Helion

# Nowoczesny C++

Zbiór praktycznych zadań dla przyszłych ekspertów


Marius Bancila

Tytuł oryginału: The Modern C++ Challenge

Tłumaczenie: Jacek Janusz

ISBN: 978-83-283-5211-7



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/nowcpp*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

ROZDZIAŁ 1

# Zadania matematyczne

## Zadania

### 1. Suma liczb naturalnych podzielnych przez 3 lub 5

Napisz program, który oblicza sumę wszystkich liczb naturalnych podzielnych przez 3 lub 5 aż do podanej wartości granicznej wprowadzonej przez użytkownika.

### 2. Największy wspólny dzielnik

Napisz program, który obliczy i wyświetli największy wspólny dzielnik dwóch dodatnich liczb całkowitych.

### 3. Najmniejsza wspólna wielokrotność

Napisz program, który obliczy i wyświetli najmniejszą wspólną wielokrotność dla dwóch lub więcej dodatnich liczb całkowitych.

### 4. Największa liczba pierwsza mniejsza od podanej

Napisz program, który obliczy i wyświetli największą liczbę pierwszą mniejszą od liczby podanej przez użytkownika, która powinna być dodatnią liczbą całkowitą.

### 5. Liczby pierwsze szóstkowe

Napisz program, który wyświetli wszystkie liczby pierwsze szóstkowe aż do limitu wprowadzonego przez użytkownika.

## 6. Liczby obfite

Napisz program, który wyświetli wszystkie liczby obfite oraz ich obfitość aż do wartości wprowadzonej przez użytkownika.

## 7. Liczby zaprzyjaźnione

Napisz program, który wyświetli listę wszystkich par liczb zaprzyjaźnionych mniejszych niż milion.

## 8. Liczby Armstronga

Napisz program, który wypisze wszystkie liczby Armstronga zawierające trzy cyfry.

## 9. Czynniki pierwsze liczby

Napisz program, który wyświetla czynniki pierwsze liczby wprowadzonej przez użytkownika.

## 10. Kod Graya

Napisz program wyświetlający naturalne reprezentacje binarne, reprezentacje kodu Graya i dekodowane wartości kodu Graya dla wszystkich liczb 5-bitowych.

## 11. Przekształcanie liczb arabskich na rzymskie

Napisz program, który biorąc pod uwagę liczbę wprowadzoną przez użytkownika, wyświetla jej odpowiednik w postaci liczby rzymskiej.

## 12. Najdłuższy ciąg Collatza

Napisz program, który ustali i wyświetli, jaka liczba mniejsza od miliona stworzy najdłuższy ciąg Collatza oraz jaka będzie jego długość.

## 13. Wyznaczanie liczby π

Napisz program, który obliczy wartość $\pi$ z dokładnością do dwóch cyfr dziesiętnych.

## 14. Sprawdzanie numerów ISBN

Napisz program, który potwierdzi, że 10-cyfrowa wartość wprowadzona przez użytkownika reprezentuje poprawny identyfikator ISBN-10.

# Rozwiązania

## 1. Suma liczb naturalnych podzielnych przez 3 lub 5

Rozwiązanie tego zadania polega na przetwarzaniu wszystkich liczb, począwszy od 3 (1 i 2 nie są podzielne przez 3, więc nie ma sensu ich sprawdzanie) aż do limitu wprowadzonego przez użytkownika. Zastosuj operację modulo, aby sprawdzić, czy reszta z dzielenia liczby przez 3 i 5 wynosi 0. Jednak sztuczka w przypadku wyższego limitu polega na użyciu dla sumy typu `long long`, a nie `int` lub `long`, co skutkowałoby przepełnieniem podczas sumowania do wartości 100 000:

```
int main()
{
   setlocale(LC_ALL, "polish");

   unsigned int limit = 0;
   std::cout << "Ograniczenie górne:";
   std::cin >> limit;

   unsigned long long sum = 0;
   for (unsigned int i = 3; i < limit; ++i)
   {
      if (i % 3 == 0 || i % 5 == 0)
         sum += i;
   }

   std::cout << "suma=" << sum << std::endl;
}
```

