Marek Gągolewski

Algorytmy i podstawy programowania w języku C++

v1.2.1, 2022-08-03

Dr hab. inż. Marek Gągolewski
Wydział Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
https://www.gagolewski.com

Warszawa 2012 [v1.0], 2016 [v1.1]; Melbourne 2022 [v1.2]

Podziękowania: Maciej Bartoszuk, Piotr Bródka, Anna Cena, Michał Dębski, Sławomir Kwasiborski, Piotr Sapiecha, Małgorzata Śleszyńska-Nowak, Barbara Żogała-Siudem.

Kody źródłowe: https://github.com/gagolews/aipp
ISBN: xxx-xx-xxxxxx-x-x (PDF v1.2)

Spis treści

Etapy tworzenia oprogramowania. Algorytm

1

Science is what we understand well enough to explain to a computer.
Art is everything else we do.
Donald Knuth

1.1 Wprowadzenie

Przygodę z algorytmami i programowaniem czas zacząć! W trakcie naszych zajęć poznasz wiele nowych, ciekawych zagadnień. Celem, który chcemy pomóc Ci osiągnąć, jest nie tylko bliższe poznanie komputera (jak wiadomo, bliskość sprzyja nawiązywaniu przyjaźni, często na całe życie), ale i rozwijanie umiejętności rozwiązywania problemów, także matematycznych.

Opanowanie umiejętności programowania komputerów wcale nie będzie atutem w życiu zawodowym. Będzie warunkiem koniecznym Twojego sukcesu. Komputery bowiem towarzyszą nam na (prawie) każdym kroku, są nieodłącznym elementem współczesnego świata.

Ważne
Nasz kurs łączy teorię (algorytmy) z praktyką (programowanie). Pamiętaj, że nie da się nabrać biegłości w żadnej dziedzinie życia bez systematycznych ćwiczeń. Praktyka mistrza czyni.

1.2 Etapy tworzenia oprogramowania

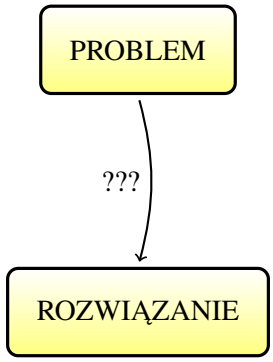
1.2.1 Sformułowanie i analiza problemu

1.2.1.1 Sformułowanie problemu

Punktem wyjścia naszych rozważań niech będzie pewne **zagadnienie problemowe**. Problem ten może być bardzo ogólnej natury, np. numerycznej, logicznej, społecznej czy nawet egzystencjalnej. Oto kilka przykładów:

- obliczenie wartości pewnego wyrażenia arytmetycznego,
- znalezienie rozwiązania układu 10000 równań różniczkowych,
- zaplanowanie najkrótszej trasy podróży od miasta X do miasta Y,
- postawienie diagnozy medycznej na podstawie listy objawów chorobowych,
- rozpoznanie twarzy przyjaciół na zdjęciu,
- dokonanie predykcji wartości indeksu giełdowego,
- zaliczenie (na piątkę, rzecz jasna) niniejszego przedmiotu,
- rozwiązanie dylematu filozoficznego, np. czym jest szczęście i jak być szczęśliwym?

Znalezienie **rozwiązania** danego problemu jest dla nas często — z różnych względów — bardzo istotne. Kluczowym pytaniem jest tylko: jak to zrobić? Sytuację tę obrazuje rys. 1.1. Innymi słowy, interesuje nas, w jaki sposób *dojść* do celu?



Rysunek 1.1. Punkt wyjścia naszych rozważań.

1.2.1.2 Analiza problemu

Po sformułowaniu problemu, czyli stwierdzeniu, że odczuwamy potrzebę uzyskania odpowiedzi na postawione pytanie, przechodzimy do etapu jego **analizy**. Tutaj doprecyzowujemy, o co nam naprawdę chodzi, co rozumiemy pod pewnymi pojęciami, jakich wyników się spodziewamy i do czego ewentualnie mogą się nam one przydać w przyszłości.

W przypadku pewnych zagadnień (np. matematycznych) zadanie czasem wydaje się względnie proste. Wszystkie pojęcia mają swoją definicję formalną, można udowodnić, że pewne kroki prowadzą do spodziewanych, jednoznacznych wyników itd.

Jednakże niektóre zagadnienia mogą przytłaczać swoją złożonością. Na przykład „zaliczenie tego przedmiotu" wymaga określenia, jaki stan końcowy jest pożądany (ocena bardzo dobra?), jakie czynności są kluczowe do osiągnięcia rozwiązania (udział w ćwiczeniach i laboratoriach, słuchanie wykładu, zadawanie pytań, dyskusje na konsultacjach), jakie czynniki mogą wpływać na powodzenie na poszczególnych etapach nauki (czytanie książek, wspólna nauka?), a jakie je wręcz uniemożliwiać (codzienne imprezy? brak prądu w akademiku? popsuty komputer?).

Komputery. Istotną cechą wielu zagadnień jest to, że nadają się do rozwiązania za pomocą *komputera*. Jak pokazuje rozwój współczesnej informatyki, tego typu problemów jest wcale niemało. W innych przypadkach często mamy do czynienia z sytuacją, w której komputery mogą nam pomóc uzyskać **rozwiązanie częściowe** lub zgrubne przybliżenie rozwiązania. Może to być lepsze niż zupełny brak rozwiązania.

Często słyszymy o niesamowitych „osiągnięciach" komputerów, np. wygraniu w szachy z mistrzem świata, uczestnictwie w pełnym podchwytliwych pytań teleturnieju *Jeopardy!* (pierwowzór niegdyś emitowanego w Polsce *Va Banque*), samodzielnym sterowaniu pojazdem kosmicznym, wirtualną obsługą klienta w banku itp. Wyobraźnię o ich potędze podsycają opowiadania *science-fiction*, których autorzy przepowiadają, że te maszyny — pewnego dnia — będą potrafiły zrobić prawie wszystko, co jest do pomyślenia.

Niestety, z drugiej strony komputery podlegają trzem bardzo istotnym ograniczeniom. By łatwiej można było je sobie uzmysłowić, posłużymy się analogią z dziedziny motoryzacji.

1 Komputer ma *ograniczoną moc obliczeniową*. Każda instrukcja wykonuje się przez pewien czas. Im bardziej złożone zadanie, tym jego rozwiązywanie trwa dłużej. Mimo że stan rzeczy poprawia się wraz z rozwojem technologii, zawsze będziemy ograniczeni zasadami fizyki.

Samochody mają np. ograniczoną prędkość, ograniczone przyspieszenie (także zasadami fizyki).

2 Komputer „rozumie” określony *język* (języki), do którego **syntaktyki** (składni) trzeba się dostosować, którego konstrukcję trzeba poznać, by móc się z nim „dogadać”. Języki są zdefiniowane formalnie za pomocą ściśle określonej gramatyki. Nie toleruje on najczęściej żadnych od niej odstępstw lub toleruje tylko nieliczne z nich.

Aby zwiększyć liczbę obrotów silnika, należy wykonać jedną ściśle określoną czynność — wcisnąć mocniej pedał gazu. Nic nie da uśmiechnięcie się lub próba sympatycznej konwersacji z deską rozdzielczą na temat zalet jazdy z inną prędkością.

3 Komputer ograniczony jest ponadto przez tzw. *czynnik ludzki*. Potrafi zrobić tylko to (i aż tyle), co sami dokładnie powiemy, co ma zrobić, krok po kroku, instrukcja po instrukcji. Nie domyśli się, co tak naprawdę nam chodzi po głowie. Każdy wydawany rozkaz ma bowiem określoną **semantykę** (znaczenie). Komputer jedynie potrafi go posłusznie wykonać.

Samochód zawiezie nas, gdzie chcemy, jeśli będziemy odpowiednio posługiwać się kierownicą i innymi przyrządami. Nie będzie protestował, gdy skręcimy nie na tym skrzyżowaniu, co trzeba. Albo gdy wjedziemy na drzewo.

Ważne

Celem przewodnim naszych rozważań w tym semestrze jest więc takie poznanie natury i *języka* komputerów, by mogły zrobić *dokładnie* to, co *my* chcemy.

Na początek, dla porządku, przedyskutujemy definicję obiektu naszych zainteresowań. Otóż słowo **komputer** pochodzi od łacińskiego czasownika *computare*, który oznacza *obliczać*. Jednak powiedzenie, że komputer zajmuje się tylko obliczaniem, to stanowczo za wąskie spojrzenie.

Informacja

Komputer to *programowalne* urządzenie *elektroniczne* służące do *przetwarzania informacji*.

Aż cztery słowa w tej definicji wymagają wyjaśnienia. **Programowalność** to omówiona powyżej zdolność do przyjmowania, interpretowania i wykonywania wydawanych poleceń zgodnie z zasadami syntaktyki i semantyki używanego języka.

Po drugie, pomimo licznych eksperymentów przeprowadzanych m.in. przez biologów molekularnych i fizyków kwantowych, współczesne komputery to w znakomitej większości maszyny **elektroniczne**. Ich „wnętrzności" są w swej naturze więc dość nieskomplikowane (ot, kilkaset milionów tranzystorów upakowanych na płytce drukowanej). O implikacjach tego faktu dowiemy się więcej z wykładu poświęconego organizacji i działaniu tych urządzeń.

Dalej, jednostkę **informacji** rozumiemy jako *ciąg liczb lub symboli* (być może jednoelementowy). Oto kilka przykładów: (6) (liczba naturalna), (*PRAWDA*) (wartość logiczna), (”STUDENT MiNI”) (napis, czyli ciąg znaków drukowanych), (3,14159) (liczba rzeczywista), (4,-3,-6,34) (ciąg liczb całkowitych), (011100) (liczba w systemie binarnym), (”WARSZAWA”, 52°13’56”N, 21°00’30”E) (współrzędne GPS pewnego miejsca na mapie).

Ciekawostka

Wiele obiektów spotykanych na co dzień może mieć swoje **reprezentacje** w postaci ciągów liczb lub symboli. Często te reprezentacje nie muszą odzwierciedlać wszystkich cech opisywanego obiektu, lecz tylko te, które są potrzebne w danym zagadnieniu. Ciąg (”Jola Kowalska”, ”Matematyka II rok”, 4,92, 21142) może być wystarczającą informacją dla pani z dziekanatu, by przyznać pewnej studentce stypendium naukowe. W tym przypadku kolor oczu Joli lub jej pasje muzyczne, to zbędne (miejmy nadzieje) szczegóły.

I wreszcie, **przetwarzanie** informacji to wykonywanie różnych działań na liczbach (np. operacji arytmetycznych, porównań) lub symbolach (np. zamiana elementów, łączenie ciągów). Rodzaje wykonywanych operacji są ściśle określone. Nie znaczy to jednak, że nie można próbować samodzielnie tworzyć nowych na podstawie tych, które są już dostępne.

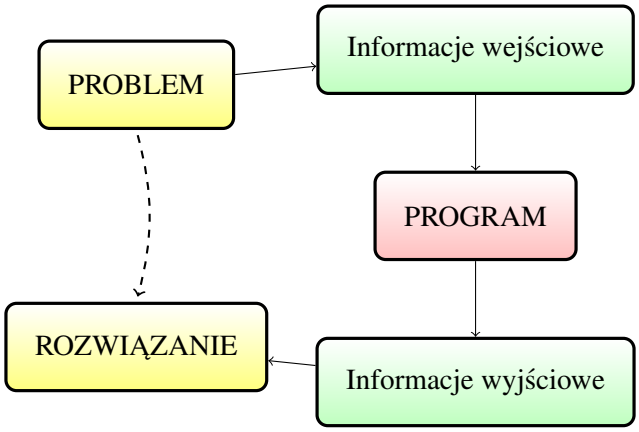
Jako przykładowe operacje przetwarzające odpowiednio: liczby naturalne, wartości logiczne i napisy, możemy przytoczyć:

$$\begin{array}{rcc} 2+2 & \rightarrow & 4 \\ \pi < e & \rightarrow & \textit{FAŁSZ} \\ \text{”KATA”} \oplus \text{(”STREFA”/(E} \rightsquigarrow \text{O))} & \rightarrow & \text{”KATASTROFA”} \end{array}$$

Nietrudno odgadnąć ich znaczenie. Co ważne: właśnie tak działa komputer!

Ważne

Ciąg operacji, które potrafi wykonać komputer, nazywamy **programem komputerowym**.



Rysunek 1.2. Rozwiązanie problemu przy użyciu programu komputerowego.

Dokonajmy syntezy naszych dotychczasowych rozważań. Otóż **problem, który da się rozwiązać za pomocą komputera** posiada dwie istotne cechy (por. rys. 1.2):

a) daje się wyrazić w postaci pewnych danych (informacji) wejściowych,
b) istnieje dla niego program komputerowy, przetwarzający dane wejściowe w taki sposób, że informacje wyjściowe mogą zostać przyjęte jako reprezentacja poszukiwanego rozwiązania.

1.2.2 Projektowanie

Po etapie analizy problemu następuje etap **projektowania algorytmów**.

Ważne

Algorytm to abstrakcyjny przepis (proces, metoda, „instrukcja obsługi") pozwalający na uzyskanie, przy użyciu *skończonej* liczby działań, oczekiwanych danych wyjściowych na podstawie poprawnych danych wejściowych.

Przykładowym algorytmem „z życia" jest przepis na sernik na zimno,[1] przedstawiony w tab. 1.1.

[1] Przepis ten pochodzi z serwisu Kwestia Smaku:
https://www.kwestiasmaku.com/desery/serniki/przepisy.html.

Tabela 1.1. Przepis na sernik na zimno.

Dane wejściowe: 225 g (1 opakowanie) herbatników Digestive; 100 g masła, roztopionego; 3 czubate łyżeczki żelatyny w proszku; 300 ml śmietanki kremówki 30%, bardzo zimnej; 4 czubate łyżki cukru pudru; 600 g serka mascarpone najlepiej włoskiego, schłodzonego; 1 biała czekolada, roztopiona w kąpieli wodnej lub mikrofalówce; 1 szklanka malin (zamrożonych lub świeżych), ewentualnie innych ulubionych owoców np. truskawek; 1 galaretka o smaku owoców leśnych.

Algorytm:

a) Formę o średnicy 23 cm z odpinaną obręczą wysmarować tłuszczem i wyłożyć przezroczystą folią. Herbatniki drobno pokruszyć i wymieszać z roztopionym masłem. Wyłożyć je na przygotowany spód formy. Żelatynę namoczyć w kilku łyżkach zimnej wody.
b) Ubić śmietankę kremową na średnio sztywną pianę, następnie dodać cukier puder i ubić na sztywną pianę.
c) Dodawać po jednej czubatej łyżce (w dość krótkich odstępach czasowych) serka mascarpone, wciąż ubijając na gęstą masę. Namoczoną żelatynę rozpuścić szybko mieszając w około 1/4 szklanki gorącej wody.
d) Masę serową zmiksować z roztopioną czekoladą, a następnie z rozpuszczoną, ostudzoną ale lekko ciepłą żelatyną. Masę wyłożyć na spód z herbatników i wstawić do lodówki. Galaretkę rozpuścić we wrzącej wodzie, w ilości podanej na opakowaniu galaretki, ale odejmując 1/2 szklanki wody. Ostudzić i wstawić do lodówki.
e) Zamrożone lub świeże maliny ułożyć na masie serowej i zalać tężejącą galaretką. Wstawić do lodówki do całkowitego stężenia na kilka godzin lub najlepiej na całą noc. Sernik wyjąć na 15 minut przed podaniem.

Dane wyjściowe: Pyszny sernik.

Widzimy, jak wygląda *zapisany* algorytm. Ważną umiejętnością, którą będziemy ćwiczyć, jest odpowiednie jego *przeczytanie*. Dobrze to opisał Prof. Donald Knuth (2002, s. 4):

> *Na wstępie trzeba jasno powiedzieć, że algorytmy to nie beletrystyka. Nie należy czytać ich ciurkiem. Z algorytmem jest tak, że jak nie zobaczysz, to nie uwierzysz. Najlepszą metodą poznania algorytmu jest wypróbowanie go.*

A zatem — do kuchni!

Informacja

Oto najistotniejsze **cechy algorytmu** (por. Knuth, 2002, s. 4):

- skończoność — wykonanie algorytmu musi zatrzymać się po skończonej liczbie kroków;
- dobre zdefiniowanie — każdy krok algorytmu musi być opisany precyzyjne, ściśle i jednoznacznie, tj. być sformułowanym na takim poziomie ogólności, by każdy, kto będzie go czytał, był w stanie zrozumieć, jak go wykonać;
- efektywność — w algorytmie nie ma operacji niepotrzebnie wydłużających czas wykonania;
- dane wejściowe są ściśle określone, pochodzą z dobrze określonych zbiorów;
- dane wyjściowe, czyli wartości powiązane z danymi wejściowymi, odpowiadają specyfikacji oczekiwanego poprawnego rozwiązania.

Wygląda na to, że nasz przepis na sernik posiada wszystkie wyżej wymienione cechy. Wykonując powyższe czynności, ciasto to uda nam się kiedyś zjeść (skończoność). Wszystkie czynności są zrozumiałe nawet dla mało wprawionego kucharza. Sam proces przygotowania jest efektywny (nie każe kucharce np. umalować się w trakcie mieszania składników lub pojechać w międzyczasie na zagraniczną wycieczkę). Dane wejściowe i wyjściowe są dobrze określone.

Ważne

Jeden program rozwiązujący rozpatrywany problem może zawierać realizację **wielu algorytmów**, np. gdy złożoność zagadnienia wymaga podzielenia go na kilka **podproblemów**.

I tak zdolny kucharz wykonujący program „obiad” powinien podzielić swą pracę na podprogramy „rosół”, „kotlet schabowy z frytkami” oraz „sernik” i skupić się najpierw na projektowaniu podzadań.

Ważne jest, że algorytm nie musi być wyrażony przy użyciu języka zrozumiałego dla komputera. Taki sposób opisu algorytmów nazywa się często **pseudokodem**. Stanowi on etap pośredni między analizą problemu a implementacją, opisaną w kolejnym paragrafie. Pseudokod ma po prostu pomóc w procesie formalizowania programu.

Na przykład, w przytoczonym powyżej przepisie nie jest dokładnie wytłumaczone — krok po kroku — co oznacza „ubić na sztywną pianę”. Wszak zadanie to wymaga wielu czynności (np. odpowiednie dobranie mieszadeł w mikserze, ustawienie odpowiedniej prędkości obrotowej itp.). Jednakże, jak już wspomnieliśmy, czynność ta jest dobrze zdefiniowana.

Jakby tego było mało, może istnieć wiele algorytmów służących do rozwiązania tego samego problemu. Przyjrzyjmy się następującemu przykładowi.

1.2.2.1 Przykład: Problem młodego Gaussa

Szeroko znana jest historia Karola Gaussa, z którym miał problemy jego nauczyciel matematyki. Aby zająć czymś młodego chłopca, profesor kazał mu wyznaczyć sumę liczb $a, a+1, \ldots, b$, gdzie $a, b \in \mathbb{N}$ (jak głosi historia, było to $a = 1$ i $b = 100$). Zapewne nauczyciel myślał, że tamten użyje algorytmu I i spędzi niemałą ilość czasu na tej rozrywce.

Algorytm I wyznaczania sumy kolejnych liczb naturalnych.

```
// Input: a, b ∈ ℕ (a < b)
// Output: a + a + 1 + ... + b ∈ ℕ

let sum, i ∈ ℕ;
sum = 0;

for (i = a, a + 1, ..., b)
{
    sum = sum + i;
}
return sum;
```

Jednakże sprytny Gauss zauważył, że $a + a + 1 + \cdots + b = \frac{a+b}{2}(b - a + 1)$. Dzięki temu wyznaczył bardzo szybko wartość rozwiązania, korzystając z algorytmu II.

Algorytm II wyznaczania sumy kolejnych liczb naturalnych.

```
// Input: a, b ∈ ℕ (a < b)
// Output: a + a + 1 + ... + b ∈ ℕ

let sum ∈ ℕ;
sum = (a+b)/2 (b − a + 1);
return sum;
```

Zauważmy, że **obydwa algorytmy rozwiązują poprawnie to samo zagadnienie**. Mimo to inna jest liczba operacji arytmetycznych $(+, -, *, /)$ potrzebna do uzyskania oczekiwanego wyniku. Poniższa tabelka zestawia tę miarę efektywności dla obydwu rozwiązań oraz różnych danych wejściowych. Zauważmy, że w przypadku pierwszego algorytmu liczba wykonywanych operacji dodawania jest równa $(b - a + 1)$ [instrukcja sum = sum + i] + $(b - a)$ [dodawanie występuje także w pętli **for**...].

Tabela 1.2. Liczba operacji arytmetycznych potrzebna do znalezienia rozwiązania problemu młodego Gaussa.

a	b	Alg. I	Alg. II
1	10	19	5
1	100	199	5
1	1000	1999	5

Ciekawostka
Badaniem efektywności algorytmów zajmuje się dziedzina zwana **analizą algorytmów**. Czerpie ona obficie z wyników teoretycznych takich działów matematyki jak matematyka dyskretna oraz rachunek prawdopodobieństwa. Z jej elementami zapoznamy się podczas innych wykładów.

1.2.3 Implementacja

Na kolejnym etapie abstrakcyjne algorytmy, zapisane często w postaci pseudokodu (np. w języku polskim), należy przepisać w formie, która jesteśmy w stanie zrozumieć my i komputer. Innymi słowy, jest to tzw. **implementacja** algorytmów.

Na tym etapie wreszcie tłumaczymy komputerowi, co dokładnie chcemy, by zrobił, tzn. go *programujemy*. Efektem naszej pracy będzie **kod źródłowy** programu.

Ponadto, jeśli zachodzi taka potrzeba, na tym etapie należy dokonać scalenia (powiązania) podzadań („rosół", „kotlet schabowy z frytkami" oraz „sernik") w jeden spójny projekt („obiad").

Ważne
Formalnie rzecz ujmując, zbiór zasad określających, jaki ciąg symboli tworzy kod źródłowy programu, nazywamy **językiem programowania**.

Reguły składniowe (ang. *syntax*) języka ściśle określają, które (i tylko które) wyrażenia są poprawne. **Reguły znaczeniowe** (ang. *semantics*) określają precyzyjnie, jak komputer ma rozumieć dane wyrażenia. Dziedziną zajmującą się analizą języków programowania, jest *lingwistyka matematyczna*.

W trakcie naszych zajęć będziemy poznawać podstawy **języka C++**, zaprojektowanego ok. 1983 r. przez B. Stroustrupa (aktualny standard: ISO/IEC 14882:2003). Należy pamiętać, że **C++** jest tylko jednym z wielu (naprawdę wielu) sposobów, w jakie można wydawać polecenia komputerowi.

Ciekawostka
Język **C++** powstał jako rozwinięcie języka C. Wśród innych podobnych doń składniowo języków można wymienić takie popularne narzędzia jak Python, R, Java, C#, PHP czy JavaScript.

Język **C++** wyróżnia się zwięzłością składni, względną łatwością nauki i efektywnością generowanego kodu wynikowego. Jest ponadto jednym z najbardziej popularnych języków ogólnego zastosowania.

Kod źródłowy programu zapisujemy zwykle w postaci kilku plików tekstowych (tzw. plików źródłowych, ang. *source files*). Pliki te można edytować przy użyciu dowolnego edytora tekstowego, np. *Notatnika*. Jednak do tego celu lepiej przydają się środowiska programistyczne. Na przykład my podczas laboratoriów będziemy używać *Microsoft Visual **C++***.

Ważne
Narzędziem, które pozwala przetworzyć pliki źródłowe (zrozumiałe dla człowieka i komputera) na kod maszynowy programu komputerowego (zrozumiały tylko dla komputera) nazywamy **kompilatorem** (ang. *compiler*).

Zanotujmy, że środowisko Visual **C++** posiada zintegrowany (wbudowany) kompilator tego języka.

1.2.4 Testowanie

Gotowy program należy **przetestować**, to znaczy sprawdzić, czy dokładnie robi to, o co nam chodziło. Jest to, niestety, często najbardziej żmudny etap tworzenia oprogramowania. Jeśli coś nie działa jak powinno, należy wrócić do któregoś z poprzednich etapów i naprawić błędy.

Warto zwrócić uwagę, że przyczyn niepoprawnego działania należy zawsze szukać w swojej omylności. Jak powiedzieliśmy, komputer robi tylko to, co mu każemy. Zatem jeśli nakazaliśmy wykonać instrukcję, której skutków ubocznych nie jesteśmy do końca pewni, nasza to odpowiedzialność, by skutki te opanować.

