Podstawy programowania w języku C++

Tomasz Tarkowski

Centrum Dydaktyczne Wydziału Fizyki
Wydział Fizyki Uniwersytetu Warszawskiego
Pasteura 5, 02-093 Warszawa

18 stycznia 2025

Spis treści

1 Wstęp do programowania

1.1 Fizyka, informatyka, programowanie

Można w przybliżeniu uznać, że informatyka ma się tak do programowania, jak fizyka do inżynierii mechanicznej. Fizyka jest nauką ścisłą, która opisuje świat z użyciem języka jakim jest matematyka. Inżynieria mechaniczna zaś wykorzystuje prawa fizyki do opisu i konstrukcji urządzeń. Tak samo informatyka jest nauką ścisłą a programowania wykorzystuje jej osiągnięcia do tworzenia programów uruchamianych na urządzeniach elektronicznych. *Informatyka jest nauką o przetwarzaniu informacji w sposób automatyczny.*

Przybliżenia potrafią jednak wprowadzać w błąd. Informatykę i fizykę łączy więcej niż to widać na pierwszy rzut oka. I nie chodzi tu o wspólną ulicę pomiędzy Wydziałem Fizyki a Wydziałem Matematyki, Informatyki i Mechaniki UW. (A przynajmniej nie chodzi tylko o to.) Dwoma przykładami związków informatyki z fizyką są pojęcie "it from bit" (John Archibald Wheeler, 1989) oraz prace z zakresu teorii informacji i mechaniki statystycznej (Edwin Thompson Jaynes, 1957). Poza tym, fizyka obliczeniowa – trzecia gałąź fizyki poza teorią i doświadczeniem – jest częścią nauk fizycznych, która czerpie z informatyki. I tak jak językiem fizyki jest matematyka, tak językiem fizyki obliczeniowej – informatyka. A narzędziem fizyki obliczeniowej jest programowanie – służące do wykonywania obliczeń użytecznych z punktu widzenia fizyki. Stąd też można wywnioskować obecność kursu programowania (a dalej też przedmiotu poświęconego metodom numerycznym) w standardowej ścieżce nauczania fizyki.

1.2 Algorytm. Maszyna Turinga

Pojęcie algorytmu ma centralne miejsce w informatyce. Algorytm to skończony ciąg ściśle określonych instrukcji, zazwyczaj używanych do rozwiązania jakiegoś problemu lub wykonania obliczeń. W szerszym sensie można uznać przykłady z dnia codziennego, takie jak przyrządzanie potraw, robienie zakupów, wyjście do kina, za przykłady algorytmów. W ścisłym sensie poprzednie przykłady nie są algorytmami, ponieważ nie zawierają w sobie bezpośrednio obliczeń czy struktur danych i nie są wyrażone w języku programowania. Za przykłady algorytmów w sensie ścisłym można natomiast uznać sortowanie skończonego ciągu liczb, dekodowanie strumienia wideo H.264 (MPEG-4) czy też metodę gradientu sprzężonego stosowanej np. obliczeniowej fizyce materiałowej. Zastanówmy się, gdzie algorytmy mogą być wykonywane. Aby odpowiedzieć na to pytanie, zdefiniujmy wpierw *maszynę Turinga*.

Maszyna Turinga została zaproponowana, jak nazwa sama wskazuje, przez Alana Turinga a stało się to w 1936. Maszyna jest modelem obecnie stosowanego komputera. Ponieważ *wszystko powinno być tak proste, jak to tylko możliwe, ale nie prostsze*, dlatego maszyna Turinga składa się tylko z trzech elementów: z nieskończonej taśmy, głowicy i mechanizmu sterującego. (Czyż nie ma nic prostszego aniżeli nieskończona taśma?)

- Taśma jest podzielona na pola (komórki). Na każdym polu może zostać zapisany symbol w języku maszyny.
- Głowica służy do odczytywania i zapisywania symboli. Może ona się znaleźć nad dowolnym polem taśmy.
- Mechanizm sterujący jest elementem maszyny, który decyduje o jej działaniu. Może on się znaleźć w danej chwili czasu w jednym ze skończenie wielu stanów.

Mechanizm sterujący podejmuje decyzje do wykonania przez maszynę Turinga. Decyzja jest podejmowana na podstawie bieżącego stanu mechanizmu a także symbolu znajdującego się w polu pod głowicą. Taką decyzją może być:

- przesunięcie głowicy względem taśmy;
- zmiana zawartości pola taśmy;
- zmiana stanu mechanizmu.

Jak widać maszyna Turinga jest środowiskiem do wykonywania algorytmów. Jest też ona znacznie prostsza w opisie aniżeli dowolny fizyczny komputer, dlatego znacznie łatwiej jest dokonywać rozumowania na temat wykonywania algorytmów. Może to być zastosowane w opisie *złożoności obliczeniowej algorytmu*.[1]

Warto przy okazji zauważyć, że komputery fizyczne mają formalnie mniejsze możliwości wykonywania programów aniżeli maszyna Turinga, ponieważ ta ostatnia ma nieskończenie wiele miejsca na taśmie. W granicy odpowienio małych programów komputerowych ta różnica nie ma jednak znaczenia.

1.3 Hipoteza Churcha-Turinga

Hipoteza sformułowana przez Alonzo Churcha oraz Alana Turinga głosi, że każdy problem intuicyjnie uznawany za obliczalny, jest obliczalny przez maszynę Turinga. Chociaż hipoteza Churcha-Turinga nie została udowodniona z powodu braku precyzji określenia „*intuicyjnie uznawany* za obliczalny", to jednak żaden znany model obliczeń nie jest w stanie wyrazić więcej aniżeli maszyna Turinga.

[1] Maciej Ślusarek i in. *Złożoność obliczeniowa*. URL: https://wazniak.mimuw.edu.pl/.

1.4 Języki programowania

Język programowania $\mathcal{L}$ jest systemem notacji umożliwiającym zapisywanie programów komputerowych. Co można zrobić z językiem programowania? Oczywiście można w nim pisać programy! Aby to sformalizować wprowadźmy następującą definicję: funkcja częściowa $f\colon X \to Y$ to funkcja $f\colon X' \to Y$ dla $X' \subseteq X$. Oznacza to, że funkcja częściowa jest uogólnieniem pojęcia funkcji, gdzie nie dla każdego argumentu musi nastąpić odwzorowanie, tzn. mogą istnieć takie $x \in X$ dla których $f(x)$ jest nieokreślone. Korzystając z tego pojęcia można zdefiniować czym jest program (komputerowy).

Jeśli $\mathcal{D}$ jest zbiorem danych, to program można uznać jako następującą funkcję częściową:

$$\mathcal{P}^{\mathcal{L}}\colon \mathcal{D} \to \mathcal{D} \tag{1}$$

taką, że:

$$\mathcal{P}^{\mathcal{L}}(\text{dane wejściowe}) = \text{dane wyjściowe} \tag{2}$$

Program $\mathcal{P}^{\mathcal{L}}$ może nigdy nie kończyć się dla niektórych danych wejściowych, dlatego też jest on określany jako funkcja częściowa – ma to związek z problemem stopu. Zbiór wszystkich programów w języku $\mathcal{L}$ można oznaczyć jako $\text{Prog}^{\mathcal{L}}$.

Gdzie można wykonać wyżej określony program? Do tego celu służy *maszyną abstrakcyjna* $\mathcal{M}_{\mathcal{L}}$ dla języka $\mathcal{L}$, która jest zbiorem algorytmów i struktur umożliwiających przechowywanie i wykonywanie programów napisanych w języku $\mathcal{L}$. Najbardziej podstawowy przykład maszyny abstrakcyjnej już znamy – jest nią wspomniana wcześniej maszyna Turinga. Dla maszyny abstrakcyjnej $\mathcal{M}_{\mathcal{L}}$ język $\mathcal{L}$ jest *zrozumiały* przez interpreter tejże maszyny i jest nazywany *językiem maszynowym*. Dla maszyny Turinga językiem maszynowym są pewne napisy nad alfabetem złożonym z symboli maszyny Turinga. Współczesny komputer fizyczny jest również przykładem maszyny abstrakcyjnej i dla niego językiem maszynowym są pewne napisy nad alfabetem zero-jedynkowym związane z brakiem lub przepływem prądu w procesorze komputera. Napisy te są równoważne semantycznie (znaczeniowo) z językiem asemblera danej architektury komputera.

1.5 Języki interpretowane i kompilowane

Implementacja danego języka programowania może mieć formę *interpretowaną* lub *kompilowaną*. Niech $\mathcal{M}_{0\mathcal{L}_0}$ oznacza maszynę abstrakcyjną z językiem maszynowym $\mathcal{L}_0$. Poprzez czysto interpretowaną implementację języka programowania $\mathcal{L}$ dla maszyny abstrakcyjnej $\mathcal{M}_{0\mathcal{L}_0}$ rozumie się tzw. interpreter, który może być zdefiniowany jako funkcja częściowa:

$$\mathcal{I}^{\mathcal{L}_0}_{\mathcal{L}}\colon \text{Prog}^{\mathcal{L}} \times \mathcal{D} \to \mathcal{D} \tag{3}$$

która spełnia warunek:

$$\mathcal{I}^{\mathcal{L}_0}_{\mathcal{L}}(\mathcal{P}^{\mathcal{L}}, \text{dane wejściowe}) = \mathcal{P}^{\mathcal{L}}(\text{dane wejściowe}) \tag{4}$$

Natomiast poprzez czysto kompilowaną implementacją języka programowania $\mathcal{L}$ dla maszyny abstrakcyjnej $\mathcal{M}_{0\mathcal{L}_0}$ rozumie się tzw. kompilator, który może być zdefiniowany jako funkcja częściowa:

$$\mathcal{C}^{\mathcal{L}_0}_{\mathcal{L}}\colon \text{Prog}^{\mathcal{L}} \to \text{Prog}^{\mathcal{L}_0} \tag{5}$$

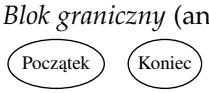
która spełnia warunek:

$$\mathcal{C}^{\mathcal{L}_0}_{\mathcal{L}}(\mathcal{P}^{\mathcal{L}})(\text{dane wejściowe}) = \mathcal{P}^{\mathcal{L}}(\text{dane wejściowe}) \tag{6}$$

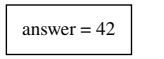
1.6 Schematy blokowe

W celu zapisu algorytmu można wykorzystać *pseudokod*, będący czymś pośrednim między potencjalnie nieprecyzyjnym językiem naturalnym a precyzyjnym językiem programowania, lub *schemat blokowy*, będący graficzną wizualizacją wykonania algorytmu. Niektórzy lubują się w schematach blokowych. Ja za nimi nie przepadam (uważam to za kwestię gustu) – jednak, aby nie ograniczać wiedzy Czytelnika warto chociaż zdawkowo o nich wspomnieć. W tym podpunkcie zostaną zatem przedstawione elementy notacji z użyciem schematów blokowych.

- *Blok graniczny* (ang. *terminal*), nazywany również *etykietą*, służy do określania początku lub końca algorytmu.

- *Blok operacyjny* (ang. *process*) oznacza zestaw operacji zmieniających dane, np. przypisanie wartości do zmiennej. (Nazwa zmiennej w poniższym rysunku została celowo pozostawiona w języku angielskim.)

- *Predykat* lub *decyzja* (ang. *decision*) polega na sprawdzeniu czy dany warunek jest spełniony i wyborze jednej z dwóch możliwości.

- *Blok wejścia/wyjścia* (ang. *input/output*) reprezentuje operacje wejścia (np. wczytanie danych z klawiatury) oraz wyjścia (np. wypisanie danych na ekran).

- *Łącznik* i *strzałka* (ang. *flowline*, *arrowhead*) są wykorzystywane do wskazania kolejności wykonywanych w algorytmie operacji. Standardowo korzysta się z łącznika gdy przepływ jest naturalny (od góry do dołu oraz z lewej do prawej). W przeciwnym razie należy użyć strzałki.

1.7 Notacja BNF

Notacja BNF (ang. *Backus-Naur form*) jest w informatyce używana m.in. do opisu składni języków programowania (jest więc tzw. *metaskładnią*). Polega ona na wyspecyfikowaniu tzw. *reguł produkcji*:

```
<symbol> ::= __wyrazenie__
```

gdzie *<symbol>* jest *zmienną nieterminalną*, znak *::=* oznacza, że *<symbol>* zostanie zastąpiony przez *__wyrazenie__*, *__wyrazenie__* składa się z sekwencji symboli terminalnych i nieterminalnych. Każda sekwencja w wyrażeniu *__wyrazenie__* jest odseparowana od następnej symbolem *|* oznaczającym wybór. Symbole *<*, *>*, *::=* oraz *|* to symbole metajęzyka. Pozostałe symbole są znakami lub symbolami definiowanego języka.

Jako przykład notacji BNF można podać sposób zwykłego zapisu liczb całkowitych.

```
<minus> ::= −
<zero> ::= 0
<cyfra niezerowa> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<cyfra> ::= <zero> | <cyfra niezerowa>
<sekwencja cyfr> ::= <cyfra> | <cyfra><sekwencja cyfr>
<liczba nieujemna> ::= <cyfra> | <cyfra niezerowa><sekwencja cyfr>
<liczba ujemna> :: <minus><cyfra niezerowa> | <minus><cyfra niezerowa><cyfra>
<liczba calkowita> ::= <liczba nieujemna> | <liczba ujemna>
```

Symbole *−*, *0*, *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8* oraz *9* są terminalne. Pozostałe symbole, tzn. *<minus>*, *<zero>*, *<cyfra niezerowa>*, *<cyfra>*, *<sekwencja cyfr>*, *<liczba nieujemna>*, *<liczba ujemna>* oraz *<liczba calkowita>*, to symbole nieterminalne.

Notacja BNF, choć bardzo użyteczna, ma pewne wady. Nie ma chociażby możliwości wyrażenia wprost powtórzeń (należy użyć rekurencji, por. *<sekwencja cyfr>*) oraz symboli opcjonalnych (trzeba w tym celu użyć symboli pośrednich). Dlatego też, w celu zaradzenia tym niedogodnościom, stworzono rozszerzenia dla BNF. Jedym z nich jest *EBNF* (ang. *extended Backus-Naur form*).

2 Przykładowe style programowania

2.1 Różne języki – różne style

Zanim przejdziemy do programowania *stricte* w C++ przyjrzymy się różnym językom programowania, które oferują różne *style*. Prawdą jest, że jeden język programowania może wspierać więcej niż jeden styl programowania. Na przykład C++ wspiera styl imperatywny, proceduralny, funkcyjny, obiektowy i generyczny. Prawdą jest też, że istnieją języki programowania, w których programuje się w jednym stylu. Na przykład Haskell jest językiem czysto funkcyjnym. Warto to wszystko teraz zobaczyć z wysokiej perspektywy, bez wdawania się w szczegóły.

2.2 Imperatywny styl programowania

W imperatywnym stylu programowania sterowanie programem jest zaprezentowane wprost w kodzie źródłowym a stan programu (np. wartości zmiennych) ulegają zmianom. Zobaczmy zastosowanie stylu imperatywnego na przykładzie napisanym w asemblerze.

Kod źródłowy 1: Asembler x86-64 (NASM)

```
;; Styl imperatywny w asemblerze

SYS_EXIT equ 60

section bss

section data
```

```
8
9   section .text
10          global _start
11
12  _start:
13          mov rcx, 1000000000
14
15  loop:
16          dec rcx ; Dekrementacja rejestru rcx (licznika petli).
17          jnz loop ; Skok do etykiety rozpoczynajacej petle.
18          ;; (Powyzsze dwie instrukcje mozna skrocic za pomoca loop.)
19
20  exit:
21          mov eax, SYS_EXIT
22          xor edi, edi
23          syscall
```

Nawet jeśli asembler nie jest najbardziej popularnym rozwiązaniem to niewątpliwie imperatywny styl programowania jest jednym z najczęściej spotykanych. Jest szeroko stosowany w różnych językach programowania i jest pierwszym wyborem praktycznie każdego początkującego programisty.

2.3 Strukturalny/proceduralny styl programowania

W strukturalnym (nazywanym też proceduralnym) stylu programowania wciąż stosuje się techniki imperatywne. Są one jednak wzbogacone o bardziej zaawansowane środki kontroli przepływu dzięki zastosowaniu instrukcji warunkowych, pętli oraz procedur (funkcji). Oznacza to unikanie instrukcji typu *goto*. Przykład strukturalnego stylu programowania został zaprezentowany w języku Python.

Kod źródłowy 2: Python 3

```
1   import numpy as np
2   import matplotlib.pyplot as p
3
4   # Transformacja przeksztalcajaca wektor [x, y] na nowy.
5   def transformation(c, x, y):
6       x_new = c[0] * x[-1] + c[1] * y[-1]
7       y_new = c[2] * x[-1] + c[3] * y[-1] + c[4]
8       x.append(x_new)
9       y.append(y_new)
10
11  # Paproc Barnsley'a (fraktal).
12  def Barnsley(x, y):
13      cs = [[ 0.00, 0.00, 0.00, 0.16, 0.00], \
14            [ 0.85, 0.04, -0.04, 0.85, 1.60], \
15            [ 0.20, -0.26, 0.23, 0.22, 1.60], \
16            [-0.15, 0.28, 0.26, 0.24, 0.44]]
17      r = np.random.uniform(0,1)
18      if r < 0.01:
19          transformation(cs[0], x, y)
20      elif r < 0.01 + 0.85:
21          transformation(cs[1], x, y)
22      elif r < 0.01 + 0.85 + 0.07:
23          transformation(cs[2], x, y)
24      else:
25          transformation(cs[3], x, y)
26
27  # Rysowanie fraktala.
28  def fractal(x, y, name):
29      figure = p.figure()
30      ax = p.gca()
31      ax.set_facecolor('black')
32      p.scatter(x, y, s=4, lw=0, marker='.')
33      figure.savefig(name)
34
35  # Tutaj rozpoczynaja sie obliczenia i rysowanie.
36  x, y = [0], [0]
37  for i in range(50000):
38      Barnsley(x, y)
```

```
fractal(x, y, "Barnsley")
```

2.4 Obiektowy styl programowania

W obiektowym stylu programowania procedury są zastąpione metodami klas. Działanie programu polega m.in. na tworzeniu obiektów i wykonywaniu operacji na nich. Dane są kapsułkowane, tzn. ukrywane przed „światem zewnętrznym" w obiektach, a same operacje są wykonywane jedynie poprzez ściśle określone interfejsy. Przykład stylu obiektowego został podany w języku C++.