## 2. Największy wspólny dzielnik

Największy wspólny dzielnik (w skrócie NWD; ang. *gcd*) dwóch lub więcej niezerowych liczb całkowitych, znany również jako największy wspólny podzielnik, to największa dodatnia liczba całkowita dzieląca każdą z nich. Istnieje kilka sposobów obliczania NWD — skuteczną metodą jest algorytm Euklidesa. W przypadku dwóch liczb całkowitych algorytm ten ma taką postać:

```
gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)
```

Powyższy wzór może zostać w bardzo prosty sposób zaimplementowany w języku C++ jako funkcja rekurencyjna:

```
unsigned int gcd(unsigned int const a, unsigned int const b)
{
   return b == 0 ? a : gcd(b, a % b);
}
```

Nierekurencyjna implementacja algorytmu Euklidesa przedstawia się następująco:

```
unsigned int gcd(unsigned int a, unsigned int b)
{
   while (b != 0) {
      unsigned int r = a % b;
      a = b;
      b = r;
   }
   return a;
```

W języku C++ 17 istnieje funkcja `constexpr`, nazwana `gcd()` i zdefiniowana w pliku nagłówkowym `<numeric>`, która oblicza największy wspólny dzielnik dwóch liczb.

# 3. Najmniejsza wspólna wielokrotność

Najmniejsza wspólna wielokrotność (NWW — ang. *lcm*) dwóch lub więcej niezerowych liczb całkowitych to najmniejsza dodatnia liczba całkowita, której dzielnikiem jest każda z nich. Możliwym sposobem obliczenia najmniejszej wspólnej wielokrotności jest zredukowanie problemu do obliczenia największego wspólnego dzielnika. W tym przypadku używany jest następujący wzór:

```
lcm(a, b) = abs(a, b) / gcd(a, b)
```

Funkcja obliczająca najmniejszą wspólną wielokrotność może wyglądać tak:

```
int lcm(int const a, int const b)
{
   int h = gcd(a, b);
   return h ? (a * (b / h)) : 0;
}
```

Aby obliczyć NWW dla więcej niż dwóch liczb całkowitych, możesz użyć algorytmu `std::accumulate`, zdefiniowanego w pliku nagłówkowym `<numeric>`:

```
template<class InputIt>
int lcmr(InputIt first, InputIt last)
{
   return std::accumulate(first, last, 1, lcm);
}
```

W języku C++ 17 istnieje funkcja `constexpr`, nazwana `lcm()` i zdefiniowana w pliku nagłówkowym `<numeric>`, która oblicza najmniejszą wspólną wielokrotność dwóch liczb.

# 4. Największa liczba pierwsza mniejsza od podanej

Liczba pierwsza ma tylko dwa dzielniki: 1 i siebie samą. Aby znaleźć największą liczbę pierwszą mniejszą od podanej wartości, należy najpierw napisać funkcję, która sprawdza, czy dana liczba jest liczbą pierwszą, a następnie wywołać tę funkcję, zaczynając od wprowadzonej wartości. W dalszej kolejności trzeba się przemieszczać w kierunku jedynki aż do napotkania pierwszej liczby pierwszej. Istnieją różne algorytmy ustalania, czy liczba jest liczbą pierwszą. Typowa implementacja służąca do sprawdzania liczb pierwszych jest następująca:

```
bool is_prime(int const num)
{
   if (num <= 3)
   {
      return num > 1;
   }
   else if (num % 2 == 0 || num % 3 == 0)
   {
      return false;
   }
   else
   {
      for (int i = 5; i * i <= num; i += 6)
      {
         if (num % i == 0 || num % (i + 2) == 0)
         {
            return false;
         }
      }

      return true;
   }
}
```

Powyższa funkcja może zostać użyta w taki sposób:

```
int main()
{
   setlocale(LC_ALL, "polish");

   int limit = 0;
   std::cout << "Ograniczenie górne:";
   std::cin >> limit;

   for (int i = limit; i > 1; i--)
   {
      if (is_prime(i))
      {
         std::cout << "Największa liczba pierwsza:" << i << std::endl;
         return 0;
      }
   }
}
```

# 5. Liczby pierwsze szóstkowe

Liczby pierwsze szóstkowe to dwie liczby pierwsze, które różnią się od siebie o wartość sześć (na przykład 5 i 11 lub 13 i 19). Istnieją również *bliźniacze liczby pierwsze*, które różnią się o wartość dwa, a także *liczby pokrewne* (lub *stryjeczne*) różniące się o wartość cztery.

