicę `{42, 55, 39, 71, 20, 18, 6, 84}`. Znajdziemy dla niej następujące przypadki:

- **Analiza najgorszego przypadku** jest obliczeniem górnego kresu czasu wykonywania algorytmu. W przypadku funkcji `Search()` górny kres określa element, który nie pojawia się w `arr`, na przykład `60`; funkcja musi wtedy iterować po wszystkich elementach `arr`, aby i tak nie znaleźć wskazanego elementu.

- **Analiza średniego przypadku** jest obliczeniem uwzględniającym wszystkie możliwe dane wejściowe w ramach wykonywania algorytmu, w tym także element, który nie występuje w tablicy `arr`.
- **Analiza najlepszego przypadku** jest obliczeniem dolnego kresu czasu wykonywania algorytmu. W przypadku funkcji `Search()` dolny kres określa pierwszy element tablicy `arr`, czyli `42`. Kiedy szukamy elementu `42`, funkcja iteruje po tablicy `arr` jednokrotnie, wobec czego `arrSize` nie ma znaczenia.

Notacja Θ, O i Ω

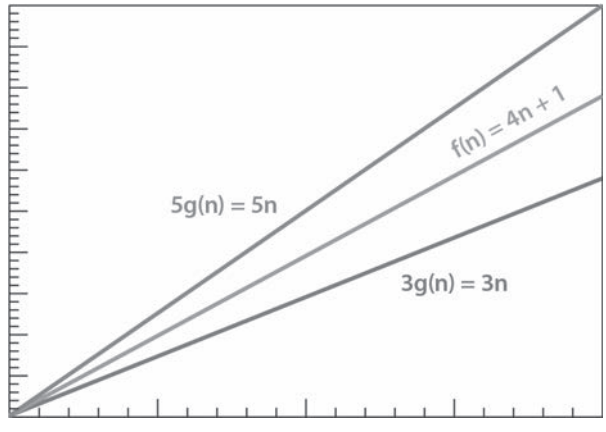
Po omówieniu analizy asymptotycznej i trzech przypadków algorytmów zajmijmy się przedstawianiem złożoności czasowej algorytmów za pomocą notacji asymptotycznej. Istnieją trzy notacje asymptotyczne wykorzystywane z algorytmami: Θ, O i Ω.

Notacja Θ ogranicza funkcję od góry i od dołu tak jak w omówionej wcześniej analizie asymptotycznej i również pomija stałe.

Przyjmijmy, że mamy funkcję o złożoności czasowej *4n + 1*. Jako że jest to funkcja liniowa, możemy zapisać ją następująco:

```
f(n) = 4n + 1
```

Załóżmy, że mamy funkcję *g(n)*, gdzie *f(n)* to Θ *g(n)*, jeżeli wartość *f(n)* znajduje się pomiędzy *c1 · g(n)* (*ograniczenie dolne*) a *c2 · g(n)* (*ograniczenie górne*). Ponieważ dla *f(n)* stała kierunkowa ma wartość *4*, wybieramy losowe ograniczenie dolne mniejsze niż *4* — czyli *3* — oraz ograniczenie górne większe niż *4* — czyli *5*. Wykres wygląda następująco:



Ze złożoności czasowej *f(n)* możemy uzyskać złożoność asymptotyczną *n*. Otrzymujemy wtedy *g(n) – n*, które oparte jest na złożoności asymptotycznej *4n + 1*. Możemy określić górne i dolne ograniczenie dla *g(n) = n*. Wybierzmy *3* dla ograniczenia dolnego i *5* dla górnego. Możemy teraz przekształcić funkcję *g(n) = n* następująco:

```
3g(n) = 3n
5g(n) = 5n
```

Notacja O jest notacją ograniczającą funkcję od góry wyłącznie przy użyciu górnej granicy algorytmu. Na podstawie poprzedniej notacji, $f(n) = 4n + 1$, możemy wskazać, że złożoność czasowa $f(n)$ to $O(n)$. Gdybyśmy skorzystali z notacji Θ, moglibyśmy powiedzieć, że najgorszy przypadek złożoności czasowej $f(n)$ to $\Theta(n)$, a najlepszy przypadek to $\Theta(1)$.

Notacja Ω jest przeciwieństwem notacji O. Wykorzystuje ona dolną granicę algorytmu do przeanalizowania złożoności czasowej. Innymi słowy, oparta jest na najlepszym przypadku notacji Θ. Moglibyśmy zatem powiedzieć, że złożoność czasowa dla $f(n)$ wynosi $\Omega(1)$.

Metoda rekurencyjna

W poprzednim przykładzie obliczyliśmy złożoność metody iteracyjnej. Zróbmy teraz to samo z metodą rekurencyjną. Przyjmijmy, że musimy obliczyć silnię pewnej liczby, na przykład 6, czyli uzyskać wynik $6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$. W tym celu możemy skorzystać z metody rekurencyjnej:

```
int Factorial(int n)
{
   if(n == 1)
      return 1;

   return n * Factorial(n - 1);
}
```

Dla powyższej funkcji możemy obliczyć złożoność w sposób podobny do wykorzystanego w przypadku metody iteracyjnej. Wzór to $f(n) = n$, ponieważ złożoność zależna jest od ilości przetwarzanych danych (n). Możemy użyć współczynnika stałego, na przykład c, aby obliczyć dolną i górną granicę.