Kod źródłowy 3: C++

```
#include <cassert>

class stack {
  struct item {
    int value;
    item* lower;
  };

public:
  stack() : top_{nullptr} {}

  void push(int val) {
    item* aux = top_;
    top_ = new item;
    top_->value = val;
    top_->lower = aux;
  }

  void pop() {
    if (top_) {
      item* aux = top_->lower;
      delete top_;
      top_ = aux;
    }
  }

  int top() const {
    assert(top_);
    return top_->value;
  }

  bool is_empty() const {
    return top_ == nullptr;
  }

  ~stack() {
    while (!this->is_empty()) {
      this->pop();
    }
  }

private:
  item* top_;
};

int main() {
  stack s{};
  assert(s.is_empty());

  for (int i = 0; i < 5; ++i) {
    s.push(i);
  }
  assert(!s.is_empty());
  assert(s.top() == 4);

  s.pop();
```

```
    assert(s.top() == 3);
}
```

2.5 Generyczny styl programowania (metaprogramowanie)

W generycznym stylu programowania implementuje się algorytmy w sposób uogólniony, zapominając niejako o konkretnych reprezentacjach typów danych. W ten sposób tworzy się szablony, które później mogą być zastosowane do dowolnych typów danych spełniających konkretne wymogi. Przykład stylu generycznego został napisany w języku Object Pascal.

Kod źródłowy 4: Object Pascal

```
program SortExample;

type
  generic BubbleSort<T> = class
    class procedure sort(var a: array of T);
  end;

  generic PrintArray<T> = class
    class procedure print(var a: array of T);
  end;

  BubbleSortInteger = specialize BubbleSort<integer>;
  PrintArrayInteger = specialize PrintArray<integer>;

  class procedure BubbleSort.sort(var a: array of T);
  var
    n, newn, i : integer;
    temp : T;
  begin
    n := high(a);
    repeat
      newn := low(a);
      for i := low(a) + 1 to n do
      begin
        if a[i - 1] > a[i] then
        begin
          temp := a[i - 1];
          a[i - 1] := a[i];
          a[i] := temp;
          newn := i;
        end;
      end;
    n := newn;
    until n = low(a);
  end;

  class procedure PrintArray.print(var a : array of T);
  var
    i : integer;
  begin
    for i := low(a) to high(a) do
    begin
      write(a[i]);
      write(' ');
    end;
  end;

procedure draw(lo, hi: integer; var a: array of integer);
var
  i : integer;
begin
  for i := low(a) to high(a) do
  begin
    a[i] := Random(10); //Random(hi - lo) + lo;
  end;
```

```
56  end;
57
58  var
59    arr : array[0..9] of integer;
60  begin
61    draw(low(arr), high(arr) + 1, arr);
62    write('Before: ');
63    PrintArrayInteger.print(arr);
64    writeln('');
65
66    BubbleSortInteger.sort(arr);
67    write('After: ');
68    PrintArrayInteger.print(arr);
69    writeln('');
70  end.
```

2.6 Funkcyjny styl programowania

Programowanie w stylu funkcyjnym polega na operowaniu na funkcjach wyższego rzędu (tzn. funkcjach, które jako swoje argumenty mogą przyjmować inne funkcje a także zwracać funkcje jako swój rezultat). Przykład stylu funkcyjnego można zobaczyć w kodzie podanym w języku Scheme.

Kod źródłowy 5: Scheme

```
1  (define true (lambda (x y) (x)))
2  (define false (lambda (x y) (y)))
3  (define if_then_else (lambda (a b c) (a b c)))
4
5  (define f0 (lambda () (write 'f0)))
6  (define f1 (lambda () (write 'f1)))
7
8  (if_then_else false f0 f1)
9  (newline)
10 (exit)
```

2.7 Programowanie w logice

Ostatnim stylem programowania do omówienia na ten moment jest programowanie w logice. Jest to jeden z najmniej popularnych styli programowania, ale ma istotne znaczenie dla rozwoju informatyki. Opiera się on na tzw. klauzulach Horna, przestrzeni Herbranda i teorii unifikacji. Programy napisane w tym stylu mogą służyć do natychmiastowego rozwiązywania zagadek logicznych. Rozwiążmy zatem słynną zagadkę opublikowaną w pewnym amerykańskim magazynie.

1. *Jest pięć domów.*
2. *Anglik mieszka w czerwonym domu.*
3. *Hiszpan posiada psa.*
4. *Kawa jest pijana w zielonym domu.*
5. *Ukrainiec pija herbatę.*
6. *Zielony dom znajduje się bezpośrednio na prawo od domu z kości słoniowej.*
7. *Palacz Old Gold posiada ślimaki.*
8. *Kools jest palony w żółtym domu.*
9. *Mleko jest pijane w środkowym domu.*
10. *Norweg mieszka w pierwszym domu.*
11. *Mężczyzna, który pali Chesterfieldy, mieszka w domu obok mężczyzny z lisem.*
12. *Kools jest palone w domu obok domu, gdzie trzymany jest koń.*
13. *Palacz Lucky Strike pija sok pomarańczowy.*
14. *Japończyk pali Parliaments.*
15. *Norweg mieszka obok niebieskiego domu.*

Teraz, kto pije wodę? Kto posiada zebrę?
Dla przejrzystości należy dodać, że każdy z pięciu domów jest pomalowany na inny kolor, a ich mieszkańcy mają odmienne narodowości, posiadają inne zwierzęta, piją inne napoje i palą amerykańskie papierosy różnych marek [sic!]. Jeszcze jedno: w zdaniu 6 „prawo" oznacza Twoje prawo.

— "Life International", 17 grudnia 1962

Powyższe zadanie można rozwiązać stosując programowanie w logice w języku Prolog – tak jak pokazano to na poniższym przykładzie. (Brak wielkich liter wynika z wymogów języka Prolog.)

Kod źródłowy 6: Prolog

```
% Dom jest okreslony przez: kolor budynku, narodowosc mieszkanca, pijany napoj,
% palone papierosy oraz posiadane zwierze.
% Uwaga: Brak wielkich liter wynika z wymogow jezyka Prolog.

% Rozwiazanie mozna otrzymac wpisujac po znaku zachety intepretera Prolog
% nastepujacych polecen:
% zebra_owner(Owner).
% water_drinker(Drinker).
% houses(Houses).

houses(Hs) :- length(Hs, 5),
  member(h(czerwony, anglik, _, _, _), Hs),
  member(h(_, hiszpan, _, _, pies), Hs),
  member(h(zielony, _, kawa, _, _), Hs),
  member(h(_, ukrainiec, herbata, _, _), Hs),
  right(h(zielony, _, _, _, _), h(kosc_sloniowa, _, _, _, _), Hs),
  member(h(_, _, _, old_gold, slimaki), Hs),
  member(h(zolty, _, _, kool, _), Hs),
  Hs = [_, _,h(_, _, mleko, _, _), _, _],
  Hs = [h(_, norweg, _, _, _)|_],
  next(h(_, _, _, _, lis), h(_, _, _, chesterfield, _), Hs),
  next(h(_, _, _, kool, _), h(_, _, _, _, kon), Hs),
  member(h(_, _, sok_pomaranczowy, lucky_strike, _), Hs),
  member(h(_, japonczyk, _, parliament, _), Hs),
  next(h(_, norweg, _, _, _), h(niebieski, _, _, _, _), Hs),
  member(h(_, _, _, _, zebra), Hs), member(h(_, _, woda, _, _), Hs).
zebra_owner(Owner) :- houses(Hs), member(h(_, Owner, _, _, zebra), Hs).
water_drinker(Drinker) :- houses(Hs), member(h(_, Drinker, woda, _, _), Hs).

% Definicje pomocnicze:
next(A, B, Ls) :- append(_, [A, B | _], Ls).
next(A, B, Ls) :- append(_, [B, A | _], Ls).

right(A, B, Ls) :- append(_, [B, A | _], Ls).
```

2.8 Generacje języków programowania

Języki programowania można klasyfikować na różne sposoby.[2] Jednym z nich jest podział na tzw. *generacje*. Współczesna klasyfikacja generacji języków programowania składa się z pięciu pozycji.

- Język programowania 1 generacji (1GL, ang. *1st generation language*) to język maszynowy (zob. definicję wyżej) danego komputera. Komputer jest wtedy programowany bezpośrednio w swoim własnym języku bez potrzeby użycia procedury kompilacji/interpretacji. W przeszłości programowanie odbywało się poprzez użycie odpowiednich interfejsów sprzętowych. Obecnie 1GL jest stosowane np. do programowania sterowników obsługujących sprzęt.
- Język programowania 2 generacji (2GL) to język asemblera. Język taki jest podobny do 1GL z tą istotną różnicą, że zamiast ciągów binarnych stosuje się mnemoniki, tzn. skróty od wyrazów języka naturalnego (np. 'mov' lub 'dec'), które z jednej strony ułatwiają pisanie i rozumienie kodów źródłowych, a z drugiej wymagają zastosowania procedury asemblacji (kompilacji z kodu 2GL na kod 1GL). Przykładem jest asembler x86-64.
- Język programowania 3 generacji (3GL) to język niezależny od danej architektury sprzętowej znacznie ułatwiający tworzenie programów komputerowych. Język taki jest znacznie bardziej abstrakcyjny aniżeli 2G i ukrywa szczegóły danej architektury przed programistą. Przykładami są C++, Object Pascal, Python, Scheme.
- Język programowania 4 generacji (4GL) jest obecnie różnie definiowany. Można go rozumieć jako język, który operuje na zbiorach danych zamiast na bajtach, albo jako język dziedzinowy (mający wąskie zastosowanie aplikacyjne). Przykładem jest R.
- Język programowania 5 generacji (5GL) to język, w którym programista, aby rozwiązać problem, nie implementuje algorytmu a jedynie specyfikuje ograniczenia czego przykładem jest wyżej opisane rozwiązanie zagadki z zebrą. Przykładem takiego języka jest więc Prolog.

[2] Ray Toal. *Classifying Programming Languages*. URL: https://cs.lmu.edu/~ray/notes/pltypes/.

3 Język C++ – wprowadzenie

3.1 Historia

Językiem programowania używanym w dalszej części książki będzie C++. Język ten łączy długą historię[3] z nowoczesnymi technikami. Praca nad C++, ówcześnie jeszcze nazywanym „C z klasami", rozpoczęła się w 1979. Wtedy to duński informatyk Bjarne Stroustrup mierzył się w New Jersey (Stany Zjednoczone) z pewnym problemem z zakresu sieci komputerowych i systemów operacyjnych a nowy język miał być właśnie rozwiązaniem tego problemu. Stworzony język okazał się być rozwiązaniem wszechstronnym i z czasem zaczął być coraz bardziej popularny. W 1979 opracowana została pierwsza implementacja nowego języka programowania, tzn. C z klasami. W 1982 opublikowano pierwszy podręcznik użytkownika dla nowego języka. W 1985 ukazały się program Cfront (program służący do translacji kodu nowego języka na język programowania C) oraz pierwsze wydanie książki "The C++ Programming Language" (TC++PL). W 1989 program Cfront został wydany w wersji 2.0. W 1990 opublikowano "The Annotated C++ Reference Manual". W 1991 ukazały się program Cfront w wersji 3.0 oraz drugie wydanie TC++PL. W 1988 C++ miał około 15 tysięcy użytkowników i wtedy pojawił się pomysł na jego standardyzację. Od tego czasu wydano standardy w latach 1998 (z nieznaczną korektą w 2003), 2011, 2014, 2017, 2020, oraz 2023. W dalszej części książki będzie wykorzystywany standard z 2020, tzn. tzw. C++20.

3.2 Właściwości

C++ jest językiem trzeciej generacji wspierającym imperatywny, proceduralny, funkcyjny, obiektowy i generyczny styl programowania. C++ wywodzi się z dwóch tradycji programistycznych, które bardzo dobrze opisują czym jest i czym powinien pozostać język C++ mimo swojej ciągłej ewolucji. Pierwsza tradycja jest powiązana z językiem C, BCPL oraz asemblerem. Wniosła ona wysoką wydajność kodu wykonywalnego poprzez *bezpośrednie odwzorowanie w sprzęcie*. Druga tradycja jest związana z!językiem Simula. Wprowadziła ona *abstrakcję bez narzutu*.

3.3 Źródła wiedzy o języku

Językowi C++ jest poświęconych wiele materiałów edukacyjnych. Ważnym uzupełnieniem kursu języka C++ jest serwis https://cppreference.com[4] oraz materiały wideo z konferencji CppCon.[5] Ponadto warto wspomnieć, że istnieje dokument opisujący standard języka. Standard jest zasadniczo przydatny dla twórców języka lub twórców kompilatorów – niezwykle ciężko jest się z niego uczyć programowania od podstaw. Anglojęzyczne sugestie dotyczące materiałów w postaci książek do przeczytania można znaleźć na stronach popularnego serwisu dla programistów.[6] Twórca języka, Bjarne Stroustrup,[7] napisał również książkę na temat C++ przeznaczoną początkującym programistom[8] (jedna ze starszych edycji posiada polskie tłumaczenie). Znanymi ekspertami w zakresie języka C++ są Herb Sutter[9] oraz, nieaktywny już w tej dyscyplinie, Scott Meyers,[10] których artykuły oraz nagrania są dostępne w sieci. Zdecydowanie warto również zerknąć na materiały Tomasza Wernera.[11] Istnieją również książki poświęcone C++, które każdy może edytować.[12]

4 Instrukcje

4.1 Czym są instrukcje?

Można stwierdzić nieprecyzyjnie, że *instrukcja*[13] programu napisanego w języku C++ jest *fragmentem kodu źródłowego, który po kompilacji wykonuje się w sekwencji*. Oznacza to, że jeśli fragment kodu źródłowego *instrukcja_B* następuje po fragmencie *instrukcja_A*, to w momencie działania programu wykonuje się wpierw operacja *A* a następnie operacja *B*. W języku C++ istnieje 9 typów instrukcji – ich nazwy to *instrukcja*:

1. *wyrażenia* (ang. *expression statement*);
2. *złożona* (ang. *compound statement*);
3. *wyboru* (ang. *selection statement*);

[3] Bjarne Stroustrup. "A History of C++: 1979–1991". W: *History of Programming Languages—II* (1996), s. 699. DOI: 10.1145/234286.1057836; URL: https://en.cppreference.com/w/cpp/language/history.

[4] URL: https://cppreference.com/.

[5] URL: https://cppcon.org/; URL: https://www.youtube.com/CppCon.

[6] *The Definitive C++ Book Guide and List*. URL: https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list/.

[7] Bjarne Stroustrup. *Bjarne Stroustrup's homepage*. URL: https://stroustrup.com/.

[8] Bjarne Stroustrup. *Programming—Principles and Practice Using C++*. 3rd Edition. https://stroustrup.com/programming.html. Addison-Wesley, 2024.

[9] Herb Sutter. *Sutter's Mill. Herb Sutter on software development*. URL: https://herbsutter.com/.

[10] Scott Meyers. URL: https://www.aristeia.com/.

[11] Tomasz R. Werner. *Programowanie 2. Język C++*. URL: https://www.fuw.edu.pl/~werner/pmn/CPP_HTML/CPP.html.

[12] URL: https://en.wikibooks.org/wiki/C%2B%2B_Programming; URL: https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms.

[13] URL: https://en.cppreference.com/w/cpp/language/statements.

4. *iteracji* (ang. *iteration statement*);
5. *skoku* (ang. *jump statement*);
6. *z etykietą* (ang. *labeled statement*);
7. *deklaracji* (ang. *declaration statement*)

oraz *blok*:

8. *try* (ang. *try block*);
9. *atomowy i synchronizowany* (ang. *atomic and synchronized block*).

W dalszej części książki zostaną omówione typy instrukcji o numerach od 1 do 6. Do numeru 7 wrócimy jeszcze w ramach tego kursu, aczkolwiek w ograniczony sposób. Numery 8 i 9 pominiemy na ten moment – jest to materiał odpowiednio na zaawansowany kurs języka C++ oraz na kurs poświęcony przetwarzaniu współbieżnemu w języku C++. Dla uproszczenia w opisie składni poszczególnych instrukcji pominięte zostaną tzw. *atrybuty* a czasami zastosowana będzie składnia nieformalna. Opis elementów składni również zostanie uproszczony dla klarowności podstawowego kursu programowania. Zainteresowany Czytelnik z łatwością znajdzie więcej informacji o składni można znaleźć w serwisie https://cppreference.com.[14]

4.2 Instrukcja wyrażenia i operatory

Instrukcja wyrażenia[15] to jeden z najczęściej spotykanych typów instrukcji w języku C++. Instrukcja wyrażenia składa się z wyrażenia, tzn. sekwencji operatorów i operandów, oraz znaku ;:

```
wyrazenie;
```

Instrukcja wyrażenia bez wyrażenia jest nazywana *instrukcją pustą*:

```
;
```

Wyrażenie jest, jak wspomniano, sekwencją operatorów i operandów. Jest to dobry moment, aby wspomnieć o operatorach języka C++. Poniżej znajduje się ich lista.

Operatory języka C++:

- *przypisania:* `a = b, a += b, a -= b, a *=b , a /= b, a %= b, a &= b, a |= b, a ^= b, a <<= b, a >>= b;`
- *inkrementacji:* `++a, a++;`
- *dekrementacji:* `--a, a--;`
- *arytmetyczny:* `+a, -a, a + b, a - b, a * b, a / b, a % b;`
- *arytmetyczny bitowy:* `~a, a & b, a | b, a ^ b, a << b, a >> b;`
- *logiczny:* `!a, a && b, a || b;`
- *porównania:* `a == b, a != b, a < b, a > b, a <= b, a >= b, a <=> b;`
- *dostępu:* `a[...], *a, &a, a->b, a.b, a->*b, a.*b;`
- *wywołania funkcji:* `a(...);`
- *przecinek:* `a, b;`
- *warunkowy:* `a ? b : c;`
- *specjalny:* `static_cast, dynamic_cast, const_cast, reinterpret_cast`, *operator rzutowania w stylu C,* `new, delete, sizeof, sizeof..., typeid, noexcept, alignof.`

4.3 Instrukcja złożona

Instrukcja złożona nazywana też *blokiem* jest sekwencją instrukcji zgrupowaną i traktowaną jak jedna instrukcja:

```
{ instrukcja... (opcjonalnie) }
```

gdzie *instrukcja...* jest sekwencją instrukcji. Zauważmy, że sekwencja instrukcji może być pusta. Co więcej, instrukcja złożona przyda się nam jako ciało pętli oraz w instrukcji warunkowej omawianych dalej. Instrukcja złożona wprowadza tzw. *zakres*, który określa widoczność nazwy. Zobaczymy to w poniższym przykładzie.

Kod źródłowy 7: Instrukcja złożona i zakres

```
#include <iostream>

int
main()
{
  // Blok #1
```

[14] URL: https://cppreference.com/.
[15] URL: https://en.cppreference.com/w/cpp/language/expressions.

```
    {
      int n = 42;
      std::cout << n << '\n';
    }
    // std::cout << n << '\n'; // Blad! Zmienna n nie jest tu widoczna.

    const int n = 42;
    const int div = 7;
    int count = 0;
    for (int i = 0; i < n; ++i) { // Blok #2
      std::cout << "Obliczam podzielnosc liczby " << i << " przez " << div
                << ".\n";
      if (i % 7 == 0) { // Blok #3
        ++count;
        std::cout << "Liczba " << i << " jest podzielna przez " << div << ".\n";
      }
    }

    { // Blok #4
      { // Blok #5
        std::cout << "W podanym zakresie jest " << count
                  << " liczb podzielnych przez " << div << ".\n";
      }
    }
  }
```

4.4 Instrukcja wyboru

Instrukcja wyboru steruje przepływem programu wybierając jedną z wielu możliwości. Istnieją następujące instrukcje wyboru o nazwach *if, if-else* oraz *switch*. Zacznijmy od omówienia dwóch pierwszych.