W poprzednim zadaniu zaimplementowaliśmy funkcję, która sprawdza, czy dana liczba całkowita jest liczbą pierwszą. Ta funkcja zostanie ponownie użyta w tym ćwiczeniu. Musisz sprawdzić, czy jeśli liczba `n` jest liczbą pierwszą, również liczba `n + 6` jest liczbą pierwszą — jeżeli tak, wówczas wyświetlisz te dwie liczby w konsoli:

```
int main()
{
   setlocale(LC_ALL, "polish");

   int limit = 0;
   std::cout << "Ograniczenie górne:";
   std::cin >> limit;

   for (int n = 2; n <= limit; n++)
   {
      if (is_prime(n) && is_prime(n + 6))
      {
         std::cout << n << "," << n + 6 << std::endl;
      }
   }
}
```

Dodatkowym ćwiczeniem mogłoby być obliczanie i wyświetlanie ciągów liczb pierwszych szóstkowych o długości trzy, cztery i pięć.

# 6. Liczby obfite

Liczba obfita jest liczbą, dla której suma jej dzielników właściwych jest większa od niej samej. Dzielnikami właściwymi liczby są dodatnie czynniki pierwsze różniące się od niej. Wartość, o jaką suma dzielników właściwych przekracza liczbę, nazywa się obfitością. Na przykład liczba 12 ma dzielniki właściwe 1, 2, 3, 4 i 6. Ich suma wynosi 16, co czyni liczbę 12 obfitą. Jej obfitość wynosi 4 (czyli 16 – 12).

Aby określić sumę dzielników właściwych, próbujemy wszystkie liczby od 2 do pierwiastka kwadratowego liczby (wszystkie czynniki pierwsze są mniejsze od tej wartości lub jej równe). Jeżeli bieżąca wartość (nazwijmy ją `i`) podzieli liczbę, wówczas `i` oraz `num / i` są dzielnikami. Jeśli jednak są one równe (na przykład jeżeli `i = 3`, a `n = 9`, wówczas `i` dzieli 9, lecz `n / i = 3`), dodajemy tylko `i`, ponieważ dzielniki właściwe mogą zostać dodane tylko raz. W przeciwnym razie dodajemy zarówno `i`, jak i `num / i` i kontynuujemy algorytm:

```
int sum_proper_divisors(int const number)
{
   int result = 1;
```

```
    for (int i = 2; i <= std::sqrt(number); i++)
    {
       if (number%i == 0)
       {
          result += (i == (number / i)) ? i : (i + number / i);
       }
    }

    return result;
}
```

Wyświetlanie liczb obfitych jest proste — polega na wykonaniu iteracji aż do osiągnięcia określonej wartości granicznej, obliczeniu sumy dzielników właściwych i porównaniu jej z liczbą:

```
void print_abundant(int const limit)
{
   for (int number = 10; number <= limit; ++number)
   {
      auto sum = sum_proper_divisors(number);
      if (sum > number)
      {
         std::cout
            << number
            << ", obfitość=" << sum - number << std::endl;
      }
   }
}

int main()
{
   setlocale(LC_ALL, "polish");

   int limit = 0;
   std::cout << "Ograniczenie górne:";
   std::cin >> limit;

   print_abundant(limit);
}
```

## 7. Liczby zaprzyjaźnione

Uważa się, że dwie liczby są zaprzyjaźnione, jeśli suma dzielników właściwych jednej liczby jest równa takiej samej sumie obliczonej dla drugiej z nich. Dzielniki właściwe danej liczby są jej dodatnimi czynnikami pierwszymi różnymi od niej. Liczby zaprzyjaźnione nie powinny być mylone z *liczbami przyjacielskimi[JJ1]*. Na przykład liczba 220 ma dzielniki właściwe 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 i 110, których suma wynosi 284. Dzielniki właściwe liczby 284 to 1, 2, 4, 71 i 142; ich suma wynosi 220. Dlatego też liczby 220 i 284 są uznawane za zaprzyjaźnione.