1.2.5 Eksploatacja

Gdy program jest przetestowany, może służyć do rozwiązywania wyjściowego problemu (wyjściowych problemów). Czasem zdarza się, że na tym etapie dochodzimy do wniosku, iż czegoś nam brakuje lub że nie jest to do końca, o co nam na początku chodziło. Wtedy oczywiście pozostaje powrót do wcześniejszych etapów pracy.

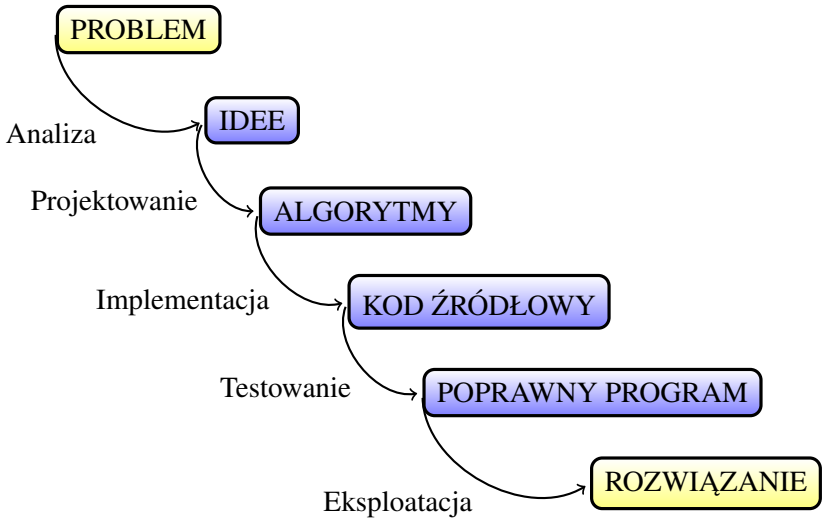
Jak widać, nauka programowania komputerów może nam pomóc rozwiązać wiele problemów, które inaczej wcale nie byłyby dla nas dostępne. Poza tym jest wspaniałą rozrywką.

1.2.6 Podsumowanie

Omówiliśmy następujące etapy tworzenia oprogramowania, które są niezbędne do rozwiązania zagadnień przy użyciu komputera:

- sformułowanie i analiza problemu,
- projektowanie,
- implementacja,
- testowanie,
- eksploatacja.

Efekty pracy po każdym z etapów podsumowuje rys. 1.3.



Rysunek 1.3. Efekty pracy po każdym z etapów tworzenia oprogramowania.

Godne polecenia rozwinięcie materiału omawianego w tej części skryptu można znaleźć w książce Harela (2001, s. 9–31).

Na zakończenie przytoczmy fragment książki F.P. Brooksa (*Mityczny osobomiesiąc. Eseje o inżynierii oprogramowania*, wyd. WNT). Programowanie przynosi wiele radości;

> *Pierwsza to zwyczajna rozkosz tworzenia czegoś. Jak dziecko lubi stawiać domki z piasku, tak dorosły lubi budować coś, zwłaszcza według własnego projektu. [...] Druga to przyjemność robienia czegoś przydatnego dla innych ludzi. W głębi duszy chcemy, żeby inni korzystali z naszej pracy i*

uznawali ją za użyteczną. [...] Trzecia przyjemność to fascynacja tworzenia złożonych przedmiotów, przypominających łamigłówki składające się z zazębiających się, ruchomych części, i obserwowanie ich działania [...]. Czwarta przyjemność wypływa z ciągłego uczenia się i wiąże się z niepowtarzalnością istoty zadania. W taki czy inny sposób problem jest wciąż nowy, a ten, kto go rozwiązuje, czegoś się uczy [...]

Brooks twierdzi także, że to, co przynosi radość, czasem musi powodować ból:
- należy działać perfekcyjnie,
- poprawianie i testowanie programu jest uciążliwe,

a także, w przypadku aplikacji komercyjnych:
- ktoś inny ustala cele, dostarcza wymagań; jesteśmy zależni od innych osób,
- należy mieć świadomość, że program, nad którym pracowało się tak długo, w chwili ukończenia najczęściej okazuje się już przestarzały.

1.3 Ćwiczenia

Zadanie 1.1. Rozważ algorytm Euklidesa, który wyznacza największy wspólny dzielnik (NWD) dwóch liczb $a, b \in \mathbb{Z}$, $0 \leqslant a < b$. Oto jego pseudokod.

```
1  //   Input: 0 ⩽ a < b
2  //   Output: NWD(a, b)
3  let c ∈ ℕ;
4  while (a ≠ 0)
5  {
6    c = b modulo a (division remainder);
7    b = a;
8    a = c;
9  }
10 return b;
```

Prześledź działanie algorytmu Euklidesa (jakie wartości przyjmują zmienne a, b, c w każdym kroku) dla następujących par liczb: a) 42, 56; b) 192, 348; c) 199, 544; d) 2166, 6099.

Zadanie 1.2. Pokaż, jak można przestawić wartości dwóch zmiennych (a, b) przy użyciu ciągu kilku przypisań tak by otrzymać (b, a).

Zadanie 1.3. Pokaż, jak można przestawić wartości trzech zmiennych (a, b, c) przy użyciu ciągu kilku przypisań tak by otrzymać (c, a, b).

Zadanie 1.4. Pokaż, jak można przestawić wartości czterech zmiennych (a, b, c, d) przy użyciu ciągu kilku przypisań tak by otrzymać (c, d, b, a).

Zadanie 1.5. Dany jest ciąg n liczb rzeczywistych $\mathbf{x} = (x[0], \ldots, x[n-1])$ (umawiamy się, że elementy ciągów numerujemy od 0). Rozważmy ich średnią arytmetyczną:

$$\frac{1}{n}\sum_{i=0}^{n-1} x[i] = \frac{1}{n}\left(x[0] + x[1] + \cdots + x[n-1]\right).$$

Wyznacz za pomocą poniższego algorytmu wartość średniej arytmetycznej dla ciągów $(1, -1, 2, 0, -2)$ oraz $(34, 2, -3, 4, 3.5)$.

```
// Input: n > 0, x[0], x[1], ..., x[n-1] ∈ ℝ
// Output: srednia arytmetyczna x[0], x[1], ..., x[n-1])
let m ∈ ℝ;
let i ∈ ℕ;
m = 0;
for (i = 0, 1, ..., n-1)   // to jest pseudokod, nie C++
    m = m + x[i];
m = m / n;
return m;
```

Zadanie 1.6. Dany jest ciąg n dodatnich liczb rzeczywistych $\mathbf{x} = (x[0], \ldots, x[n-1])$ (różnych od 0). Napisz algorytm, który wyznaczy ich średnią harmoniczną:

$$\frac{n}{\sum_{i=0}^{n-1} \frac{1}{x[i]}} = \frac{n}{1/x[0] + 1/x[1] + \cdots + 1/x[n-1]}.$$

Wyznacz za pomocą tego algorytmu wartość średniej harmonicznej dla ciągów $(1, 4, 2, 3, 1)$ oraz $(10, 2, 3, 4)$.

(*) Zadanie 1.7. Dany jest ciąg n liczb rzeczywistych $\mathbf{x} = (x[0], x[1], \ldots, x[n-1])$. Rozważmy tzw. sumę kwadratów odchyleń elementów od ich średniej arytmetycznej:

$$\mathrm{SKO}(\mathbf{x}) = \sum_{i=0}^{n-1}\left(x[i] - \left(\frac{1}{n}\sum_{j=0}^{n-1} x[j]\right)\right)^2.$$

Rozważmy następujący algorytm służący do wyznaczania SKO.

```
// Input: n > 0, x[0], x[1], ..., x[n-1] ∈ ℝ
// Output: SKO(x[0], x[1], ..., x[n-1])
let sko, m ∈ ℝ;
let i, j ∈ ℕ;

sko = 0;
for (i = 0, 1, ..., n-1)
{
```

```
    m = 0;
    dla (j = 0, 1, ..., n-1)
        m = m + x[j];
    m = m / n;
    sko = sko + (x[i] - m)*(x[i] - m);
}
return sko;
```

a) Wyznacz wartość SKO dla $\mathbf{x} = (5, 3, -1, 7, -2)$.
b) Policz, ile łącznie operacji arytmetycznych $(+, -, *, /)$ potrzebnych jest do wyznaczenia SKO dla ciągów wejściowych o $n = 5, 10, 100, 1000, 10000$ elementach. Wyraź tę liczbę jako funkcję długości ciągu wejściowego n.
c) Zastanów się, jak usprawnić powyższy algorytm, by nie wykonywać wielokrotnie zbędnych obliczeń. Ile operacji arytmetycznych będzie teraz potrzebnych?

Zadanie 1.8. Pokaż, w jaki sposób przy użyciu ciągu przypisań i podstawowych operacji arytmetycznych, mając na wejściu ciąg zmiennych (a, b, c), można otrzymać a) $(b, c + 10, a/b)$; b) $(a + b, c^2, a + b + c)$; c) $(a - b, -a, c - a + b)$. Postaraj się zminimalizować liczbę użytych instrukcji i zmiennych pomocniczych.

(*) Zadanie 1.9. „Szklana pułapka". Masz do dyspozycji pojemniki na wodę o objętości x i y litrów oraz dowolną ilość wody w basenie. Czy przy ich użyciu (pojemniki wypełnione do pełna) można wybrać z litrów wody? Napisz pseudokod algorytmu, który to sprawdza i dokonaj obliczeń dla a) $x = 13, y = 31, z = 1111$; b) $x = 12, y = 21, z = 111$.

1.4 Laboratoria

Zadanie 1.10. W środowisku Visual C++ (albo jakimkolwiek innym, którego używasz) utwórz nowy projekt o nazwie `PoznajmySie`. Dodaj do niego plik źródłowy o nazwie `przedstawienie.cpp`. Utwórz funkcję *main()*, która wypisze na ekran Twoje imię i nazwisko, ile masz lat oraz co lubisz robić najbardziej. Uruchom program.

Zadanie 1.11. Utwórz nowy projekt o nazwie `PoznajmySieNaprawiony`. Przekopiuj do niego plik `przedstawienie.cpp` z poprzedniego zadania. Uruchom program.

Zadanie 1.12. W pliku źródłowym z poprzedniego zadania wprowadź następujące „poprawki" (każdą z nich oddzielnie) i zobacz, jakie błędy zostaną zgłoszone przez kompilator.

a) Usuń linijkę `#include`.
b) Usuń linijkę `using namespace`.
c) Usuń nawias klamrowy na końcu pliku.
d) Usuń średnik po `return 0`.

Zadanie 1.13. Utwórz nowy projekt o nazwie `Psuj`. Umieść w nim plik źródłowy pobrany ze strony https://raw.githubusercontent.com/gagolews/aipp/master/zrodla/-zadanie_01_popsuty.cpp. Kod ten zawiera błędy. Postaraj się je usunąć i uruchomić program. Zwróć uwagę, że gdy kompilator zgłasza więcej niż jeden błąd, należy za każdym razem poprawić pierwszy z listy, a potem od nowa przekompilować program.

Podstawy organizacji i działania komputerów

2

2.1 Historia i organizacja współczesnych komputerów

2.1.1 Zarys historii informatyki (*)

Historia informatyki nieodłącznie związana jest z historią matematyki, a w szczególności zagadnieniem zapisu liczb i rachunkami. Najważniejszym powodem powstania komputerów było, jak łatwo się domyślić, **wspomaganie wykonywania żmudnych obliczeń**.

Oto wybór istotnych wydarzeń z dziejów tej dziedziny. Pomogą nam rzucić światło na przedmiot naszych zainteresowań.

- Ok. 2400 r. p.n.e. w Babilonii używany jest już kamienny pierwowzór liczydła, na którym rysowano piaskiem (specjaliści umiejący korzystać z tego urządzenia nie byli liczni). Podobne przyrządy w Grecji i Rzymie pojawiły się dopiero ok. V–IV w. p.n.e. Tzw. *abaki*, żłobione w drewnie, pozwalały dokonywać obliczeń w systemie dziesiętnym.
- Ok. V w. p.n.e. indyjski uczony Pãnini sformalizował teoretyczne reguły gramatyki sanskrytu. Można ten fakt uważać za pierwsze badanie teoretyczne w dziedzinie lingwistyki.
- Pierwszy algorytm przypisywany jest Euklidesowi (ok. 400–300 r. p.n.e.). Ten starożytny uczony opisał operacje, których wykonanie krok po kroku pozwala wyznaczyć największy wspólny dzielnik dwóch liczb (por. zadanie 1.1).
- Matematyk arabski Al Kwarizmi (IX w.) określa reguły podstawowych operacji arytmetycznych dla liczb dziesiętnych. Od jego nazwiska pochodzi ponoć pojęcie **algorytmu**.
- W epoce baroku dokonuje się „obliczeniowy przełom". W 1614 r. John Napier, szkocki teolog i matematyk, znalazł zastosowanie logarytmów do wykonywania szybkich operacji mnożenia (zastąpił je dodawaniem). W roku 1622 William Oughtred stworzył **suwak logarytmiczny**, który jeszcze bardziej ułatwił wykony-

wanie obliczeń.
W. Schickard (1623 r.) oraz B. Pascal (1645 r.) tworzą pierwsze **mechaniczne sumatory**. Ciekawe, czy Isaac Newton (1643–1727) z nich korzystał?

Zadanie
Zastanów się przez chwilę, do czego może być przydatna umiejętność szybkiego wykonywania operacji arytmetycznych w różnych dziedzinach życia.

- Jacques Jacquard (ok. 1801 r.) skonstruował krosno tkackie sterowane dziurkowanymi kartami. Było to pierwsze **sterowane programowo** urządzenie w dziejach techniki. Podobny ideowo sposób działania miały *pianole* (poł. XIX w.), czyli automatycznie sterowane pianina.
- Do czasów Charlesa Babbage'a żadne urządzenie służące do wykonywania obliczeń nie było **programowalne**. Ok. 1837–1839 r. opisał on więc wymyśloną przez siebie „maszynę analityczną" (parową!), która to umożliwiała. Była to jednak tylko koncepcja teoretyczna, nigdy nie udało się jej zbudować. Pomagająca mu Ada Lovelace może być uznana za **pierwszego programistę**.
- H. Hollerith (1890) — maszyna wspomagająca spis powszechny w USA, czyli zorientowana na przetwarzanie dużej ilości danych.
- Niemiecki inżynier w 1918 r. patentuje maszynę szyfrującą Enigma. (Nota bene jej kod w 1934 r. łamie polski matematyk Marian Rejewski).
- W 1936 r. Alan Turing i Alonzo Church definiują formalnie *algorytm* jako ciąg instrukcji matematycznych. Określają dzięki temu to, **co daje się policzyć**. Kleene stawia później tzw. **hipotezę Churcha–Turinga**.

Informacja
Hipoteza Churcha–Turinga mówi o tym, że jeśli dla jakiegoś problemu istnieje efektywny algorytm korzystający z nieograniczonych zasobów, to da się go wykonać na pewnym prostym, abstrakcyjnym modelu komputera:

> *Każdy problem, który może być intuicyjnie uznany za obliczalny, jest rozwiązywalny przez maszynę Turinga.*

Tutaj jednak pojawia się problem: co może być uznane za „intuicyjnie obliczalne" — zagadnienie to do dziś nie jest do końca rozstrzygnięte.

- Claude E. Shannon w swojej pracy magisterskiej w 1937 r. opisuje możliwość użycia **elektronicznych przełączników** do wykonywania operacji logicznych (implementacja algebry Boole'a). Jego praca teoretyczna staje się podstawą konstrukcji wszystkich współczesnych komputerów elektronicznych.
- W latach 40' XX w. w Wielkiej Brytanii, USA i Niemczech **powstają pierwsze komputery elektroniczne**, np. Z1, Z3, Colossus, Mark I, ENIAC (zob. rys. 2.1),

EDVAC, EDSAC. Pobierają bardzo duże ilości energii elektrycznej i potrzebują dużych przestrzeni. Np. ENIAC miał masę ponad 27 ton, zawierał około 18 000 lamp elektronowych i zajmował powierzchnię ok. 140 m^2. Były bardzo wolne (jak na dzisiejsze standardy), np. Z3 wykonywał jedno mnożenie w 3 sekundy. Pierwsze zastosowanie: łamanie szyfrów, obliczanie trajektorii lotów balistycznych (zastosowania wojskowe).

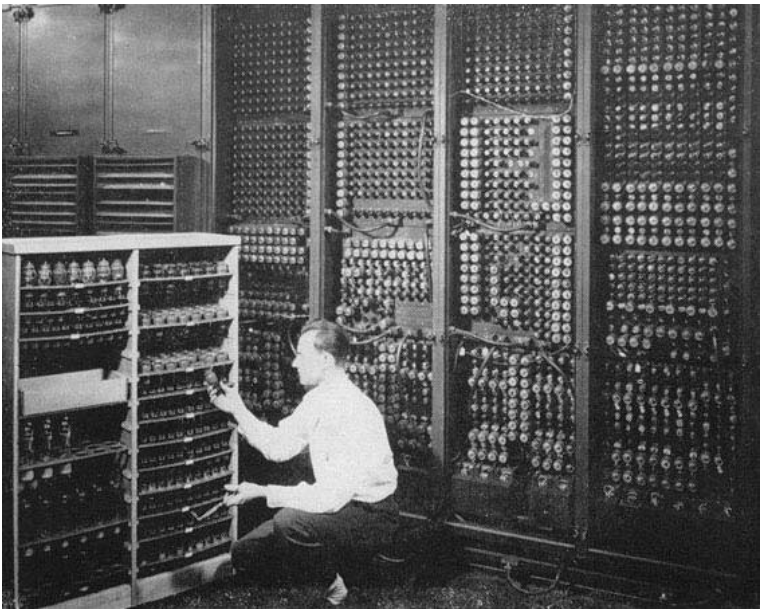
- W 1947 r. Grace Hopper odkrywa pierwszego robaka (ang. *bug*) komputerowego w komputerze Harvard Mark II — dosłownie!
- W międzyczasie okazuje się, że komputery nie muszą być wykorzystywane tylko w celach przyspieszania obliczeń. Powstają **teoretyczne podstawy informatyki** (np. Turing, von Neumann). Zapoczątkowywane są nowe kierunki badawcze, m.in. sztuczna inteligencja (por. np. słynny test Turinga sprawdzający „inteligencję" maszyny).
- Pierwszy **kompilator** języka Fortran zostaje uruchomiony w 1957 r. Kompilator języka LISP powstaje rok później.
- Informatyka staje się **dyscypliną akademicką** dopiero w latach sześćdziesiątych XX w. (wytyczne programowe ACM). Pierwszy wydział informatyki powstał na Uniwersytecie Purdue w 1962 r. Pierwszy doktorat w tej dziedzinie został obroniony już w 1965 r.
- W 1969 r. następuje pierwsze **połączenie sieciowe** pomiędzy komputerami w ramach projektu ARPAnet (prekursora Internetu). Początkowo ma ono mieć głównie zastosowanie (znowu!) wojskowe.
- Dalej rozwój następuje bardzo szybko: powstają bardzo ważne algorytmy (np. wyszukiwanie najkrótszych ścieżek w grafie Dijkstry, sortowanie szybkie Hoara itp.), teoria relacyjnych baz danych, okienkowe wielozadaniowe systemy operacyjne, nowe języki programowania, systemy rozproszone i wiele, wiele innych...
- Współcześnie komputer (o mocy kilka miliardów razy większej niż ENIAC) podłączony do ogólnoświatowej sieci Internet znajdziemy w prawie każdej kieszeni, a elementy informatyki wykładane są już w szkołach podstawowych.

Zadanie

Zastanów się, jak wyglądał świat 30–50 lat temu. A jak 100 lat temu. W jaki sposób ludzie komunikowali się ze sobą. W jaki sposób spędzali czas wolny.

Tak przyzwyczailiśmy się do wszechobecnych komputerów, że wiele osób nie potrafi sobie wyobrazić bez nich życia.

Wielu znaczących informatyków XX wieku było z wykształcenia matematykami. Informatyka teoretyczna była początkowo poddziałem matematyki. Z czasem dopiero stała się niezależną dziedziną akademicką. Nie oznacza to oczywiście, że obszary badań obu dyscyplin są rozłączne.



Rysunek 2.1. Komputer ENIAC.

Ważne
Wielu matematyków zajmuje się informatyką, a wielu informatyków – matematyką. Bardzo płodnym z punktu widzenia nauki jest obszar na styku obydwu dziedzin (ale i fizyki, biologii, medycyny itp.).

Można pokusić się o następującą klasyfikację głównych kierunków badań w informatyce[1]:

a) matematyczne podstawy informatyki:
   i. kryptografia (kodowanie informacji niejawnej),
   ii. teoria grafów (np. algorytmy znajdowania najkrótszych ścieżek, algorytmy kolorowania grafów),
   iii. logika (w tym logika wielowartościowa, logika rozmyta),
   iv. teoria typów (analiza formalna typów danych, m.in. badane są efekty związane z bezpieczeństwem programów),
b) teoria obliczeń:
   i. teoria automatów (analiza abstrakcyjnych maszyn i problemów, które można z ich pomocą rozwiązać),
   ii. teoria obliczalności (co jest obliczalne?),
   iii. teoria złożoności obliczeniowej (które problemy są „łatwo" rozwiązywalne?),

[1] Por. http://www.newworldencyclopedia.org/entry/Computer_science

c) algorytmy i struktury danych:
   i. analiza algorytmów (ze względu na czas działania i potrzebne zasoby),
   ii. projektowanie algorytmów,
   iii. struktury danych (organizacja informacji),
d) języki programowania i kompilatory:
   i. kompilatory (budowa i optymalizacja programów przetwarzających kod w danym języku na kod maszynowy),
   ii. języki programowania (formalne paradygmaty języków, własności języków),
e) bazy danych:
   i. teoria baz danych (m.in. systemy relacyjne, obiektowe),
   ii. data mining (wydobywanie wiedzy z baz danych),
f) systemy równoległe i rozproszone:
   i. systemy równoległe (bada wykonywanie jednoczesnych obliczeń przez wiele procesorów),
   ii. systemy rozproszone (rozwiązywanie tego samego problemu z użyciem wielu komputerów połączonych w sieć),
   iii. sieci komputerowe (algorytmy i protokoły zapewniające niezawodny przesył danych pomiędzy komputerami),
g) architektura komputerów:
   i. architektura komputerów (projektowanie i organizacja komputerów),
   ii. systemy operacyjne (systemy zarządzające zasobami komputera i pozwalające uruchamiać inne programy),
h) inżynieria oprogramowania:
   i. metody formalne (np. automatyczne testowanie programów),
   ii. inżynieria (zasady tworzenia dobrych programów, zasady organizacji procesu analizy, projektowania, implementacji i testowania programów),
i) sztuczna inteligencja:
   i. sztuczna inteligencja (systemy, które wydają się cechować inteligencją),
   ii. automatyczne wnioskowanie (imitacja zdolności samodzielnego rozumowania),
   iii. robotyka (projektowanie i konstrukcja robotów i algorytmów nimi sterujących),
   iv. algorytmy przybliżone (tworzenie heurystyk do rozwiązywania problemów, których rozwiązanie dokładne kosztuje zbyt wiele czasu/zasobów),
   v. maszynowe uczenie się (tworzenie zestawów reguł w oparciu o dane referencyjne),
j) grafika komputerowa:
   i. podstawy grafiki komputerowej (algorytmy generowania i filtrowania obrazów),
   ii. przetwarzanie obrazów (wydobywanie informacji z obrazów),
   iii. interakcja człowiek-komputer (zagadnienia tzw. interfejsów służących do komunikacji z komputerem),

k) zastosowania w innych naukach:
   i. statystyka obliczeniowa,
   ii. bioinformatyka,
   iii. ... i wiele innych.

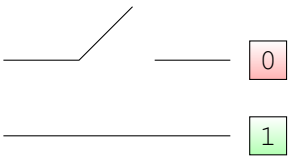
Na koniec tego paragrafu warto zwrócić uwagę (choć zdania na ten temat są podzielone), że to, co w języku polskim określamy mianem **informatyki** czy też **nauk informacyjnych**, jest przez wielu uznawane za pojęcie szersze od angielskiego *computer science*, czyli nauk o komputerach. Informatyka jest więc ogólnym przedmiotem dociekań dotyczących przetwarzania informacji, w tym również przy użyciu urządzeń elektronicznych.