Składnia instrukcja *if* oraz *if-else* jest następująca:[16]

```
if (instrukcja_0 (opcjonalnie) warunek) instrukcja_1
if (instrukcja_0 (opcjonalnie) warunek) instrukcja_1 else instrukcja_2
```

Powyżej, *instrukcja_0* jest instrukcją wyrażenia, w tym instrukcją pustą ;, lub pewną deklaracją, zazwyczaj pojedynczej zmiennej połączonej z jej zainicjowaniem. *warunek* jest wyrażeniem konwertowalnym na wartość logiczną lub jest deklaracją zainicjowanej zmiennej nietablicowej. *instrukcja_1* i *instrukcja_2* są dowolnymi instrukcjami, w tym złożonymi. *instrukcja_1* jest wykonywana jeśli *warunek* ewaluuje się do *true*. *instrukcja_2* jest wykonywana jeśli *warunek* ewaluuje się do *false*. Zauważmy, że w przypadku użycia opcjonalnej instrukcji *instrukcja_0* składnię:

```
if (instrukcja_0 warunek) instrukcja_1
if (instrukcja_0 warunek) instrukcja_1 else instrukcja_2
```

można *w przybliżeniu* przepisać odpowiednio jako:

```
{
  instrukcja_0
  if (warunek)
    instrukcja_1
}
```

oraz:

```
{
  instrukcja_0
  if (warunek)
    instrukcja_1
  else
    instrukcja_2
}
```

Instrukcja *switch* ma następującą składnię:[17]

```
switch (instrukcja_0 (opcjonalnie) warunek) instrukcja_1
```

[16] URL: https://en.cppreference.com/w/cpp/language/if.
[17] URL: https://en.cppreference.com/w/cpp/language/switch.

Powyżej, *instrukcja_0* jest instrukcją wyrażenia, w tym instrukcją pustą ';', lub pewną deklaracją, zazwyczaj pojedynczej zmiennej połączonej z jej zainicjowaniem. *warunek* jest wyrażeniem lub jest deklaracją zainicjowanej zmiennej nietablicowej. Wartość wyrażenia lub deklaracji *warunek* powinna być typu całkowitoliczbowego lub enumeracyjnego, lub klasy konwertowalnej na typ całkowitoliczbowy lub enumeracyjny. *instrukcja_1* jest dowolną instrukcją, w tym złożoną, w której zezwala się na etykiety *case:* oraz *default:* a instrukcja *break* ma szczególne znaczenie.

Zobaczmy przykład użycia instrukcji *switch*.

Kod źródłowy 8: Instrukcja *switch*

```
#include <iostream>

int
main()
{
  std::cout << "Podaj liczbe naturalna z zakresu [0, 10]: ";
  int n;
  std::cin >> n;

  switch (n) {
    case 0:
      std::cout << "Zero.";
      break;
    case 1:
      std::cout << "Jeden.";
      break;
    case 2:
    case 3:
    case 5:
    case 7:
      std::cout << "Liczba pierwsza.";
      break;
    case 4:
    case 6:
    case 8:
    case 9:
    case 10:
      std::cout << "Liczba zlozona.";
      break;
    default:
      std::cout << "Liczba poza zakresem.";
  }

  std::cout << '\n';
}
```

Wynik działania przykładu jest następujący:

```
Podaj liczbe naturalna z zakresu [0, 10]: 5
Liczba pierwsza.
```

4.5 Instrukcja iteracji

Instrukcja iteracji wielokrotnie powtarza ten sam kod. Istnieją następujące instrukcje iteracji: pętle *while*, *do-while*, *for* oraz zakresowa pętla *for*. Ta ostatnia została wprowadzona w standardzie języka C++ z roku 2011 i nie jest ona omawiana w tym rozdziale.

Pętla *while* ma składnię:

```
while ( warunek ) instrukcja
```

warunek jest wyrażeniem konwertowalnym na wartość logiczną, które jest ewaluowane przed każdą potencjalną iteracją pętli, lub jest deklaracją zmiennej, która jest inicjowana przed każdą potencjalną iteracją pętli. Jeśli otrzymana przez ewaluację lub inicjację wartość jest konwertowalna na *true*, wtedy faktycznie wykonywana jest iteracja pętli. W przeciwnym razie pętla jest kończona. *instrukcja* jest dowolną instrukcją, w tym złożoną. *instrukcja* jest wykonywana w każdej iteracji pętli. Przykład użycie pętli *while* został przedstawiony poniżej.

Kod źródłowy 9: Pętla *while*

```
#include <iostream>

```

```
int
main()
{
  std::cout << "Sposob nr 1:\n";
  int n = 10;
  while (n > 0) {
    std::cout << n << "^2 = " << n * n << '\n';
    --n;
  }

  std::cout << '\n';

  std::cout << "Sposob nr 2:\n";
  int i = 10;
  while (int j = i * i) {
    std::cout << i-- << "^2 = " << j << '\n';
  }
}
```

Wynik działania przykładu jest następujący:

```
Sposob nr 1:
10^2 = 100
9^2 = 81
8^2 = 64
7^2 = 49
6^2 = 36
5^2 = 25
4^2 = 16
3^2 = 9
2^2 = 4
1^2 = 1

Sposob nr 2:
10^2 = 100
9^2 = 81
8^2 = 64
7^2 = 49
6^2 = 36
5^2 = 25
4^2 = 16
3^2 = 9
2^2 = 4
1^2 = 1
```

Pętla *do-while* ma następującą składnię:

```
do instrukcja while ( warunek ) ;
```

Powyżej, *instrukcja* jest dowolną instrukcją, w tym złożoną. *instrukcja* jest wykonywana w każdej iteracji pętli. *warunek* jest wyrażeniem konwertowalnym na wartość logiczną, które jest ewaluowane po każdej iteracji pętli. Jeśli otrzymana przez ewaluację wartość jest konwertowalna na *false*, wtedy pętla jest kończona. Przykład zastosowania pętli *do-'while* jest pokazany poniżej.

Kod źródłowy 10: Pętla *do-while*

```
#include <iostream>

int
main()
{
  char c = 'a';
  do {
    std::cout << c << ": " << static_cast<int>(c) << '\n';
  } while (c++ != 'z');
}
```

Wynik działania przykładu jest następujący:

```
a: 97
b: 98
c: 99
d: 100
e: 101
f: 102
g: 103
h: 104
i: 105
j: 106
k: 107
l: 108
m: 109
n: 110
o: 111
p: 112
q: 113
r: 114
s: 115
t: 116
u: 117
v: 118
w: 119
x: 120
y: 121
z: 122
```

Pętla *for* ma składnię:

```
for ( instrukcja_0 warunek (opcjonalnie) ; wyrazenie (opcjonalnie) ) instrukcja
```

Powyżej, *instrukcja_0* jest instrukcją wyrażenia, w tym instrukcją pustą ';', lub pewną deklaracją, zazwyczaj zmiennej licznika pętli połączonej z jego zainicjowaniem. *warunek* jest wyrażeniem konwertowalnym na wartość logiczną, które jest ewaluowane przed każdą potencjalną iteracją pętli, lub jest deklaracją zmiennej, która jest inicjowana przed każdą potencjalną iteracją pętli. Jeśli otrzymana przez ewaluację lub inicjację wartość jest konwertowalna na *true*, wtedy faktycznie wykonywana jest iteracja pętli. W przeciwnym razie pętla jest kończona. *wyrazenie* jest dowolnym wyrażeniem, które jest wykonywane po każdej iteracji pętli i zazwyczaj polega ono na inkrementacji licznika pętli. *instrukcja* jest dowolną instrukcją, w tym złożoną. *instrukcja* jest wykonywana w każdej iteracji pętli.

Powyższy opis może być nieco nieintuicyjny, dlatego warto rozważyć pewne dwa szczególne przypadki. Jeśli *instrukcja_0* jest instrukcją pustą a *warunek* i *wyrazenie* zostały pominięte to otrzymujemy pętlę nieskończoną, np.:

```
for (;;) {
  // Kod wykonywany w petli nieskonczonej.
}
```

Jeśli *instrukcja_0* jest deklaracją zmiennej licznika połączonej z jego zainicjowaniem, *warunek* jest wyrażeniem konwertowalnym na wartość logiczną a *wyrazenie* polega na inkrementacji licznika pętli to otrzymujemy klasyczną pętlę *for*, np.:

```
for (int i = 0; i < 42; ++i) {
 // Kod wykonywany 42 razy dla i rownego odpowiednio 0, 1, ..., 41.
}
```

Zobaczmy jeszcze jeden przykład użycia pętli *for*.

Kod źródłowy 11: Pętla *for*

```
#include <iostream>

int
main()
{
  for (int i = 0; i < 10; i += 2) {
    std::cout << i << '\n';
  }
}
```

Wynik działania przykładu jest następujący:

```
0
2
4
6
8
```

4.6 Instrukcja skoku

Instrukcja skoku przekazuje przepływ programu. Istnieją następujące instrukcje skoku: *break, continue, return* z opjonalnym wyrażeniem albo z użyciem listy inicjującej oraz *goto*. Instrukcja *return* zostanie omówiona przy okazji punktu poświęconego funkcjom. Instrukcja *goto* jest zaszłością historyczną (aczkolwiek ważną) i prawie nigdy nie powinna być używana – zostanie ona na ten moment pominięta. Pozostałe dwie instrukcje są wytłumaczone poniżej.

Instrukcja *break* ma bardzo prostą składnię:

```
break ;
```

Instrukcja ta powoduje zakończenie pętli *while, do-while, for* (w tym zakresowej) oraz instrukcji *switch*. Zobaczmy to na przykładzie.

Kod źródłowy 12: Instrukcja *break*

```
#include <iostream>

int
main()
{
  for (int i = 0; i < 10; ++i) {
    for (int j = 0; ; ++j) {
      std::cout << "* ";
      if (j == i) {
        break; // Zakonczenie wewnetrznej petli.
      }
    }
    std::cout << '\n';
  }
}
```

Wynik działania przykładu jest następujący:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

Instrukcja *continue* ma podobnie jak *break* bardzo prostą składnię.

```
continue ;
```

Instrukcja *continue* powoduje ominięcie pozostałej części bieżącej iteracji w pętli *while, do-while* oraz *for* (w tym zakresowej). Czas wię na kolejny przykład.

Kod źródłowy 13: Instrukcja *continue*

```
#include <iostream>

int
main()
{
  for (int i = 0; i < 10; ++i) {
    if (i == 5)
      continue;
    for (int j = 0; j < 10; ++j) {
      if (i == j)
        continue;
```

```
12            std::cout << "(" << i << ", " << j << ") ";
13          }
14          std::cout << '\n';
15        }
16    }
```

Wynik działania przykładu jest następujący:

```
(0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9)
(1, 0) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (1, 9)
(2, 0) (2, 1) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9)
(3, 0) (3, 1) (3, 2) (3, 4) (3, 5) (3, 6) (3, 7) (3, 8) (3, 9)
(4, 0) (4, 1) (4, 2) (4, 3) (4, 5) (4, 6) (4, 7) (4, 8) (4, 9)
(6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 7) (6, 8) (6, 9)
(7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6) (7, 8) (7, 9)
(8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7) (8, 9)
(9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8)
```

5 Typy danych i zmienne

5.1 Instrukcja prostej deklaracji zmiennej

W tym miejscu wrócimy jeszcze na chwilę do typów instrukcji omawiając szczególny przypadek instrukcji deklaracji[18] – instrukcję prostej deklaracji. *Instrukcja prostej deklaracji* jest instrukcją, która wprowadza i stwarza jeden lub więcej identyfikatorów, zazwyczaj zmiennych, które może opcjonalnie zainicjować. Na ten moment, jak się okazuje, ograniczenie do instrukcji prostej deklaracji jest wciąż nazbyt obszerne. Rozważmy zatem *instrukcję prostej deklaracji zmiennej*. Ma ona następującą składnię:

```
1  sekwencja_specyfikatorow lista_deklaratorow ;
```

Powyżej, *sekwencja_specyfikatorow* jest sekwencją specyfikatorów opisaną poniżej. *lista_deklaratorow* jest listą deklaratorów, tzn. identyfikatorów – nazw zmiennych, z opcjonalnymi wartościami, którymi te zmienne są inicjowane. Zauważmy, że deklarator może zostać zmodyfikowany z użyciem pewnych operatorów, np. *[]* dla typu tablicowego – będzie to opisane dalej. *Sekwencję specyfikatorów* ograniczymy na ten moment wyłącznie do wybranych *prostych specyfikatorów typu* oraz do kwalifikatora *const*. Przykładami prostych specyfikatorów typu[19] są:

- *bool* – typ logiczny (a zarazem całkowitoliczbowy);
- *char* – typ znakowy (a zarazem całkowitoliczbowy);
- *int* – typ całkowitoliczbowy;
- *float*, *double* – typy zmiennoprzecinkowe;
- wcześniej zdefiniowana nazwa klasy, np. *std::string*.

Istnieją także specyfikatory typu całkowitoliczbowego, które modyfikują rozmiar (*short*, *long*) oraz kodowanie znaku (*signed*, *unsigned*). Kwalifikator *const* określa, że typ jest stałą.[20] Zobaczmy krótki przykład.

```
1  char c = 'A';
2  const unsigned int answer = 42;
3  double temperature;
```

5.2 Reguły tworzenia identyfikatorów

Istnieją reguły tworzenia identyfikatorów, w tym nazw zmiennych. Precyzyjne określenie tych reguł jest dość długie, dlatego ograniczymy się w tym momencie do następującego opisu. *Identyfikatorem* może być dowolnie długa sekwencja cyfr *0-9*, znaków _, małych (*a-z*) i wielkich (*A-Z*) znaków alfabetu łacińskiego a także znaków Unicode, które posiadają właściwość `XID_Start` lub `XID_Continue`, z zastrzeżeniem, że:

1. pierwszy znak jest małą lub wielką literą alfabetu łacińskiego, znakiem _ lub znakiem Unicode posiadającym właściwość `XID_Start`;
2. pozostałe znaki (jeśli istnieją) są cyframi, małymi lub wielkimi literami alfabetu łacińskiego, znakami _ lub znakami Unicode posiadającymi właściwość `XID_Continue`;
3. cała sekwencja nie została zarezerwowana przez standard języka.

Powyższy opis byłby kompletną definicją, gdyby nie ostatni punkt, którego zresztą nie będziemy tutaj omawiać. Warto zaznaczyć, że wsparcie kompilatorów dla identyfikatorów posiadających znaki Unicode jest ograniczone, dlatego warto unikać ich stosowania.

[18] URL: https://en.cppreference.com/w/cpp/language/declarations.
[19] URL: https://en.cppreference.com/w/cpp/language/types.
[20] URL: https://en.cppreference.com/w/cpp/language/cv.

5.3 Szerokość i zakres danych poszczególnych typów fundamentalnych

Szerokość **typu logicznego** *bool* jest zależna od implementacji. Zmienne tego typu mogą przechowywać jedną z dwóch wartości: *true* lub *false*.

Standard określa minimalną szerokość **typów całkowitoliczbowych** (do których zalicz się też **typy znakowe**) a rzeczywista wielkość bywa czasami większa (zależy to od tzw. *modelu danych* stosowanego przez daną architekturę komputera i systemu operacyjnego). Minimalne szerokości poszczególnych typów całkowitoliczbowych z wyłączeniem *bool* (jest określany osobno i został już opisany wyżej) według standardu języka C++ są następujące:

- *char* – 8 bitów;
- *short int*, *int* – 16 bitów;
- *long int* – 32 bity;
- *long long int* – 64 bity.

Warto w tym miejscu zauważyć, że specyfikator kodowania znaku *signed*/*unsigned* nie zmienia szerokości typu. Może jednak zadziwiać, że typy *short int* oraz *int* mają taką samą szerokość minimalną. Z drugiej strony dla komputerów osobistych praktyczna realizacja sytuacji gdzie rozmiar *short int* oraz *int* jest jednakowa właściwie już nie występuje (miało to miejsce w przypadku wycofanej z użycia architektury Win16). Na współczesnych 64-bitowych komputerach osobistych szerokości typów przedstawiają się następująco:

- *char* – 8 bitów;
- *short int* – 16 bitów;
- *int* – 32 bity;
- *long int* – 32 lub 64 bity;
- *long long int* – 64 bity.

Szerokość typu wpływa na zakres reprezentowalnych wartości i dla typów całkowitoliczbowych zakres można łatwo wyznaczyć:

- W przypadku liczb bez znaku (*unsigned*) wszystkie bity są wykorzystywane do kodowania cyfr. Dla szerokości równej n bitów można zaprezentować liczby z zakresu $[0, 2^n - 1]$. Poniżej przedstawiono interesujące przypadki szerokości i zakresów wartości:
  - 8 bitów: $[0, 255]$;
  - 16 bitów: $[0, 65535]$;
  - 32 bity: $[0, 4294967295]$;
  - 64 bity: $[0, 18446744073709551615]$.
- W przpadku liczb ze znakiem (*signed*) jeden bit jest wykorzystywany do kodowania znaku a pozostałe – do kodowania cyfr. Aby uniknąć problemu podwójnej reprezentacji wartości zero ($+0$, -0) wprowadzono kod uzupełnieniowy do dwójki i jest to jedyny kod zaakceptowany przez standard języka C++. Z użyciem tego kodu dla szerokości równej n bitów można zaprezentować liczby z zakresu $[2^{n-1}, 2^{n-1} - 1]$. Poniżej przedstawiono interesujące przypadki szerokości i zakresów wartości:
  - 8 bitów: $[-128, 127]$;
  - 16 bitów: $[-32768, 32767]$;
  - 32 bitów: $[-2147483648, 2147483647]$;
  - 64 bity: $[-9223372036854775808, 9223372036854775807]$.

Zostawmy już typy całkowitoliczbowe w spokoju i przejdźmy dalej. Istnieją trzy **typy zmiennoprzecinkowe** określone w standardzie języka C++:

- *float* – typ zmiennoprzecinkowy o pojedynczej precyzji: 32 bity;
- *double* – podwójna precyzja: 64 bity;
- *long double* – rozszerzona precyzja: od 64 do 128 bitów w zależności od platformy (najpopularniejszym obecnie wyborem jest 80 bitów, choć można też z łatwością spotkać implementację opartą o 64 bity).

Sposób kodowania liczb zmiennoprzecinkowych określa standard IEEE 754:

- `binary32`: 1 bit znaku, 8 bitów wykładnika, 23 bity mantysy;
- `binary64`: odpowienio 1, 11 i 52 bitów;
- `binary128`: 1, 15 i 112 bitów.

W przypadku szerokości 80 bitów wykorzystuje się 1 bit znaku, 15 bitów wykładnika oraz 64 bitów mantysy.

Pole wykładnika bywa też nazywane cechą liczby zmiennoprzecinkowej.