Rozwiązanie tego zadania polega na przetwarzaniu wszystkich liczb aż do określonego limitu. Dla każdej liczby oblicz sumę jej dzielników właściwych. Nazwijmy ją `sum1`. Powtórz proces i wyznacz sumę dzielników właściwych. Jeżeli wynik będzie równy liczbie pierwotnej, wówczas ona oraz `sum1` utworzą liczby zaprzyjaźnione:

```
void print_amicables(int const limit)
{
   for (int number = 4; number < limit; ++number)
   {
      auto sum1 = sum_proper_divisors(number);
      if (sum1 < limit)
      {
         auto sum2 = sum_proper_divisors(sum1);
         if (sum2 == number && number != sum1)
         {
            std::cout << number << "," << sum1 << std::endl;
         }
      }
   }
}
```

W powyższym przykładzie `sum_proper_divisors()` jest funkcją utworzoną w celu rozwiązania zadania wyznaczania liczb obfitych.

> Powyższa funkcja wyświetla pary liczb dwukrotnie, na przykład 220,284 i 284,220. Zmodyfikuj tę implementację w taki sposób, aby wyświetlać tylko jedną parę.

## 8. Liczby Armstronga

Liczba Armstronga (nazwana tak na cześć Michaela F. Armstronga), zwana również liczbą narcystyczną, to liczba, która jest sumą swoich cyfr podniesionych do potęgi równej ich liczbie. Przykładowo, najmniejsza liczba Armstronga to 153, która jest równa $1^3 + 5^3 + 3^3$.

Aby ustalić, czy liczba z trzema cyframi jest liczbą narcystyczną, musisz najpierw określić te cyfry, aby zsumować ich potęgi. Jednak wymaga to dzielenia i operacji modulo, które są kosztowne. Znacznie szybszym sposobem obliczenia jest poleganie na tym, że liczba jest sumą cyfr pomnożonych przez wartość 10 podniesioną do potęgi zależnej od położenia danej cyfry. Innymi słowy, dla liczb do wartości 1000 używamy wzoru `a * 10 ^ 2 + b * 10 ^ 1 + c`. Ponieważ musisz tylko ustalać liczby trzycyfrowe, oznacza to, że wartość `a` powinna się zaczynać od 1. Ten sposób obliczeń jest lepszy niż inne metody, ponieważ mnożenie jest szybsze niż dzielenie i operacja modulo. Implementacja funkcji wyglądałaby tak:

```
void print_narcissistics()
{
   for (int a = 1; a <= 9; a++)
   {
      for (int b = 0; b <= 9; b++)
      {
```

```
            for (int c = 0; c <= 9; c++)
            {
               auto abc = a * 100 + b * 10 + c;
               auto arm = a * a * a + b * b * b + c * c * c;
               if (abc == arm)
               {
                  std::cout << arm << std::endl;
               }
            }
         }
      }
   }
```

Dodatkowym ćwiczeniem mogłoby być napisanie funkcji, która wyznacza liczby narcystyczne aż do podanego limitu, niezależnie od liczby jej cyfr. Taka funkcja działałaby wolniej, ponieważ najpierw musiałbyś określić zestaw cyfr, zapisać go w jakimś kontenerze, a następnie zsumować cyfry podniesione do odpowiedniej potęgi (równej liczbie cyfr).