Informacja
Informatyka jest nauką o komputerach w takim stopniu jak astronomia — nauką o teleskopach. (Dijkstra)

2.1.2 Organizacja komputerów

Podstawy teoretyczne działania współczesnych komputerów elektronicznych zawarte zostały w pracy C. Shannona (1937 r.). Mogą być one pojmowane jako zestaw odpowiednio sterowanych **przełączników**, tzw. *bitów* (ang. *binary digits*).

Ważne
Każdy przełącznik może znajdować się w jednym z dwóch stanów (rys. 2.2):
- prąd nie płynie (co oznaczamy jako `0`),
- prąd płynie (co oznaczamy przez `1`).



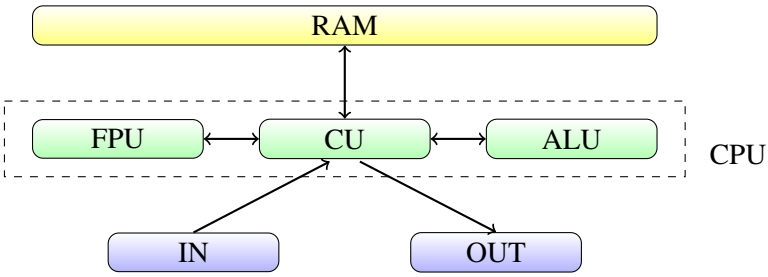
Rysunek 2.2. Przełączniki.

Na współczesny komputer osobisty (PC, od ang. *personal computer*) składają się następujące elementy:
a) jednostka obliczeniowa (CPU, od ang. *central processing unit*):
   - jednostki arytmetyczno-logiczne (**ALU**, ang. *arithmetic and logical units*),
   - jednostka do obliczeń zmiennopozycyjnych (**FPU**, ang. *floating point unit*),

  - układy sterujące (**CU**, ang. *control units*),
  - rejestry (akumulatory), pełniące funkcję pamięci podręcznej,

b) pamięć **RAM** (od ang. *random access memory*) — zawiera dane i program,
c) urządzenia wejściowe (ang. ***in**put devices*),
d) urządzenia wyjściowe (ang. ***out**put devices*).

Jest to tzw. **architektura von Neumanna** (por. rys. 2.3). Najważniejszą jej cechą jest to, że w pamięci operacyjnej znajdują się zarówno instrukcje programów jak i dane. To od kontekstu zależy, jak powinny być one interpretowane.



Rysunek 2.3. Architektura współczesnych komputerów osobistych.

2.2 Reprezentacje liczb całkowitych

Rozważmy najpierw reprezentacje **nieujemnych liczb całkowitych**. Systemy liczbowe można podzielić na pozycyjne i addytywne. Oto przykłady każdego z nich.

Pozycyjne systemy liczbowe:
a) system dziesiętny (decymalny, indyjsko-arabski) — o podstawie 10,
b) system dwójkowy (binarny) — o podstawie 2,
c) system szesnastkowy (heksadecymalny) — o podstawie 16,
d) ...

Addytywne systemy liczbowe:
a) system rzymski,
b) system sześćdziesiątkowy (Mezopotamia).

Nas szczególnie będą interesować systemy pozycyjne, w tym używany na co dzień system dziesiętny.

2.2.1 System dziesiętny

System dziesiętny (decymalny, ang. *decimal*), przyjęty w Europie zachodniej w XVI w., to pozycyjny system liczbowy o podstawie 10. Do zapisu liczb używa się w nim następujących symboli (cyfr): $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Jako przykład rozważmy liczbę 1945_{10} (przyrostek $_{10}$ wskazuje, że chodzi nam dokładnie o liczbę w systemie dziesiętnym). Jak w każdym systemie pozycyjnym, istotne jest tu, na którym miejscu stoi każda z cyfr.

	1	9	4	5	$_{10}$
=	**1**	0	0	0	
+		**9**	0	0	
+			**4**	0	
+				**5**	

	1	9	4	5	$_{10}$
=	**1**				$\times 1000$
+		**9**			$\times 100$
+			**4**		$\times 10$
+				**5**	$\times 1$

	1	9	4	5	$_{10}$
=	**1**				$\times 10^3$
+		**9**			$\times 10^2$
+			**4**		$\times 10^1$
+				**5**	$\times 10^0$

Pierwszą z cyfr znajdujemy na tzw. pozycji 0. Kolejne oznaczamy w kierunku od prawej do lewej, czyli od najmniej do najbardziej znaczącej.

	1	9	4	5	$_{10}$
=	**1**				$\times 10^3$
+	$\vdots$	**9**			$\times 10^2$
+	$\vdots$	$\vdots$	**4**		$\times 10^1$
+	$\vdots$	$\vdots$	$\vdots$	**5**	$\times 10^0$
	3	2	1	0	

Ogólnie rzecz biorąc, mając daną liczbę n-cyfrową, zapisaną w postaci *ciągu* cyfr dziesiętnych $b_{n-1}b_{n-2}\ldots b_1b_0$, gdzie $b_i \in \{0,1,\ldots,9\}$ dla $i = 0,1,\ldots,n-1$, jej wartość można określić za pomocą wzoru

$$\sum_{i=0}^{n-1} b_i \times 10^i.$$

Liczbę 10 występującą we wzorze nazywamy **podstawą systemu**.

Jak łatwo się domyślić, można określić systemy pozycyjne o innych podstawach. My szczególnie skupimy się na systemach o podstawie 2 i 16, które są najbardziej popularne w informatyce.

2.2.2 System dwójkowy

Dwójkowy pozycyjny system liczbowy (binarny, ang. *binary*) to system pozycyjny o podstawie 2. Używanymi weń symbolami (cyframi) są tylko 0 oraz 1. Zauważmy, że cyfry te mają „przełącznikową" interpretację: prąd nie płynie/płynie, przełącznik wyłączony/włączony (por. wyżej). Zatem n-cyfrowa liczba w tym systemie może opisywać naraz stan n przełączników.

Jako przykład rozpatrzmy liczbę 101001_2.

	1	0	1	0	0	1	$_2$
=	**1**						$\times 2^5$
+		**0**					$\times 2^4$
+			**1**				$\times 2^3$
+				**0**			$\times 2^2$
+					**0**		$\times 2^1$
+						**1**	$\times 2^0$

	1	0	1	0	0	1	$_2$
=	**1**						$\times 32$
+		**0**					$\times 16$
+			**1**				$\times 8$
+				**0**			$\times 4$
+					**0**		$\times 2$
+						**1**	$\times 1$

Zatem $101001_2 = 32_{10} + 8_{10} + 1_{10} = 41_{10}$. Jest to przykład **konwersji** (zamiany) podstawy liczby dwójkowej na dziesiętną.

Przy dokonywaniu operacji na liczbach binarnych przydatna może się okazać tab. 2.1, przedstawiająca wybrane potęgi liczby 2. Pierwszych kilkanaście potęg liczby 2 dobrze jest nauczyć się na pamięć.

Informacja

Współczesne komputery przetwarzają liczby całkowite w „pakietach", najczęściej 32- lub 64-bitowych. Zależne jest to m.in. od systemu operacyjnego.

Tabela 2.1. Wybrane potęgi liczby 2.

k	2^k
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1 024
11	2 048

k	2^k
12	4 096
13	8 192
14	16 384
15	32 768
16	65 536
⋮	⋮
31	2 147 483 648
32	4 294 967 296
⋮	⋮
63	9 223 372 036 854 775 808
64	18 446 744 073 709 551 616

Tabela 2.2. Przykład — konwersja liczby 1945_{10} na dwójkową.

	k	2^i	i	
	1945	~~2048~~		
	1945	1024	10	1
1945-1024 =	921	512	9	1
921-512 =	409	256	8	1
409-256 =	153	128	7	1
153-128 =	25	~~64~~	6	0
	25	~~32~~	5	0
	25	16	4	1
25-16=	9	8	3	1
9-8=	1	~~4~~	2	0
	1	~~2~~	1	0
	1	1	0	1
1-1=	0			↑

Konwersja liczb do systemu dziesiętnego jest stosunkowo prosta. Przyjrzyjmy się, jak przekształcić liczbę dziesiętną na dwójkową. Algorytm 2.1 może być wykorzystany w tym właśnie celu.

Rozważmy na przykład liczbę 1945_{10}. Tabela 2.2 opisuje kolejno wszystkie kroki potrzebne do uzyskania wyniku w postaci dwójkowej. Możemy z niej odczytać, iż $1945_{10} = 11110011001_2$.

Listing 2.1. Konwersja liczby dziesiętnej na dwójkową.

```
// Wejscie: k ∈ ℕ.
// Output: reprezentacja binarna k (wypisana na konsole)
let i be the greatest integer such that k ⩾ 2^i;
while (i ⩾ 0) // to jest pseudokod, nie C++
{
    if (k ⩾ 2^i)
    {
        print(1);
        k = k − 2^i;
    }
    else
        print(0);

    i = i − 1;
}
```

2.2.3 System szesnastkowy

System szesnastkowy (heksadecymalny, ang. *hexadecimal*) jest systemem pozycyjnym o podstawie 16. Zgodnie z przyjętą konwencją, używanymi symbolami (cyframi) są: $0, 1, \ldots, 9, A, B, C, D, E$ oraz F. Jak widzimy, wykorzystujemy tutaj (z konieczności) także kolejne litery alfabetu, które mają swoją liczbową (a właściwie cyfrową) interpretację.

Po co taki system, skoro komputer właściwie opera się na liczbach w systemie binarnym? Jak można zauważyć, liczba w postaci dwójkowej prezentuje się na kartce lub na ekranie dosyć... okazale. Jako że jest to czasem problematyczne, w zastosowaniach informatycznych zwykło się zapisywać liczby właśnie jako szesnastkowe, gdyż $16 = 2^4$. Dzięki temu można użyć jednego symbolu zamiast 4 w systemie dwójkowym. Fakt ten sprawia również, że konwersja z systemu binarnego na heksadecymalny i odwrotnie jest bardzo prosta.

Tabela 2.3 przedstawia cyfry systemu szesnastkowego i ich wartości w systemie dwójkowym oraz dziesiętnym.

Zatem na przykład $1011110_2 = 5\mathrm{E}_{16}$, gdyż grupując kolejne cyfry dwójkowe czwórkami, od prawej do lewej, otrzymujemy

$$\overbrace{0101}^{5}\overbrace{1110}^{\mathrm{E}}{}_2.$$

Tabela 2.3. Cyfry systemu szesnastkowego i ich wartości w systemie dwójkowym oraz dziesiętnym.

BIN	DEC	HEX
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

BIN	DEC	HEX
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Podobnie postępujemy, dokonując konwersji w przeciwną stronę, np. $2\mathrm{D}_{16} = 101101_2$, bowiem

$$\overbrace{2}^{0010}\overbrace{\mathrm{D}}^{1101}{}_{16}.$$

Jak widzimy, dla uproszczenia zapisu początkowe zera możemy pomijać bez straty ogólności (zob. jednak wyjątek w rozdz. 2.2.4).

Konwersji z i do postaci dziesiętnej możemy dokonywać za pośrednictwem opanowanej już przez nas postaci binarnej.

2.2.4 System U2 reprezentacji liczb ze znakiem

Interesującym jest pytanie, w jaki sposób można reprezentować w komputerze liczby ze znakiem, np. -1001_2? Jak widzimy, znak jest nowym (trzecim) symbolem, a elektroniczny przełącznik może znajdować się tylko w dwóch stanach.

Spośród kilku możliwych sposobów rozwiązania tego problemu, obecnie w większości komputerów osobistych używany jest tzw. **kod uzupełnień do dwóch**, czyli **kod U2**. Niewątpliwą jego zaletą jest to, iż pozwala na bardzo łatwą realizację operacji dodawania i odejmowania (zob. ćwiczenia).

Ważne

W notacji U2 najstarszy (czyli stojący po lewej stronie) bit determinuje znak liczby:

- najstarszy bit równy 0 oznacza liczbę nieujemną,
- najstarszy bit równy 1 oznacza liczbę ujemną.

Jeśli dana jest n cyfrowa liczba w systemie U2 postaci $b_{n-1}b_{n-2}\dots b_1b_0$, gdzie $b_i \in \{0,1\}$ dla $i = 0,1,\dots,n-1$, to jej wartość wyznaczamy następująco:

$$-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i.$$

Zatem dla liczby n bitowej najstarszy bit mnożymy nie przez 2^{n-1}, lecz przez -2^{n-1}.

Na przykład, $0110_{U2} = 110_2$ oraz

$$1011_{U2} = -\mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{1} \times 2^0 = -5_{10} = -101_2.$$

Co ważne, tutaj najstarszego bitu równego 0 nie można dowolnie pomijać w zapisie. Mamy bowiem $0b_{n-2}\dots b_{0\mathrm{U2}} \neq b_{n-2}\dots b_{0\mathrm{U2}}$, gdy $b_{n-2} = 1$. Jednakże,

$$0b_{n-2}\dots b_{0\mathrm{U2}} = 00\dots 0b_{n-2}\dots b_{0\mathrm{U2}}.$$

Z drugiej strony, można udowodnić (zob. ćwiczenia), że również

$$1b_{n-2}\dots b_{0\mathrm{U2}} = 11\dots 1b_{n-2}\dots b_{0\mathrm{U2}}.$$

Informacja

Łatwo pokazać, że użycie k bitów pozwala na zapis liczb $-2^{k-1},\dots,2^{k-1}-1$, np. liczby 8 bitowe w systemie U2 mogą mieć wartości od -128 do 127.

2.3 Reprezentacje liczb rzeczywistych

Ostatnim, ważnym dla nas zagadnieniem jest problem **reprezentacji liczb rzeczywistych**, a właściwie odpowiedniego jego podzbioru, przydatnego podczas dokonywania obliczeń numerycznych. Rozważymy tutaj tzw. reprezentację stałoprzecinkową i zmiennoprzecinkową. Ta ostatnia jest powszechnie używana we współczesnych komputerach.

2.3.1 System stałoprzecinkowy (*)

W **reprezentacji stałoprzecinkowej** liczb rzeczywistych w systemie dwójkowym liczba bitów używanych do zapisu części „dziesiętnej" i części ułamkowej jest z góry ustalona.

Rozpatrzmy liczbę postaci $b_{n-1}b_{n-2}\dots b_k,b_{k-1}\dots b_0$. Jest to liczba n bitowa, w której na część ułamkową przypada k bitów. Jej wartość można wyznaczyć ze wzoru

$$\sum_{i=0}^{n-1} b_i \times 2^{i-k}.$$

Tym samym np. $1011{,}101_2 = 8 + 2 + 1 + \frac{1}{2} + \frac{1}{8} = 11\frac{5}{8}{}_{10}$. Niestety, ustalenie liczby k z góry uniemożliwia przybliżanie z zadowalającą dokładnością liczb o względnie dużej i małej wartości.

2.3.2 System zmiennoprzecinkowy

Jak wspomnieliśmy wyżej, współczesne procesory są w stanie przechowywać i przetwarzać liczby o długości dokładnie 8 (tzw. bajt, ang. *byte*), 16, 32, 64 lub nawet 128 bitów. Własność ta umożliwia zdefiniowanie **postaci zmiennopozycyjnej** (zmiennoprzecinkowej) liczb rzeczywistych (ang. *floating-point numbers*).

Liczba zmiennoprzecinkowa przechowywana jest w pamięci komputera w postaci:

$$x = s \times m \times 2^e,$$

gdzie:
- $s \in \{0, 1\}$ — **znak** (1 bit, interpretowany jako -1^s),
- m — **mantysa** (odpowiednio znormalizowana),
- e — **wykładnik**.

Ważne
Liczba cyfr mantysy i wykładnika jest ustalona z góry, np. w większości komputerów (standard IEEE 754) jest to 23+8 (liczba 32-bitowa) albo 52+11 (64-bitowa liczba zmiennoprzecinkowa).

Jak widzimy, dzięki zastosowaniu części wykładnikowej, pozycja „przecinka" jest dość elastyczna: może „przepływać" (ang. *float*) między cyframi. Jeśli podstawą systemu byłoby 10 (a nie 2 jak wyżej), w przypadku sześciocyfrowej mantysy i wykładnika o odpowiedniej pojemności, moglibyśmy reprezentować m.in. liczby 1,23456, 123,456, 0,0000123456 oraz 123456000000. W porównaniu do systemu stałoprzecinkowego, dla dwóch cyfr po przecinku liczbom tym odpowiadają, odpowiednio, 1,23, 123,456, 0,00 oraz 123456000000,00; widzimy, że nie jest to zbyt zadowalające przybliżenie.

Ważne
Dodatkowo w tym systemie w specjalny sposób zdefiniowane są trzy specjalne wartości:
- ∞ (`Inf`, ang. *infinity*),
- $-\infty$ (`-Inf`),
- nie-liczba (`NaN`, ang. *not a number*).

Np. $1/0$ = `Inf`, $0/0$ = `NaN`.

Informacja

Dokładne omówienie zagadnienia reprezentacji i arytmetyki liczb zmiennopozycyjnych wykracza poza ramy naszego wykładu. Więcej szczegółów, w tym rachunek błędów arytmetyki tych liczb, przedstawiony będzie na wykładzie z metod numerycznych.

Zainteresowanym osobom można polecić następujący angielskojęzyczny artykuł: D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys* **21**(1), 1991, 5–48.

Ważne

Pomimo, z konieczności, dość skrótowego potraktowania tego bardzo ciekawego tematu warto mieć na uwadze dwa problemy związanych z użyciem liczb zmiennopozycyjnych.

a) Nie da się reprezentować całego zbioru liczb rzeczywistych, a tylko jego podzbiór (właściwie jest to tylko podzbiór liczb wymiernych!). Wszystkie liczby są zaokrąglane do najbliższej reprezentowalnej.
b) Arytmetyka zmiennoprzecinkowa nie spełnia w pewnych szczególnych przypadkach własności łączności i rozdzielności.

2.4 Ćwiczenia

Zadanie 2.14. Przedstaw następujące liczby całkowite nieujemne w postaci binarnej, dziesiętnej i szesnastkowej: $10_{10}, 18_{10}, 18_{16}, 101_2, 101_{10}, 101_{16}, \mathrm{ABCDEF}_{16}, 64135312_{10}, 110101111001_2, \mathrm{FFFFF0C2}_{16}$.

Zadanie 2.15. Dana jest n cyfrowa liczba w systemie U2 zapisana jako ciąg cyfr $b_{n-1}b_{n-2}\ldots$ $\ldots b_1b_0$, gdzie $b_i \in \{0,1\}$ dla $i = 0,1,\ldots,n-1$. Pokaż, że powielenie pierwszej cyfry dowolną liczbę razy, nie zmienia wartości danej liczby, tzn. $b_{n-1}b_{n-2}\ldots b_1b_0 = b_{n-1}b_{n-1}\ldots\ldots b_{n-1}b_{n-2}\ldots b_1b_0$.

Zadanie 2.16. Przedstaw jako liczby dziesiętne następujące liczby dane w notacji U2: 10111100, 00111001, 1000000111001011.

Zadanie 2.17. Przedstaw w notacji U2 (do zapisu użyj 8, 16 lub 32 bitów) następujące liczby dziesiętne: $-12, 54, -128, -129, 53263, -32000, -56321, -3263411$.

Zadanie 2.18. Dana jest zmienna $x \in \mathbb{N}$. Napisz wzór matematyczny, który opisze najmniejszą liczbę cyfr potrzebnych do zapisania wartości x w systemie a) binarnym; b) dziesiętnym; c) szesnastkowym.

(*) Zadanie 2.19. Rozważmy operacje dodawania i odejmowania liczb nieujemnych w reprezentacji binarnej. Korzystając z metody analogicznej do sposobu „szkolnego" (dodawania i odejmowanie słupkami), oblicz wartość następujących wyrażeń. Pamiętaj jednak, że np. $1_2 + 1_2 = 10_2$.

a) $1000_2 + 111_2$,
b) $1000_2 + 1111_2$,
c) $110110_2 + 11100111_2$,
d) $11111_2 + 11111111_2$,
e) $1111_2 - 0001_2$,
f) $1000_2 - 0001_2$,
g) $10101010_2 - 11101_2$,
h) $111111001101_2 - 1110010111_2$.

(*) Zadanie 2.20. Okazuje się, że liczby w systemie U2 można dodawać i odejmować tą samą metodą, co liczby nieujemne w reprezentacji binarnej. Oblicz wartość następujących wyrażeń i sprawdź otrzymane wyniki, przekształcając je do postaci dziesiętnej. Uwaga: operacji dokonuj na dwóch liczbach n bitowych, a wynik podaj również jako n bitowy.

a) $1000_{\mathrm{U2}} + 0111_{\mathrm{U2}}$,
b) $10110110_{\mathrm{U2}} + 11100111_{\mathrm{U2}}$,
c) $10101010_{\mathrm{U2}} - 00011101_{\mathrm{U2}}$,
d) $11001101_{\mathrm{U2}} - 10010111_{\mathrm{U2}}$.
e) $10001101_{\mathrm{U2}} - 01010111_{\mathrm{U2}}$.

(*) Zadanie 2.21. Dana jest liczba w postaci U2. Pokaż, że aby uzyskać liczbę do niej przeciwną, należy odwrócić wartości jej bitów (dokonać ich negacji, tzn. zamienić zera na jedynki i odwrotnie) i dodać do wyniku wartość 1. Ile wynoszą wartości $0101_{\mathrm{U2}}, 1001_{\mathrm{U2}}$ i 0111_{U2} po dokonaniu tych operacji? Sprawdź uzyskane rezultaty, konwertując je do systemu dziesiętnego.

(*) Zadanie 2.22. Dana jest liczba w postaci binarnej. W jaki sposób przy użyciu jednej operacji logicznej, jednej bitowej oraz jednej arytmetycznej sprawdzić, czy dana liczba jest potęgą liczby 2?

Zadanie 2.23. Dany jest ciąg n liczb $\mathbf{x} = (x[0], x[1], \ldots, x[n-1]) \in \{0,1\}^n$, który reprezentuje pewną liczbę w postaci U2, gdzie $x[0]$ tym razem oznacza najstarszy bit, a $x[n-1]$ — najmłodszy. Napisz algorytm, który wyznaczy wartość tej liczby.

Zadanie 2.24. Napisz algorytm, który znajdzie binarną reprezentację dodatniej liczby naturalnej. Wskazówka: jako wynik zwróć tablicę $x[0], x[1], \ldots, x[k-1]$ zawierającą kolejne cyfry dwójkowe.

2.5 Laboratoria

Zadanie 2.25. Wczytuj kolejno liczby całkowite z klawiatury. Zakończ wczytywanie, gdy użytkownik wprowadzi liczbę ujemną. Wypisz na ekran, ile wprowadzono liczb nieujemnych.

Zadanie 2.26. Wczytuj kolejno liczby całkowite z klawiatury. Zakończ wczytywanie, gdy użytkownik wprowadzi liczbę ujemną. Wypisz na ekran sumę wszystkich wprowadzonych liczb nieujemnych.

Zadanie 2.27. Wczytuj kolejno liczby całkowite z klawiatury. Zakończ wczytywanie, gdy użytkownik wprowadzi liczbę ujemną. Wypisz na ekran średnią arytmetyczną wszystkich wprowadzonych liczb nieujemnych.

Zadanie 2.28. Wczytuj kolejno liczby całkowite z klawiatury. Zakończ wczytywanie, gdy użytkownik wprowadzi liczbę ujemną **lub** wprowadzi 10 liczb. Wypisz na ekran średnią wszystkich wprowadzonych liczb nieujemnych.

Zadanie 2.29. Wypisz na ekranie wartości 10^i dla kolejnych $i = 0, 1, \ldots, n$, gdzie n jest liczbą całkowitą nieujemną wprowadzoną z klawiatury. Program po podaniu $n < 0$ powinien wyświetlać tylko komunikat o błędzie.

Podstawy języka C++ 3

3.1 Zmienne w języku C++ i ich typy

3.1.1 Pojęcie zmiennej

Czytając pseudokody algorytmów z pierwszego zestawu zadań, kilka razy napotykaliśmy instrukcję podobną do następującej.

```
let x ∈ ℝ;
```

Mieliśmy przez to na myśli „niech **od tej pory** x będzie **zmienną** ze zbioru liczb rzeczywistych". Dzięki temu możemy użyć x np. do obliczenia wartości pewnego podwyrażenia w skomplikowanym wzorze, tak by rozwiązanie uczynić czytelniejszym.