Podczas kodowania liczb zmiennoprzecinkowych może zostać użyta reprezentacja *normalna* lub *subnormalna*. Na chwilę obecną pominiemy reprezentację subnormalną i skoncentrujemy się jedynie na reprezentacji normalnej. W reprezentacji normalnej stosuje się mantysę, która zawsze rozpoczyna się od wartości 1 i nie jest ona wtedy kodowana w pamięci komputera. Precyzję dziesiętną można zatem wtedy obliczyć jako $\log_{10} 2^{\text{szerokość mantysy kodowanej} + 1}$ co dla poszczególnych sposobów kodowania oznacza następującą liczbę cyfr znaczących rozwinięcia dziesiętnego (wartości przybliżone):

- `binary32` (23 bity mantysy kodowanej): 7;

- `binary64` (52): 15;
- kodowanie na 80 bitach (64): 19;
- `binary128` (112): 34.

Zakres wykładników dla reprezentacji normalnej:

- `binary32`: [-38, +38];
- `binary64`: [-308, +308];
- 80 bitów oraz `binary128`: [-4932, +4932].

Skąd biorą się powyższe wartości? Aby to wyjaśnić rozważmy przykład kodowania `binary32`. Poniższa reprezentacja binarna zawiera informację o znaku, wykładniku i mantysie i jest reprezentacją normalną pewnej liczby w kodowaniu `binary32`:

```
0 00000001 00000000000000000000000
```

Co oznaczają te liczby? Pierwsze zero oznacza, że liczba jest dodatnia a ciąg ostatnich 23 zer informuje, że mantysa jest równa 1 zgodnie z wcześniej wspomnianą konwencją dla reprezentacji normalnej.

Informacja o wykładniku jest jednak nieco zawoalowana. Jedynka na najmniej znaczącym bicie dawałaby wartość wykładnika 2^0, czyli 1, co można byłoby błędnie zinterpretować jako finalną wartość. W kodowaniu zmiennoprzecinkowym stosuje się jednak przesunięcie wykładnika i od wykładnika o n bitach należy odjąć liczbę $2^{n-1} - 1$. W tym przypadku $n = 8$ (reprezentacja `binary32`), co oznacza, że należy odjąć liczbę 127.

Finalnie otrzymujemy wartość $2^{-126} \approx 1{,}175494 \cdot 10^{-38}$, przy czym reprezentacja dziesiętna jest przybliżona z 7 cyframi rozwinięcia dziesiętnego (1 cyfra przed separatorem dziesiętnym i 6 cyfr po separatorze).

6 Złożone typy danych

6.1 Tablice

Przypomnijmy, że instrukcja prostej deklaracji zmiennej ma składnię:

```
sekwencja_specyfikatorow lista_deklaratorow ;
```

gdzie *lista_deklaratorow* jest listą deklaratorów. Deklarator typu tablicowego ma składnię opisaną niżej, gdzie wykorzystano operator *[]*:

```
deklarator [wyrazenie (opcjonalnie)]
```

Powyżej, *deklarator* jest deklaratorem, tzn. identyfikatorem. *wyrazenie* jest wyrażeniem stałym konwertowalnym na typ *std::size_t* i określa rozmiar tablicy, który powinien być większy od zera. (Wyrażenie stałe oznacza, że jego wartość powinna być możliwa do obliczenia w momencie kompilacji.) Kompilatory mogą rozszerzyć *wyrazenie* na wyrażenie niebędące wyrażeniem stałym – jest to jednak niezgodne ze standardem (choć wykorzystanie tej możliwości bywa nierzadko praktykowane). Warto zauważyć, że typ *std::size_t* jest typem całkowitoliczbowym bez znaku adekwatnym do reprezentowania rozmiarów kontenerów w języku C++. Zamiast zwykłych tablic często lepiej jest korzystać z kontenera *std::vector*. Będzie to opisane dalej.

Zobaczmy przykłady tablic:

```
int arr[42];
char hello[] = { 'H', 'e', 'l', 'l', 'o', ',', ' ',
                 'w', 'o', 'r', 'l', 'd', '!', '\0' };
```

Zobaczmy teraz przykład tablicy wielowymiarowej:

```
int arr[3][2] = {{0, 1}, {2, 3}, {4, 5}};
```

W przypadku tablic wielowymiarowych podtablice są umieszczane kolejno w pamięci. Podobnie zresztą elementy każdej podtablicy są umieszczane kolejno w pamięci. Powyższy przykład ma więc następujący układ danych w pamięci:

```
0 1 2 3 4 5
```

Oznacza to również, że najbardziej wydajny sposób na przetwarzanie elemenentów tablicy polega na iterowaniu ostatniego indeksu w najbardziej zagnieżdżonej pętli (pozwala to na wykorzystanie mechanizmu tzw. *cache prefetching*):

```
const int max_x = 7;
const int max_y = 42;
double data[max_x][max_y];

// ...

```

```
  double sum = 0.;
  for (int x = 0; x < max_x, ++x) {
    for (int y = 0; y < max_y; ++y) {
      sum += data[x][y];
    }
  }
```

W języku C++, gdy jest mowa o tablicach, nie sposób nie wspomnieć o *std::vector*, więc wspomnijmy.

Rozważmy sytuację, w której programista potrzebuje zadeklarować tablicę, której rozmiar nie jest znany w momencie kompilacji. Jest to sytuacja często spotykana – tablica taka nosi nazwę VLA (ang. *variable length array*). Przykładowy program, który demonstruje wykorzystanie VLA jest zaprezentowany poniżej.

Kod źródłowy 14: Tablica VLA

```
#include <cstdlib>
#include <iostream>
#include <random>

int
main()
{
  std::size_t n;
  std::cin >> n;
  int tab[n];

  std::mt19937 engine(std::random_device{}());
  for (std::size_t i = 0; i < n; ++i) {
    std::cout << (tab[i] = std::uniform_int_distribution<int>{ 0, 9 }(engine))
              << '\n';
  }
}
```

Ten program można skompilować i uruchomić. Dociekliwy Czytelnik zauważy jednak, że coś jest nie tak z procedurą kompilacji powyższego programu. Kompilator GCC wypisuje następujące ostrzeżenie:

```
vla.cc: In function ‘int main()’:
vla.cc:10:7: warning: ISO C++ forbids variable length array ‘tab’ [-Wvla]
```

Okazuje się bowiem, że VLA nie jest częścią standardu języka C++ (choć dopuszcza go język C w standardzie z roku 1999) i jest jedynie rozszerzeniem kompilatora. Czy można znaleźć lepsze rozwiązanie?

Zanim odpowiemy sobie na to pytanie, rozważmy kolejny problem. Otóż nierzadko zdarza się, że nie tylko rozmiar tablicy nie jest znany w momencie kompilacji, ale przydałoby się go zmienić w trakcie wykonania programu. Do tego celu służy dynamiczna alokacja pamięci, która w C++ może zostać zrealizowana poprzez użycie operatorów *new* oraz *delete* w wersji tablicowej, tzn. poprzez *new[]* oraz *delete[]*. Niestety taka forma dynamicznej alokacji pamięci jest rozwiązaniem niewygodnym i podatnym na poważne błędy programistyczne, które mogą wieść do katastrofalnych konsekwencji! Czy można znaleźć lepsze rozwiązanie?

Okazuje się, że zarówno problem VLA jak i dynamicznej alokacji pamięci w formie tablic ma rozwiązanie w języku C++, które jest zarówno zgodne ze standardem jak i bardzo wygodne w użyciu. Tym rozwiązaniem, jest kontener *std::vector*. Szczegółowy opis kontenera *std::vector* można znaleźć np. na stronie https://en.cppreference.com/w/cpp/container/vector.[21] W pozostałej części tego podpunktu skoncentrujemy się wyłącznie na wybranych własnościach tego kontenera.

std::vector jest jednym z wielu kontenerów dostępnych w języku C++. Swoją funkcjonalnością może on w zdecydowanej większości przypadków zastępować zwykłą tablicę. Precyzyjnie można ująć, że *std::vector* jest kontenerem sekwencyjnym kapsułkującym tablicę o dynamicznym rozmiarze. Warto pamiętać, że *std::vector* zajmuje więcej pamięci aniżeli zwykła tablica, ponieważ kontener ten wyprzedzająco rezerwuje więcej pamięci aniżeli jest to w danej chwili potrzebne. Kontener ten zwiększa ponadto swój rozmiar, gdyby zaalokowana wcześniej pamięć nie wystarczała do przechowania dodatkowych danych.

Zobaczmy jak niestandardowy przykład wykorzystania VLA może zostać przepisany w kanonicznej formie języka C++.

Kod źródłowy 15: Kontener *std::vector*

```
#include <cstdlib>
#include <iostream>
#include <random>
#include <vector>

```

[21] URL: https://en.cppreference.com/w/cpp/container/vector.

```
int
main()
{
  std::vector<int> v;
  std::mt19937 engine(std::random_device{}());

  std::size_t n;
  std::cin >> n;

  for (std::size_t i = 0; i < n; ++i) {
    v.push_back(std::uniform_int_distribution<int>{ 0, 9 }(engine));
    std::cout << v.back() << '\n';
  }
}
```

6.2 Struktury

Struktura jest typem złożonym składającym się z elementów zapisanych sekwencyjnie według podanej kolejności. Specyfikator typu struktury ma podczas jej definiowania składnię, której uproszczony i przybliżony opis jest podany niżej. Do nieco bardziej kompletnej składni będzie można wrócić przy okazji programowania w stylu obiektowym.

```
struct nazwa (opcjonalnie) { lista_deklaracji }
```

Powyżej, *nazwa* jest nazwą definiowanej struktury. *lista_deklaracji* jest listą dowolnej długości prostych deklaracji zmiennych. Trzeba zauważyć, że powyższy opis jest mocno uproszczony i nie uwzględnia wielu kategorii składników, które mogą pojawić się podczas definiowania struktury, m.in. tych związanych z programowaniem w stylu obiektowym lub generycznym. Na ten moment zadowolimy się jednak taką uproszczoną wersją. A będziemy jeszcze bardziej zadowoleni, gdy zobaczymy jakiś przykład.

Kod źródłowy 16: Struktura

```
#include <cmath>
#include <iostream>

struct point_2d
{
  double x;
  double y;
};

int
main()
{
  point_2d zero(0., 0.);
  point_2d e_x(1., 0.);
  point_2d e_y(0., 1.);
  point_2d p(1., 1.);

  std::cout << "Poczatek ukladu wspolrzednych: (" << zero.x << ", " << zero.y
            << ").\n";
  std::cout << "Wersor e_x: (" << e_x.x << ", " << e_x.y << ").\n";
  std::cout << "Wersor e_y: (" << e_y.x << ", " << e_y.y << ").\n";

  std::cout << "Odleglosc punktu (1, 1) od (0, 0): "
            << std::hypot(p.x - zero.x, p.y - zero.y) << ".\n";
}
```

7 Funkcje

7.1 Wprowadzenie

Czym są funkcje[22] w języku C++? *Funkcja* to element języka, który umożliwia połączenie sekwencji instrukcji z nazwą oraz listą argumentów. Sekwencja instrukcji jest nazywana *ciałem funkcji*. Lista argumentów może być pusta.

Funkcja może zostać *zadeklarowana* oraz *zdefiniowana*. Deklaracja funkcji wprowadza jej nazwę (identyfikator) i umożliwia użycie (wywołanie). Definicja funkcji jest potrzebna do opisu wykonywanych czynności (jest to

[22] URL: https://en.cppreference.com/w/cpp/language/functions; URL: https://en.cppreference.com/w/cpp/language/function.

implementacja zadeklarowanego identyfikatora). Funkcja może zostać zadeklarowana wielokrotnie lecz zdefiniowana tylko jeden raz. Definicja funkcji może pojawić się w miejscu, gdzie oczekiwana byłaby deklaracja.

7.2 Deklaracja funkcji

Zacznijmy więc od deklaracji funkcji. Uproszczona składnia deklaracji funkcji jest poniżej.

```
typ_zwracany deklarator ( lista_argumentow )
```

Powyżej, *typ_zwracany* jest typem wartości zwracanej przez funkcję z użyciem instrukcji *return* lub jest słowem *void*, jeśli funkcja nic nie zwraca. *deklarator* jest identyfikatorem funkcji, tzn. jej nazwą. *lista_argumentow* jest listą argumentów odseparowanych przecinkami. Lista może być pusta. Zobaczmy przykłady deklaracji funkcji.

```
#include <string>

int f(int);
double g();
std::string h(const char* str);
```

7.3 Definicja funkcji

Uproszczona składnia definicji funkcji jest podobna do jej deklaracji. Jedyna istotna na ten moment różnica jest taka, że definicja funkcji posiada *ciało*.

```
typ_zwracany deklarator ( lista_argumentow ) cialo
```

typ_zwracany, *deklarator* oraz *lista_argumentow* zostały opisane przy okazji deklaracji funkcji. *cialo* jest ciałem funkcji, tzn. instrukcją złożoną zawierającą zero lub więcej instrukcji otoczonych nawiasami klamrowymi *{* oraz *}*. Instrukcja złożona jest wykonywana w momencie wywołania funkcji. Zobaczmy przykłady definicji funkcji.

```
#include <cmath>
#include <string>
#include <vector>

int id(int x) {
  return x;
}

double answer() {
  return 42;
}

std::string to_string(const char* str) {
  return std::string(str);
}

std::vector<double> quadratic_eq(double a, double b, double c) {
  std::vector<double> res;
  double delta = b * b - 4 * a * c;
  if (delta >= 0) {
    res.push_back((-b + std::sqrt(delta)) / (2 * a));
    if (delta > 0) {
      res.push_back((-b - std::sqrt(delta)) / (2 * a));
    }
  }
  return res;
}
```

Zauważmy, że lista argumentów w definicji funkcji zawiera tzw. *argumenty formalne*, czyli identyfikatory wykorzystywane w ciele definiowanej funkcji. W momencie wywołania funkcji, w miejsce argumentów formalnych wstawiane są *argumenty właściwe* (nazywane też *argumenty rzeczywistymi* lub, korzystając błędnie z kalki z języka angielskiego, *argumentami aktualnymi*). W poniższym przykładzie *x* jest argumentem formalnym a *42* oraz *n* – właściwymi.

```
int id(int x) {
  return x;
}

int main() {
  id(42);
```

```
7     int n = 7;
8     id(n);
9   }
```

7.4 Instrukcja *return*

Instrukcja *return* jest instrukcją skoku o bardzo prostej składni:

```
1   return wyrazenie (opcjonalnie) ;
```

Powyżej, *wyrazenie* jest wyrażeniem konwertowalnym na typ zwracany przez funkcję. W ramach instrukcjij skoku *return* wyrażenie *wyrazenie* jest ewaluowane, po czym bieżąca funkcja kończona a wartość zwrócona po ewentualnej konwersji do funkcji wywołującej. Przykład użycia instrukcji *return* znajduje się poniżej.

Kod źródłowy 17: Instrukcja *return*

```
1   #include <iostream>
2   #include <vector>
3
4   double
5   accumulate(std::vector<double> v);
6
7   int
8   main()
9   {
10    std::vector<double> piggy_bank;
11    piggy_bank.push_back(10.23); // styczen
12    piggy_bank.push_back(9.35); // luty
13    // ...
14    piggy_bank.push_back(25.12); // grudzien
15
16    double total = // (*) Tutaj jest "wkladana" wartosc res
17      accumulate(piggy_bank); // z instrukcji return z funkcji accumulate.
18
19    std::cout << "Calkowite oszczednosci w skarbonce: " << total << " zl\n";
20  }
21
22  double
23  accumulate(std::vector<double> v)
24  {
25    double res = 0.;
26    for (std::size_t i = 0; i < v.size(); ++i) {
27      res += v.at(i);
28    }
29    return res; // Instrukcja return wykonuje skok z tego miejsca do linijki (*).
30  }
```

7.5 Rekurencja

Rekurencja polega na wywołaniu w definicji danej funkcji jej samej. Zobaczmy to na przykładzie.

Kod źródłowy 18: Rekurencja

```
1   #include <iostream>
2
3   using ulli_t = unsigned long long int;
4
5   ulli_t
6   factorial(ulli_t n);
7
8   int
9   main()
10  {
11    // factorial(21) daje nieprawidlowy wynik dla typu ulli_t.
12    for (int i = 0; i < 21; ++i) {
13      std::cout << i << "! = " << factorial(i) << '\n';
14    }
15  }
```

```
16
17  ulli_t
18  factorial(ulli_t n)
19  {
20    return n == 0 ? 1 : n * factorial(n - 1);
21  }
```

W powyższym przykładzie w definicji funkcji *factorial* następuje wywołanie funkcji *factorial* – mamy więc do czynienia z rekurencją. Wynik działania przykładu jest następujący:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
```

Z rekurencją trzeba jednak uważać, co obrazuje poniższy przykład.

Kod źródłowy 19: Ciąg Fibonacciego

```
1   #include <iostream>
2
3   int counter = 0;
4
5   int
6   Fibonacci(int n)
7   {
8     counter++;
9     return n == 0 ? 0 : n == 1 ? 1 : Fibonacci(n - 1) + Fibonacci(n - 2);
10  }
11
12  int
13  main()
14  {
15    const int i = 42;
16    const int res = Fibonacci(i);
17    std::cout << "Fibonacci(" << i << ") = " << res << " (" << counter
18              << " wywolan funkcji Fibonacci)\n";
19  }
```

Wynik działania przykładu jest następujący:

```
Fibonacci(42) = 267914296 (866988873 wywolan funkcji Fibonacci)
```

7.6 Przeciążanie funkcji

W języku C++ istnieje możliwość *przeciążania* funkcji, tzn. specyfikowania więcej niż jednej funkcji o takiej samej nazwie. Zobaczmy przykład.

Kod źródłowy 20: Przeciążanie funkcji

```
1   #include <iostream>
```

```
#include <string>
#include <vector>

void
print(int i)
{
  std::cout << "Liczba calkowita: " << i << '\n';
}

void
print(double d)
{
  std::cout << "Liczba zmiennoprzecinkowa: " << d << '\n';
}

void
print(std::vector<std::string> v)
{
  std::cout << "Kontener std::vector zawierajacy napisy std::string:\n";
  for (std::size_t i = 0; i < v.size(); ++i) {
    std::cout << i + 1 << ". " << v.at(i) << '\n';
  }
  std::cout << "Calkowita liczba napisow: " << v.size() << '\n';
}

int
main()
{
  print(42);
  print(137.035999084);
  std::vector<std::string> v = {
    "\"Answer to the Ultimate Question of Life, The Universe, and Everything\"",
    "odwrotnosc stalej struktury subtelnej"
  };
  print(v);
}
```

Wynik działania przykładu jest następujący:

```
Liczba calkowita: 42
Liczba zmiennoprzecinkowa: 137.036
Kontener std::vector zawierający napisy std::string:
1. "Answer to the Ultimate Question of Life, The Universe, and Everything"
2. odwrotnosc stalej struktury subtelnej
Calkowita liczba napisow: 2
```

Czym powinny się różnić funkcje o tej samej nazwie, aby zostały prawidłowo przeciążone? Mogą one się różnić liczbą argumentów formalnych lub ich typem. Jeśli dwie funkcje o tej samej nazwie różnią się wyłącznie typem zwracanej wartości to nastąpi błąd kompilacji. Lista prawidłowych i nieprawidłowych różnic jest dłuższa, lecz na ten moment ograniczymy się wyłącznie do przed chwilą wymienionych.

8 Zakres

8.1 Wprowadzenie

Każda deklaracja w C++ jest widoczna w jakimś zakresie.[23] Zostaną tutaj omówione czym są *widoczność* oraz *zakres*. Zostaną przedstawione wybrane przykłady zakresów. Rozważmy na początek prosty przykład.

Kod źródłowy 21: Zakres

```
#include <iostream>

void
f()
{
  int a = 42;
```

[23] URL: https://en.cppreference.com/w/cpp/language/scope.