## 9. Czynniki pierwsze liczby

Czynnikami pierwszymi dodatniej liczby całkowitej są liczby pierwsze, które dokładnie dzielą tę liczbę całkowitą. Na przykład czynniki pierwsze liczby 8 to $2 \cdot 2 \cdot 2$, a czynniki pierwsze liczby 42 to $2 \cdot 3 \cdot 7$. Aby określić czynniki pierwsze, należy zastosować następujący algorytm:

1. Gdy liczba `n` jest podzielna przez 2, oznacza to, że wartość 2 jest jej czynnikiem pierwszym i musi zostać dodana do listy, natomiast `n` powinna stać się równa `n / 2`. Po wykonaniu tego kroku `n` będzie liczbą nieparzystą.
2. Rozpocznij szereg iteracji, zaczynając od wartości 3 aż do pierwiastka kwadratowego z liczby `n`. Jeżeli bieżąca wartość (nazwijmy ją `i`) dzieli liczbę `n`, wówczas jest jej czynnikiem pierwszym i musi zostać dodana do listy, natomiast `n` powinna stać się równa `n / i`. Gdy `i` nie dzieli już `n`, zwiększ jej wartość o 2 (aby uzyskać następną liczbę nieparzystą).
3. Gdy `n` będzie liczbą pierwszą większą niż 2, powyższe kroki nie spowodują, że stanie się ona równa 1. Jeśli więc pod koniec kroku 2. liczba `n` jest wciąż większa niż 2, oznacza to, że jest czynnikiem pierwszym.

```
std::vector<unsigned long long> prime_factors(unsigned long long n)
{
   std::vector<unsigned long long> factors;
   while (n % 2 == 0)
   {
      factors.push_back(2);
      n = n / 2;
   }

   for (unsigned long long i = 3; i <= std::sqrt(n); i += 2)
   {
      while (n%i == 0)
      {
```

```
            factors.push_back(i);
            n = n / i;
         }
      }

      if (n > 2)
         factors.push_back(n);

      return factors;
   }

   int main()
   {
      setlocale(LC_ALL, "polish");

      unsigned long long number = 0;
      std::cout << "liczba:";
      std::cin >> number;

      auto factors = prime_factors(number);

      std::copy(
         std::begin(factors), std::end(factors),
         std::ostream_iterator<unsigned long long>(std::cout, " "));
   }
```

W ewentualnym następnym ćwiczeniu określ największy czynnik pierwszy dla liczby 600 851 475 143.

## 10. Kod Graya

Kod Graya, znany również pod nazwą kodu refleksyjnego lub odzwierciedlonego binarnie, jest formą kodowania binarnego, w którym dwie kolejne liczby różnią się od siebie tylko jednym bitem. Aby przeprowadzić kodowanie do kodu Graya, musimy użyć następującego wzoru:

```
if b[i-1] = 1 then g[i] = not b[i]
else g[i] = b[i]
```

Jest to równoznaczne następującemu zapisowi:

```
g = b xor (wartość b jeden raz logicznie przesunięta w prawo)
```

W celu zdekodowania kodu Graya używany jest poniższy wzór:

```
b[0] = g[0]
b[i] = g[i] xor b[i-1]
```

Dla liczb całkowitych nieujemnych można go zapisać w języku C++ w postaci poniższego programu:

```
unsigned int gray_encode(unsigned int const num)
{
   return num ^ (num >> 1);
}

unsigned int gray_decode(unsigned int gray)
{
   for (unsigned int bit = 1U << 31; bit > 1; bit >>= 1)
   {
      if (gray & bit) gray ^= bit >> 1;
   }
   return gray;
}
```

Aby wyświetlić wszystkie 5-bitowe liczby całkowite, ich binarną reprezentację, zakodowany kod Graya i zdekodowaną wartość, możemy wykorzystać poniższy listing:

```
std::string to_binary(unsigned int value, int const digits)
{
   return std::bitset<32>(value).to_string().substr(32-digits, digits);
}

int main()
{
   setlocale(LC_ALL, "polish");

   std::cout << "Liczba\tWart. binarna\tKod Graya\tWart. zdekodowana\n";
   std::cout << "------\t-------------\t----------\t-----------------\n";

   for (unsigned int n = 0; n < 32; ++n)
   {
      auto encg = gray_encode(n);
      auto decg = gray_decode(encg);

      std::cout
         << n << "\t" << to_binary(n, 5) << "\t\t"
         << to_binary(encg, 5) << "\t\t" << decg << "\n";
   }
}
```