> **Ważne**
> **Zmienna** służy do przechowywania w pamięci komputera (lub, w przypadku pseudokodu, czymś co pamięć komputera reprezentuje) dowolnych wartości z pewnego ustalonego zbioru. Taki zbiór nazywamy **typem** zmiennej.

W powyższy sposób dokonaliśmy więc **deklaracji** zmiennej (ang. *variable declaration*) typu rzeczywistego.

3.1.2 Typy liczbowe proste

W niniejszym paragrafie omówimy dostępne w języku **C++** typy zmiennych liczbowych (są to tzw. **typy proste**). Będą to: typy całkowite, zmiennoprzecinkowe i typ logiczny.

3.1.2.1 Typy całkowite

Do **typów całkowitych** zaliczamy:

Nazwa typu	Liczba bitów[1]
`char`	8
`short`	16
`int`	32
`long long`	64

My najczęściej będziemy korzystać z typu `int`. Jest to liczba 32-bitowa w systemie U2 (a więc ze znakiem). Zatem typ ten reprezentuje zbiór

$$\texttt{int} = \{-2^{31}, -2^{31}+1, \ldots, 2^{31}-1\} \subset \mathbb{Z}.$$

Co więcej, do każdego z wyżej wymienionych typów można zastosować modyfikator `unsigned` — wtedy otrzymujemy liczbę nieujemną (w zwykłym systemie binarnym), np.

$$\texttt{unsigned int} = \{0, 1, \ldots, 2^{32}-1\} \subset \mathbb{N}_0.$$

Informacja

Stałe całkowite, np. `0`, `-5`, `15210` są reprezentantami typu `int` danymi w postaci dziesiętnej. Oprócz tego można używać stałych w postaci szesnastkowej, poprzedzając liczby przedrostkiem `0x`, np. `0x53a353` czy też `0xabcdef`, oraz ósemkowej, korzystając z przedrostka `0`, np. `0777`. Nie ma, niestety, możliwości definiowania bezpośrednio stałych w postaci dwójkowej.

Ważne

`010` oraz `10` oznaczają dwie różne liczby!

3.1.2.2 Typy zmiennoprzecinkowe

Z kolei do **typów zmiennoprzecinkowych** zaliczamy:

Nazwa typu	Liczba bitów
`float`	32
`double`	64

Mamy `float` $\subset \bar{\mathbb{R}}$ i `double` $\subset \bar{\mathbb{R}}$ (oczywiście pamiętamy, że typy zmiennopozycyjne przechowują też wartości specjalne `Inf`, `-Inf` oraz `NaN`). W przypadku liczby

[1] Liczba bitów może się różnić w zależności od systemu operacyjnego i kompilatora.

32-bitowej, najmniejsza reprezentowalna wartość dodatnia to ok. $1{,}18 \times 10^{-38}$, a największa — ok. $3{,}4 \times 10^{38}$. Dla liczby 64-bitowej mamy, odpowiednio, ok. $2{,}23 \times 10^{-308}$ i $1{,}80 \times 10^{308}$.

Informacja

Stałe zmiennoprzecinkowe wprowadzamy, podając zawsze część dziesiętną i część ułamkową rozdzieloną kropką (nawet gdy część ułamkowa jest równa 0), np. `3.14159`, `1.0` albo `-0.000001`. Domyślnie należą one do typu **`double`**.

Dodatkowo liczby takie można wprowadzać w **notacji naukowej**, używając do tego celu separatora `e`, który oznacza „razy dziesięć do potęgi". I tak liczba `-2.32e-4` jest stałą o wartości $-2{,}32 \cdot 10^{-4}$.

3.1.2.3 Typ logiczny

Oprócz powyższych, wśród typów prostych, dostępny jest jeszcze typ **`bool`**, służący do reprezentowania **zbioru wartości logicznych**.

Informacja

Zmienna typu logicznego może znajdować się w dwóch stanach, określonych przez **stałe logiczne** **`true`** (prawda) oraz **`false`** (fałsz).

3.1.3 Identyfikatory

Mamy dwie możliwości użycia zmiennej: możemy **przypisać** jej pewną wartość, a także przechowywaną weń wartość **odczytać**. W tym sensie zmienną można porównać do szuflady, która ma swoją etykietkę (nazwę, **identyfikator**), jednoznacznie odróżniającą ją od innych obiektów. Do takiej szuflady można coś włożyć (jednocześnie pozbywając się wcześniejszej zawartości) albo coś z niej wyjąć.

Każda zmienna musi mieć zatem jakąś nazwę, aby można się było do niej odwołać. Identyfikatory mogą składać się z następujących znaków:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _
```

oraz z poniższych, pod warunkiem, że nie są one pierwszym znakiem identyfikatora:

```
0 1 2 3 4 5 6 7 8 9
```

Wobec tego przykładami poprawnych identyfikatorów są: *`i`*, *`suma`*, *`_2m1a`* oraz

`zmienna_pomocnicza1`. Zgodnie jednak z podaną regułą np. `3523aaa`, `ala ma kota` oraz `:P` nie mogą być identyfikatorami.

Identyfikatorami nie mogą być też **zarezerwowane słowa kluczowe języka C++** (np. **`int`**, **`if`**). Z tego powodu wyróżniamy je w niniejszym skrypcie pogrubioną czcionką.

Ważne
W języku **C++** wielkość liter ma znaczenie.

Na przykład `T` oraz `t` to identyfikatory dwóch różnych zmiennych. Z tego też powodu identyfikator `Int` nie jest tożsamy z **`int`**.

Informacja
Dobrą praktyką jest nadawanie takich nazw zmiennym, by w pewnym sensie „same się objaśniały".

Zmienne `poleKwadratu`, `delta`, `wspolczynnikX` w pewnych kontekstach posiadają tę cechę. Często używamy też identyfikatorów `i`, `j`, `k` jako liczników w pętlach. Z drugiej strony, np. zmienne `dupa` albo `ratunku` niezbyt jasno mówią czytelnikowi, do czego mogą służyć.

3.1.4 Deklaracja zmiennych

Dzięki zmiennym pisane przez nas programy (ale także np. twierdzenia matematyczne) nie opisują jednego, szczególnego przypadku pewnego interesującego nas problemu (np. rozwiązania konkretnego układu dwóch równań liniowych), lecz wszystkie możliwe w danym kontekście (np. rozwiązania dowolnego układu równań liniowych). Dzięki takiemu mechanizmowi mamy więc możliwość **uogólniania** naszych wyników.

Oto kolejny przykład. Otóż ciąg twierdzeń:

Twierdzenie 1. *Pole koła o promieniu* 1 *wynosi* π.

Twierdzenie 2. *Pole koła o promieniu* 2 *wynosi* 4π.

Twierdzenie 3. ...

przez każdego matematyka (a nawet przyszłego matematyka) byłby prawdopodobnie uznany za żart. Dopiero uogólnienie powyższych wyników przez odwołanie się do pewnej, wcześniej zadeklarowanej zmiennej sprawi, że wynik stanie się interesujący.

Twierdzenie 4. *Niech* $r \in \mathbb{R}_+$. *Pole koła o promieniu* r *wynosi* πr^2.

Zauważmy, że w matematyce zdanie deklarujące zmienną „niech $r \in \mathbb{R}_+$" nie musi się pojawiać w takiej właśnie formie. Większość naukowców podchodzi do tego, rzecz

jasna, w sposób elastyczny: jesteśmy bowiem dość inteligentni. Mimo to, pisząc np. „pole koła o promieniu $r > 0$ wynosi πr^2" bądź nawet „pole koła o promieniu r wynosi πr^2", domyślamy się, że chodzi właśnie o powyższą konstrukcję (czyli założenie, że r jest liczbą rzeczywistą dodatnią).

Wiemy już z pierwszego wykładu, że komputery nie są jednak tak domyślne jak my.

Ważne

W języku **C++** wszystkie zmienne muszą zostać **zadeklarowane przed** ich **pierwszym użyciem**.

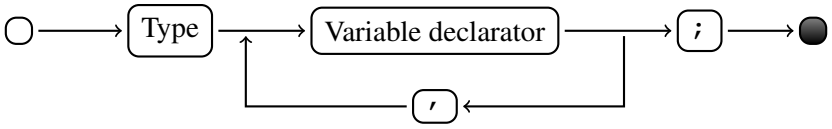
Aby zadeklarować zmienną, należy napisać w kodzie programu odpowiednią instrukcję o ściśle określonej składni.

Informacja

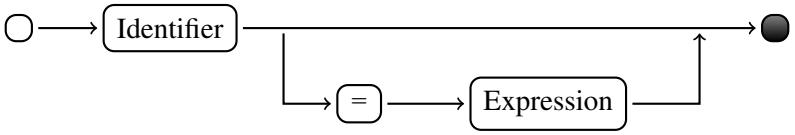
Instrukcja (ang. *statement*) jest podstawowym składnikiem pisanego programu. Dzięki niej możemy nakazać komputerowi wykonać pewną ustaloną czynność (być może bardzo złożoną), której znaczenie jest ściśle określone przez reguły semantyczne języka programowania.

Odpowiednikiem instrukcji w języku polskim jest w pewnym sensie zdanie rozkazujące. Na przykład „wynieś śmieci!", „zacznij się wreszcie uczyć!" są dobrymi instrukcjami. Z drugiej strony zdanie „Jasiu, źle mi, zrób coś!" instrukcją nie jest, ponieważ nie ma ściśle określonego znaczenia (semantyki).

Składnia **instrukcji deklarującej zmienną lokalną** w języku **C++** (ang. *local variable declaration statement*), czyli np. takiej, którą możemy umieścić w funkcji `main()`, przedstawiona jest na poniższym diagramie:



gdzie Type to typ zmiennej (np. `int` lub `double`), a Variable declarator to tzw. deklarator zmiennej. Jest on postaci:



Tutaj Identifier oznacza identyfikator zmiennej (o omówionej wyżej postaci), a Expression

jest pewnym wyrażeniem. Więcej o wyrażeniach dowiemy się za chwilę. Na razie wystarczy nam wiedzieć, że wyrażeniem może być m.in. stała całkowita lub zmiennoprzecinkowa albo pewna operacja arytmetyczna.

Ważne
Większość instrukcji w języku **C++** musi być zakończona **średnikiem**!

Oto kilka przykładów instrukcji deklarujących zmienne:

```
1 int x;
2 bool p = false;
3 int y = 10;
4 double a, b;
5 int u = y+10, v = -y;
```

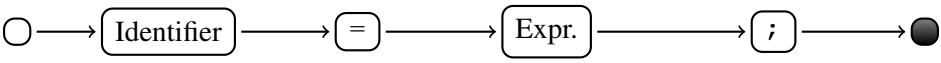
3.1.5 Instrukcja przypisania

Deklaracja zmiennej powoduje przydzielenie jej pewnego miejsca w pamięci RAM komputera (poznamy to bardzo dokładnie zagadnienie na wykładzie o wskaźnikach).

Informacja
Zmienna nie jest inicjowana automatycznie. Dopóki nie przypiszemy jej jawnie jakiejś wartości, będzie przechowywać „śmieci”.

Próba odczytania wartości niezainicjowanej zmiennej może (lecz nie musi, jest to zależne od kompilatora) skutkować nagłym zakończeniem uruchomionego programu.

Instrukcja przypisania (ang. *assignment statement*), służy do nadania pewnej zmiennej określonej wartości. Jej składnię obrazuje poniższy diagram:



Tutaj = oznacza tzw. operator przypisania (ang. *assignment operator*).

Instrukcja przypisania działa w następujący sposób. Wpierw obliczana jest wartość wyrażenia Expression. Dopiero następnie wynik wpisywany jest do zmiennej o nazwie Identifier.

Prześledźmy poniższy kod.

```
int i;       // deklaracja zmiennej
int j;
i = 4;       // niech i od teraz bedzie rowne 4
j = i;       /* niech j bedzie rowne wartosci aktualnie
                przechowywanej w zmiennej i */
cout << j; // wypisuje "4" na konsoli
```

Informacja

Tekst drukowany kursywą i szarą czcionką to **komentarze**. Komentarze są albo zawarte między znakami /* i */ (i wtedy mogą zajmować wiele wierszy kodu), albo następują po znakach // (i wtedy sięgają do końca aktualnego wiersza).

Treść komentarzy jest ignorowana przez kompilator i nie ma wpływu na wykonanie programu. Opisywanie tworzonego kodu jest bardzo dobrym nawykiem, bo sprawia, że jest on bardziej zrozumiały dla czytelnika.

W języku **C++** istnieje specjalny obiekt o nazwie `cout`, umożliwiający wypisywanie wartości zmiennych różnych typów na ekran monitora (a dokładnie, na tzw. standardowe wyjście). Zmienną „wysyła" się na ekran za pomocą operatora <<.

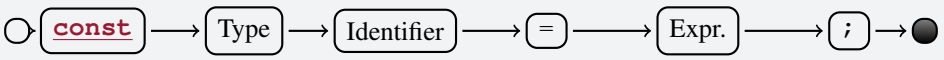
Istnieje jeszcze inny sposób przypisywania wartości zmiennym, dzięki któremu dane nie muszą być z góry ustalane podczas pisania kodu. Informacje te można podać bowiem w trakcie działania programu, prosząc użytkownika, aby je wprowadził za pomocą klawiatury. Dokonujemy tego za pośrednictwem obiektu `cin` i operatora >>.

Przykład:

```
int x;
cout << "Podaj wartosc x: ";
cin >> x; /* wczytanie wartosci x z tzw. standardowego
     wejscia (domyslnie jest to klawiatura komputera) */
cout << "Teraz x=" << x << endl;
```

Ciekawostka

Przy okazji omawiania operatora przypisania, warto wspomnieć o możliwości definiowania **stałych symbolicznych**, według składni:



Przy tworzeniu stałej symbolicznej należy od razu przypisać jej wartość. Cechą zasadniczą stałych jest to, że po ich zainicjowaniu nie można wcale zmieniać ich wartości. Warto je więc

stosować, by wyeliminować możliwość pomyłki programisty.

Przykłady:

```
const int dlugoscCiagu = 10;
const double pi = 3.14159;
const double G = 6.67428e-11; /* to samo, co 6,67428 · 10^-11. */
pi = 4.3623; // blad - pi jest "tylko do odczytu"
```

3.1.6 Rzutowanie (konwersja) typów. Hierarchia typów

Zastanówmy się, co się stanie, jeśli zechcemy przypisać zmiennej jednego typu wartość innego typu.

```
double x; // deklaracja zmiennej
int i = 4, j; /* deklaracja dwoch zmiennych,
       zmienna i zostanie zainicjowana wartoscia 4,
       zmienna j jest niezainicjowana */

x = i;       /* inny typ... niejawna konwersja na 4.0,
                tzw. promocja */

x = 3.14159;

j = x;       // blad! (typ docelowy jest "mniej pojemny")
j = (int)x; // jawna konwersja (tzw. rzutowanie) - OK

cout << j << ", " << x; // wypisze na ekran "3, 3.14159"
```

Przypisanie wartości typu **double** do zmiennej typu **int** zakończy się błędem kompilacji. Jest to spowodowane tym, że w tym miejscu mogłaby wystąpić utrata informacji. W języku **C++** obowiązuje następująca **hierarchia typów**:

$$\overbrace{\texttt{bool}}^{\text{typ logiczny}} \subset \overbrace{\texttt{char} \subset \texttt{short} \subset \texttt{int} \subset \texttt{long long}}^{\text{typy całkowite}} \Subset \overbrace{\texttt{float} \subset \texttt{double}}^{\text{typy zmiennoprzecinkowe}}.$$

Promocja typu, czyli konwersja na typ silniejszy, bardziej „pojemny”, odbywa się automatycznie. Dlatego w powyższym kodzie instrukcja `x=i;` wykona się poprawnie.

Z drugiej strony, rzutowanie do słabszego typu musi być zawsze jawne. Należy *explicité* oznajmić kompilatorowi, że postępujemy świadomie. Służy do tego tzw. **operator**

rzutowania. Taki operator umieszczamy przed rzutowanym wyrażeniem. Oprócz zastosowanej w powyższym przykładzie składni `(typ)wyrazenie`, dostępne są ponadto: `typ(wyrazenie)` oraz `static_cast<typ>(wyrazenie)`.

Ważne

W przypadku rzutowania typu zmiennopozycyjnego do całkowitoliczbowego następuje **obcięcie części ułamkowej**, np. `(int)3.14` da w wyniku `3`, a `(int)-3.14` zwróci wartość `-3`.

Z kolei dla konwersji do typu `(bool)`, wartość równa `0` da zawsze wynik `false`, a różna od `0` będzie przekształcona na `true`.

Przy konwersji typu `bool` do całkowitoliczbowego wartość `true` zostaje zamieniona zawsze na `1`, a `false` na `0`.

3.2 Wyrażenia. Operatory

Ważne

Wyrażenie (ang. *expression*) to ciąg operatorów i tzw. operandów, czyli argumentów, na których dokonujemy operacji.

Najprostsze wyrażenia konstruujemy, nie używając operatorów wcale, podając po prostu:

- stałą całkowitą, zmiennoprzecinkową lub logiczną (np. `7`, `-4.1e7`, `false`),
- identyfikator zmiennej lub stałej symbolicznej (wartość zostanie z niej pobrana),
- wartość zwracaną przez pewną funkcję (więcej o funkcjach w kolejnych wykładach).

Wyrażenia bardziej złożone możemy konstruować za pomocą dostępnych operatorów unarnych (jednoargumentowych) lub binarnych (dwuargumentowych). Jako parametry (czyli właśnie operandy) przyjmują one inne wyrażenia.

Informacja

Zauważmy, że definicja wyrażenia jest rekurencyjna; działania na wyrażeniach też są wyrażeniami.

Ciekawostka
Na przykład `2+3*7` jest de facto wyrażeniem skonstruowanym przy użyciu dwóch wyrażeń a i b połączonych ze sobą operatorem `+`, gdzie a jest wyrażeniem o wartości stałej (całkowitej) `2`, a b jest wynikiem obliczenia wyrażenia `3*7` (które z kolei rozbija się na prostsze podwyrażenia).

Rozkład każdego wyrażania jest wykonywany właśnie w ten sposób przez kompilator podczas przetwarzania kodu źródłowego na kod wynikowy.

3.2.1 Operatory arytmetyczne

Operatory arytmetyczne mogą byś stosowane (poza wyszczególnionymi wyjątkami) na wyrażeniach typu całkowitego i zmiennoprzecinkowego. Operatory binarne prezentujemy w poniższej tabeli.

Operator	Znaczenie
`+`	dodawanie
`-`	odejmowanie
`*`	mnożenie
`/`	dzielenie rzeczywiste lub całkowite
`%`	reszta z dzielenia (tylko liczby całkowite)

Informacja
Zauważmy, że w języku **C++** nie ma określonego operatora potęgowania.

Przykłady:

```
1 cout << 2+2 << endl;
2 cout << 3.0/2.0 << endl;
3 cout << 3/2 << endl;
4 cout << 7%4 << endl;
5 cout << 1/0.0 << endl; // uzgodnienie typow == 1.0/0.0
```

Zastosowanie operatorów do wyrażeń dwóch różnych typów powoduje konwersję operandu o typie słabszym do typu silniejszego operandu. Wynikiem wykonania powyższego kodu będzie:

```
4
1.5
1     (dlaczego?)
3
Inf
```

Zajmijmy się teraz **operatorami unarnymi**.

Operator	Znaczenie
+	unarny plus (nic nie robi)
−	unarny minus (zmiana znaku na przeciwny)

Przykład:

```
int i = -4; // zmiana znaku stalej 4
i = -i;     // zmiana znaku zmiennej i
cout << i;  // wynik: 4
```

Ponadto określone zostały dwa operatory unarne, które można zastosować tylko do *zmiennych* typu całkowitego. Operator **inkrementacji** (++) służy do zwiększania wartości zmiennej o 1, a operator **dekrementacji** (−−) zmniejszania o 1.

Każdy z powyższych operatorów posiada dwa warianty: przedrostkowy (ang. *prefix*) oraz przyrostkowy (ang. *suffix*). W przypadku operatora inkrementacji/dekrementacji **przedrostkowego**, wartością całego wyrażenia jest wartość zmiennej po wykonaniu operacji (operator ten działa tak, jakby wpierw aktualizował stan zmiennej, a potem zwracał swoją wartość w wyniku). Z kolei dla operatora **przyrostkowego** wartością wyrażenia jest stan zmiennej sprzed zmiany (najpierw zwraca wartość zmiennej, potem aktualizuje jej stan).

Zadanie

Przeanalizuj następujący przykład.

```
int i = 5, j = 9;

--j;      // wynik: j==8, to samo to j = j-1;
j--;      //        j==7, to samo to j = j-1;

j = ++i; // wynik: i==6, j==6 (przedrostkowy)
j = i++; //        j==6, i==7 (przyrostkowy)

j = ++100; // blad, 100 jest stala
```

Podsumowując, `j = ++i;` można zatem zapisać jako dwie instrukcje: `i = i+1; j = i;` Z drugiej strony, `j = i++;` jest równoważne operacjom: `j = i; i = i +1;`

Niewątpliwą zaletą tych dwóch operatorów jest możliwość napisania zwięzłego fragmentu kodu. Niestety, jak widać, odbywa się to kosztem czytelności.

3.2.2 Operatory relacyjne

Operatory relacyjne służą do porównywania wartości innych wyrażeń. Wynikiem takiego porównania jest wynik typu **bool**. Są to operatory binarne.

Operator	Znaczenie
==	czy równe?
!=	czy różne?
<	czy mniejsze?
<=	czy nie większe?
>	czy większe?
>=	czy nie mniejsze?

Ważne
Często popełnianym błędem jest omyłkowe użycie operatora przypisania (=) w miejscu, w którym powinien wystąpić operator porównania (==).

Przykłady:

```
bool w1 = 1==1; // true
int  w2 = 2>=3; // 0, czyli (int)false
bool w3 = (0.0 == (((1e34 + 1e-34) - 1e34) - 1e-34));
     // false - blad zaokraglen liczb zmiennoprzecinkowych
```

3.2.3 Operatory logiczne

Operatory logiczne służą do działań na wyrażeniach typu **bool** (lub takich, które są sprowadzalne do **bool**).

Operator	Znaczenie
!	negacja (unarny)
||	alternatywa (lub)
&&	koniunkcja (i)

Wyniki powyższych działań na wszystkich możliwych wartościach operandów zestawione są w poniższych tzw. **tablicach prawdy**.

!	
false	**true**
true	**false**

||	**false**	**true**
false	**false**	**true**
true	**true**	**true**

&&	false	true
false	false	false
true	false	true

Kilka przykładów:

```
bool w1 = (!true);        // false
bool w2 = (1<2 || 0<1); // true -> (true || false)
bool w3 = (1.0 && 0.0); // false -> (true && false)
```

Ciekawostka

Okazuje się, że komputery są czasem leniwe. W przypadku wyrażeń składających się z wielu podwyrażeń logicznych, obliczane jest tylko to, co jest potrzebne do ustalenia wyniku. I tak w przypadku koniunkcji, jeśli jeden operand ma wartość `false`, to wyznaczanie wartości drugiego nie jest potrzebne. Podobnie jest dla alternatywy i jednego operandu o wartości `true`. Na przykład:

```
int a = 0;
bool p = (true || (++a==0)); // leniwy
cout << a;                   // a==0
bool q = (true && (++a==0)); // tutaj juz musi policzyc
cout << a;                   // a==1
```

3.2.4 Operatory bitowe (*)

Jak już zdążyliśmy się przekonać, każda informacja, w tym i liczby, są przechowywane w pamięci komputera w postaci ciągu bitów. Operatory bitowe działają na poszczególnych bitach operandów typu całkowitego. Zwracany wynik jest też liczbą całkowitą.

Operator	Znaczenie
~	bitowa negacja (unarny)
|	bitowa alternatywa
&	bitowa koniunkcja
^	bitowa alternatywa wyłączająca (albo, ang. *exclusive-or*)
<< k	przesunięcie w lewo o k bitów (ang. *shift-left*)
>> k	przesunięcie w prawo o k bitów (ang. *shift-right*)

Operator bitowej negacji zamienia każdy bit liczby na przeciwny.

Ważne
Nie należy milić operatorów ~ i !.

Operatory bitowej alternatywy, koniunkcji i alternatywy wyłączającej zestawiają bity na odpowiadających sobie pozycjach dwóch operandów według następujących reguł.

~	
0	1
1	0

|	0	1
0	0	1
1	1	1

&	0	1
0	0	0
1	0	1

^	0	1
0	0	1
1	1	0

Np. `0xb6 ^ 0x5f == 0xe9`, gdyż

$$\begin{array}{rcccccccc} & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hat{} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ \hline & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

Operatory przesunięcia zmieniają pozycje bitów w liczbie. Bity, które nie „mieszczą" się, są tracone. W przypadku operatora <<, na zwalnianych pozycjach wstawiane są zera. Operator >> wstawia zaś na zwalnianych pozycjach bit znaku. Dokonajmy przesunięcia bitów przykładowej liczby 8-bitowej o trzy pozycje w lewo.

$$\begin{array}{ccccc} 10101011 & & & & \overbrace{101\mathbf{01011000}}^{\text{wynik}} \\ 0xab & << 3 & == & & 0x58 \end{array}$$

Oto kolejne przykłady:

```
1 int p1 = 1 | 2;      /* 3,   bo  00000001 | 00000010 ->
                                    00000011 */
2 int p2 = ~0xf0;      /* 15,  bo ~11110000            ->
                                    00001111 */
3 int p3 = 0xb ^ 5;    /* 14,  bo  00001011 ^ 000000101->
                                    00001110 */
4 int p4 = 5 << 2;     /* 20,  bo  00000101 << 2       ->
                                    00010100 */
5 int p5 = 7 >> 1;     /* 3,   bo  00000111 >> 1       ->
                                    00000011 */
6 int p6 = -128 >> 4;  /* -8,  bo  10000000 >> 4       ->
                                    11111000 */
```

Ciekawostka
Uwaga: n<<k równoważne jest (o ile nie nastąpi przepełnienie) $n \times 2^k$, a n>>k równoważne jest całkowitoliczbowej operacji $n \times 2^{-k}$ (k-krotne dzielenie całkowite przez 2).

Informacja
Zauważmy, że operatory << i >> mają inne znaczenie w przypadku użycia ich na obiektach, odpowiednio, `cout` i `cin`. Jest to praktyczny przykład zastosowania tzw. przeciążania operatorów, czyli różnicowania ich znaczenia w zależności od kontekstu.

3.2.5 Operatory łączone

Dla skrócenia zapisu zostały też wprowadzone tzw. **operatory łączone**, które w zwięzły sposób łączą operacje arytmetyczne lub bitowe i przypisanie:

`+=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=.`

Np. `x += 10;` znaczy to samo, co `x = x + 10;`

3.2.6 Priorytety operatorów

Zaobserwowaliśmy, że w jednym wyrażeniu można używać wielu operatorów. Kolejność wykonywania działań jest jednak ściśle określona. Można ją zawsze wymusić za pomocą nawiasów okrągłych (...).

Domyślnie jednak wyrażenia obliczane są według priorytetów, zestawionych w tab. 3.1 w kolejności od największego do najmniejszego. W przypadku operatorów o tym samym priorytecie, operacje wykonywane są w kolejności od lewej do prawej.

Przykłady:

```
1 int p1  = 2+4*3/2;     // p1 = (2+((4*3)/2));
2 p1 += 2>3 == !true;    /* p1 += ((2>3) == (!true));
3                                  czyli to samo, co p1++; */
4 cout << p1;            // 9
```

Tabela 3.1. Priorytety operatorów.

13	`++, --` (przyrostkowe)
12	`++, --` (przedrostkowe), `+, -` (unarne), `!, ~`
11	`*, /, %`
10	`+, -`
9	`<<, >>`
8	`<, <=, >, >=`
7	`==, !=`
6	`&`
5	`^`
4	`|`
3	`&&`
2	`||`
1	`=, +=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=`

3.3 Instrukcje sterujące

3.3.1 Bezpośrednie następstwo instrukcji

Znamy już następujące typy instrukcji:
- instrukcje deklarujące zmienne i stałe lokalne,
- instrukcje przypisania.

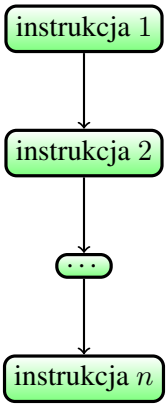
Wiemy też, w jaki sposób wypisywać wartości wyrażeń na ekran oraz jak pobierać wartości z klawiatury.

Zauważmy, iż do tej pory nasz kod w języku **C++** nie miał żadnych rozgałęzień. Wszystkie instrukcje były wykonywane jedna po drugiej. Owo tzw. **bezpośrednie następstwo** instrukcji obrazuje rys. 3.1. Jest to tak zwany schemat blokowy algorytmu (ang. *control flow diagram*, czyli schemat przepływu sterowania).

W kolejnych podrozdziałach poznamy konstrukcje, które pozwolą nam sterować przebiegiem programu. Dzięki temu będziemy mogli dostosowywać jego działanie do różnych szczególnych przypadków — w zależności od wartości przetwarzanych danych.

3.3.2 Instrukcja warunkowa if

Podczas zmagania się z różnymi problemami prawie zawsze spotykamy sytuację, gdy musimy dokonać jakiegoś wyboru. Na przykład, rozwiązując równanie kwadratowe inaczej postępujemy, gdy ma ono dwa pierwiastki rzeczywiste, a inaczej gdy nie ma ich wcale. Policjant mierzący fotoradarem prędkość nadjeżdżającego samochodu inaczej postąpi, gdy stwierdzi, że dopuszczalna prędkość została przekroczona o 50 km/h, niż gdyby się okazało, że kierowca jechał prawidłowo. Student przed sesją głowi się, czy

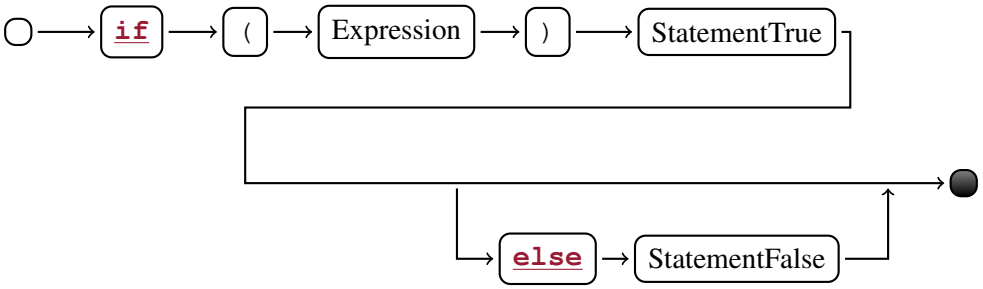


Rysunek 3.1. Schemat blokowy bezpośredniego następstwa instrukcji.

przyłożyć się bardziej do programowania, algebry, do obu na raz (jedynie słuszna koncepcja) czy też dać sobie spokój i iść na imprezę itp.

W języku **C++** dokonywanie tego typu wyborów umożliwia **instrukcja warunkowa** `if` (od ang. *jeśli*). Wykonuje ona pewien fragment kodu **wtedy i tylko wtedy**, gdy pewien dany warunek logiczny jest spełniony. Może też, opcjonalnie, zadziałać w inny sposób w przeciwnym przypadku (`else`).

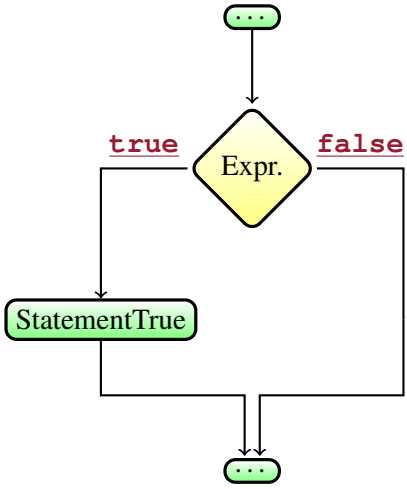
Jej składnię przedstawia poniższy diagram.



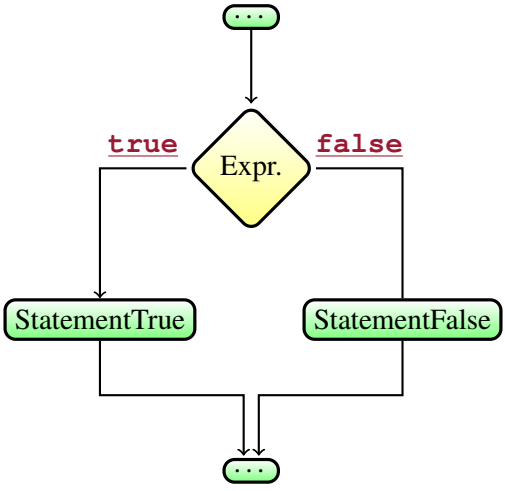
Expression jest wyrażeniem (koniecznie ujętym w nawiasy) sprowadzalnym do typu `bool`, reprezentującym sprawdzany warunek logiczny. StatementTrue jest instrukcją wykonywaną wtedy i tylko wtedy, gdy wyrażenie Expression ma wartość `true`. Z kolei StatementFalse wykonywane jest w przeciwnym przypadku.

Zauważmy, że podinstrukcja `else` jest opcjonalna. Schemat blokowy instrukcji `if` bez części `else` przedstawia rys. 3.2.

Jeśli jednak podinstrukcja `else` jest obecna, stosowany jest schemat zobrazowany na rys. 3.3.



Rysunek 3.2. Schemat blokowy instrukcji warunkowej `if`.

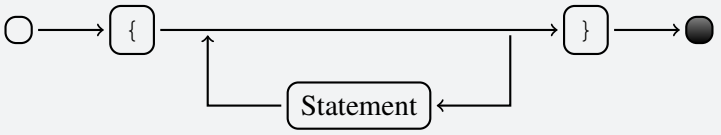


Rysunek 3.3. Schemat blokowy instrukcji warunkowej `if`...`else`.

Ważne

Jeżeli chcemy wykonać warunkowo więcej niż jedną instrukcję tworzymy w tym celu tzw. blok instrukcji, czyli **instrukcję złożoną** (ang. *compound statement*), ograniczoną z obu stron nawiasami klamrowymi {...}.

Składnia instrukcji złożonej przedstawiona jest na poniższym diagramie.



Informacja

Aby zwiększyć czytelność kodu, instrukcje w bloku powinniśmy **wyróżnić wcięciem**. Jest to jedna z zasad dobrego pisania programów.

Rozważmy przykład, w którym wyznaczane jest minimum z dwóch liczb całkowitych.

```
int x, y;
cin >> x >> y;   // wprowadz x i y z klawiatury

if (x < y)       // tutaj nie ma srednika
   cout << x;
else             // tutaj nie ma srednika
   cout << y;
```

Instrukcje warunkowe można, rzecz jasna, zagnieżdżać. Oto przykład służący do znajdowania minimum z trzech liczb.

```
int x, y, z;
cin >> x >> y >> z;

if (x < y) {
   if (z < x)
      cout << z;
   else
      cout << x;
}
else {
```

```
    if (z < y)
        cout << z;
    else
        cout << y;
}
```

Informacja

Z zagnieżdżaniem instrukcji warunkowych trzeba jednak uważać. Słowo kluczowe **else** dotyczy najbliższej instrukcji **if**. Zatem poniższy kod zostanie wykonany przez komputer inaczej niż sugerują to wcięcia.

```
int x, y, z;
cin >> x >> y >> z;

if (x != 0)
    if (y > 0 && z > 0)
        cout << y*z/x;
else  // else dotyczy if (y > 0 && z > 0)
    cout << ":(";
```

Aby dostosować ten kod do wyraźnej intencji programisty, należałoby objąć fragment

```
if (y > 0 && z > 0)cout << y*z/x;
```

w nawiasy klamrowe.

Warto pamiętać, że jako StatementFalse może się pojawić kolejna instrukcja **if**... **else**. Dzięki temu można obsłużyć w danym fragmencie programu więcej niż dwa rozłączne przypadki rozpatrywanego problemu na raz.

```
int x;
cout << "Okresl swoj nastroj w skali 1-3:";
cin >> x;

if (x == 1)
    cout << ":(";
else if (x == 2)
    cout << ":|";
else
    cout << ":)";
```

3.3.3 Pętle

Oprócz instrukcji warunkowej `if`, rozgałęziającej przebieg sterowania „w dół”, możemy skorzystać z tzw. **pętli** (ang. *loops* albo *iterative statements*). Umożliwiają one wykonywanie tej samej instrukcji (albo, rzecz jasna, bloku instrukcji) wielokrotnie, być może na innych danych, **dopóki** pewien warunek logiczny jest spełniony.

Pomysł ten jest oczywiście bliski naszemu życiu. Pętle znajdują zastosowanie, jeśli zachodzi konieczność np. wykonania pewnych podobnych operacji na każdym z elementów danego zbioru. Np. życząc sobie zsumować wydatki poczynione na przyjemności podczas każdego dnia wakacji, rozpatrujemy kolejno sumę “odpływów” w dniu pierwszym, potem drugim, a potem trzecim, a potem itd., czyli tak naprawdę w dniu i-tym, gdzie $i = 1, 2, \ldots, n$.

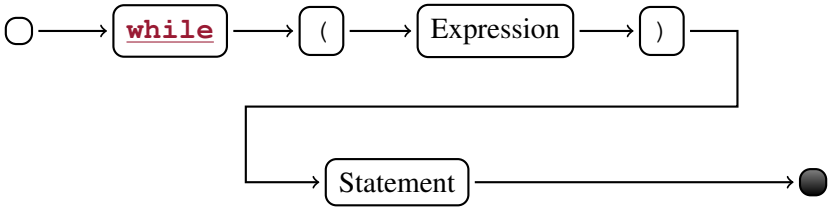
Dalej, ileż to razy słyszeliśmy słowa mądrej mamy „dopóki (!nauczysz_się) zostajesz_w_domu;”. To jest również przykład (niekoniecznie świadomie użytej) pętli.

W języku **C++** zdefiniowane zostały trzy instrukcje realizujące tego typu ideę:
- `while`,
- `for`,
- `do ... while`.

Przyjrzyjmy się im dokładniej.

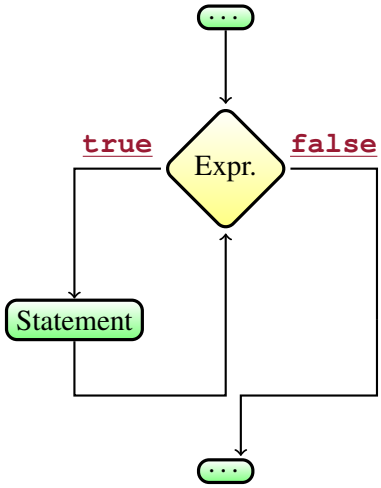
3.3.3.1 Pętla while

Najprostszą konstrukcją realizującą pętlę, jest instrukcja `while`. Jej składnię przedstawia poniższy diagram:



Expression jest wyrażeniem sprowadzalnym do typu `bool`. Najlepiej będzie, gdy będzie istnieć możliwość jego zmiany na skutek wykonywania instrukcji podanej w części Statement. W przeciwnym wypadku stworzymy program, który nigdy się nie zatrzyma.

Schemat blokowy przedstawionej pętli zobrazowany jest na rys. 3.4.



Rysunek 3.4. Schemat blokowy pętli `while`.

Oto w jaki sposób możemy wypisać na ekranie kolejno liczby 1,2,...,100 i zsumować ich wartości (przypomnijmy sobie w tym miejscu problem młodego Gaussa z pierwszego wykładu):

```
int i = 1;
int suma = 0;

while (i <= 100) /* ten warunek przestanie
                    byc kiedys spelniony... */
{
    cout << i << endl;
    suma += i;
    i++;  /* ...gdyz jest zalezny od wartosci zmiennej i,
             ktora sie w tym miejscu zmienia */
}

cout << "Suma=" << suma << endl;
```

Ważne

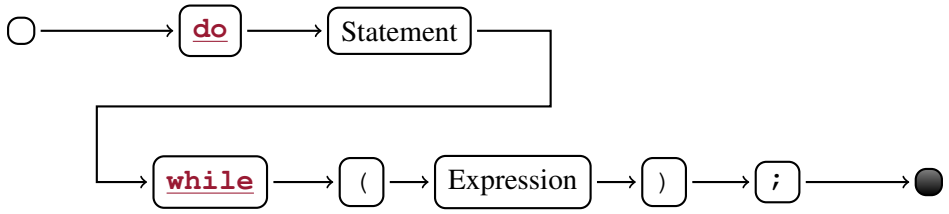
Sam średnik (;) oznacza instrukcję pustą. Dlatego pętla

```
while (true) ;
```

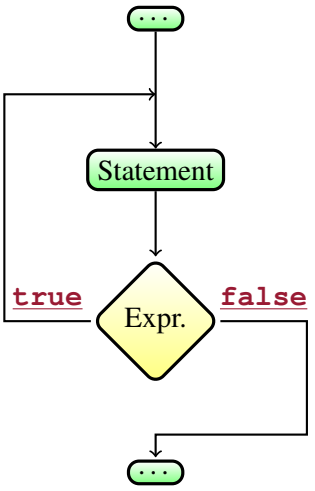
będzie w nieskończoność nie robić nic. Nie bierzmy z niej przykładu.

3.3.3.2 Pętla do... while

Inną odmianą pętli jest konstrukcja **do** ... **while**, określona według składni:



Jak widać na schemacie blokowym na rys. 3.5, używamy jej, gdy chcemy zapewnić, by instrukcja Statement została wykonana **co najmniej** jeden raz.



Rysunek 3.5. Schemat blokowy pętli **do**... **while**.

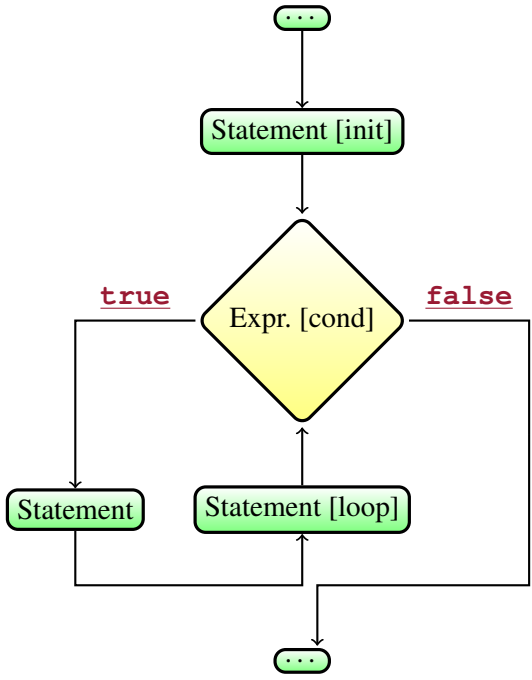
Zapiszmy dla ćwiczenia powyższy przykład za pomocą pętli **do** ... **while**, nieco go jednak urozmaicając.

```
int i, suma = 0;
cin >> i;
do
{
    suma += i;
    cout << i << endl;
    ++i;
}
while (i<=100);
```

Zauważmy, że tym razem wartość początkowa zmiennej `i` zostaje wprowadzona z klawiatury. Za pomocą wprowadzonej pętli możemy wymusić, aby wypisanie wartości na ekran i ustalenie odpowiedniej wartości zmiennej `suma` dokonało się zawsze; także w przypadku, gdy `i > 100`.

3.3.3.3 Pętla for

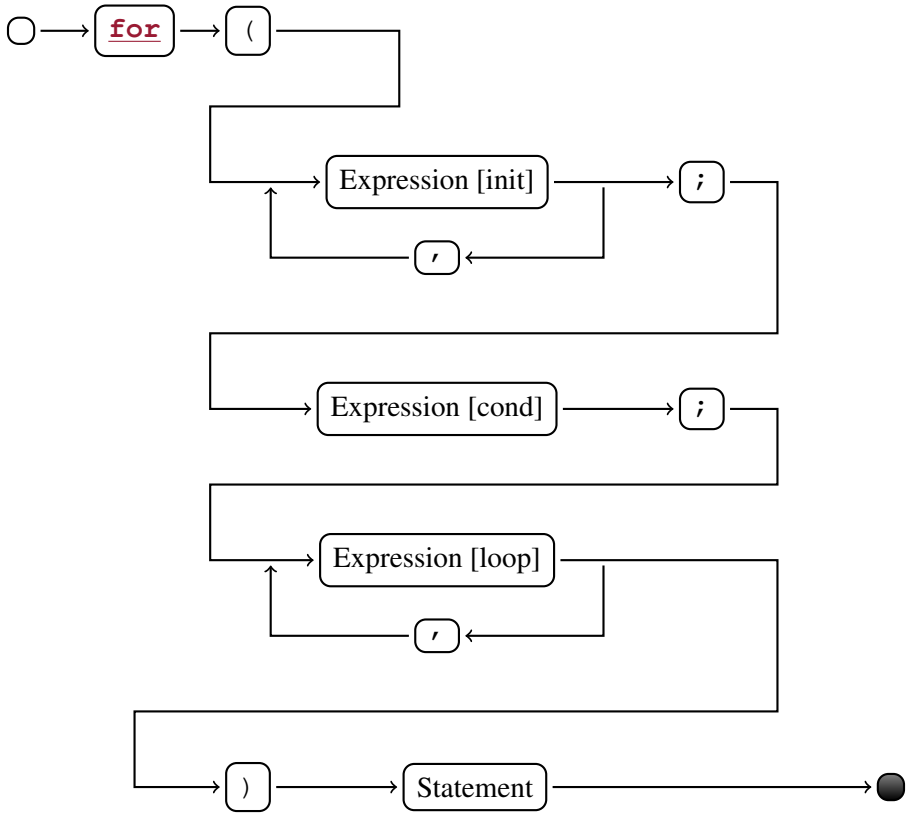
Dość często zachodzi potrzeba ustalenia przebiegu programu tak jak na schemacie blokowym przedstawionym na rys. 3.6.



Rysunek 3.6. Schemat blokowy pętli **for**.

Tutaj Statement [init] oznacza instrukcję inicjującą, wykonywaną tylko raz, na początku obliczeń (przygotowanie do wykonania pracy). Z kolei Statement [loop] jest wyrażeniem, które zostaje wykonane zawsze na końcu każdej iteracji i służy np. do aktualizacji licznika pętli (przygotowanie do kolejnej iteracji).

Tego typu schemat warto zakodować za pomocą pętli **for**, której (dość skomplikowaną na pierwszy rzut oka) składnię przedstawia poniższy diagram.



Expression [init] jest odpowiednikiem Statement [init] z rys. 3.6 bez końcowego średnika. Podobnie jest w przypadku Expression [loop]

Wróćmy do naszego wyjściowego przykładu. Można go także rozwiązać, korzystając z wprowadzonej właśnie pętli.