```
7     std::cout << "Zmienna a (f) ma wartosc " << a << ". "
8               << "Znajduje sie pod adresem " << &a << ".\n";
9   }
10
11  int
12  main()
13  {
14    int a = 7;
15    std::cout << "Zmienna a (main) ma wartosc " << a << ". "
16              << "Znajduje sie pod adresem " << &a << ".\n";
17    f();
18    std::cout << "Zmienna a (main) ma wartosc " << a << ". "
19              << "Znajduje sie pod adresem " << &a << ".\n";
20  }
```

Wynik działania przykładu jest następujący:

```
Zmienna a (main) ma wartosc 7. Znajduje sie pod adresem 0x7ffcc35f1d24.
Zmienna a (f) ma wartosc 42. Znajduje sie pod adresem 0x7ffcc35f1d04.
Zmienna a (main) ma wartosc 7. Znajduje sie pod adresem 0x7ffcc35f1d24.
```

O adresowaniu pamięci komputera poświęcony jest osobny materiał w tej książce. Na cwhilę obecną wystarczy wiedzieć, że kod *&a* oznacza adres zmiennej *a* to znaczy jej położenie w pamięci komputera oraz że adres jest nieujemną liczbą całkowitą notowaną zazwyczaj w systemie szesnastkowym. Adres zmiennej może być różny w różnych uruchomieniach tego samego programu.

Przebieg działania powyższego przykładu jest następujący. Uruchamiana jest funkcja *main*, gdzie tworzona jest zmienna o nazwie *a* o wartości *7*. Po wypisaniu komunikatu na temat tej zmiennej wywoływana jest funkcja *f*, gdzie tworzona jest zmienna *a* o wartości *42*. Po wypisaniu komunikatu działanie funkcji *f* kończy się i następuje powrót do funkcji *main*, gdzie wypisywany jest jeszcze jeden komunikat.

Szczegółowa analiza wypisywanych komunikatów pozwala dojść do wniosku, że zmienna *a* z funkcji *main* oraz zmienna *a* z funkcji *f* są innymi bytami. Widać to wprost dzięki znajomości adresów oraz pośrednio dzięki wypisywanym wartościom.

Dlaczego, pomimo identycznej nazwy, obydwie zmienne są różne? Dzieje się tak, ponieważ obydwie zmienne znajdują się w różnych *zakresach*. Zmienna zadeklarowana w danej funkcji znajduje się w zakresie powiązanym z tą funkcją. Dalej zobaczymy różne przykłady zakresów.

8.2 Zakres globalny

Zakres globalny zawiera w sobie cały program. Jest to jedyny zakres, który nie zawiera się w żadnym innym zakresie.

8.3 Zakres blokowy

Zakres blokowy jest wprowadzany poprzez użycie instrukcji wyboru *if* oraz *switch*, pętli *while*, *do-while* oraz *for* (w tym zakresowej pętli *for*) a także instrukcji złożonej. Zobaczmy to na przykładzie pętli *for*:

```
1   for (int i = 0; i < 42; ++i) // Zmienna i rezyduje w zakresie wprowadzonym przez
2     std::cout << i << '\n'; // petle for.
3   // Ten punkt jest poza zakresem wprowadzonym przez petle for. Zmienna i jest
4   // niedostepna.
```

Instrukcja złożona, nazywana też blokiem, była omówiona wraz z przykładem zawartym w kodzie źródłowym 7 na stronie 12, w którym widać bloki o numerach od 1 do 5 i każdy z nich wprowadza zakres blokowy. Zakres wprowadzony przez blok numer 5 jest zawarty w zakresie wprowadzonym przez blok 4. Zakres wprowadzony przez blok 2 jest zawarty w zakresie wprowadzonym przez pętlę *for*.

8.4 Zakres argumentu funkcji

Każdy argument z listy argumentów w deklaracji bądź definicji funkcji wprowadza nowy zakres, który rozpoczyna się tym argumentem i kończy się odpowiednio wraz z końcem deklaracji lub wraz z końcem ciała definiowanej funkcji. Widać to na poniższym kodzie źródłowym:

```
1   void f(
2           int n, // Rozpoczyna sie zakres wprowadzony przez n.
3           double d // A tutaj rozpoczyna sie zakres wprowadzony przez d.
4         ); // Konczy sie zakres wprowadzony przez n oraz d.
5
6   double
```

```
7   sum_of_squares(
8                    double x, // Poczatek zakresu wprowadzonego przez x.
9                    double y // . . . . . y.
10                  ) {
11    double res = x * x + y * y;
12    return res;
13  } // Koniec zakresow wprowadzonych przez x oraz y.
```

W powyższym przykładzie celowo nie zachowano tzw. stylu kodowania, aby wskazać początek i koniec danego zakresu.

9 Wskaźniki i referencje

9.1 Wprowadzenie

Przypomnijmy, że instrukcja prostej deklaracji zmiennej ma składnię:

```
1   sekwencja_specyfikatorow lista_deklaratorow ;
```

gdzie *lista_deklaratorow* jest listą deklaratorów. Deklarator typu wskaźnikowego ma następującą składnię:[24]

```
1   * const (opcjonalnie) deklarator
```

gdzie *const* jest opcjonalnym kwalifikatorem *const*, który określa, że wskaźnik jest stały, a *deklarator* jest deklaratorem. Zobaczmy przykłady wskaźników:

```
1   int x = 42;
2   const int c = 7;
3
4   int* p0 = &x; // wskaznik na zmienna typu int
5
6   const int* p2 = &c; // wskaznik na stala typu int
7   int const* p3 = &c; // jw.
8
9   int* const p4 = &x; // staly wskaznik na zmienna typu int
10
11  const int* const p5 = &c; // staly wskaznik na stala typu int
12  int const* const p6 = &c; // jw.
```

W powyższym przykładzie ograniczyliśmy się wyłącznie do wskaźników na typ *int* oraz *const int*. Zauważmy również, że w celu pobrania adresu wykorzystany został operator *&*.

W celu uzyskania wartości wskazywanej przez wskaźnik należy wykorzystać tzw. *dereferencję* (zwaną też *wyłuskaniem*), którą otrzymuje się poprzez zastosowanie operatora *:

```
1   int x = 42;
2   std::cout << "x = " << x << '\n';
3
4   int* p = &x;
5   std::cout << "x = " << *p << '\n';
```

Wynik działania powyższego przykładu jest następujący:

```
x = 42
x = 42
```

9.2 Wskaźnik pusty *nullptr*

Do wskaźnika, który nie wskazuje na dane, należy przypisać specjalną wartość określaną przez literał *nullptr*. Warto zrobić to już w momencie zainicjowania zmiennej wskaźnikowej:

```
1   int* p = nullptr;
2   double* q = nullptr;
```

Wskaźnik pusty nie wskazuje na żadne dane – informuje o tym jego specjalna wartość. Co więcej, poprawnie skonstruowany kod źródłowy sprawdza czy ma do czynienia z pustym wskaźnikiem. Jak to może wyglądać w praktyce? Zobaczymy to na przykładzie szkieletu aplikacji okienkowej, gdzie wykorzystano tzw. *widget* (słowo pochodzi od *window gadget*).

[24] URL: https://en.cppreference.com/w/cpp/language/pointer.

```
struct Widget;

Widget*
make_widget()
{
  Widget* res;
  // Tutaj moze znalezc sie kod tworzacy element interfejsu (widget).
  // W przypadku niepowodzenia tworzenia wykonywana jest ponizsza instrukcja.
  res = nullptr;
  return res;
}

void
update_widget(Widget* w)
{
  if (w != nullptr) {
    // Widget nalezy zaktualizowac tylko jesli zostal poprawnie utworzonony.
    // Tutaj moze znalezc sie kod aktualizujacy widget.
  }
}

void
destroy_widget(Widget* w)
{
  if (w) {
    // Instrukcja if (w) dziala tak samo jak if (w != nullptr).
  }
}

int
main()
{
  Widget* w = make_widget();
  update_widget(w);
  destroy_widget(w);
}
```

9.3 Wskaźnik typu *void*

Mimo, że nie istnieje typ danych *void* a samo to słowo ma szczególne znaczenie, istnieje możliwość zdefiniowania wskaźnika typu *void*:

```
struct s
{
  int n;
};

double x = 0.5;
const s y = { 7 };

void* p = &x;
const void* q = &y;
```

Wskaźniki typu *void* są wykorzystywane w języku C w celu osiągnięcia polimorfizmu. W języku C++ istnieją lepsze rozwiązania takie jak polimorfizm statyczny z wykorzystaniem szablonów.

9.4 Wskaźnik na funkcję

Istnieje możliwość stworzenia wskaźnika na funkcję:

```
int sum(int a, int b) { return a + b; }
int (*p)(int, int) = &sum;
```

Aby zainicjować wartość wskaźnika na funkcję należy podać nazwę funkcji poprzedzoną opcjonalnie operatorem *&*. Zamiast *int (*p)(int, int) = ∑* można opuścić operator *&* i napisać *int (*p)(int, int) = sum;*. Dzieje się tak, ponieważ następuje niejawna konwersja nazwy funkcji na adres początku jej kodu wykonywalnego w kontekście wymagającym podania adresu.

W celu wywołania funkcji, do której znany jest wskaźnik należy zwyczajowo zastosować dereferencję:

```
std::cout << (*p)(1, 2) << '\n'; // Wypisze wartosc 3.
```

A jak stworzyć tablicę wskaźników na funkcje? Oto przykład tablicy (rozmiaru 7) wskaźników na funkcje przyjmujące dwa argumenty typu *double* i niezwracające niczego: *void (*p[7])(double, double);*. Składnię można uprościć stosując alias *using F = void(double, double);*. Wtedy wystarczy deklaracja o skróconej i bardziej znajomej postaci *F* p[7];*.

9.5 Referencje

Istnieją dwa rodzaje referencji[25] w języku C++. Są to referencje do l-wartości oraz do r-wartości. Na ten moment ograniczymy się do tego pierwszego (i zarazem starszego) rodzaju. Przypomnijmy, że instrukcja prostej deklaracji zmiennej ma składnię:

```
sekwencja_specyfikatorow lista_deklaratorow ;
```

gdzie *lista_deklaratorow* jest listą deklaratorów. Deklarator typu referencyjnego ma następującą składnię:

```
& deklarator
```

gdzie *deklarator* jest deklaratorem. Należy zauważyć, że referencja musi zostać zainicjowana. Zobaczmy przykłady:

```
int n = 0;
int& r = n;
r = 42;
std::cout << n << '\n'; // Wypisze wartosc 42.
```

Jak widać na powyższym przykładzie referencje służą do tworzenia *aliasów* (nowych nazw) dla zmiennych. Referencja *r* jest nową nazwą dla *n* i modyfikacja wartości *r* oznacza modyfikację *n*, ponieważ *n* oraz *r* są tym samym obiektem. Jakie jest typowe zastosowanie referencji? Zobaczmy to na przykładzie argumentów funkcji.

Kod źródłowy 22: Referencja

```
#include <iostream>
#include <string>

void
replace(std::string& s, char from, char to)
{
  for (std::size_t i = 0; i < s.size(); ++i) {
    if (s.at(i) == from) {
      s.at(i) = to;
    }
  }
}

int
main()
{
  std::string s = "Programowanie w C++";
  std::cout << "Napis oryginalny: " << s << '\n';
  replace(s, ' ', '*');
  std::cout << "Napis zmodyfikowany: " << s << '\n';
}
```

Wynik działania przykładu jest następujący:

```
Napis oryginalny: Programowanie w C++
Napis zmodyfikowany: Programowanie*w*C++
```

Zauważmy, że dzięki zastosowaniu referencji obiekt *s* z funkcji *main* powyższego przykładu nie jest kopiowany do funkcji *replace*. Dzięki temu funkcja *replace* modyfikuje oryginał, co było w tym przypadku zamierzone. Co więcej, brak kopiowania obiektu oznacza niejednokrotnie oszczędność (w pamięci oraz w czasie wykonania). Dlatego referencje warto wykorzystywać. Trzeba tylko wiedzieć jak prawidłowo użyć referencji. Rozważmy poniższy przykład.

```
#include <iostream>
#include <string>

std::string
join(std::string& s1, std::string& s2)
{
  return s1 + s2;
```

[25] URL: https://en.cppreference.com/w/cpp/language/reference.

```
}

int
main()
{
  std::string s = "Programowanie ";
  std::string t = "jest proste!\n";
  std::cout << join(s, t);
}
```

Wynik działania przykładu jest następujący:

```
Programowanie jest proste!
```

Funkcja *join* łączy dwa napisy podane jako referencje. Dzięki temu nie następuje kopiowanie napisów typu *std::string* do funkcji. Wszystko działa bardzo dobrze... do czasu gdy nie zmodyfikujemy lekko wywołania funkcji. Oto więc nowa funkcja *main*:

```
int
main()
{
  std::string s = "Programowanie ";
  std::cout << join(s, "jest proste!\n");
}
```

Co się zmieniło? Zamiast podać drugi argument jako zmienną *t* postanowiliśmy przekazać go do funkcji *join* jako napis w stylu języka C z intencją, aby w międzyczasie został wywołany konstruktor *std::string* a wynik przypisany do argumentu *s2* funkcji *join*. I faktycznie konstruktor *std::string* został wywołany – została stworzona tymczasowa nienazwana wartość typu *std::string*. Po tym próbowano przypisać tę tymczasową wartość do referencji *s2* a to się nie udało – oto odpowiednie fragmenty błędu generowanego przez kompilator GCC:

```
error: cannot bind non-const lvalue reference of type ‘std::string&’ [...] to an
rvalue of type ‘std::string’
```

Płynie stąd ważny wniosek: do referencji (do l-wartości) nie można przypisać wartości tymczasowej. Istnieją dwa rozwiązania tego problemu – można zastosować stałą referencję do l-wartości lub referencję do r-wartości. Na ten moment ograniczymy się do pierwszego rozwiązania. Czym jest stała referencja (do l-wartości)? Rozważmy poniższy przykład.

```
int n = 42;
const int& r = n;
n++; // Teraz n jest rowne 43.
r--; // Blad! Ta linijka sie nie skompiluje.
```

Fragment *const int& r = n;* jest stałą referencją, co oznacza, że *r* jest referencją (aliasem, nową nazwą dla *n*), ale z jej użyciem nie można zmodyfikować wartości zmiennej *n*. Korzystając z tej wiedzy można już rozwiązać poprawnie wcześniejszy przykład.

Kod źródłowy 23: Stała referencja (do l-wartości)

```
#include <iostream>
#include <string>

std::string
join(const std::string& s1, const std::string& s2)
{
  return s1 + s2;
}

int
main()
{
  std::string s = "Programowanie ";
  std::cout << join(s, "jest proste!\n");
}
```

Powyższy przykład działa już poprawnie. Prawda, że programowanie jest proste?

10 Programowanie obiektowe

10.1 Wprowadzenie

W języku C++ istnieje możliwość definiowania własnych typów danych. Widzieliśmy to na przykładzie struktur, gdzie zdefiniowaliśmy typ *point_2d* (zob. kod źródłowy 16 na stronie 22. Rozszerzmy teraz typ *point_2d*.

Kod źródłowy 24: Klasa

```
#include <cmath>
#include <iostream>

class point_2d
{
public:
  point_2d(double x, double y)
    : x_(x)
    , y_(y)
  {
  }

  double get_x() const { return x_; }
  double get_y() const { return y_; }

  double distance_from_zero() const { return std::hypot(x_, y_); }

private:
  double x_;
  double y_;
};

int
main()
{
  point_2d zero(0., 0.);
  point_2d e_x(1., 0.);
  point_2d e_y(0., 1.);
  point_2d p(1., 1.);

  std::cout << "Poczatek ukladu wspolrzednych: (" << zero.get_x() << ", "
            << zero.get_y() << ").\n";
  std::cout << "Wersor e_x: (" << e_x.get_x() << ", " << e_x.get_y() << ").\n";
  std::cout << "Wersor e_y: (" << e_y.get_x() << ", " << e_y.get_y() << ").\n";

  std::cout << "Odleglosc punktu (1, 1) od (0, 0): " << p.distance_from_zero()
            << ".\n";
}
```

Mimo, że powyższy program jest krótki, to pojawiło się w nim kilka nowości:

- słowo *class* służące do deklarowania i definiowania klas;
- specyfikatory dostępu *public* oraz *private*;
- konstruktor (*point_2d::point_2d*);
- metody klasy (*get_x*, *get_y*, *distance_from_zero*);
- kapsułkowanie danych (*x_*, *y_*).

W dalszej części tego punktu przyjrzymy się dokładniej wyżej wymienionym pojęciom.

10.2 Klasy i obiekty

Mowi się, że *klasa jest modelem dla zbioru obiektów*.[26] Tylko co to oznacza? Przyjrzyjmy się ponownie przykładowi z kodu źródłowego 24. Została tam zdefiniowana klasa *point_2d* oraz kilka obiektów tej klasy o nazwach *zero*, *e_x*, *e_y* oraz *p*. Możemy więc w ramach języka C++ stwierdzić, że klasa jest typem a obiekty są zmiennymi (instancjami) tego typu. Klasa posiada identyfikującą ją nazwę, np. *point_2d* lub *std::vector<int>*. Dzięki tej nazwie można tworzyć obiekty tej klasy. Do tego celu wykorzystywane są konstruktory, o których powiemy sobie więcej za chwilę. Klasa określa jakie dane są przechowywane w jej instancjach. W przypadku *point_2d* są to dwie dane typu *double* o nazwach *x_* oraz *y_*. Określone są też prawa dostępu do tych danych i w naszym przykładzie te dane są dostępne wyłącznie

[26] URL: https://en.cppreference.com/w/cpp/language/classes; URL: https://en.cppreference.com/w/cpp/language/class.

z poziomu kodu samej klasy. Zostanie to omówione obszerniej nieco dalej. Klasa posiada metody, nazywane też funkcjami składowymi. Funkcje składowe są, jak nazwa wskazuje, funkcjami. Jednakże funkcje składowe mają pewnę szczególną własność odróżniającą je od zwykłych funkcji – są one wywoływane na rzecz obiektów danej klasy.

10.3 Kapsułkowanie danych

Kapsułkowanie (*hermetyzacja*) polega na ograniczeniu dostępu do danych przechowywanych przez obiekt poprzez ukrycie ich w części prywatnej. Zauważmy, że dane składowe `x_` oraz `y_` przykładu w kodzie źródłowym 24 nie są dostępne wprost poza kodem klasy. W szczególności próba odwołania się do pola `x_` lub `y_` np. z poziomu funkcji *main()* poprzez kod `e_x.x_` zakończy się błędem kompilacji. Kapsułkowanie wraz z metodami klasy umożliwia precyzyjną kontrolę dostępu do przechowywanych damych. W ww. przykładzie po zainicjowaniu obiektu danymi nie było możliwości ich modyfikacji a dostęp do nich był ograniczony wyłącznie do ich odczytu poprzez metody *point_2d::get_x* oraz *point_2d::get_y*. Programista może modyfikować zestaw metod aby umożliwiać i uniemożliwiać odczyt i modyfikację prywatnych (zakapsułkowanych) danych. Kapsułkowanie danych jest w języku C++ realizowane poprzez specyfikator dostępu *private*.

10.4 Specyfikatory dostępu *public* oraz *private*

Nazwa każdego składnika klasy ma przypisane prawa dostępu.[27] Dostęp może być:

- publiczny (*public*);
- chroniony (*protected*);
- prywatny (*private*).

Kiedy nazwa składnika klasy jest używana gdzieś w programie, sprawdzane są prawa dostępu. Publiczne prawa dostępu umożliwiają dostęp z dowolnego miejsca programu a prywatne prawa dostępu – wyłącznie dostęp z poziomu danej klasy. Chronione prawa dostępu zostaną na ten moment pominięte. Zobaczmy jak publiczne i prywatne prawa dostępu działają w praktyce na poniższym przykładzie.