# 11. Przekształcanie liczb arabskich na rzymskie

Liczby rzymskie w formie takiej, jaką znamy obecnie, używają siedmiu znaków: I = 1, V = 5, X = 10, L = 50, C = 100, D = 500 i M = 1000. System wykorzystuje dodawanie i odejmowanie w celu tworzenia symboli liczbowych. Symbole od 1 do 10 są takie: I, II, III, IV, V, VI, VII, VIII, IX i X. Rzymianie nie używali symbolu zera i w celu jego reprezentacji pisali słowo *nulla*. W tym systemie największe symbole znajdują się po lewej stronie, a najmniej znaczące — po prawej. Przykładowo, liczbą rzymską reprezentującą rok 1994 jest MCMXCIV. Jeśli nie znasz reguł dotyczących liczb rzymskich, poszukaj dodatkowych informacji w internecie.

Aby zdefiniować liczbę rzymską dla danej liczby, użyj następującego algorytmu:

1. Weź pod uwagę każdy podstawowy symbol liczb rzymskich, zaczynając od największego (M), a kończąc na najmniejszym (I).
2. Jeśli wartość bieżąca jest większa niż wartość symbolu, dodaj symbol do liczby rzymskiej i odejmij jego wartość od bieżącej.
3. Powtarzaj proces, aż wartość bieżąca będzie równa zero.

Jako przykładu użyjmy liczby 42: pierwszy rzymski symbol podstawowy mniejszy niż 42 to XL, który wynosi 40. Dodajemy go do docelowej liczby i otrzymujemy XL, a jednocześnie odejmujemy od bieżącej liczby, w wyniku czego uzyskujemy 2. Pierwszy rzymski symbol podstawowy mniejszy od 2 to I, czyli 1. Dodajemy I do liczby, w wyniku czego uzyskujemy XLI, a następnie odejmujemy 1 od bieżącej liczby, co daje nam 1. Dodajemy jeszcze jeden symbol I do liczby, która staje się równa XLII, i odejmujemy ponownie 1 od bieżącej liczby, osiągając 0, a zatem kończymy algorytm:

```
std::string to_roman(unsigned int value)
{
   std::vector<std::pair<unsigned int, char const*>> roman {
      { 1000, "M" },{ 900, "CM" }, { 500, "D" },{ 400, "CD" },
      { 100, "C" },{ 90, "XC" }, { 50, "L" },{ 40, "XL" },
      { 10, "X" },{ 9, "IX" }, { 5, "V" },{ 4, "IV" }, { 1, "I" }};

   std::string result;
   for (auto const & kvp : roman) {
      while (value >= kvp.first) {
         result += kvp.second;
         value -= kvp.first;
      }
   }
   return result;
}
```

Powyższa funkcja może zostać użyta w następujący sposób:

```
int main()
{
   for(int i = 1; i <= 100; ++i)
   {
      std::cout << i << "\t" << to_roman(i) << std::endl;
   }

   int number = 0;
   std::cout << "liczba:";
   std::cin >> number;

   std::cout << to_roman(number) << std::endl;
}
```

# 12. Najdłuższy ciąg Collatza

Problem Collatza, znany również jako problem Ulama, problem *3x + 1*, problem Kakutaniego lub problem syrakuzański, jest nieudowodnioną hipotezą, która stwierdza, że ciąg zdefiniowany w sposób, który opisano poniżej, zawsze uzyskuje wartość 1.

Definicja ciągu jest następująca: rozpocznij od dowolnej całkowitej liczby dodatniej `n` i uzyskaj każdy nowy element za pomocą poprzedniego: jeśli poprzedni składnik będzie parzysty, następny powinien być równy jego połowie — w przeciwnym razie musi zostać zdefiniowany jako 3 razy większy od poprzedniego oraz dodatkowo zwiększony o 1.