```
int suma = 0;

for (int i=1; i<=100; ++i)
{
    cout << i << endl;
    suma += i;
}

cout << "Suma=" << suma << endl;
```

Informacja

Jeśli chcemy wykonać więcej niż jedną instrukcję inicjującą lub aktualizującą, należy każdą z nich oddzielić przecinkiem (nie średnikiem, ani nie tworzyć dlań bloku).

Zatem powyższy przykład można zapisać i tak:

```
int suma = 0; /* chcemy, by ta zmienna byla dostepna
    rowniez po zakonczeniu dzialania petli for... */

for (int i=1; i<=100; suma += i, ++i)
    cout << i << endl;

cout << "Suma=" << suma << endl;
```

3.3.3.4 break i continue

Niekiedy (choć dość rzadko) zachodzi konieczność zmiany domyślnego przebiegu wykonywania pętli.

Informacja

Instrukcja **break** służy do natychmiastowego wyjścia z pętli (niezależnie od wartości warunku testowego).

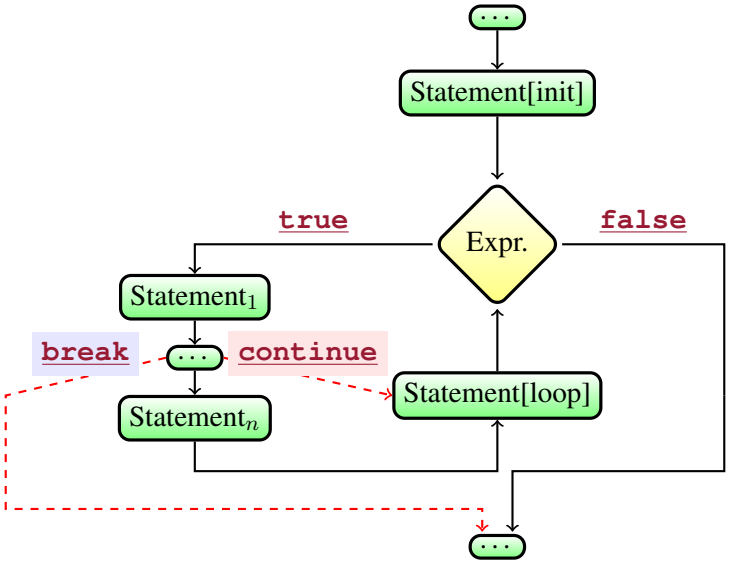
Z kolei instrukcja **continue** służy do przejścia do kolejnej iteracji pętli (ignorowane są instrukcje następujące po **continue**). W przypadku pętli **for** instrukcja aktualizująca jest jednak wykonywana.

Schemat blokowy z rys. 3.7 przedstawia zmianę przebiegu programu za pomocą omawianych instrukcji na przykładzie pętli **for**.

W przypadku zastosowania kilku pętli zagnieżdżonych, instrukcje te dotyczą tylko jednej (wewnętrznej) pętli.

Przykład (niezbyt pomysłowy): wypisywanie kolejnych liczb parzystych od 2 do 100:

```
for (int i=2; i<=100; ++i)
{
    if (i % 2 == 1)
        continue;
    cout << i;
}
```



Rysunek 3.7. Instrukcje `break` i `continue` a pętla `for`.

3.4 Ćwiczenia

Zadanie 3.30. Jaką wartość mają następujące liczby zmiennoprzecinkowe dane w notacji naukowej: a) `4.21e6`, b) `1.95323e2`, c) `2.314e-4`, d) `4.235532e-2`?

Zadanie 3.31. Wyznacz wartość następujących wyrażeń. Ponadto określ typ zwracany przez każde z nich.

a) `10.0+15.0/2+4.3`,
b) `10.0+15/2+4.3`,
c) `3.0*4/6+6`,
d) `20.0-2/6+3`,
e) `10+17*3+4`,
f) `10+17/3.0+4`,
g) `3*4%6+6`,
h) `3.0*4%6+6`,
i) `10+17%3+4`.

Zadanie 3.32. Dane są trzy zmienne zadeklarowane w następujący sposób.

```
double a = 10.6, b = 13.9, c = -3.42;
```

Oblicz wartość poniższych wyrażeń.

a) `int(a)`,
b) `int(c)`,
c) `int(a+b)`,
d) `int(a)+b+c`,
e) `int(a+b)*c`,
f) `double(int(a))/c`.

Zadanie 3.33. Korzystając z przekształceń logicznych (np. praw De Morgana, praw rozdzielności), uprość następujące wyrażenia.

a) `!(!p)`,
b) `!p && !q`,
c) `!(!p || !q || !r)`,
d) `!(b>a && b<c)`,
e) `!(a>=b && b>=c && a>=c)`,
f) `(a>b && a<c) || (a<c && a>d)`,
g) `p || !p`.

Zakładamy, że `a`, `b`, `c`, `d` są typu **`double`**, a `p`, `q`, `r` są typu **`bool`**.

(*) Zadanie 3.34. Jaki wynik dadzą poniższe operacje bitowe dla danych typu **`short`**?

a) `0x0FCD | 0xFFFF`,
b) `364 & 0x323`,
c) `~163`,
d) `0xFC93 ^ 0x201D`,
e) `14 << 4`,
f) `0xf5a3 >> 8`.

Zadanie 3.35. Dane są zmienne `ax`, `ay`, `bx`, `by`, `cx`, `cy` typu **`double`**, reprezentujące współrzędne trzech punktów w $\mathbb{R}^2$: $\mathbf{a} = (ax, ay)$, $\mathbf{b} = (bx, by)$, $\mathbf{c} = (cx, cy)$. Napisz program, który wyznaczy kwadrat promienia okręgu przechodzącego przez $\mathbf{a}$, $\mathbf{b}$ i $\mathbf{c}$, określony wzorem

$$r^2 = \frac{|\mathbf{a}-\mathbf{c}|^2\,|\mathbf{b}-\mathbf{c}|^2\,|\mathbf{a}-\mathbf{b}|^2}{4\,|(\mathbf{a}-\mathbf{c})\times(\mathbf{b}-\mathbf{c})|^2},$$

gdzie $|\mathbf{a}-\mathbf{b}| = \sqrt{(ax-bx)^2+(ay-by)^2}$ oraz $\mathbf{a}\times\mathbf{b} = axby - aybx$.

Zadanie 3.36. Napisz program (a potem dokładnie przetestuj), który dla danych $a, b, c, d, e, f \in \mathbb{R}$ rozwiąże układ dwóch równań liniowych względem niewiadomych $x, y \in \mathbb{R}$:

$$\begin{cases} ax + by &= c, \\ dx + ey &= f. \end{cases}$$

Poprawnie identyfikuj przypadki (np. wypisując stosowny komunikat na ekranie), w których dany układ jest sprzeczny albo nieoznaczony. Współczynniki układu pobierz z klawiatury. Do reprezentacji zbioru $\mathbb{R}$ użyj typu **`double`**.

(*) Zadanie 3.37. Dane są liczby rzeczywiste $x_1, \ldots, x_4, y_1, \ldots, y_4 \in \mathbb{R}$. Sprawdź, korzystając z jak najmniejszej liczby warunków logicznych, czy prostokąty $[x_1, x_2] \times [y_1, y_2]$ oraz $[x_3, x_4] \times [y_3, y_4]$ mają część wspólną (tj. przecinają się).

Zadanie 3.38. Napisz fragment kodu, który dla danego $x \in \mathbb{R}$ oblicza wartość funkcji:

a) $f(x) = x\,|x| + \left\lfloor \frac{|x-5|}{|\frac{x}{2}|+1} \right\rfloor$.

Ponadto dla $n \in \mathbb{N}_0$:

b) $h(n) = \sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor$ jeśli $n > 0$ oraz $h(0) = 0$.

Zadanie 3.39. Wyraź następujące pętle przy użyciu instrukcji korzystającej z pętli **`for`**.

a) **dla** $i = 0, 1, \ldots, n-1$ wypisz i (dla pewnego $n \in \mathbb{N}$),
b) **dla** $i = n, n-1, \ldots, 0$ wypisz i (dla pewnego $n \in \mathbb{N}$),

c) **dla** $j = 1, 3, \ldots, 2k - 1$ wypisz j (dla pewnego $k \in \mathbb{N}$),
d) **dla** $i = 1, 2, 4, 7, \ldots, n$ wypisz i (dla pewnego $n \in \mathbb{N}$),
e) **dla** $j = 1, 2, 4, 8, 16, \ldots n$ wypisz j (dla pewnego $n \in \mathbb{N}$),
f) **dla** $j = 1, 2, 4, 8, 16, \ldots, 2^k$ wypisz j (dla pewnego $k \in \mathbb{N}$),
g) **dla** $x = a, a + \delta, a + 2\delta, \ldots, b$ wypisz x (dla pewnych $a, b, \delta \in \mathbb{R}, a < b, \delta > 0$).

Zadanie 3.40. Zaprogramuj w języku **C++** algorytm Euklidesa do wyznaczania największego wspólnego dzielnika dwóch liczb całkowitych nieujemnych a, b pobranych z klawiatury (zob. rozdz. 1). Poprawnie identyfikuj wszystkie możliwe przypadki danych wejściowych, w tym m.in. $a < b$, $b < a$, $a < 0$, $b < 0$.

Zadanie 3.41. Spośród liczb $1, 2, \ldots, 100$ wypisz na ekran:
a) wszystkie podzielne przez 7, tzn. $7, 14, 21, \ldots$,
b) wszystkie podzielne przez 2 lecz niepodzielne przez 5, tzn. $2, 4, 6, 8, 12, \ldots$,
c) co drugą podzielną przez 5 lub podzielną przez 7, tzn. $5, 10, 15, 21, 28, \ldots$

Zadanie 3.42. Napisz fragment kodu, który sprawdzi, czy dana liczba naturalna jest liczbą pierwszą, czy też liczbą złożoną.

Zadanie 3.43. Napisz fragment kodu, który dla danej liczby $n \in \mathbb{N}$ oblicza wartość $\max\{k : 2^k \leqslant n\}$.

Zadanie 3.44. Napisz fragment kodu, który znajduje minimum z danych liczb naturalnych. Liczby odczytuj z klawiatury, póki użytkownik nie wprowadzi wartości $\leqslant 0$.

Zadanie 3.45. Napisz fragment kodu, który znajduje różnicę między maksimum a minimum z danych liczb rzeczywistych nieujemnych. Liczby odczytuj z klawiatury, póki użytkownik nie wprowadzi liczby ujemnej.

Zadanie 3.46. Napisz fragment kodu, który oblicza średnią harmoniczną i średnią arytmetyczną z danych liczb rzeczywistych dodatnich. Liczby odczytuj z klawiatury, póki użytkownik nie wprowadzi wartości $\leqslant 0$.

Zadanie 3.47. Napisz program, który dla danego n-elementowego (n jest dane z klawiatury na początku działania programu) uporządkowanego nierosnąco ciągu liczb naturalnych (użytkownik wprowadza z klawiatury kolejno wartości $a_1 \geqslant a_2 \geqslant \ldots \geqslant a_n \geqslant 1$) wyznacza wartości tzw. indeksu h Hirscha oraz indeksu g Egghego, danych wzorami:
a) $\max\{h = 1, \ldots, n : a_h \geqslant h\}$ (indeks h),
b) $\max\{g = 1, \ldots, n : \sum_{i=1}^{g} a_i \geqslant g^2\}$ (indeks g).
Postaraj się wymusić poprawność danych wejściowych, np. w przypadku nie wprowadzenia danych w określonym porządku, wypisz komunikat o błędzie.

Zadanie 3.48. Napisz fragment kodu, który aproksymuje wartość liczby π na podstawie wzoru

$$\pi \simeq 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right).$$

Wypisz kolejne przybliżenia, korzystając z $1, 2, 3, \ldots, 25$ początkowych wyrazów tego

szeregu.

Zadanie 3.49. Napisz fragment kodu, który aproksymuje wartość liczby e na podstawie wzoru

$$e \simeq 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots,$$

gdzie $n! = 1 \times 2 \times \cdots \times n$. Wypisz wynik dopiero wtedy, gdy różnica między kolejnymi wyrazami szeregu będzie mniejsza niż 10^{-9}.

Zadanie 3.50. Napisz fragmenty kodu, które posłużą do wyznaczenia wartości następujących wyrażeń.

a) 2^n dla pewnego $n \in \mathbb{N}$,
b) $\sum_{i=1}^{10} i$,
c) $\sum_{i=1}^{100} \frac{1}{i!}$,
d) $\prod_{i=1}^{5} \frac{i}{i+1}$,
e) $e^x \simeq \sum_{n=0}^{100} \frac{x^n}{n!}$ dla pewnego $x \in \mathbb{R}$,
f) $\ln(1+x) \simeq \sum_{n=1}^{100} \frac{(-1)^{n+1}}{n} x^n$ dla pewnego $x \in [-1, 1]$,
g) $\sin x \simeq \sum_{n=0}^{100} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$ dla pewnego $x \in \mathbb{R}$,
h) $\cos x \simeq \sum_{n=0}^{100} \frac{(-1)^n}{(2n)!} x^{2n}$ dla pewnego $x \in \mathbb{R}$,
i) $\arcsin x \simeq \sum_{n=0}^{100} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1}$ dla pewnego $x \in (-1, 1)$.

Zadanie 3.51. Korzystając ze wzoru na przybliżoną wartość funkcji sin podanego w zad. 3.50, utwórz program, który wydrukuje tablice przybliżonych wartości $\sin x$ dla $x = \frac{k}{n}\pi$, $k = 0, 1, \ldots, n$ i pewnego n, np. $n = 10$. Wynik niech będzie podany w postaci podobnej do poniższej.

```
x          sin(x)
0.0000000  0.0000000
0.3141593  0.3090170
...
3.1415927  0.0000000
```

(*) Zadanie 3.52. Napisz programy sprawdzające, czy następujące zdania logiczne są tautologiami.

a) $p \wedge \neg p$,
b) $\neg(\neg p) \Leftrightarrow p$,
c) $p \vee q \Leftrightarrow p \vee q$,
d) $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$,
e) $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$,
f) $(p \Leftrightarrow q) \Leftrightarrow (p \vee q) \wedge (\neg p \vee \neg q)$.

(*) Zadanie 3.53. Przeanalizuj poniższy kod.

```
1 int n, w=0, r=1;
2 cin >> n;
3 while (n>=r && r!=0) {
4     w = w + n&r;
5     r = r<<1;
```

```
}
cout << w;
```

a) Jaką funkcję oblicza podany fragment kodu?
b) Co się stanie, gdy zmienną `r` zainicjujemy wartością `2`?
c) Co się zepsuje, jeśli z pętli **`while`** usuniemy sprawdzenie warunku `r!=0`?

(*) Zadanie 3.54. Przeanalizuj poniższy kod.

```
int n, w=0, i=0;
cin >> n;
while (n>>i != 0) {
  w += (~n & (1<<i))>>i;
  i++;
}
cout << w;
```

a) Jaką funkcję oblicza podany fragment kodu?
b) Co się zepsuje, jeśli n będzie miało wartość ujemną?

3.5 Laboratoria

Zadanie 3.55. Ciąg Fibonacciego określony jest wzorem

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{dla } n \geqslant 2, \\ 1 & \text{dla } n = 0 \text{ lub } n = 1. \end{cases}$$

Napisz program, który dla danej liczby całkowitej $k \geqslant 0$ wypisze wartość F_k, czyli k-ty wyraz ciągu Fibonacciego. W przypadku podania $k < 0$ program powinien wyświetlić tylko komunikat o błędzie.

Zadanie 3.56. Napisz program, który sprawdzi, czy dana z klawiatury liczba całkowita f jest pewnym wyrazem ciągu Fibonacciego, tzn. czy istnieje takie $k \geqslant 0$, że $F_k = f$. Jeśli tak jest, to wypisujemy stosowną wartość k. W przeciwnym przypadku wyświetlamy komunikat, że dana wartość nie jest wyrazem ciągu Fibonacciego.

Wskazówka. Nie zapomnij o poprawnym obsłużeniu przypadków $f \leqslant 0$ oraz $f = 1$.

Zadanie 3.57. Zgadula. Napisz program do grania z komputerem w „zgadulę”. Na początku komputer prosi użytkownika o wymyślenie całkowitej liczby ze zbioru $\{1, 2, \ldots, 100\}$ i zapamiętanie jej (w „głowie”).

Następnie komputer próbuje tę liczbę odgadnąć, posługując się wskazówkami danymi przez użytkownika. W każdej iteracji komputer przedstawia użytkownikowi liczbę-kandydatkę i pyta się, jak „kandydatka” ma się do liczby wymyślonej. Jeśli użytkownik wprowadzi wartość 1 z klawiatury, oznacza to, że „kandydatka” jest mniejsza; wartość 2, że równa (sukces, koniec programu); a wartość 3, że większa od wymyślonej.

Jako algorytm „sztucznej inteligencji” użyj następującej „heurystyki”. W każdej iteracji komputer wie, że odgadywana liczba należy na pewno do zbioru $\{a, a+1, \ldots, b\}$, gdzie w pierwszej iteracji mamy $a = 1$ oraz $b = 100$. Jako „kandydatkę”, c, komputer wybiera „środek” tego przedziału, czyli $c = (b+a)/2$ (dzielenie całkowite!). Na podstawie informacji zwrotnej od użytkownika albo kończymy obliczenia (2), albo zmieniamy b na $c - 1$ (1), albo a na $c + 1$ (3). Dzięki temu „obszar poszukiwań” się zmniejszy i w kolejnej iteracji komputer będzie bliżej prawidłowej odpowiedzi. Procedurę odgadywania kończymy, gdy a stanie się równe b, tzn. komputer nie będzie dalej już musiał zgadywać.

Zadanie 3.58. Generator Parka-Millera. Rozważmy ciąg liczb naturalnych, którego n-ty wyraz ($n > 0$) dany jest wzorem

$$r_n = 75 r_{n-1} \mod 65537,$$

gdzie $\mod$ oznacza resztę z dzielenia, a $r_0 \in \{1, 2, \ldots 65536\}$ jest liczbą wprowadzoną z klawiatury (tzw. ziarno generatora).

Wzór ten określa pewien szczególny przypadek tzw. kongruencyjnego generatora liczb pseudolosowych (zwanego czasem generatorem Parka–Millera)[1]. Istotne jest, że kolejne liczby generowane przy użyciu tego wzoru „wyglądają” jak losowe. Co więcej, cechują się całkiem dobrymi własnościami statystycznymi.

Wypisz na ekran liczby $r_1, \ldots, r_{10}$ i zobacz jaki wpływ ma ziarno generatora r_0 na postać wynikowego ciągu.

Zadanie 3.59. Nietrudno pokazać, że powyższy generator ma skończony okres, tzn. po wygenerowaniu k różnych liczb zaczną pojawiać się na wyjściu znów te same wartości, w identycznej kolejności. Napisz program, który dla danego z klawiatury ziarna r_0 znajdzie wartość k. Czy okres tego generatora jest zależny od ziarna?

Zadanie 3.60. Można zauważyć, że powyższy generator typu Parka–Millera $(r_0, r_1, \ldots)$ generuje liczby ze zbioru $\{1, 2, \ldots, 65536\}$. Zatem ciąg liczb rzeczywistych, którego n-ty wyraz (dla $n > 0$) jest określony wzorem

$$s_n = r_n / 65537.0,$$

może służyć do generowania liczb „rzeczywistych” z przedziału $(0, 1)$.

Zasymuluj $n \in \mathbb{N}$ rzutów monetą, gdzie np. $n = 10000$. Przyjmij, że gdy $s_n > 0.5$, to został wyrzucony orzeł, a gdy $s \leqslant 0.5$, to reszka. Wyznacz proporcję (ułamek) wyrzuconych orłów w n rzutach.

Jeśli generator Parka–Millera cechuje się dobrymi własnościami statystycznymi, to spodziewamy się, że proporcja ta powinna być bliska 0.5 (choć będziemy obserwować pewną zmienność dla różnych r_0). Tym sposobem dokonasz prostego testu zaimplementowanego generatora.

[1] Można ustalać różne stałe we wzorze na r_n, najlepiej, by były m.in. względnie pierwsze.

(*) Zadanie 3.61. (Zgadula 2) Rozpatrzmy ciąg $(s_0, s_1, \dots)$ z powyższego zadania. Przy użyciu instrukcji `c = (int)(`s_i`*(b-a+1)+a);` (dla pewnego $i \geqslant 0$) dokonujemy przekształcenia wartości $s_i \in (0, 1)$ na zbiór liczb całkowitych $\{a, a+1, \dots, b\}$.

Dokonaj modyfikacji algorytmu „sztucznej inteligencji" w zgaduli, tak żeby nieco urozmaicić „strzały" komputera, zastępując wybór liczby-„kandydatki" `c=(a+b)/2` wyborem „pseudolosowym".

(*) Zadanie 3.62. (Mieszanka generatorów I) Stwórz „ulepszony" generator liczb rzeczywistych z przedziału $(0, 1)$, który jest tzw. mieszanką dwóch generatorów typu Parka–Millera. Niech $(s_0, s_1, \dots)$ oznacza ciąg generowany przez pierwszy generator (zainicjowany ziarnem s_0), a $(s'_0, s'_1, \dots)$ ciąg generowany przez drugi generator (ziarno s'_0). Na ich podstawie tworzymy nowy ciąg, którego n-ty wyraz ($n \geqslant 0$) dany jest wzorem

$$t_n = (s_n + s'_n)/2.0.$$

Aby sprawdzić, czy generator daje oczekiwane wyniki (okaże się, że nie będzie to już generator liczb z rozkładu jednostajnego, czyli takiego, dla którego prawdopodobieństwo wylosowania każdej liczby jest takie samo), przeprowadź symulację $k = 10000$ rzutów czterościenną kostką do gry fantasy. Przyjmij, że jedno oczko wypada dla $t_n < 0.25$, 2 oczka dla $t_n \in [0.25, 0.5)$, 3 dla $t_n \in [0.5, 0.75)$ oraz 4 dla $t_n \geqslant 0.75$. Wypisz uzyskane proporcje dla każdego z czterech oczek. Co zauważasz?

(*) Zadanie 3.63. (Mieszanka generatorów II) Oto kolejne „ulepszenie" generatora mieszanego. n-ty wyraz ciągu ($n \geqslant 0$) dany jest tym razem wzorem

$$t_n = \begin{cases} (s_n + s'_n)^2 & \text{jeśli } (s_n + s'_n) < 1, \\ (2 - s_n - s'_n)^2 & \text{w p.p..} \end{cases}$$

Przetestuj ten generator metodą taką samą jak w powyższym zadaniu (spodziewamy się, że będzie się on zachowywał zgodnie z oczekiwaniami).

Tablice jednowymiarowe. Algorytmy sortowania

4

4.1 Tablice jednowymiarowe o ustalonym rozmiarze

Do tej pory przechowywaliśmy dane, używając jedynie pojedynczych zmiennych. Były to tzw. **zmienne skalarne** (atomowe). Jedna zmienna odpowiadała pewnej „jednostce informacji" (liczbie).

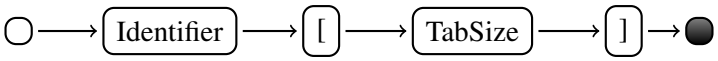
Często jednak na wejściu mamy dany ciąg n zmiennych tego samego typu. Rozważmy fragment programu dokonujący podsumowania rocznych zarobków pewnej doktorantki.

```
1  double zarobki1, zarobki2, /* ... */, zarobki12;
2      // deklaracja 12 zmiennych
3  zarobki1 = 1399.0;  // styczen
4  zarobki2  = 1493.0; // luty
5  // ...
6  zarobki12 = 999.99; // grudzien
7
8  double suma = 0.0;
9  suma += zarobki1;
10 suma += zarobki2;
11 //...
12 suma += zarobki12;
13
14 cout << "Zarobilam w 2011 r. " << suma << " PLN.";
```

Dochód z każdego miesiąca przechowywany jest w oddzielnej zmiennej. Wszystkie z nich są tego samego typu. Nietrudno zauważyć, że operowanie na nich nie jest zbyt wygodne.

W tym wykładzie zajmiemy się **tablicami jednowymiarowymi o ustalonym rozmiarze**. Tablice jednowymiarowe (ang. *one-dimensional arrays*) są reprezentacją znanych nam obiektów matematycznych: **ciągów skończonych** lub **wektorów**.

Do tworzenia tablic jednowymiarowych służy następujący deklarator zmiennych (por. poprzedni wykład):



TabSize jest stałą (!) całkowitą dodatnią określającą rozmiar tablicy.

Ważne

Dostęp do poszczególnych elementów tablicy odbywa się za pomocą **operatora indeksowania** $[\cdot]$. Elementy tablicy są numerowane od 0 do $n-1$, gdzie n to rozmiar tablicy.

Na przykład:

```
int t[5]; // deklaracja tablicy (5 elementow typu int)

// t[0] to pierwszy element (!)
// t[4] to ostatni element (!)
```

Operator indeksowania przyjmuje jako parametr dowolną wartość całkowitą (np. stałą lub wyrażenie arytmetyczne). Każdy element tablicy traktujemy tak, jakby był zwykłą zmienną.

Informacja

W języku **C++** nie ma mechanizmów sprawdzania poprawności indeksów! Następujący kod być może nie spowoduje błędu natychmiast po uruchomieniu.

```
int t[5];
t[-100]  = 15123;  // :(
t[10000] = 25326;  // :(
```

Jednakże powyższe instrukcje zmieniają wartości komórek pamięci wykorzystywanych przez inne funkcje. Skutki tego działania mogą się objawić w innym miejscu programu, powodując nieprzewidywalne i trudne do wykrycia błędy.

Więcej na temat tego zagadnienia dowiemy się w rozdziale dotyczącym dynamicznego przydziału pamięci.

Wróćmy do przykładu opisującego życiowe perypetie zaradnej doktorantki. Oto w jaki sposób można wykorzystać tablice do podsumowania jej zarobków. Zauważmy analogię między tym a poprzednim fragmentem kodu.