```
1  class test
2  {
3  public:
4    void public_f() const { private_f(); }
5
6  private:
7    void private_f() const {}
8  };
9
10 int
11 main()
12 {
13   test t;
14   t.public_f();
15   t.private_f(); // Ten wiersz jest bledny!
16 }
```

Powyższy przykład nie skompiluje się. Fragment komunikatu błędu zgłoszonego przez kompilator GCC jest następujący:

```
error: ‘void test::private_f() const’ is private within this context
```

Można zobaczyć na tym przykładzie, że składowe publiczne są dostępne wszędzie, ale prywatne są dostępne tylko z poziomu kodu samej klasy.

10.5 Słowo *class* a słowo *struct*

Ze specyfikatorami dostępu wiąże się pewien fakt. Otóż w języku C++ struktury i klasy są tym samym z dokładnością do domyślnych praw dostępu. Jeśli w definicji struktury nie zostanie użyty specyfikator dostępu, to jej składniki są dostępne publicznie:

```
1  struct point {
2    double x;
3    double y;
4  };
5
6  point p;
```

[27] URL: https://en.cppreference.com/w/cpp/language/access.

```
p.x = 0.0;
p.y = 1.0;
```

Jeśli w definicji klasy nie zostanie użyty specyfikator dostępu, to jej składniki mają dostępność prywatną:

```
class point {
  double x;
  double y;
};

point p;
// Nie mozna odwolac sie do p.x ani p.y z poziomu tego miejsca!
```

Można stwierdzić, że *struct x { skladowe }* jest tym samym co *class x { public: skladowe }* zaś *class y { skladowe }* jest tym samym co *struct y { private: skladowe }*.

10.6 Konstruktory

Czym jest konstruktor? Konstruktor[28] jest specjalną funkcją składową klasy, która jest wywoływana automatycznie podczas tworzenia obiektów tej klasy i służy do inicjowania tych obiektów. Wróćmy do przykładu z kodu źródłowego 24 i przyjrzyjmy się bliżej konstruktorowi klasy *point_2d*:

```
point_2d(double x, double y)
  : x_(x)
  , y_(y)
{
}
```

Korzystając z innego stylu kodowania można ten konstruktor przepisać tak:

```
point_2d(double x, double y) : x_(x), y_(y) {
}
```

Nazwa konstruktora klasy *point_2d* to *point_2d*. Jest regułą w języku C++, że nazwa konstruktora jest nazwą klasy. Konstruktor jest funkcją (choć specjalną), dlatego może on przyjmować argumenty. W przykładzie konstruktor *point_2d* przyjmuje dwa argumenty typu *double*, które służą do zainicjowania danych składowych obiektu *point_2d*, tzn. *x_* oraz *y_*. Do zainiacjowania służy tzw. *lista inicjująca*, która rozpoczyna się od znaku dwukropka i składa się z elementów inicjujących poszczególne składowe:

```
point_2d(double x, double y) : x_(x), y_(y) {
//                             ^^^^^^^^^^^^^^
//                             To jest lista
//                             inicjujaca.
```

Konstruktor jest funkcją (choć specjalną), dlatego posiada ciało, które jest zawarte między klamerkami *{* oraz *}*. Często się zdarza, że ciało jest puste. Konstruktor jest funkcją (choć specjalną). Przy okazji specyfikowania konstruktora nie podaje się typu zwracanego przez tę funkcję tak jak robi się to przy zwykłych funkcjach.

Na zakończenie warto jeszcze przytoczyć jedną uwagę dotyczącą listy inicjującej oraz ciała funkcji. Otóż niektórzy programiści C++ unikają listy inicjującej i zamiast napisać:

```
point_2d(double x, double y) : x_(x), y_(y) {
}
```

wolą użyć:

```
point_2d(double x, double y) {
  x_ = x;
  y_ = y;
}
```

Jest to jednak błędna strategia. Można to łatwo sprawdzić testując obydwa rozwiązania ze zmienioną częścią prywatną ww. przykładu:

```
private:
  const double x_;
  const double y_;
```

[28] URL: https://en.cppreference.com/w/cpp/language/constructor.

10.7 Metody klas (funkcje składowe)

Metoda klasy jest funkcją, którą można wywołać na rzecz obiektu stosując notację z operatorem dostępu do składowej.[29] Jeśli *X* jest klasą, to dla obiektu *x* tej klasy wywołanie:

```
x.f();
```

oznacza użycie metody *f* na rzecz *x* a operator . użyty do wywołania jest operatorem dostępu do składowej. W przykładzie z kodu źródłowego 24 metodami klasy *point_2d* są *get_x*, *get_y* oraz *distance_from_zero*. Widać tutaj, że metody klasy definiuje się w sposób podobny do zwykłych funkcji, aczkolwiek z pewnymi wyjątkami. Pierwszy jest raczej dość oczywisty – metoda klasy należy do klasy, więc można zdefiniować ją wewnątrz definicji klasy. Drugi wyjątek jest mniej oczywisty – metoda klasy może mieć kwalifikator *const*, co będzie wyjaśnione dalej.

10.8 Metody klas z kwalifikatorem *const*

Zacznijmy od przykładu.

Kod źródłowy 25: Metody z kwalifikatorem *const*

```
#include <iostream>

class counter
{
public:
  void increment() { ++n_; }
  void reset() { n_ = 0; }
  int get() const { return n_; }

private:
  int n_;
};

int
main()
{
  counter c{};
  for (int i = 0; i < 3; ++i) {
    c.increment();
    std::cout << c.get() << '\n';
  }
  c.reset();
  std::cout << c.get() << '\n';
}
```

Wynik działania przykładu jest następujący:

```
1
2
3
0
```

10.9 Destruktory

Destruktor[30] jest specjalną funkcją składową. Jest on wywoływany w momencie zakończenia czasu istnienia obiektu, aby zwolnić zasoby zaalokowane ten przez obiekt. Jeżeli klasa nie posiada zdefiniowanego destruktora może zostać on wygenerowany przez kompilator (tzw. *definicja niejawna*). Destruktor zdefiniowany niejawnie ma puste ciało. W przypadku ręcznego zarządzania zasobami, destruktor nie powinien mieć pustego ciała. Nazwa destruktora jest nazwą klasy poprzedzoną znakiem tyldy (~). Zobaczmy przykład prostego destruktora o dyskusyjnej użyteczności.

Kod źródłowy 26: Destruktor

```
#include <iostream>

class verbose
{
public:
```

[29] URL: https://en.cppreference.com/w/cpp/language/member_functions.
[30] URL: https://en.cppreference.com/w/cpp/language/destructor.

```
6     verbose() { std::cout << "Konstruktor\n"; }
7     ~verbose() { std::cout << "Destruktor\n"; }
8     void get() const { std::cout << "Metoda skladowa\n"; }
9   };
10
11  int
12  main()
13  {
14    for (int i = 0; i < 3; ++i) {
15      verbose v{};
16      v.get();
17    }
18  }
```

Wynik działania przykładu jest następujący:

```
Konstruktor
Metoda skladowa
Destruktor
Konstruktor
Metoda skladowa
Destruktor
Konstruktor
Metoda skladowa
Destruktor
```

10.10 *this*

Wyrażenie *this* ma wartość równą adresowi obiektu, na rzecz którego wywoływana jest funkcja składowa (z pewnymi wyjątkami, które na ten moment musimy pominąć).[31] Jeśli funkcja składowa posiada kwalifikator *const* wtedy wyrażenie **this* jest traktowane jako stałe.

Kod źródłowy 27: *this*

```
1   #include <iostream>
2
3   class counter
4   {
5   public:
6     counter& increment()
7     {
8       ++n_;
9       return *this;
10    }
11
12    counter& reset()
13    {
14      n_ = 0;
15      return *this;
16    }
17
18    int get() const { return n_; }
19
20  private:
21    int n_;
22  };
23
24  int
25  main()
26  {
27    counter c{};
28    for (int i = 0; i < 3; ++i) {
29      std::cout << c.increment().get() << '\n';
30    }
31    std::cout << c.reset().get() << '\n';
32  }
```

[31] URL: https://en.cppreference.com/w/cpp/language/this.

Wynik działania przykładu jest taki sam jak przykładu z kodu źródłowego 25 ze strony 35. W tym momencie powinien być już zrozumiały przykład z kodu źródłowego 3 ze strony 7.

10.11 Deklaracja przyjaźni

Deklaracja przyjaźni[32] może wystąpić w ciele klasy. Umożliwia ona funkcji lub innej klasie dostęp do prywatnych (i chronionych) składowych klasy.

Kod źródłowy 28: Deklaracja przyjaźni

```
1  #include <iostream>
2  #include <string>
3
4  class hello_kitty
5  {
6  public:
7    hello_kitty(const std::string& text)
8      : my_secret_{ "Hello, kitty! " + text + '\n' }
9    {
10   }
11
12   friend std::ostream& operator<<(std::ostream& os, const hello_kitty& kitty);
13
14 private:
15   std::string my_secret_;
16 };
17
18 std::ostream&
19 operator<<(std::ostream& os, const hello_kitty& kitty)
20 {
21   os << kitty.my_secret_;
22   return os;
23 }
24
25 int
26 main()
27 {
28   hello_kitty kitty{ "I love milk too!" };
29   std::cout << kitty; // Wywolanie funkcji operator<<.
30 }
```

Wynik działania przykładu jest następujący:

```
Hello, kitty! I love milk too!
```

11 Pliki

11.1 Wprowadzenie

Obsługa plików ma wiele wspólnego z poznaną już w międzyczasie obsługą wejścia (strumień *cin*) oraz wyjścia (strumień *cout*). Plik nagłówkowy odpowiedzialny za obsługę plików to *fstream*:[33]

```
1  #include <fstream>
```

Klasy odpowiedzialne za obsługę strumieni plikowych to *std::ifstream* (strumienie wejściowe), *std::ofstream* (strumienie wyjściowe) oraz *std::fstream* (strumienie dwukierunkowe).

11.2 Klasa *std::ifstream*

Zacznijmy od przykładu.

Kod źródłowy 29: Strumień *std::ifstream*

```
1  #include <fstream>
2  #include <iostream>
3
```

[32] URL: https://en.cppreference.com/w/cpp/language/friend.
[33] URL: https://en.cppreference.com/w/cpp/header/fstream.

```
4   int
5   main()
6   {
7     char c;
8     std::ifstream file{ "ifstream.cc" };
9     while (file >> std::noskipws >> c) {
10      std::cout << c;
11    }
12  }
```

Wynikiem działania przykładu jest wypisanie kodu źródłowego programu przykładu. W powyższym przykładzie konstruktor *std::ifstream* otwiera plik *ifstream.cc* do odczytu (tzn. z obiektu *file* można teraz korzystać podobnie jak z obiektu *cin*) a destruktor wywołany automatycznie przed zakończeniem funkcji *main* zamyka ten plik. Program działa poprawnie ponieważ plik *ifstream.cc* znajduje się w bieżącym katalogu. Funkcja *std::noskipws* umożliwia wczytanie białych znaków (takich jak znak spacji).[34] Obiekt strumienia plikowego (nie tylko *std::ifstream*) w kontekście logicznym zwraca wartość logiczną, która informuje czy strumień nadaje się do operacji wejścia/wyjścia.

11.3 Klasa *std::ofstream*

Znowu zacznijmy od przykładu.

Kod źródłowy 30: Strumień *std::ofstream*

```
1   #include <ctime>
2   #include <fstream>
3
4   int
5   main()
6   {
7     std::ofstream file{ "time.txt" };
8     file << std::time(nullptr) << '\n';
9   }
```

Wynikiem działania przykładu jest plik wyjściowy z bieżącym czasem (zob. punkt *Notes* na stronie dokumentacji funkcji *std::time*[35]). W powyższym przykładzie plik *time.txt* nie musi istnieć w bieżącym katalogu, aby program działał poprawnie. Jeśli taki plik istnieje to jego zawartość zostanie nadpisana co można sprawdzić uruchamiając wielokrotnie program i sprawdzając zawartość pliku wynikowego.

11.4 Klasa *std::fstream*

Klasa *std::fstream* łączy własności klas *std::ifstream* oraz *std::ofstream* umożliwiając obsługę wejścia/wyjścia plikowego.

11.5 Tryby otwarcia pliku

Plik może być otwarty na wiele różnych sposobów:[36]

- *in* – otwiera plik do odczytu (domyślne zachowanie *std::ifstream*);
- *out* – otwiera plik do zapisu (domyślne zachowanie *std::ofstream*);
- *binary* – otwiera plik w trybie binarnym;
- *app*, *ate* – przechodzi do końca strumienia odpowiednio przed każdym zapisem (dla *app*) lub bezpośrednio po otwarciu (dla *ate*);
- *trunc* – pomija zawartość strumienia w momencie otwarcia.

Poniżej znajduje się lekko zmodyfikowany program jednego z wcześniejszych przykładów.

Kod źródłowy 31: Łączenie sposobów otwarcia pliku

```
1   #include <ctime>
2   #include <fstream>
3
4   int
5   main()
6   {
7     std::ofstream file{ "time.txt", std::ios::out | std::ios::app };
8     file << std::time(nullptr) << '\n';
9   }
```

[34] URL: https://en.cppreference.com/w/cpp/io/manip/skipws.
[35] URL: https://en.cppreference.com/w/cpp/chrono/c/time.
[36] URL: https://en.cppreference.com/w/cpp/io/ios_base/openmode.

Tym razem wynikiem działania przykładu jest dodawanie do końca stworzonego pliku wyjściowego nowej informacji o czasie.

11.6 Tryb binarny

Zacznijmy od przykładu.

Kod źródłowy 32: Tryb binarny

```
#include <fstream>
#include <iostream>
#include <string>

void
encode_to_binary_file(const std::string& filename, int value)
{
  std::ofstream file{ filename, std::ios::binary };
  file.write(reinterpret_cast<const char*>(&value), sizeof(value));
}

int
decode_from_binary_file(const std::string& filename)
{
  std::ifstream file{ filename, std::ios::binary };
  int n;
  file.read(reinterpret_cast<char*>(&n), sizeof(n));
  return n;
}

int
main()
{
  const std::string file{ "file.bin" };
  encode_to_binary_file(file, 42);
  std::cout << decode_from_binary_file(file) << '\n';
}
```

Wynik działania przykładu jest następujący:

```
42
```

Poza wypisaniem na standardowy strumień wyjściowy wynikiem działania programu jest też stworzenie pliku binarnego zawierającego binarną reprezentację liczby 42 w systemie kodowania odpowiednim dla maszyny, na którym program został uruchomiony. Operator *reinterpret_cast* w powyższym przykładzie służy do tzw. *paronomazji typów* (*type punning*) i jest potencjalnie niebezpieczny – paronomazja może wieść do naruszenia tzw. *ścisłych zasad aliasowania* (*strict aliasing rule*[37]).

11.7 Wskaźniki strumienia

Strumień wejścia/wyjścia posiada dwa wskaźniki: wejścia oznaczonego symbolem *g* (od słowa *get*) oraz wyjścia oznaczonego symbolem *p* (od *put*). Do obsługi wskaźników wejścia/wyjścia można wykorzystać metody odczytujące (*tellg*,[38] *tellp*[39]) oraz ustawiające pozycję wskaźnika (*seekg*,[40] *seekp*[41]).

Kod źródłowy 33: Rozmiar pliku

```
#include <fstream>
#include <iostream>
#include <string>

std::streamsize
file_size(const std::string& filename)
{
  // Uwaga: Lepiej jest korzystac ze strumienia wejsciowego std::ifstream
```

[37] URL: https://stackoverflow.com/questions/98650/what-is-the-strict-aliasing-rule.
[38] URL: https://en.cppreference.com/w/cpp/io/basic_istream/tellg.
[39] URL: https://en.cppreference.com/w/cpp/io/basic_ostream/tellp.
[40] URL: https://en.cppreference.com/w/cpp/io/basic_istream/seekg.
[41] URL: https://en.cppreference.com/w/cpp/io/basic_ostream/seekp.

```
9     // anizeli ze strumienia wyjsciowego std::ofstream do pomiaru rozmiaru pliku.
10    // Czy wiesz dlaczego?
11    std::ifstream file{ filename };
12    file.seekg(0, std::ios::end); // Ustawienie wskaznika wejsciowego.
13    return file.tellg(); // Pobranie pozycji wskaznika wejsciowego.
14  }
15
16  int
17  main()
18  {
19    std::cout << "Rozmiar pliku: " << file_size("size.cc") << '\n';
20  }
```

Wynik działania przykładu jest następujący:

```
Rozmiar pliku: 564
```

Jest to prawidłowy rozmiar pliku w bajtach przy założeniu, że powyższy kod źródłowy zostanie umieszczony w pliku o nazwie *size.cc*.

12 Szablony

12.1 Wprowadzenie

Szablony[42] są funkcjonalnością języka C++ umożliwiającą tworzenie m.in. rodzin klas lub funkcji. Szablony C++ są mechanizmem czasu kompilacji kompletnym w sensie Turinga. Oznacza to, że można z ich użyciem symulować dowolną maszynę Turinga i w efekcie wykonać dowolne jej obliczenia (por. hipotezę Churcha-Turinga) już w trakcie kompilacji. W przypadku podziału programu komputerowego na pliki nagłówkowe i implementacji, szablony są umieszczane w plikach nagłówkowych. Szablony służą do realizacji stylu generycznego, z którego przykładem w języku Object Pascal mieliśmy do czynienia już na początku książki w kodzie źródłowym 4 na stronie 8. Zobaczmy odpowiednik tego przykładu napisany w języku C++.

12.2 Szablony funkcji

Zacznijmy od przykładu szablonu funkcji.[43]

Kod źródłowy 34: Sortowanie generyczne

```
1   #include <algorithm>
2   #include <cstddef>
3   #include <iostream>
4   #include <iterator>
5
6   template<typename T>
7   void
8   sort(T* a, std::size_t n)
9   {
10    do {
11      std::size_t new_n = 0;
12      for (std::size_t i = 1; i < n; ++i) {
13        if (a[i - 1] > a[i]) {
14          std::swap(a[i - 1], a[i]);
15          new_n = i;
16        }
17      }
18      n = new_n;
19    } while (n > 0);
20  }
21
22  template<typename T>
23  void
24  print_array(std::ostream& os, T* a, std::size_t n)
25  {
26    for (std::size_t i = 0; i < n; ++i) {
27      os << a[i] << ' ';
```

[42] URL: https://en.cppreference.com/w/cpp/language/templates.
[43] URL: https://en.cppreference.com/w/cpp/language/function_template.