Zadanie, które musisz rozwiązać, polega na wygenerowaniu ciągów Collatza dla wszystkich dodatnich liczb całkowitych mniejszych od miliona, określeniu, który z nich jest najdłuższy, a następnie wyświetleniu jego długości oraz liczby początkowej, z której powstał. Chociaż w celu utworzenia ciągu dla każdej z liczb oraz wyznaczenia elementów pozwalających na osiągnięcie wartości 1 możemy zastosować metodę „na siłę" (ang. *brute-force*), szybszym rozwiązaniem będzie zapamiętanie długości wszystkich ciągów, które już zostały wygenerowane. Gdy bieżący element ciągu rozpoczynającego się od wartości `n` stanie się mniejszy od niej, oznacza to, że jest liczbą, dla której ciąg został już określony. Możemy więc po prostu pobrać jego zapamiętaną długość i dodać do bieżącej, aby określić długość ciągu rozpoczynającego się od `n`. Takie podejście ogranicza jednak maksymalną liczbę wyznaczanych ciągów Collatza, ponieważ w pewnym momencie zajętość pamięci podręcznej przekroczy ilość pamięci, którą system może przydzielić:

```
std::pair<unsigned long long, long> longest_collatz(
   unsigned long long const limit)
{
   long length = 0;
   unsigned long long number = 0;
   std::vector<int> cache(limit + 1, 0);
   for (unsigned long long i = 2; i <= limit; i++)
   {
      auto n = i;
      long steps = 0;
      while (n != 1 && n >= i)
      {
         if ((n % 2) == 0) n = n / 2;
         else n = n * 3 + 1;
         steps++;
      }
      cache[i] = steps + cache[n];

      if (cache[i] > length)
      {
         length = cache[i];
         number = i;
      }
   }

   return std::make_pair(number, length);
}
```

# 13. Wyznaczanie liczby π

Właściwym rozwiązaniem zadania przybliżonego określenia wartości $\pi$ jest użycie symulacji *Monte Carlo*. Jest to metoda wykorzystująca losowe próbki danych wejściowych do badania zachowania złożonych procesów lub systemów. Ta metoda ma wiele różnych zastosowań i jest wykorzystywana w rozmaitych obszarach, w tym w fizyce, inżynierii, informatyce, finansach czy biznesie.

Aby zaimplementować rozwiązanie, wykorzystujemy następujące założenia: powierzchnia koła o średnicy `d` wynosi `PI * d ^ 2 / 4`. Pole kwadratu o długości boków równych `d` wynosi `d ^ 2`. Jeśli podzielimy te dwa wzory przez siebie, otrzymamy `PI / 4`. Gdy umieścimy koło wewnątrz kwadratu, a następnie wygenerujemy losowe liczby rozmieszczone w nim równomiernie, ich liczba w kole powinna być wprost proporcjonalna do powierzchni koła, a liczba wewnątrz kwadratu powinna być wprost proporcjonalna do powierzchni kwadratu. Oznacza to, że podzielenie całkowitej liczby trafień zawartych w kwadracie i kole powinno dać wynik `PI / 4`. Im więcej punktów zostanie wygenerowanych, tym dokładniejszy będzie rezultat.

Do generowania liczb pseudolosowych użyjemy algorytmu *Mersenne Twister* i rozkładu jednostajnego ciągłego:

```
template <typename E = std::mt19937,
          typename D = std::uniform_real_distribution<>>
double compute_pi(E& engine, D& dist, int const samples = 1000000)
{
   auto hit = 0;
   for (auto i = 0; i < samples; i++)
   {
      auto x = dist(engine);
      auto y = dist(engine);
      if (y <= std::sqrt(1 - std::pow(x, 2))) hit += 1;
   }
   return 4.0 * hit / samples;
}

int main()
{
   std::random_device rd;
   auto seed_data = std::array<int, std::mt19937::state_size> {};
   std::generate(std::begin(seed_data), std::end(seed_data),
                 std::ref(rd));
   std::seed_seq seq(std::begin(seed_data), std::end(seed_data));
   auto eng = std::mt19937{ seq };
   auto dist = std::uniform_real_distribution<>{ 0, 1 };

   for (auto j = 0; j < 10; j++)
      std::cout << compute_pi(eng, dist) << std::endl;
}
```

# 14. Sprawdzanie numerów ISBN

**Międzynarodowy Znormalizowany Numer Książki** (ISBN) to unikatowy numeryczny identyfikator książek. Obecnie używany jest format 13-cyfrowy. W przypadku naszego zadania należy jednak zweryfikować poprzedni format, w którym używano 10 cyfr. Ostatnia z 10 cyfr to suma kontrolna. Ta cyfra musi być wybrana w taki sposób, by suma wszystkich 10 cyfr, z których każda została pomnożona przez swoją wagę (liczbę całkowitą) zmniejszającą się od 10 do 1, była wielokrotnością 11.