Analiza kosztu zamortyzowanego

W poprzednim punkcie omówiliśmy obliczanie złożoności dla pojedynczej danej wejściowej n. Czasami jednak mamy do czynienia z większą liczbą danych. Przyjrzyjmy się poniższej implementacji funkcji `SumOfDivision()`:

```
int SumOfDivision(
   int nArr[], int n, int mArr[], int m)
{
   int total = 0;

   for(int i = 0; i < n; ++i)
   {
      for(int j = 0; j < m; ++j)
      {
         total += (nArr[i] * mArr[j]);
      }
   }

   return total;
}
```

Tutaj należy skorzystać z analizy kosztu zamortyzowanego. Taka analiza polega na obliczeniu złożoności wykonywania operacji dla różnych danych wejściowych, na przykład wielu elementów umieszczonych w wielu tablicach. Złożoność nie jest uzależniona wyłącznie od zmiennej `n`, ale także od zmiennej `m`. Można ją wyrazić następująco:

```
Złożoność czasowa(n, m) = n * m
```

Te metody analizy omówimy bardziej szczegółowo w kolejnych rozdziałach.

Podsumowanie

W tym rozdziale omówiliśmy podstawy C++ (pisanie prostych programów, IDE, sterowanie przepływem) oraz wszystkie typy danych (podstawowe i zaawansowane, w tym szablony i STL), które wykorzystamy do tworzenia struktur danych w następnych rozdziałach. Wprowadziłem również podstawy analizy złożoności, którą pogłębimy w kolejnych rozdziałach.

Przejdziemy teraz do tworzenia pierwszych struktur danych, czyli list wiązanych, oraz do wykonywania operacji na strukturze danych.

Pytania

- Jaka funkcja w C++ jest wykonywana jako pierwsza?
- Wymień podstawowe typy danych C++.
- Czego można używać do sterowania przepływem kodu?
- Jaka jest różnica między typem `enum` a `struct`?
- Czym jest abstrakcja, ukrywanie informacji i enkapsulacja?
- Jakiego słowa kluczowego używa się do tworzenia szablonu w C++?
- Czym różni się szablon funkcji od szablonu klasy?
- Jakie są różnice pomiędzy notacjami Θ, O i Ω?

Dodatkowe materiały

Możesz także zapoznać się z następującymi materiałami:

- *http://www.learncpp.com/*
- *https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/*
- *https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/*
- *https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/*

Skorowidz

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

W

Z

C++. O jakości kodu decyduje algorytm i odpowiednia struktura danych!

C++ to dojrzały język programowania o wielu różnych zastosowaniach. Inżynier oprogramowania, który chce w pełni skorzystać z jego zalet, powinien płynnie posługiwać się dostępnymi w tym języku strukturami danych i algorytmami. W ten sposób łatwiej można rozwiązywać konkretne problemy. Zastosowanie odpowiedniej struktury danych oraz algorytmu jest również ważne z punktu widzenia wydajności działania kodu, co bezpośrednio przekłada się na szybkość pracy aplikacji. Bez dogłębnego zrozumienia tych zagadnień bardzo trudno nauczyć się biegle programować w C++.

Dzięki tej książce dowiesz się, na czym polega implementacja klasycznych struktur danych i algorytmów w C++. Znajdziesz tu również przystępne wprowadzenie do podstawowych konstrukcji językowych oraz do korzystania z zintegrowanego środowiska programistycznego (IDE). Ponadto dowiesz się, w jaki sposób przechowywać dane za pomocą list wiązanych, tablic, stosów i kolejek, a także jak zaimplementować algorytmy sortowania, takie jak sortowanie szybkie i sortowanie przez kopcowanie, oraz algorytmy wyszukiwania, takie jak wyszukiwanie liniowe czy binarne. Kolejnym ważnym zagadnieniem ujętym w książce jest wysoka wydajność algorytmów operujących na ciągach znakowych i strukturach mieszających, jak również analiza algorytmów siłowych, zachłannych i wielu innych.

Najciekawsze zagadnienia ujęte w książce:

- podstawy C++, w tym kontrola przepływu kodu i abstrakcyjne typy danych
- listy, listy wiązane, stosy i kolejki
- algorytmy sortowania, w tym bąbelkowe, przez selekcję, wstawianie, scalanie
- tworzenie hierarchicznej struktury drzewa
- praktyczne aspekty implementacji algorytmów

Wisnu Anggoro — jest doświadczonym programistą C/C++, certyfikowanym przez Microsoft (Microsoft Certified Professional) w zakresie programowania w C++. Programowaniem zajmuje się od czasów szkolnych (czyli około 20 lat). Wyspecjalizował się w programowaniu kart elektronicznych, komputerów i aplikacji internetowych. Obecnie pracuje jako starszy programista kart elektronicznych w CIPTA, indonezyjskiej firmie specjalizującej się w innowacji i technologii kart.

ISBN 978-83-283-5185-1