```
28      }
29      os << '\n';
30    }
31
32    int
33    main()
34    {
35      int tab0[] = { 0, 7, 3, 42, 12, 100, 50 };
36      print_array(std::cout, tab0, std::size(tab0));
37      sort(tab0, std::size(tab0));
38      print_array(std::cout, tab0, std::size(tab0));
39
40      std::string tab1[] = { "fonon", "plazmon", "ekscyton", "polaryton",
41                             "dziura", "elektron", "foton" };
42      print_array(std::cout, tab1, std::size(tab1));
43      sort(tab1, std::size(tab1));
44      print_array(std::cout, tab1, std::size(tab1));
45    }
```

Wynik działania przykładu jest następujący:

```
0 7 3 42 12 100 50
0 3 7 12 42 50 100
fonon plazmon ekscyton polaryton dziura elektron foton
dziura ekscyton elektron fonon foton plazmon polaryton
```

Przyjrzyjmy się wywołaniu funkcji sortujących (a dokładniej ich specjalizacji szablonu funkcji). Okazuje się, że prawidłowymi byłyby również wywołania:

```
37      sort<int>(tab0, std::size(tab0));
```

lub:

```
43      sort<>(tab1, std::size(tab1));
```

Oznacza to, że w składni języka C++ istnieją trzy sposoby na odwołanie się do szablonu (nie tylko zresztą funkcji):

- *nazwa_szablonu<lista_parametrow>*;
- *nazwa_szablonu<>*;
- *nazwa_szablonu*.

Ostatnia forma jest najkrótsza w związku z tym najbardziej korzystna, ale nie zawsze jest możliwa do zastosowania (ponieważ może zawieść dedukcja typów). W takich sytuacjach przydatna jest pierwsza forma, gdzie *explicite* podaje się wszystkie typy specjalizacji. Drugą formę (z pustymi nawiasami ostrymi) na ten moment pozostawimy, ponieważ jest ona najbardziej przydatna w zaawansowanym metaprogramowaniu.

Należy zauważyć, że nazwę typu wykorzystanego w szablonie można użyć w ciele funkcji szablonowej dokładnie tak samo jak nazwy zwykłych typów:

```
1   template<typename T>
2   void
3   swap(T& a, T& b)
4   {
5     T temp = a;
6     a = b;
7     b = temp;
8   }
```

12.3 Szablony klas

Zacznijmy od przykładu szablonu klasy.[44]

Kod źródłowy 35: Szablon klasy

```
1   #include <cassert>
2   #include <cstddef>
3   #include <string>
4
5   template<typename T>
6   class counted_proxy
```

[44] URL: https://en.cppreference.com/w/cpp/language/class_template.

```
{
public:
  counted_proxy(const T& t)
    : t_{ t }
    , counter_{ 0 }
  {
  }

  T get() const
  {
    ++counter_;
    return t_;
  }

  counted_proxy& set(const T& t)
  {
    counter_ = 0;
    t_ = t;
    return *this;
  }

  std::size_t get_counter() const { return counter_; }

private:
  T t_;
  mutable std::size_t counter_;
};

int
main()
{
  counted_proxy<std::string> cp{ "Piggy Bank" };
  assert(cp.get_counter() == 0);
  cp.get();
  assert(cp.get_counter() == 1);
  cp.set("Hello, kitty!");
  assert(cp.get_counter() == 0);
}
```

12.4 Dygresja: Asercje

Instrukcja *assert(wyrazenie)*[45] stwierdza, że *wyrazenie* ma wartość *true*, co implikuje dalsze wykonanie programu. Gdyby jednak *wyrazenie* miało wartość *false* to program kończy swoje wykonanie z błędem. Oznacza to, że:

```
assert(true);
```

jest instrukcją, przez którą program przechodzi bezbłędnie. Natomiast:

```
assert(false);
```

jest instrukcją, która spowoduje zakończenie programu z błędem. Asercje są elementem *defensywnego* stylu programowania, służacego do wczesnego wyłapywania błędów. Służą one jako podstawa *automatycznych testów jednostkowych*. Na przykład, dla programu:

```
#include <cassert>

int
main()
{
  const double a = 0.5;
  assert(a + a == 1.0);

  const double b = 0.1;
  assert(3 * b == 0.3);
}
```

można otrzymać na wyjściu:

[45] URL: https://en.cppreference.com/w/cpp/error/assert.

```
a.out: example.cc:10: int main(): Assertion ‘3 * b == 0.3’ failed.
Przerwane (zrzut pamięci)
```

Dlaczego tak się dzieje? Ponieważ liczba *0.1* nie ma dobrej reprezentacji binarnej.[46]

Programista, który lubi asercje, ma więcej wolnego czasu.

12.5 Parametry szablonów niebędące typami

Parametrem szablonu może być również, z pewnymi ograniczeniami, argument niebędący typem. Zobaczmy to na przykładzie.

Kod źródłowy 36: Parametry szablonów niebędące typami

```
1  #include <cassert>
2  #include <cstddef>
3
4  template<typename T, std::size_t N>
5  class array
6  {
7  public:
8    T& operator[](std::size_t i)
9    {
10     check_bounds(i);
11     return a_[i];
12   }
13
14   const T& operator[](std::size_t i) const
15   {
16     check_bounds(i);
17     return a_[i];
18   }
19
20   std::size_t size() const { return N; }
21
22 private:
23   void check_bounds(std::size_t i) const { assert(i < N); }
24
25 private:
26   T a_[N] = {}; // Kompilator wygeneruje konstruktor za nas i zainicjuje tablice
27                 // domyslna wartoscia dla T...
28 };
29
30 class test
31 {
32 public:
33   test()
34     : x_{ 42 } // ...ktora w tym przypadku jest rowna 42.
35   {
36   }
37
38   int get() const { return x_; }
39   void set(int x) { x_ = x; }
40
41 private:
42   int x_;
43 };
44
45 int
46 main()
47 {
48   array<test, 7> a{};
49   assert(a.size() == 7);
50   assert(a[3].get() == 42);
51   a[3].set(100);
52   assert(a[3].get() == 100);
53 }
```

[46] David Goldberg. “What Every Computer Scientist Should Know about Floating-Point Arithmetic”. W: *ACM Computing Surveys* 23 (1991), s. 5–48. DOI: 10.1145/103162.103163.

13 Funkcje lambda

13.1 Wprowadzenie

Funkcja lambda[47] (nazywana też *funkcją anonimową*) jest funkcją, która nie posiada nazwy. Funkcje lambda są praktyczną realizacją λ-funkcji, które są elementem rachunku λ zapoczątkowanego przez Alonzo Churcha i stanowią istotny element programowania funkcyjnego (Alonzo Church, Haskell Curry). Rachunek λ może zostać bardzo zwięźle sformułowany w notacji BNF.[48] W języku C++ funkcje lambda mają bardzo szerokie możliwości. Dlatego na ten moment skoncentrujemy się wyłącznie na wybranych aspektach.

13.2 Proste funkcje lamda

Pusta funkcja lambda to *[]() {}*. Niczego ona nie robi. Możemy to zobaczyć w poniższym przykładzie, gdzie została ona wywołana:

```
int
main()
{
  []() {}();
}
```

Instrukcja *[]() {}();* składa się z funkcji lambda *[]() {}*, która została wywołana z użyciem operatora nawiasy okrągłe *()* leżącego bezpośrednio przed znakiem średnika.

Pusta funkcja lambda nie jest szczególnie użyteczna. Zobaczmy więc funkcję, która przyjmuje argument i go zwraca (w tym przypadku do systemu operacyjnego):

```
int
main()
{
  auto f = [](int x) { return x; };
  return f(0);
}
```

W powyższym przykładzie wykorzystano dedukcję typów 'auto', której nie możemy jeszcze zbyt obszernie omówić. Wystarczy założyć na ten moment, że sekwencję specyfikatorów w instrukcji prostej deklaracji zmiennej można zastąpić słowem *auto*. A więc, zamiast np. *int i = 42;* można napisać *auto i = 42;* i typ *i* zostanie wydedukowany na podstawie wartości, którą została zainicjowana zmienna *i*. Użycie *auto* w przypadku funkcji lambda jest konieczne, ponieważ każda funkcja lambda ma swój własny unikalny typ. W powyższym przykładzie wyrażenie lambda zostało przypisane do zmiennej *f*, po czym zmienną tą można wykorzystać tak jak zwykłą funkcję, tzn. wywołać z użyciem operatora nawiasy okrągłe.

13.3 Parametryzacja przez funkcję lambda

Zastanówmy się, gdzie funkcje lambda mogą się przydać. W tym celu korzystajmy z omówionego wcześniej przy okazji szablonów przykładu z sortowaniem.

Kod źródłowy 37: Parametryzacja przez obiekt wywoływalny na przykładzie funkcji lambda

```
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <iterator>

template<typename T, typename Comp>
void
sort(T* a, std::size_t n, Comp comp)
{
  do {
    std::size_t new_n = 0;
    for (std::size_t i = 1; i < n; ++i) {
      if (comp(a[i - 1], a[i])) {
        std::swap(a[i - 1], a[i]);
        new_n = i;
      }
    }
```

[47] URL: https://en.cppreference.com/w/cpp/language/lambda.

[48] *CS 3110—Data Structures and Functional Programming. Fall 2008*. 2008. URL: https://www.cs.cornell.edu/courses/cs3110/2008fa/recitations/rec26.html.

```
18          n = new_n;
19        } while (n > 0);
20      }
21
22      template<typename T>
23      void
24      print_array(std::ostream& os, T* a, std::size_t n)
25      {
26        for (std::size_t i = 0; i < n; ++i) {
27          os << a[i] << ' ';
28        }
29        os << '\n';
30      }
31
32      int
33      main()
34      {
35        int tab[] = { 0, 7, 3, 42, 12, 100, 50 };
36        print_array(std::cout, tab, std::size(tab));
37        sort(tab, std::size(tab), [](int a, int b) { return a > b; });
38        print_array(std::cout, tab, std::size(tab));
39        sort(tab, std::size(tab), [](int a, int b) { return a < b; });
40        print_array(std::cout, tab, std::size(tab));
41      }
```

Wynik działania przykładu jest następujący:

```
0 7 3 42 12 100 50
0 3 7 12 42 50 100
100 50 42 12 7 3 0
```

Podobną funkcjonalność można osiągnąć stosując zwykłe funkcje lub funktory (struktury ze zdefiniowaną funkcją składową *operator()*). Jednak obydwa rozwiązania są zazwyczaj znacznie dłuższe aniżeli zastosowanie funkcji lambda.

14 Biblioteka standardowa

14.1 Pliki nagłówkowe biblioteki standardowej języka C++

Biblioteka standardowa języka C++ (*C++ Standard Library*, por. również *Standard Template Library*[49]) jest kolekcją funkcjonalności (przede wszystkim szablonów klas i funkcji) napisanych w języku C++ (a dokładniej z użyciem jego „centralnej" części) i umieszczonych w standardzie języka. Funkcjonalność biblioteki standardowej jest podzielona na pliki nagłówkowe, których lista znajduje się poniżej. W nawiasach okrągłych podano standard języka C++, w którym wprowadzono dany plik nagłówkowy z ewentualną informacją „przestarzałe" lub „usunięte" oraz informacją czy funkcjonalność jest narzędziem zaczerpniętym z języka C.

[49] *Standard Template Library Programmer's Guide*. URL: https://www.boost.org/sgi/stl/index.html.

- <algorithm> (C++98)
- <any> (C++17)
- <array> (C++11)
- <atomic> (C++11)
- <barrier> (C++20)
- <bit> (C++20)
- <bitset> (C++98)
- <cassert> (C++98, narzędzia języka C)
- <ccomplex> (C++11, przestarzałe w C++17, usunięte w C++20)
- <cctype> (C++98, narzędzia języka C)
- <cerrno> (C++98, narzędzia języka C)
- <cfenv> (C++11, narzędzia języka C)
- <cfloat> (C++98, narzędzia języka C)
- <charconv> (C++17)
- <chrono> (C++11)
- <cinttypes> (C++11, narzędzia języka C)
- <ciso646> (C++98, usunięte w C++20)
- <climits> (C++98, narzędzia języka C)
- <clocale> (C++98, narzędzia języka C)
- <cmath> (C++98, narzędzia języka C)
- <codecvt> (C++11, przestarzałe w C++17)
- <compare> (C++20)
- <complex> (C++98)
- <concepts> (C++20)
- <condition_variable> (C++11)
- <coroutine> (C++20)
- <csetjmp> (C++98, narzędzia języka C)
- <csignal> (C++98, narzędzia języka C)
- <cstdalign> (C++11, przestarzałe w C++17, usunięte w C++20)
- <cstdarg> (C++98, narzędzia języka C)
- <cstdbool> (C++11, przestarzałe w C++17, usunięte w C++20)
- <cstddef> (C++98, narzędzia języka C)
- <cstdint> (C++11, narzędzia języka C)
- <cstdio> (C++98, narzędzia języka C)
- <cstdlib> (C++98, narzędzia języka C)
- <cstring> (C++98, narzędzia języka C)
- <ctgmath> (C++11, przestarzałe w C++17, usunięte w C++20)
- <ctime> (C++98, narzędzia języka C)
- <cuchar> (C++11, narzędzia języka C)
- <cwchar> (C++98, narzędzia języka C)
- <cwctype> (C++98, narzędzia języka C)
- <deque> (C++98)
- <exception> (C++98)
- <execution> (C++17)
- <filesystem> (C++17)
- <format> (C++20)
- <forward_list> (C++11)
- <fstream> (C++98)
- <functional> (C++98)
- <future> (C++11)
- <initializer_list> (C++11)
- <iomanip> (C++98)
- <ios> (C++98)
- <iosfwd> (C++98)
- <iostream> (C++98)
- <istream> (C++98)
- <iterator> (C++98)
- <latch> (C++20)
- <limits> (C++98)
- <list> (C++98)
- <locale> (C++98)
- <map> (C++98)
- <memory> (C++98)
- <memory_resource> (C++17)
- <mutex> (C++11)
- <new> (C++98)
- <numbers> (C++20)
- <numeric> (C++98)
- <optional> (C++17)
- <ostream> (C++98)
- <queue> (C++98)
- <random> (C++11)
- <ranges> (C++20)
- <ratio> (C++11)
- <regex> (C++11)
- <scoped_allocator> (C++11)
- <semaphore> (C++20)
- <set> (C++98)
- <shared_mutex> (C++14)
- <source_location> (C++20)
- (C++20)
- <sstream> (C++98)
- <stack> (C++98)
- <stdexcept> (C++98)
- <stop_token> (C++20)
- <streambuf> (C++98)
- <string> (C++98)
- <string_view> (C++17)
- <strstream> (C++98)
- <syncstream> (C++20)
- <system_error> (C++11)
- <thread> (C++11)
- <tuple> (C++11)
- <typeindex> (C++11)
- <typeinfo> (C++98)
- <type_traits> (C++11)
- <unordered_map> (C++11)
- <unordered_set> (C++11)
- <utility> (C++98)
- <valarray> (C++98)
- <variant> (C++17)
- <vector> (C++98)
- <version> (C++20)

Funkcjonalność udostępniana przez ww. pliki nagłówkowe to w szczególności:

- algorytmy[50]
- iteratory[51]
- koncepty/*concepts*[52]
- kontenery[53]
- lokalizacja[54]
- metaprogramowanie[55]
- napisy[56]
- narzędzia[57]
- obliczenia numeryczne[58]
- obsługa błędów[59]
- system plików[60]
- wejscie/wyjście[61]

[50] URL: https://en.cppreference.com/w/cpp/algorithm.
[51] URL: https://en.cppreference.com/w/cpp/iterator.
[52] URL: https://en.cppreference.com/w/cpp/concepts.
[53] URL: https://en.cppreference.com/w/cpp/container.
[54] URL: https://en.cppreference.com/w/cpp/locale.
[55] URL: https://en.cppreference.com/w/cpp/meta.
[56] URL: https://en.cppreference.com/w/cpp/string.
[57] URL: https://en.cppreference.com/w/cpp/utility.
[58] URL: https://en.cppreference.com/w/cpp/numeric.
[59] URL: https://en.cppreference.com/w/cpp/error.
[60] URL: https://en.cppreference.com/w/cpp/filesystem.
[61] URL: https://en.cppreference.com/w/cpp/io.

- współbieżność[62]
- wyrażenia regularne[63]
- zakresy/*ranges*[64]

14.2 Kontenery

Kontenery biblioteki standardowej języka C++ można podzielić na trzy grupy:

- kontenery sekwencyjne (*std::array*, *std::deque*, *std::forward_list*, *std::list*, *std::vector*);
- kontenery asocjacyjne (*std::map* oraz *std::multiset*, *std::set* oraz *std::multimap*);
- nieuporządkowane kontenery asocjacyjne (*std::unordered_map* oraz *std::unordered_multimap*, *std::unordered_set* oraz *std::unordered_multiset*).

Kontener sekwencyjny to taki, który samodzielnie nie zmienia kolejności wstawionych elementów. Kontener asocjacyjny może zmieniać kolejność elementów.

Kontener sekwencyjny *std::vector* został już omówiony wcześniej. Przykład użycia kontenera asocjacyjnego *std::set*[65] znajduje się poniżej.

Kod źródłowy 38: Kontener asocjacyjny *std::set*

```
#include <iostream>
#include <random>
#include <set>

std::mt19937 engine{ std::random_device{}() };

int
random(int a, int b)
{
  return std::uniform_int_distribution<int>{ a, b }(engine);
}

int
main()
{
  std::set<int> numbers;

  for (int i = 0; i < 10; ++i) {
    const int draw = random(0, 100);
    std::cout << "Wylosowano liczbe " << draw << ".\n";
    numbers.insert(draw);
  }

  std::cout << "Zawartosc kontenera set: ";
  for (auto x : numbers) {
    std::cout << x << ' ';
  }
  std::cout << std::endl;
}
```

Wynik działania przykładu może być następujący:

```
Wylosowano liczbe 11.
Wylosowano liczbe 39.
Wylosowano liczbe 90.
Wylosowano liczbe 64.
Wylosowano liczbe 93.
Wylosowano liczbe 48.
Wylosowano liczbe 30.
Wylosowano liczbe 93.
Wylosowano liczbe 42.
Wylosowano liczbe 41.
Zawartosc kontenera set: 11 30 39 41 42 48 64 90 93
```

[62] URL: https://en.cppreference.com/w/cpp/thread.
[63] URL: https://en.cppreference.com/w/cpp/regex.
[64] URL: https://en.cppreference.com/w/cpp/ranges.
[65] URL: https://en.cppreference.com/w/cpp/container/set.

Wynik działania powyższego przykładu może różnić się pomiędzy poszczególnymi wywołaniami programu. Kontener *std::multiset* jest podobny do kontenera *std::set* z tą różnicą, że *std::multiset* zezwala na wielokrotne wystąpienia klucza o tej samej wartości. Kontenery *std::unordered_set* oraz *std::unordered_multiset* są odpowiednikami odpowiednio kontenerów *std::set* oraz *std::multiset* z tą różnicą, że *std::unordered_set* oraz *std::unordered_multiset* nie zachowują porządku sortowania.

Kontener asocjacyjny *std::map* został zaprezentowany na poniższym przykładzie.

Kod źródłowy 39: Kontener asocjacyjny *std::map*

```
#include <iostream>
#include <map>
#include <string>

int
main()
{
  std::map<char, unsigned int> counter;
  std::string str{ "Hello, kitty! I love milk too!" };
  for (char c : str) {
    counter[c]++;
  }
  for (auto pair : counter) {
    std::cout << "Znak: " << pair.first << " Liczba wystapien: " << pair.second
              << '\n';
  }
}
```

Wynik działania przykładu jest następujący:

```
Znak:   Liczba wystapien: 5
Znak: ! Liczba wystapien: 2
Znak: , Liczba wystapien: 1
Znak: H Liczba wystapien: 1
Znak: I Liczba wystapien: 1
Znak: e Liczba wystapien: 2
Znak: i Liczba wystapien: 2
Znak: k Liczba wystapien: 2
Znak: l Liczba wystapien: 4
Znak: m Liczba wystapien: 1
Znak: o Liczba wystapien: 4
Znak: t Liczba wystapien: 3
Znak: v Liczba wystapien: 1
Znak: y Liczba wystapien: 1
```

W powyższym przykładzie typem klucza jest *char* a typem wartości jest *unsigned int*. W kontenerze *std::map* klucze nie mogą się powtarzać. Poza kontenerem *std::map* istnieją też *std::multimap, unordered_map* oraz *unordered_multimap*.

14.3 Algorytmy

Zacznijmy od przykładu.

Kod źródłowy 40: Algorytmy w języku C++

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <random>
#include <vector>

std::mt19937 engine{ std::random_device{}() };

template<typename C>
void
print_container(std::ostream& os, const std::string& str, const C& c)
{
  os << str << ": ";
  std::for_each(c.begin(), c.end(), [&](auto x) { os << x << ' '; });
  os << '\n';
```

```
16  }
17
18  int
19  main()
20  {
21    const std::size_t sz = 10;
22    std::vector<int> v(sz);
23    assert(v.size() == sz);
24    print_container(std::cout, "Poczatkowa zawartosc kontenera", v);
25
26    std::generate(v.begin(), v.end(), []() {
27      return std::uniform_int_distribution<int>{ 0, 9 }(engine);
28    });
29    print_container(std::cout, "Kontener po losowaniu zawartosci", v);
30
31    if (std::any_of(v.begin(), v.end(), [](auto x) { return x % 2 == 0; })) {
32      std::cout << "W kontenerze jest co najmniej jeden element parzysty.\n";
33    } else {
34      std::cout << "W kontenerze nie ma elementow parzystych.\n";
35    }
36
37    std::cout << "Liczba elementow podzielnych przez 3: "
38              << std::count_if(
39                   v.begin(), v.end(), [](auto x) { return x % 3 == 0; })
40              << '\n';
41
42    std::cout << "Minimum: " << *std::min_element(v.begin(), v.end()) << '\n';
43    std::cout << "Maksimum: " << *std::max_element(v.begin(), v.end()) << '\n';
44
45    std::reverse(v.begin(), v.end());
46    print_container(std::cout, "Kontener po odwroceniu zawartosci", v);
47
48    std::partition(v.begin(), v.end(), [](auto x) { return x % 2 == 0; });
49    print_container(std::cout, "Kontener po partycjonowaniu (parzystosc)", v);
50
51    std::sort(v.begin(), v.end());
52    print_container(std::cout, "Kontener po sortowaniu", v);
53
54    std::shuffle(v.begin(), v.end(), engine);
55    print_container(std::cout, "Kontener po losowej zmianie kolejnosci", v);
56  }
```

Wynik działania przykładu jest następujący:

```
Poczatkowa zawartosc kontenera: 0 0 0 0 0 0 0 0 0 0
Kontener po losowaniu zawartosci: 7 4 9 2 8 8 8 2 6 2
W kontenerze jest co najmniej jeden element parzysty.
Liczba elementow podzielnych przez 3: 2
Minimum: 2
Maksimum: 9
Kontener po odwroceniu zawartosci: 2 6 2 8 8 8 2 9 4 7
Kontener po partycjonowaniu (parzystosc): 2 6 2 8 8 8 2 4 9 7
Kontener po sortowaniu: 2 2 2 4 6 7 8 8 8 9
Kontener po losowej zmianie kolejnosci: 9 8 6 2 4 8 7 8 2 2
```

std::for_each aplikuje dany obiekt funkcyjny do każdego elemenentu wskazywanego przez zakres. *std::generate* przypisuje każdemu elementowi zakresu wynik otrzymany przez dany obiekt funkcyjny. *std::all_of*, *std::any_of* oraz *std::none_of* sprawdzają czy predykat podany jako obiekt funkcyjny jest spełniony dla odpowiednio wszystkich, co najmniej jednego oraz żadnego elementu zakresu. *std::count* oraz *std::count_if* zliczają elementy zakresu odpowiednio równe danemu oraz spełniające warunek dany przez obiekt funkcyjny. *std::max_element* oraz *std::min_element* zwracają iterator (uogólniony wskaźnik) do pierszego odpowiednio największego oraz najmniejszego elementu w zakresie. *std::reverse* odwraca kolejność elementów w zakresie. *std::partition* zmienia kolejność elementów w zakresie w taki sposób, aby elementy spełniające predykat dany przez obiekt funkcyjny znalazły się przed pozostałymi elementami.

std::sort sortuje elementy w zakresie w porządku niemalejącym. W przeszłości *std::sort* był implementowany z użyciem algorytmu *Quicksort*, który jest wydajnym sposobem sortowania. Jednak dzięki postępowi w algorytmice zasadnym była zmiana implementacji *std::sort* na jeszcze wydajnieszą hybrydową metodę sortowania *Introsort*. O ile nie ma się uzasadnionego powodu to we własnych programach warto stosować *std::sort*. Przytaczany w książce

algorytm sortowania bąbelkowego ma wartość dydaktyczną, ale nie jest wydajny i dlatego nie należy go stosować w profesjonalny rozwiązaniach programistycznych.

std::shuffle zmienia kolejność elementów w zakresie w sposób losowy.

14.4 Obliczenia numeryczne

Do obliczeń numerycznych można wykorzystywać m.in. *std::valarray* (tablica wartości z operacjami element po elemencie), *std::accumulate* (suma), *std::inner_product* (iloczyn skalarny), *std::adjacent_difference* (różnica sąsiednich elementów), *std::partial_sum* (suma częściowa), *std::reduce*, *std::transform_reduce*, *<cmath>* (m.in. funkcje matematyczne), *<numbers>*, *<numeric>*, *<random>* (liczby pseudolosowe) oraz *std::complex* (liczby zespolone).

A Klasyczne problemy programistyczne rozwiązane w języku C++

A.1 Problem skoczka szachowego (rekurencja i programowanie obiektowe)

Poniżej znajduje się rozwiązanie problemu skoczka szachowego[66] z wykorzystaniem rekurencji oraz programowania obiektowego. Dla niewielkiego uproszczenia przyjęto w rozwiązaniu, że skoczek rozpoczyna swoją drogę w jednym z wierzchołków szachownicy.

Kod źródłowy 41: Problem skoczka szachowego

```
#include <cassert>
#include <cstddef>
#include <fstream>
#include <iomanip>
#include <vector>

class knights_tour_problem
{
  using row_t = std::vector<int>;
  using chessboard_t = std::vector<row_t>;
  using solutions_t = std::vector<chessboard_t>;

public:
  explicit knights_tour_problem(int size)
    : size_{ size }
  {
  }

  bool solved() const { return solved_; }

  knights_tour_problem& find_all_solutions()
  {
    if (!solved_) {
      search(1, 0, 0);
      solved_ = true;
    }
    return *this;
  }

  std::ostream& print_all_solutions(std::ostream& os) const
  {
    assert(solved());
    if (!solutions_.size()) {
      os << "\nBrak rozwiazan\n";
    } else {
      for (std::size_t i = 0; i < solutions_.size(); ++i) {
        print_solution(os << '\n', i);
      }
    }
    return os << std::flush;
  }

private:
  // Rekurencyjne rozwiazanie problemu skoczka szachowego.
```

[66] URL: https://en.wikipedia.org/wiki/Knight's_tour.

```
45      void search(int move, int i, int j) const
46      {
47        const int nd = 8;
48        const int delta_i[nd] = { -2, -2, -1, -1, +1, +1, +2, +2 };
49        const int delta_j[nd] = { -1, +1, -2, +2, -2, +2, -1, +1 };
50
51        board_[i][j] = move;
52        if (is_solution()) {
53          solutions_.push_back(board_);
54        } else {
55          for (int k = 0; k < nd; ++k)
56            if (is_on_board(i + delta_i[k], j + delta_j[k]) &&
57                !was_visited(i + delta_i[k], j + delta_j[k]))
58              search(move + 1, i + delta_i[k], j + delta_j[k]);
59        }
60        board_[i][j] = 0; // Zastosowanie techniki wycofywania (ang. backtracking).
61        return;
62      }
63
64      bool is_solution() const
65      {
66        for (int i = 0; i < size_; i++)
67          for (int j = 0; j < size_; j++)
68            if (!board_[i][j])
69              return false;
70        return true;
71      }
72
73      std::ostream& print_solution(std::ostream& os, std::size_t number) const
74      {
75        auto& board = solutions_[number];
76        os << "Rozwiazanie #" << number + 1;
77        for (int i = 0; i < size_; i++) {
78          os << '\n';
79          for (int j = 0; j < size_; j++) {
80            os << std::setw(2) << board[i][j] << ' ';
81          }
82        }
83        return os << '\n';
84      }
85
86      bool is_on_board(int pos_i, int pos_j) const
87      {
88        return pos_i >= 0 && pos_i < size_ && pos_j >= 0 && pos_j < size_;
89      }
90
91      bool was_visited(int i, int j) const { return board_[i][j] > 0; }
92
93    private:
94      int size_;
95      mutable chessboard_t board_ = { static_cast<std::size_t>(size_),
96                                      row_t(static_cast<std::size_t>(size_), 0) };
97      mutable solutions_t solutions_ = {};
98      bool solved_ = false;
99    };
100
101   std::ostream&
102   operator<<(std::ostream& os, const knights_tour_problem& rhs)
103   {
104     rhs.print_all_solutions(os);
105     return os;
106   }
107
108   int
109   main()
110   {
111     std::ofstream file{ "solutions.txt" };
```

```
    for (int i = 1; i < 6; ++i) {
      file << "> Szachownica " << i << " x " << i << std::endl
           << knights_tour_problem{ i }.find_all_solutions() << std::endl;
    }
  }
```

B Biblioteki programistyczne C++ dla fizyki

B.1 `deal.II` – metoda elementu skończonego

Biblioteka dla metody elementu skończonego stworzona oryginalnie przez Wolfganga Bangertha[67] i dostępna pod adresem https://dealii.org/.

Podziękowania

Materiał dotyczący maszyny Turinga, hipotezy Churcha-Turinga, maszyn abstrakcyjnych, języków maszynowych, klasyfikacji języków oraz styli programowania powstał dzięki lekturze książki[68] zasugerowanej przez Prof. Michała Skrzypczaka z Instytutu Informatyki Wydziału Matematyki, Informatyki i Mechaniki UW. Przykład z kodu źródłowego nr 2 został zainspirowany zajęciami *Symulacje komputerowe w fizyce* u Prof. Piotra Szymczaka oraz Prof. Jakuba Tworzydło z Instytutu Fizyki Teoretycznej Wydziału Fizyki UW. Wszystkie błędy są oczywiście mojego autorstwa.

Bibliografia

[1] Maciej Ślusarek, Przemysław Broniek, Grzegorz Gutowski i Jan Jeżabek. *Złożoność obliczeniowa*. URL: https://wazniak.mimuw.edu.pl/.

[2] Ray Toal. *Classifying Programming Languages*. URL: https://cs.lmu.edu/~ray/notes/pltypes/.

[3] Bjarne Stroustrup. "A History of C++: 1979–1991". W: *History of Programming Languages—II* (1996), s. 699. DOI: 10.1145/234286.1057836.

[4] URL: https://en.cppreference.com/w/cpp/language/history.

[5] URL: https://cppreference.com/.

[6] URL: https://cppcon.org/.

[7] URL: https://www.youtube.com/CppCon.

[8] *The Definitive C++ Book Guide and List*. URL: https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list/.

[9] Bjarne Stroustrup. *Bjarne Stroustrup's homepage*. URL: https://stroustrup.com/.

[10] Bjarne Stroustrup. *Programming—Principles and Practice Using C++*. 3rd Edition. https://stroustrup.com/programming.html. Addison-Wesley, 2024.

[11] Herb Sutter. *Sutter's Mill. Herb Sutter on software development*. URL: https://herbsutter.com/.

[12] Scott Meyers. URL: https://www.aristeia.com/.

[13] Tomasz R. Werner. *Programowanie 2. Język C++*. URL: https://www.fuw.edu.pl/~werner/pmn/CPP_HTML/CPP.html.

[14] URL: https://en.wikibooks.org/wiki/C%2B%2B_Programming.

[15] URL: https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms.

[16] URL: https://en.cppreference.com/w/cpp/language/statements.

[17] URL: https://en.cppreference.com/w/cpp/language/expressions.

[18] URL: https://en.cppreference.com/w/cpp/language/if.

[19] URL: https://en.cppreference.com/w/cpp/language/switch.

[20] URL: https://en.cppreference.com/w/cpp/language/declarations.

[21] URL: https://en.cppreference.com/w/cpp/language/types.

[22] URL: https://en.cppreference.com/w/cpp/language/cv.

[67] Wolfgang Bangerth. *MATH 676: Finite element methods in scientific computing*. URL: https://www.math.colostate.edu/~bangerth/videos.html.

[68] Maurizio Gabbrielli i Simone Martini. *Programming Languages: Principles and Paradigms*. Springer-Verlag London Limited, 2010. DOI: 10.1007/978-1-84882-914-5.

[23] URL: https://en.cppreference.com/w/cpp/container/vector.
[24] URL: https://en.cppreference.com/w/cpp/language/functions.
[25] URL: https://en.cppreference.com/w/cpp/language/function.
[26] URL: https://en.cppreference.com/w/cpp/language/scope.
[27] URL: https://en.cppreference.com/w/cpp/language/pointer.
[28] URL: https://en.cppreference.com/w/cpp/language/reference.
[29] URL: https://en.cppreference.com/w/cpp/language/classes.
[30] URL: https://en.cppreference.com/w/cpp/language/class.
[31] URL: https://en.cppreference.com/w/cpp/language/access.
[32] URL: https://en.cppreference.com/w/cpp/language/constructor.
[33] URL: https://en.cppreference.com/w/cpp/language/member_functions.
[34] URL: https://en.cppreference.com/w/cpp/language/destructor.
[35] URL: https://en.cppreference.com/w/cpp/language/this.
[36] URL: https://en.cppreference.com/w/cpp/language/friend.
[37] URL: https://en.cppreference.com/w/cpp/header/fstream.
[38] URL: https://en.cppreference.com/w/cpp/io/manip/skipws.
[39] URL: https://en.cppreference.com/w/cpp/chrono/c/time.
[40] URL: https://en.cppreference.com/w/cpp/io/ios_base/openmode.
[41] URL: https://stackoverflow.com/questions/98650/what-is-the-strict-aliasing-rule.
[42] URL: https://en.cppreference.com/w/cpp/io/basic_istream/tellg.
[43] URL: https://en.cppreference.com/w/cpp/io/basic_ostream/tellp.
[44] URL: https://en.cppreference.com/w/cpp/io/basic_istream/seekg.
[45] URL: https://en.cppreference.com/w/cpp/io/basic_ostream/seekp.
[46] URL: https://en.cppreference.com/w/cpp/language/templates.
[47] URL: https://en.cppreference.com/w/cpp/language/function_template.
[48] URL: https://en.cppreference.com/w/cpp/language/class_template.
[49] URL: https://en.cppreference.com/w/cpp/error/assert.
[50] David Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". W: *ACM Computing Surveys* 23 (1991), s. 5–48. DOI: 10.1145/103162.103163.
[51] URL: https://en.cppreference.com/w/cpp/language/lambda.
[52] *CS 3110—Data Structures and Functional Programming. Fall 2008*. 2008. URL: https://www.cs.cornell.edu/courses/cs3110/2008fa/recitations/rec26.html.
[53] *Standard Template Library Programmer's Guide*. URL: https://www.boost.org/sgi/stl/index.html.
[54] URL: https://en.cppreference.com/w/cpp/algorithm.
[55] URL: https://en.cppreference.com/w/cpp/iterator.
[56] URL: https://en.cppreference.com/w/cpp/concepts.
[57] URL: https://en.cppreference.com/w/cpp/container.
[58] URL: https://en.cppreference.com/w/cpp/locale.
[59] URL: https://en.cppreference.com/w/cpp/meta.
[60] URL: https://en.cppreference.com/w/cpp/string.
[61] URL: https://en.cppreference.com/w/cpp/utility.
[62] URL: https://en.cppreference.com/w/cpp/numeric.
[63] URL: https://en.cppreference.com/w/cpp/error.
[64] URL: https://en.cppreference.com/w/cpp/filesystem.
[65] URL: https://en.cppreference.com/w/cpp/io.
[66] URL: https://en.cppreference.com/w/cpp/thread.
[67] URL: https://en.cppreference.com/w/cpp/regex.

[68] URL: https://en.cppreference.com/w/cpp/ranges.

[69] URL: https://en.cppreference.com/w/cpp/container/set.

[70] URL: https://en.wikipedia.org/wiki/Knight's_tour.

[71] Wolfgang Bangerth. *MATH 676: Finite element methods in scientific computing*. URL: https://www.math.colostate.edu/~bangerth/videos.html.

[72] Maurizio Gabbrielli i Simone Martini. *Programming Languages: Principles and Paradigms*. Springer-Verlag London Limited, 2010. DOI: 10.1007/978-1-84882-914-5.