Przedstawiona poniżej funkcja `validate_isbn_10` używa numeru ISBN w postaci ciągu znaków i zwraca wartość `true`, jeśli długość łańcucha wynosi 10, wszystkie jego elementy są cyframi, a ich suma pomnożona przez odpowiednią wagę (lub pozycję) jest wielokrotnością 11:

```
bool validate_isbn_10(std::string_view isbn)
{
   auto valid = false;
   if (isbn.size() == 10 &&
       std::count_if(std::begin(isbn), std::end(isbn), isdigit) == 10)
   {
      auto w = 10;
      auto sum = std::accumulate(
         std::begin(isbn), std::end(isbn), 0,
         [&w](int const total, char const c) {
            return total + w-- * (c - '0'); });

     valid = !(sum % 11);
   }
   return valid;
}
```

Twoim kolejnym ćwiczeniem mogłoby być ulepszenie powyższej funkcji w taki sposób, aby poprawnie weryfikowała numery ISBN-10, które zawierają łączniki (na przykład 3-16-148410-0). Możesz również napisać funkcję, która sprawdza poprawność numerów ISBN-13.

# Skorowidz

## K



## L



## Ł



## N



## O



## P



## S

## Ś



## T



## U



## W



## Z

## Oto C++. Podejmiesz wyzwanie i napiszesz kod!

C++ jest dojrzałym językiem programowania, od wielu lat wykorzystywanym przez profesjonalnych programistów do wielu różnych zastosowań, włączając w to pisanie gier, programowanie GUI czy tworzenie złożonych aplikacji użytkowych. Zaprojektowano go pod kątem maksymalizowania wydajności, jest więc najczęściej wybieranym językiem w sytuacjach, w których najważniejsza jest efektywność działania kodu. Aby jednak wykorzystać te zalety C++, trzeba *nauczyć się* nim posługiwać. A biegłość w posługiwaniu się językiem programowania przychodzi dzięki regularnym ćwiczeniom i ciągłemu testowaniu nabytych umiejętności. Innymi słowy, trzeba rozwiązywać jak najwięcej różnorodnych, rzeczywistych i praktycznych zadań problemowych.

W tej książce zawarto zestaw 100 zadań o różnym poziomie trudności, ułożonych w taki sposób, aby podczas rozwiązywania móc skorzystać z bogactwa standardowej biblioteki C++ oraz z wielu zewnętrznych bibliotek wieloplatformowych. Zadania rozmieszczono w 12 rozdziałach, z których każdy dotyczy określonego tematu. Są to problemy, których odpowiednie rozwiązanie warunkuje poprawne działanie aplikacji, takie jak bezpieczna komunikacja, szyfrowanie i autoryzacja danych, korzystanie z wątków i funkcji asynchronicznych czy implementacja algorytmów współbieżnych. Proponowane zagadnienia zostały dokładnie opisane, uwzględniono również szereg zaleceń, wyjaśnień i wskazówek. Na wypadek gdyby wykonanie któregoś zadania sprawiało trudności, do książki dołączono kod źródłowy przykładowych rozwiązań.

**W tej książce między innymi:**

- serializacja i deserializacja danych JSON i XML
- praca z bazą danych SQLite
- implementacja takich struktur jak bufor cykliczny i kolejka priorytetowa
- usługi REST i HTTP
- wzorce projektowe w rozwiązywaniu problemów

**Marius Bancila** — urodziła się w Kijowie. Dorastała w rodzinie fizyków, naukowców i profesorów. Ma duże doświadczenie jako programistka aplikacji WWW, szczególnie w zakresie języka JavaScript. Równocześnie od wielu lat rozwija swoje talenty dydaktyczne w dziedzinie technologii internetowych. Obecnie mieszka w Berlinie, gdzie pracuje jako lider zespołu w firmie Meetrics.



ISBN 978-83-283-5211-7
9 788328 352117
Cena: 57,00 zł

Packt