```
double zarobki[12];  // deklaracja 12 elementowej tablicy

zarobki[0] = 1399.0; // styczen
zarobki[1] = 1493.0; // luty
// ...
zarobki[11] = 999.99; // grudzien

double suma = 0.0;
for (int i=0; i<=11; ++i)  // rozwaz wszystkie elementy...
   suma += zarobki[i];

cout << "Zarobilam w 2011 r. " << suma << " PLN.";
```

Zauważmy, że decydując się na przetwarzanie całej tablicy za pomocą pętli **for**, dostajemy w efekcie bardziej zwięzły i czytelny kod programu.

Co więcej, fakt dokonywania na każdym, i-tym, elemencie zawsze tej samej operacji arytmetycznej powoduje, że nietrudno by było teraz zmodyfikować ten program tak, by na przykład uwzględniał zarobki z całego okresu studiów.

Dla wygody, przy tworzeniu tablicy można od razu przypisać wartości jej elementom[1]. Pozwala to jeszcze bardziej uprościć program dla naszej doktorantki:

```
const int n = 12; /* stala! */
double zarobki[n] = { 1399.0, 1493.0, /*...*/, 999.99 };

double suma = 0.0;
for (int i=0; i<n; ++i)
   suma += zarobki[i];

cout << "Zarobilam w 2011 r. " << suma << " PLN.";
```

[1]Zauważmy, że w języku **C++** do inicjacji zawartości tablicy o ustalonym rozmiarze służą nawiasy klamrowe, które w matematyce najczęściej stosowane są do definicji zbiorów. Zbiór, jak wiemy – w przeciwieństwie do ciągu, który reprezentuje tablica w języku **C++** – abstrahuje od kolejności umieszczonych weń elementów. Może być to dla nas czasem mylące, zwracajmy więc na to szczególną uwagę.

Informacja
O tablicach dowolnego rozmiaru (nie definiowanego z góry na etapie pisania kodu) oraz sposobach przekazywania tablic innym funkcjom dowiemy się więcej z wykładu na temat dynamicznej alokacji pamięci.

Na marginesie: aby wypisać zawartość całej tablicy, możemy użyć następującego fragmentu kodu:

```
const int n = ...;
int tab[n] = { ... };

for (int i=0; i<n; ++i)
   cout << tab[i] << endl;
```

Aby wczytać elementy tablicy z klawiatury, możemy napisać:

```
const int n = ...;
int tab[n];

cout << "Podaj wartosci " << n << " elementow." << endl;
for (int i=0; i<n; ++i)
   cin >> tab[i];
```

4.2 Proste algorytmy sortowania tablic

Rozpatrzymy teraz problem **sortowania tablic jednowymiarowych**, który jest istotny w wielu zastosowaniach, zarówno teoretycznych jak i praktycznych. Dzięki odpowiedniemu uporządkowaniu elementów, niektóre algorytmy (np. wyszukiwania) mogą działać szybciej, mogą być prostsze w implementacji, czy też można łatwiej formalnie udowodnić ich poprawność.

Zadanie
Wyobraź sobie, ile czasu zajęłoby Ci znalezienie hasła w słowniku (chodzi oczywiście o słownik książkowy), gdyby redaktorzy przyjęliby losową kolejność wyrazów. Przeanalizuj z jakiego „algorytmu” korzystasz, wyszukując tego, co Cię interesuje.

Co więcej, istnieje wcale niemało zagadnień, które wprost wymagają pewnego uporządkowania danych i które bez takiej operacji wcale nie mają sensu.

Zadanie

Rozważmy, w jaki sposób wyszukujemy interesujących nas stron internetowych. Większość wyszukiwarek działa w następujący sposób.

a) Znajdź wszystkie strony w bazie danych, które zawierają podane przez użytkownika słowa kluczowe (np. „kaszel", „gorączka" i „objawy").
b) Oceń każdą znalezioną stronę pod względem pewnej miary adekwatności/popularności/jakości.
c) Posortuj wyniki zgodnie z ocenami (od „najlepszej" do „najgorszej") i pokaż ich listę użytkownikowi.

Warto przypomnieć, że o rynkowym sukcesie wyszukiwarki G**gle zadecydowało to, że – w przeciwieństwie do ówczesnych konkurencyjnych serwisów – zwracała ona wyniki w dość „pożytecznej" dla większości osób kolejności.

Problem sortowania, w swej najprostszej postaci, można sformalizować następująco.

> Dana jest tablica `t` rozmiaru `n` zawierająca elementy, które można porównywać przy użyciu operatora relacyjnego <=. Należy zmienić kolejność (tj. dokonać permutacji, przestawienia, uporządkowania) elementów `t` tak, by zachodziły warunki:
>
> `t[0] <= t[1], t[1] <= t[2], ..., t[n-2] <= t[n-1].`

Ciekawostka

Dla porządku (sic!) zauważmy, że rozwiązanie takiego problemu wcale nie musi być jednoznaczne. Dla tablic zawierających elementy `t[i]` i `t[j]` takie, że dla $i \neq j$ zachodzi `t[i] <= t[j]` oraz `t[j] <= t[i]`, istnieje więcej niż jedna permutacja spełniająca powyższe warunki.

W niniejszym paragrafie omówimy trzy algorytmy sortowania:

a) sortowanie przez wybór,
b) sortowanie przez wstawianie,
c) sortowanie bąbelkowe.

Algorytmy te cechują się tym, że w **pesymistycznym** („najgorszym") przypadku liczba operacji porównań elementów tablicy jest **proporcjonalna** do n^2 (zob. dalej; podrozdz. 4.2.4). Bardziej wydajne i, co za tym idzie, bardziej złożone algorytmy sortowania będą omówione gdzie indziej (np. sortowanie szybkie, przez łączenie, przez kopcowanie). Niektóre z nich wymagają co najwyżej $kn \log n$ porównań dla pewnego k. Dzięki temu dla tablic o dużym rozmiarze działają one naprawdę szybko.

4.2.1 Sortowanie przez wybór

W algorytmie **sortowania przez wybór** (ang. *selection sort*) dokonujemy wyboru elementu najmniejszego spośród do tej pory nieposortowanych, póki cała tablica nie zostanie uporządkowana.

Ideę tę przedstawia następujący pseudokod:

```
for (i = 0, 1, ..., n-2)
{
   // zalozenie: t[0], ..., t[i-1] sa juz uporzadkowane
      wzgledem relacji <= (ponadto sa juz na swoich
      ostatecznych miejscach)
   j = indeks najmniejszego elementu
         sposrod t[i], ..., t[n-1];
   zamien elementy t[i] i t[j];
}
```

Jako przykład rozważmy, krok po kroku, przebieg sortowania tablicy

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

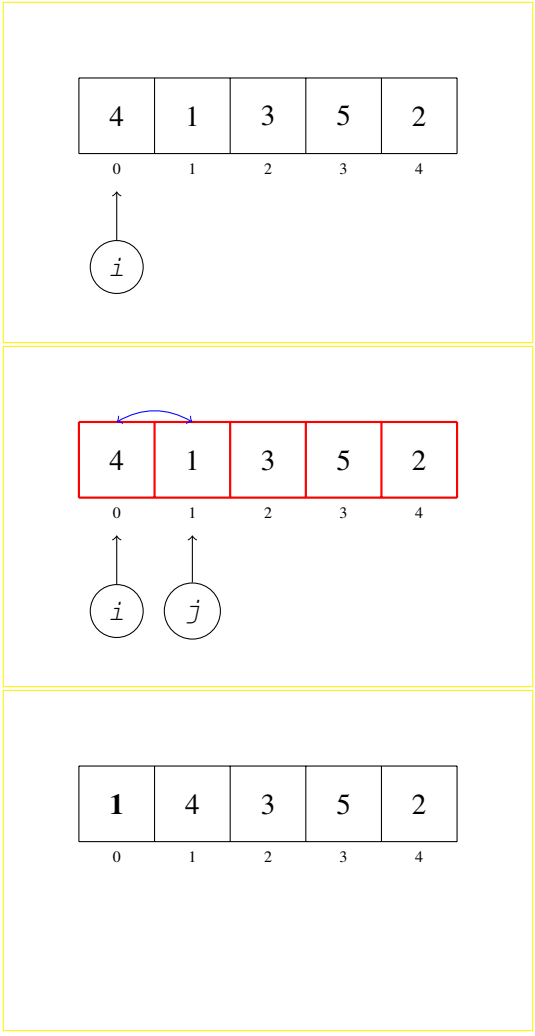
Kolejne iteracje tego algorytmu ilustrują rys. 4.1–4.5.

a) W kroku I (rys. 4.1) mamy `i==0`. Dokonując wyboru elementu najmniejszego spośród `t[0]`, ..., `t[4]`, otrzymujemy `j==1`. Zamieniamy więc elementy `t[0]` i `t[1]` miejscami.

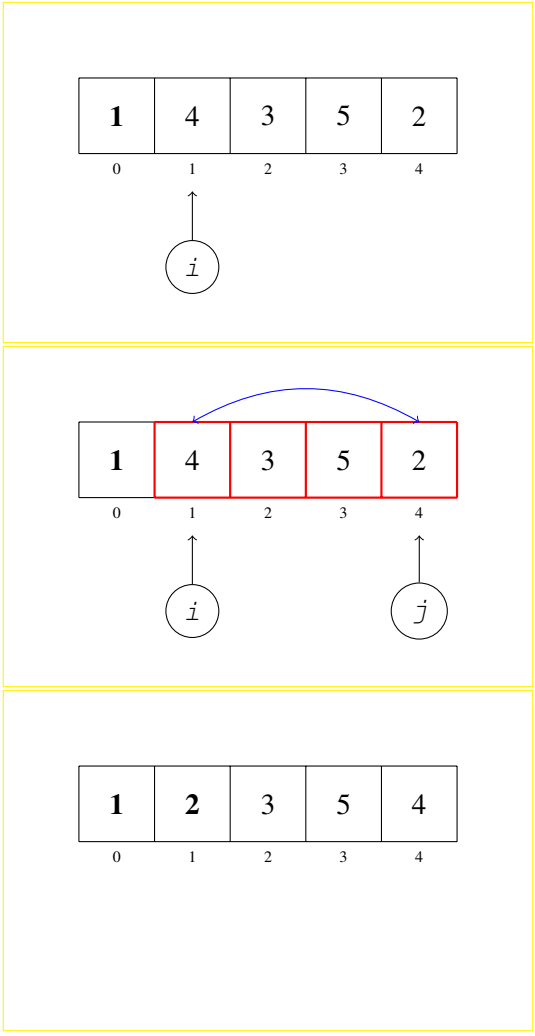
b) W kroku II (rys. 4.2) mamy `i==1`. Wybór najmniejszego elementu wśród `t[1]`, ..., `t[4]` daje `j==4` Zamieniamy miejscami zatem `t[1]` i `t[4]`.

c) Dalej (rys. 4.3), `i==2`. Elementem najmniejszym spośród `t[2]`, ..., `t[4]` jest `t[j]` dla `j==2` Zamieniamy miejscami zatem niezbyt sensownie `t[2]` i `t[2]`. Komputer, na szczęście, zrobi to bez grymasu.

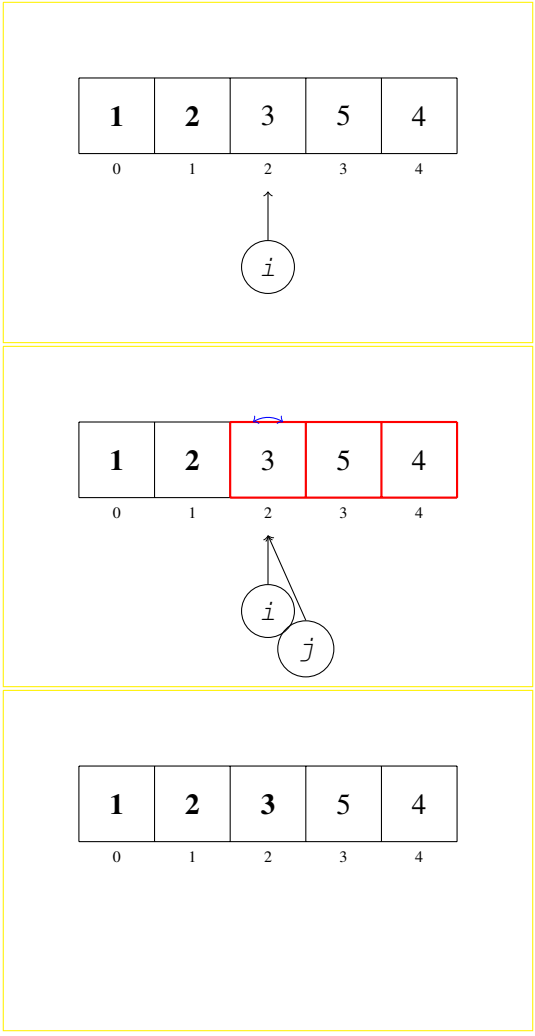
d) W ostatnim kroku (rys. 4.4) `i==3` i `j==4`, dzięki czemu możemy uzyskać ostateczne rozwiązanie (rys. 4.5).



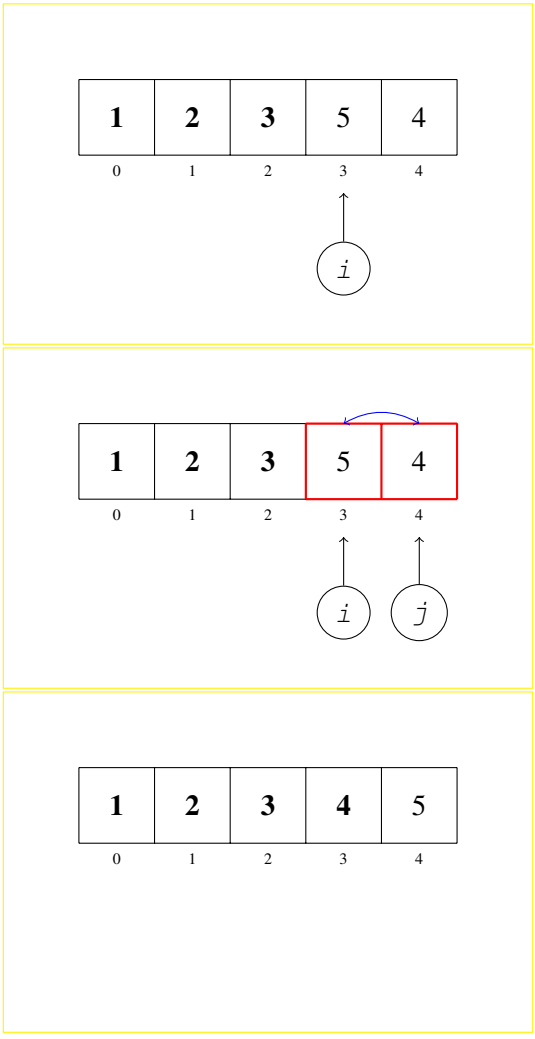
Rysunek 4.1. Sortowanie przez wybór — przykład — iteracja I.



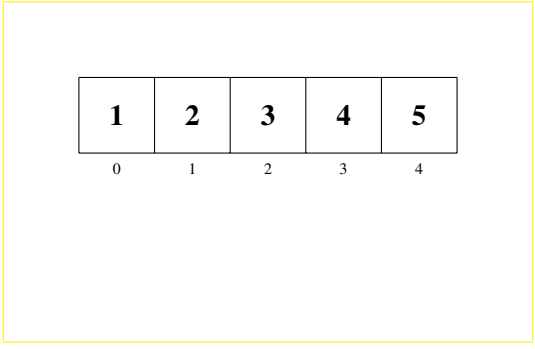
Rysunek 4.2. Sortowanie przez wybór — przykład — iteracja II.



Rysunek 4.3. Sortowanie przez wybór — przykład — iteracja III.



Rysunek 4.4. Sortowanie przez wybór — przykład — iteracja IV.



Rysunek 4.5. Sortowanie przez wybór — przykład — rozwiązanie.

4.2.2 Sortowanie przez wstawianie

Algorytm **sortowania przez wstawianie** (ang. *insertion sort*) jest metodą często stosowaną w praktyce do porządkowania małej liczby elementów (do ok. 20–30) ze względu na swą prostotę i szybkość działania.

W niniejszej metodzie, w i-tym kroku elementy `t[0], ..., t[i-1]` są już wstępnie uporządkowane względem relacji <=. Między nie wstawiamy `t[i]` tak, by nie zaburzyć porządku.

Formalnie rzecz ujmując, idea ta może być wyrażona przy użyciu pseudokodu:

```
for (i = 1, 2, ..., n-1)
{
   // t[0], ..., t[i-1] sa wstepnie uporzadkowane wzgledem
        <= (ale niekoniecznie jest to ich ostateczne
      miejsce)
   j = indeks takiego elementu sposrod t[0],...,t[i], ze
      t[u] <= t[i] dla kazdego u < j oraz
      t[i] <= t[v] dla kazdego j ⩽ v;
   wstaw t[i] przed t[j];
}
```

gdzie przez operację „`wstaw t[i] przed t[j]`”, dla $0 \leqslant j \leqslant i$ rozumiemy ciąg działań, mający na celu przestawienie kolejności elementów tablicy:

`t[0]`	...	`t[j-1]`	`t[j]`	...	`t[i-1]`	`t[i]`	`t[i+1]`	...	`t[n-1]`

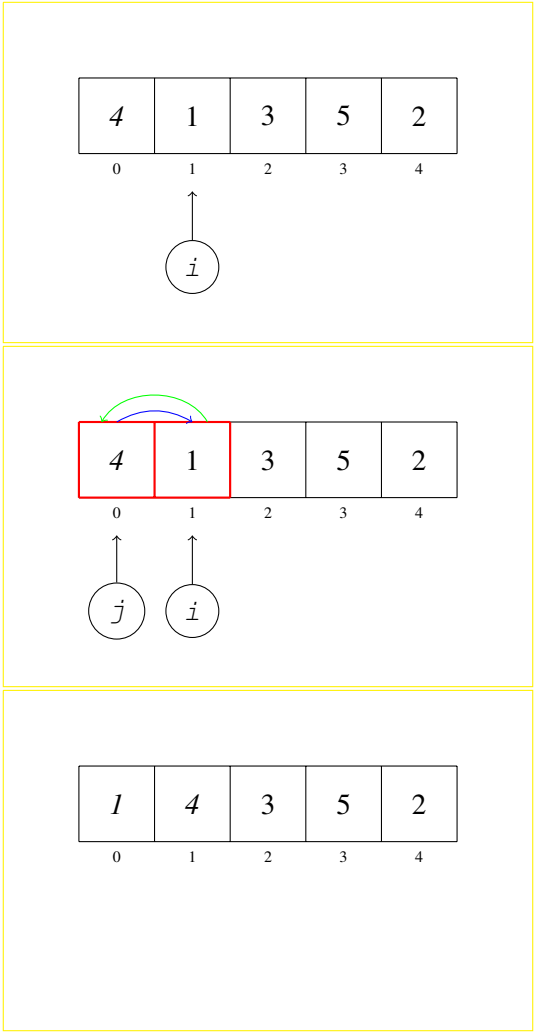
tak by uzyskać:

`t[0]`	...	`t[j-1]`	`t[i]`	`t[j]`	...	`t[i-1]`	`t[i+1]`	...	`t[n-1]`

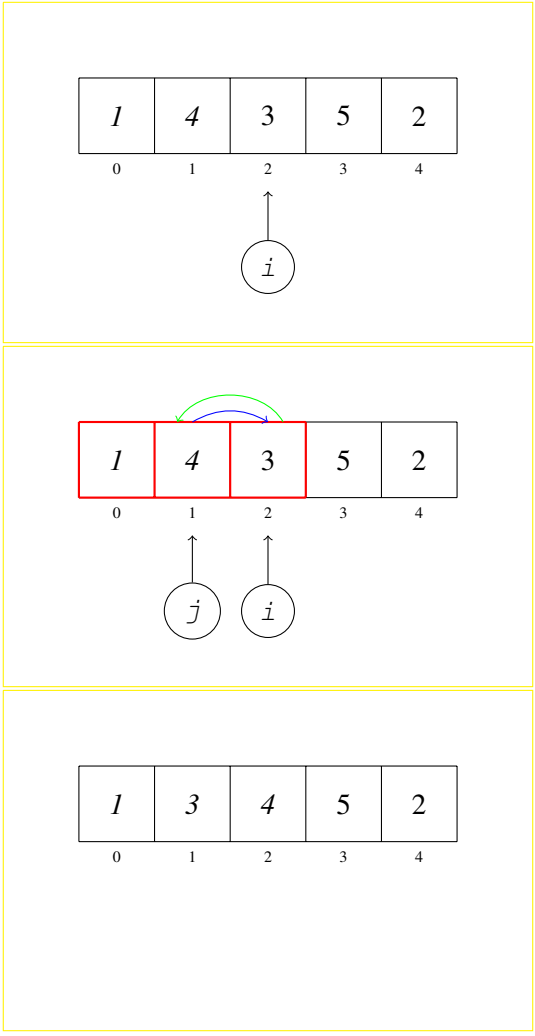
Powyższy pseudokod może być wyrażony w następującej równoważnej formie:

```
for (i = 1, 2, ..., n-1)
{
    // t[0], ..., t[i-1] sa uporzadkowane wzgledem <=
    // (ale niekoniecznie jest to ich ostateczne miejsce)
    znajdz najwieksze j ze zbioru {0,...,i} takie, ze
        j == 0 lub t[j-1] <= t[i];
    wstaw t[i] przed t[j];
}
```

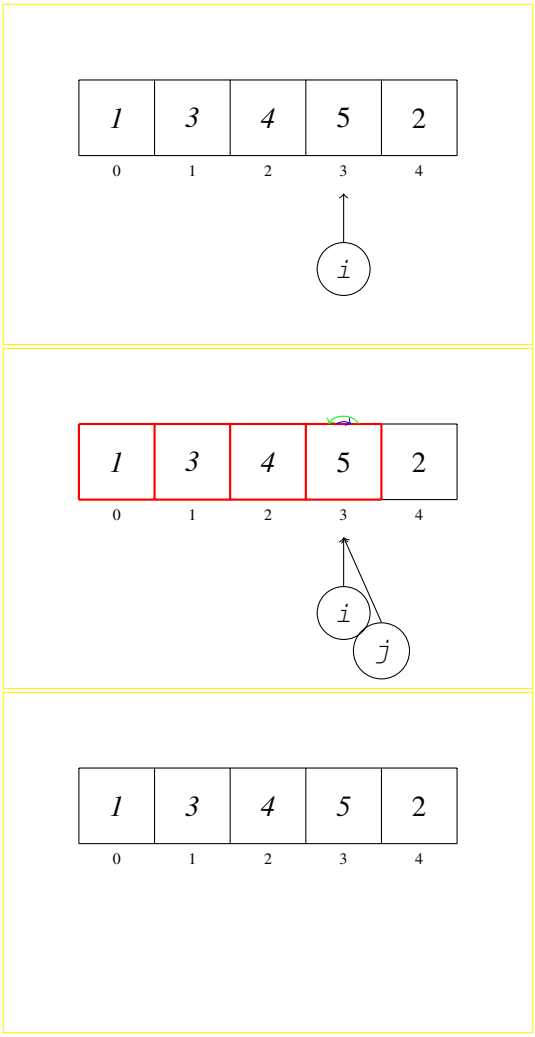
Jako przykład rozpatrzmy znów tablicę `int t[5] = {4,1,3,5,2};`
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 4.6–4.9.



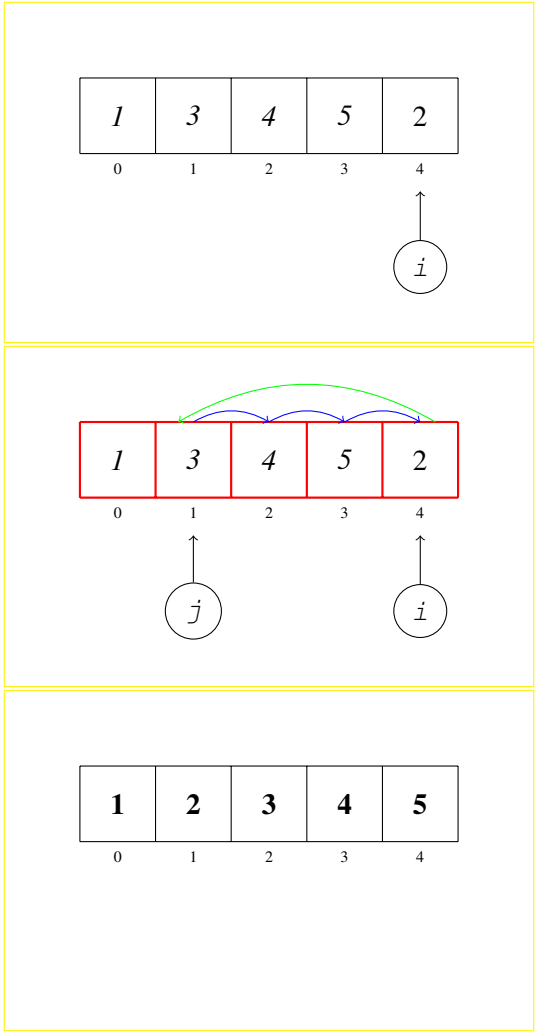
Rysunek 4.6. Sortowanie przez wstawianie — przykład — iteracja I.



Rysunek 4.7. Sortowanie przez wstawianie — przykład — iteracja II.



Rysunek 4.8. Sortowanie przez wstawianie — przykład — iteracja III.



Rysunek 4.9. Sortowanie przez wstawianie — przykład — iteracja IV.

4.2.3 Sortowanie bąbelkowe

Sortowanie bąbelkowe (ang. *bubble sort*) jest interesującym przykładem algorytmu pojawiającego się w większości podręczników akademickich dotyczących podstawowych sposobów sortowania tablic. Jednakże prawie wcale nie stosuje się go w praktyce. Jego wydajność jest bowiem bardzo słaba w porównaniu do dwóch metod opisanych powyżej. Z drugiej strony, posiada on sympatyczną „hydrologiczną" (nautyczną?) interpretację, która urzeka wielu wykładowców, w tym i skromnego autora niniejszego opracowania. Tak umotywowani, przystąpmy więc do zapoznania się z nim.

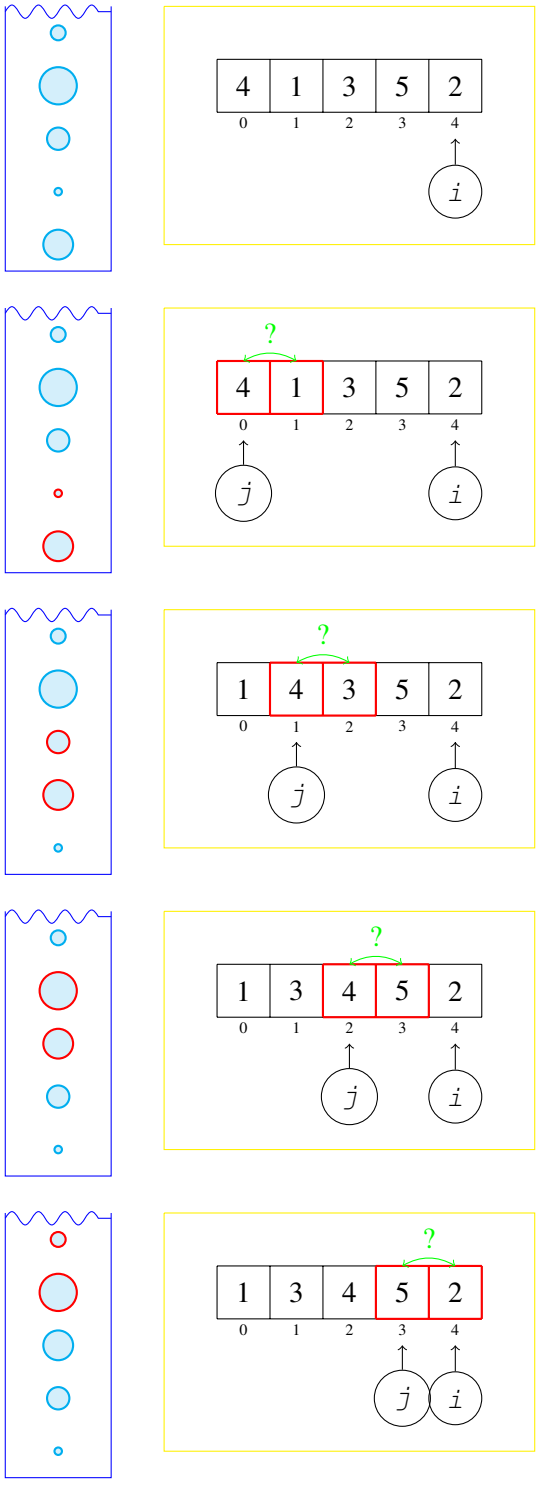
W tym algorytmie porównywane są elementy tylko ze sobą bezpośrednio sąsiadujące (parami). Jeśli okaże się, że nie zachowują one odpowiedniej kolejności względem relacji <=, element „cięższy" wypychany jest „w górę", niczym pęcherzyk powietrza (tytułowy bąbelek) pod powierzchnią wody.

A oto pseudokod:

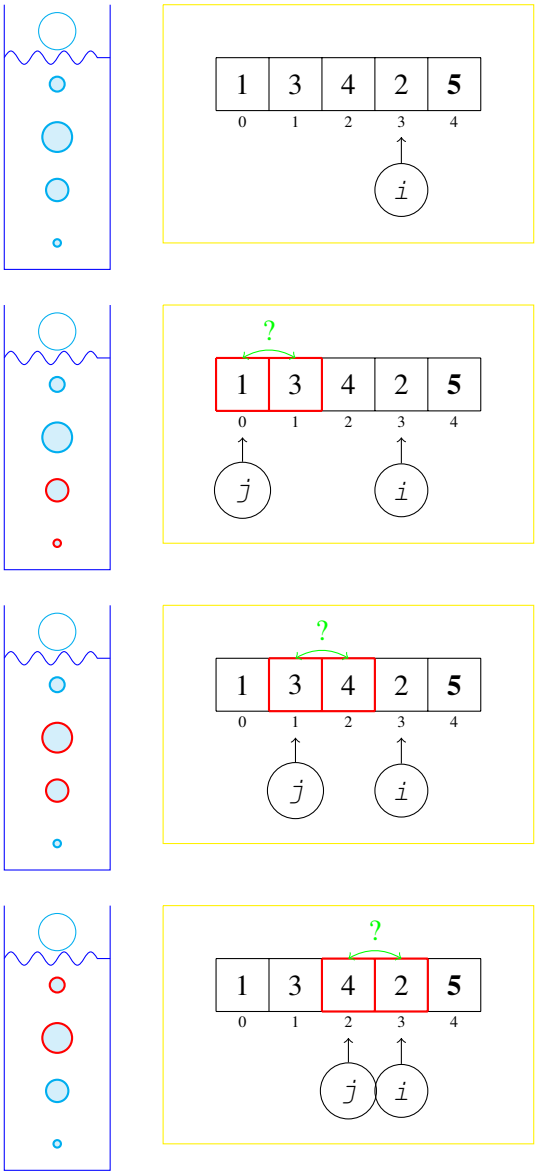
```
for (i = n-1, ..., 1)
{
   for (j = 0, ..., i-1)
   {
      // porownuj elementy parami
      jesli (t[j] > t[j+1])
         zamien t[j] i t[j+1];
         // tzn. "wypchnij" ciezszego "babelka" w gore
   }

   // tutaj elementy t[i],...,t[n-1] sa juz na swoich
      miejscach
}
```

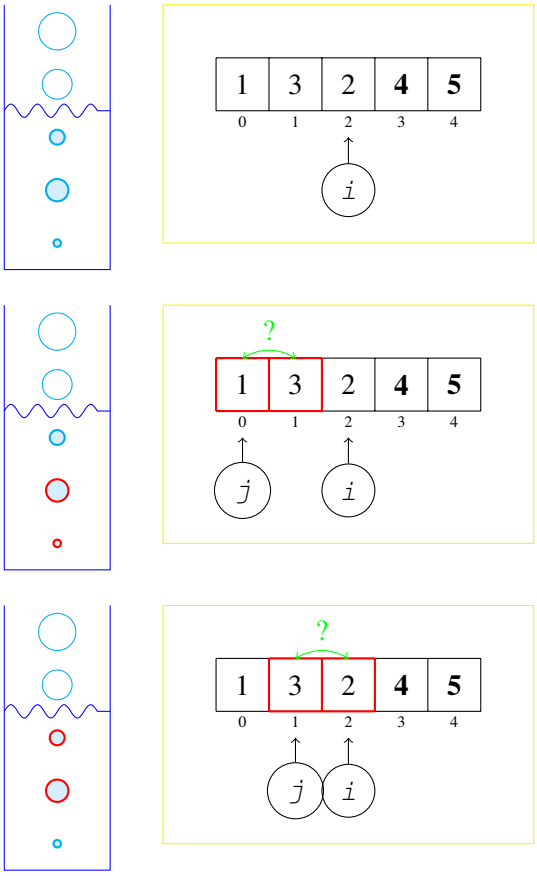
Rozpatrzmy ponownie tablicę `int t[5] = {4,1,3,5,2};`
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 4.10–4.14.



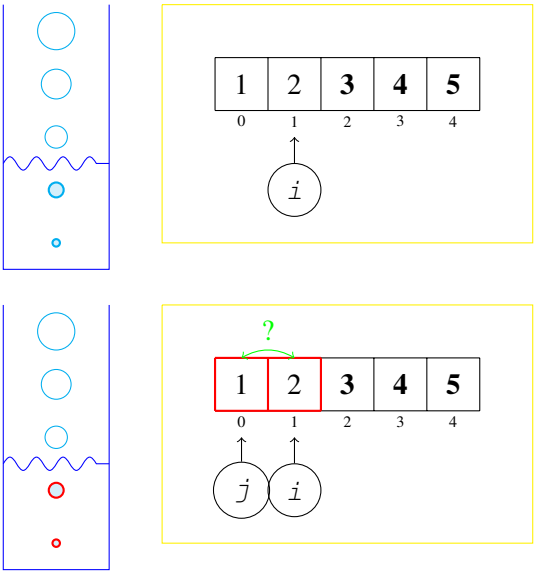
Rysunek 4.10. Sortowanie bąbelkowe — przykład — krok I.



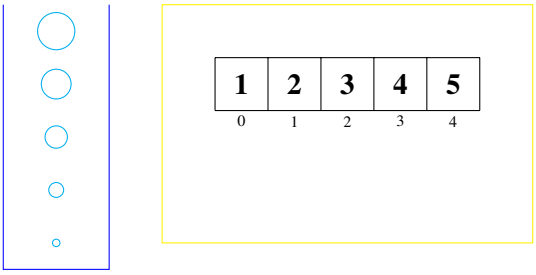
Rysunek 4.11. Sortowanie bąbelkowe — przykład — krok II.



Rysunek 4.12. Sortowanie bąbelkowe — przykład — krok III.



Rysunek 4.13. Sortowanie bąbelkowe — przykład — krok IV.



Rysunek 4.14. Sortowanie bąbelkowe — przykład — rozwiązanie.

4.2.4 Efektywność obliczeniowa

Rozważając algorytmy sortowania, w sposób oczywisty doświadczamy zapowiedzianego na pierwszym wykładzie zjawiska „jeden problem — wiele algorytmów". Rzecz jasna, trzy wprowadzone powyżej algorytmy nie wyczerpują wszystkich możliwości rozwiązania zagadnienia sortowania tablic jednowymiarowych.

Nasuwa się więc pytanie: skoro każda z metod znajduje rozwiązanie danego problemu w poprawny sposób, jakie przesłanki powinny nami kierować przy wyborze algorytmu w zastosowaniach praktycznych?

Leniuszki z pewnością wybrałyby algorytm najprostszy w implementacji. Domyślamy się jednak, że nie jest to koniecznie najlepsze kryterium.

W **analizie algorytmów** wyróżnia się najczęściej dwie następujące miary efektywności obliczeniowej:

a) **złożoność czasowa** — czyli liczbę tzw. operacji znaczących (zależną od danego problemu; mogą to być przestawienia, porównania, operacje arytmetyczne itp.) potrzebnych do rozwiązania danego zagadnienia,
b) **złożoność pamięciową** — czyli ilość dodatkowej pamięci potrzebnej do rozwiązania problemu.

Zauważmy, że w przypadku algorytmów przedstawionych powyżej, oprócz tablicy do posortowania i kilku zmiennych pomocniczych, nie jest potrzebna żadna dodatkowa liczba komórek pamięci komputera (w przypadku bardziej złożonych metod omawianych na bardziej zaawansowanych kursach będzie jednak inaczej, np. konieczne będzie użycie jeszcze jednej tablicy). Skupimy się więc teraz tylko na omówieniu **złożoności czasowej**.

Przyjrzyjmy się zatem przykładowym implementacjom dwóch algorytmów. Oto kod stosujący sortowanie przez wybór:

```
// Dane:
const int n = ...;
int t[n] = { ... };

// Sortowanie przez wybor:
for (int i=0; i<n-1; ++i)
{
    // znajdz indeks najmn. el. sposrod t[i], ..., t[n-1];
    int j = i;
    for (int k=i+1; k<n; ++k)
        if (t[k] < t[j])          // porownanie elementow
            j = k;

    // zamiana
    int p = t[i];
    t[i] = t[j];
    t[j] = p;
}
```

A oto kod sortujący tablicę „bąbelkowo”:

```
// Dane:
const int n = ...;
int t[n] = { ... };

// Sortowanie babelkowe:
for (int i=n-1; i>0; --i)
{
    for (int j=0; j<i; ++j)
    {
        if (t[j] > t[j+1])              // porownanie parami
        {
            int p = t[j];               // zamiana
            t[j] = t[j+1];
            t[j+1] = p;
        }
    }
}
```

Po pierwsze, musimy podkreślić, iż **złożoność czasową wyznacza się nie dla abstrakcyjnego algorytmu, tylko dla jego konkretnej implementacji**. Implementacja zależna jest, rzecz jasna, od stosowanego języka programowania i zdolności programisty. Nie powinno to nas dziwić, ponieważ np. za wysokim poziom abstrakcji, często stosowanym w pseudokodach, może kryć się naprawdę wiele skomplikowanych instrukcji języka programowania.

Po drugie, zauważmy, że **złożoność jest najczęściej zależna od danych wejściowych**. W naszym przypadku jest to nie tylko rozmiar tablicy, n, ale i także jej wstępne uporządkowanie. Z tego też powodu liczbę operacji znaczących powinno podawać się w postaci **funkcji rozmiaru zbioru danych wejściowych** (u nas jest to po prostu n) w różnych sytuacjach, najczęściej co najmniej:

a) w przypadku **pesymistycznym**, czyli dla danych, dla których implementacja algorytmu musi wykonać najwięcej operacji znaczących,
b) w przypadku **optymistycznym**, czyli gdy dane wejściowe minimalizują liczbę potrzebnych operacji znaczących.

Ciekawostka
Celowość przeprowadzenia badania przypadków pesymistycznych i optymistycznych pozostaje przeważnie poza dyskusją. Czasem także bada się tzw. złożoność czasową oczekiwaną (średnią), gdzie zakładamy, że dane mają np. tzw. rozkład jednostajny (w naszym przypadku znaczyłoby to, że prawdopodobieństwo pojawienia się każdej możliwej permutacji ciągu wejściowego jest takie samo). Oczywiście wątpliwości może tutaj budzić, czy taki dobór rozkładu jest adekwatny, słowem — czy rzeczywiście modeluje to, co zdarza się w praktyce.

Nie zmienia to jednak faktu, że jest to zagadnienie bardzo ciekawe i warte rozważania. Wymaga ono jednak dość rozbudowanego aparatu matematycznego, którym niestety jeszcze nie dysponujemy.

Nietrudno zauważyć, że w przypadku przedstawionych wyżej implementacji algorytmów, przypadkiem optymistycznym jest tablica już posortowana (np. $(1, 2, 3, 4, 5)$), a przypadkiem pesymistycznym – tablica posortowana odwrotnie (np. $(5, 4, 3, 2, 1)$).

Pozostaje nam już tylko wyróżnić, co możemy rozumieć w danych przykładach za operacje znaczące oraz dokonać stosownych obliczeń. Wydaje się, że najrozsądniej będzie rozważyć liczbę potrzebnych przestawień elementów oraz liczbę ich porównań – rzecz jasna, w zależności od n.

1. Analiza liczby porównań elementów.

a) Sortowanie przez wybór.
Zauważamy, że liczba porównań nie jest zależna od wstępnego uporządkowania elementów tablicy. Jest ona zawsze równa $(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = n^2/2 - n/2$.

b) Sortowanie bąbelkowe. Jak wyżej, liczba porównań nie jest zależna od wstępnego uporządkowania elementów tablicy. Uzyskujemy wartość $(n-1)+(n-2)+\cdots+1 = n^2/2 - n/2$.

Obydwie implementacje algorytmów nie różnią się liczbą potrzebnych porównań. Są zatem tak samo dobre (lub tak samo złe) pod względem tego kryterium. Zauważmy, że liczba ta jest **proporcjonalna** do n^2 (inaczej — jest rzędu n^2).

2. Analiza liczby przestawień elementów.

a) Sortowanie przez wybór.
   i. Przypadek pesymistyczny.
      Liczba potrzebnych przestawień wynosi $n-1$.
   ii. Przypadek optymistyczny.
      Liczba potrzebnych przestawień wynosi 0.

b) Sortowanie bąbelkowe.
   i. Przypadek pesymistyczny.
      Liczba potrzebnych przestawień wynosi $n(n-1)/2$.
   ii. Przypadek optymistyczny.
      Liczba potrzebnych przestawień wynosi 0.

Okazuje się, że w najgorszym przypadku liczba potrzebnych przestawień elementów dla naszej implementacji algorytmu sortowania przez wybór jest proporcjonalna do n, a dla sortowania bąbelkowego — jest aż rzędu n^2!

Tablica 4.1 zestawia liczbę potrzebnych przestawień elementów powyższych algorytmów i uzyskane na komputerze autora niniejszego opracowania czasy działania (w sekundach, tablice o elementach typu `int`, dość stary komputer).

Zauważamy, że użycie powyższych algorytmów już dla $n > 100\,000$ nie jest dobrym pomysłem (na szczęście, istnieją lepsze, naprawdę szybkie sposoby rozwiązania problemu sortowania). Ponadto sama liczba przestawień nie tłumaczy czasu wykonania obliczeń (dlatego konieczne było także zbadanie liczby potrzebnych porównań).

Tabela 4.1. Porównanie liczby potrzebnych przestawień elementów i czasów działania dwóch algorytmów sortowania w przypadku pesymistycznym.

	Liczba przestawień		Czas [s]	
n	Selection	Bubble	Selection	Bubble
100	99	4 950	~0,01	~0,01
1 000	999	499 500	~0,01	~0,01
10 000	9 999	49 995 000	0,05	0,10
100 000	99 999	4 999 950 000	6,02	10,31
1 000 000	999 999	499 999 500 000	631,38	1 003,60

4.3 Ćwiczenia

Zadanie 4.64. Znajdź w danej n-elementowej ($n \geqslant 3$) tablicy `double tab[n]` trzeci największy element; np. dla ciągu $(3.0, 5.0, 2.0, 4.0, 1.0, 6.0)$ będzie to 4.0.

Zadanie 4.65. Dla danej tablicy liczb rzeczywistych `t` rozmiaru n napisz kod, który wyznaczy wartość średniej arytmetycznej wszystkich elementów. Jest ona dana wzorem $\frac{1}{n}\sum_{i=0}^{n-1} t[i]$.

Zadanie 4.66. Dla danej tablicy liczb rzeczywistych `t` rozmiaru n napisz kod, który wyznaczy wartość średniej harmonicznej wszystkich elementów, tj. $\frac{n}{\sum_{i=0}^{n-1} 1/t[i]}$.

Zadanie 4.67. Niech dany będzie n-elementowy ciąg liczb rzeczywistych $\mathbf{a} = (a_1, a_2, \ldots, a_n)$. Średnią ważoną względem nieujemnego wektora wag $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ takiego, że $\sum_{i=1}^{n} w_i = 1$, nazywamy wartość

$$\mathsf{WM}_{\mathbf{w}}(\mathbf{a}) = \sum_{i=1}^{n} a_i w_i.$$

Napisz program, który wyznaczy wartość średniej ważonej ustalonego ciągu $(-2, -1, 0, 1, 2)$. Tablica wag powinna być wczytana z klawiatury. Należy sprawdzać, czy podane wagi są poprawne (czy są nieujemne i czy sumują się do 1 ± 0.0000001 – pozwalamy na mały błąd związany z zaokrąglaniem).

Zadanie 4.68. Niech dana będzie tablica `double x[n]` (np. dla $n = 9$), której elementy wczytane zostaną z klawiatury. Napisz program, który wyznaczy wartość normy euklidesowej x, danej wzorem

$$|\mathbf{x}| = \sqrt{\sum_{i=0}^{n-1} x[i]^2}.$$

Wskazówka. Skorzystaj z funkcji `sqrt(...)` z biblioteki `<cmath>`.

Zadanie 4.69. Dana jest tablica `int L[n]` dla pewnego $n > 0$. Napisz program, który wypisze na ekranie $L[0]$ liczb równych 0, $L[1]$ liczb 1, …, $L[n-1]$ liczb $(n-1)$.

Na przykład jeśli $L = (3, 6, 2)$ i $n = 3$, to w wyniku działania programu na konsoli powinien pojawić się napis `00011111122`.

Zadanie 4.70. Zaimplementuj algorytm sortowania przez wstawianie danej tablicy o n elementach typu `int`. Oblicz, ile jest potrzebnych operacji porównań oraz przestawień elementów w zależności od n dla tablicy już posortowanej oraz dla tablicy posortowanej w kolejności odwrotnej.

Zadanie 4.71. Zmodyfikuj podany na wykładzie algorytm sortowania bąbelkowego tablicy o n elementach typu `int`, tak by korzystał z tzw. wartownika.

Wartownik to dodatkowa zmienna w, która zapamiętuje indeks tablicy, pod którym wystąpiło ostatnie przestawienie elementów w pętli wewnętrznej. Dzięki niemu wystarczy, że w kolejnej iteracji pętla wewnętrzna zatrzyma się na indeksie w, a nie aż na pozycji $i-1$ (choć czasem może się zdarzyć, że $w = i-1$, por. algorytm z wykładu). W przypadku pomyślnego ułożenia danych w tablicy wejściowej może to bardzo przyspieszyć obliczenia.

Oblicz, ile operacji porównań oraz przestawień elementów jest potrzebnych w zależności od n, zarówno dla tablicy już posortowanej jak i dla tablicy posortowanej w kolejności odwrotnej.

Zadanie 4.72. Niech dany będzie wielomian rzeczywisty stopnia n (np. dla $n = 3$), $w(x) = w[0]x^0 + w[1]x^1 + \ldots w[n]x^n$, `w[n]` $\neq 0$, którego współczynniki przechowywane są w $(n+1)$-wymiarowej tablicy `w` o elementach typu **`double`**. Napisz program, który wczytuje z klawiatury ww. współczynniki oraz x, a następnie wyznacza wartość $w(x)$ wg powyższego wzoru.

(*) Zadanie 4.73. Zmodyfikuj program z zad. 4.72, tak by korzystał z bardziej efektywnego obliczeniowo (porównaj liczbę potrzebnych operacji arytmetycznych) wzoru na wartość $w(x)$, zwanego schematem Hornera:

$$w(x) = (\cdots(((w[n]x + w[n{-}1])\,x + w[n{-}2])\,x + w[n{-}3])\cdots)\,x + w[0].$$

(*) Zadanie 4.74. Niech dane będą wielomiany $w(x)$ i $v(x)$ stopnia, odpowiednio, n i m. Napisz program, który wyznaczy wartości współczynników wielomianu $u(x)$ stopnia $n+m$, będącego iloczynem wielomianów w i v. Dokonaj obliczeń dla $w(x) = x^4 + 4x^2 - x + 2$ oraz $v(x) = x^4 + x^3 + 10$.

(**) Zadanie 4.75. Dana jest tablica o `n>1` elementach typu **`double`**. Napisz program zawierający tylko jedną pętlę **`for`** (!), który obliczy wariancję jej elementów. Wariancja ciągu $\mathbf{x} = (x_1, \ldots, x_n)$ dana jest wzorem

$$s^2(\mathbf{x}) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{\mathrm{x}})^2\,,$$

gdzie $\overline{\mathrm{x}}$ jest średnią arytmetyczną ciągu $\mathbf{x}$. Na przykład $s^2(-0.40, 1.82, 2.71, -0.76, 0.04, -2.35, -0.75) \simeq 1.706213$.

(*) Zadanie 4.76. Dana jest tablica o `n>1` elementach typu **`double`**. Sprawdzić, czy każde trzy liczby z tablicy mogą być długościami boków pewnego trójkąta.

(*) Zadanie 4.77. Dane są dwie uporządkowane niemalejąco tablice **`int`** $x[n]$ i **`int`** $y[m]$ dla pewnych $n, m > 0$. Napisz program, który utworzy uporządkowaną niemalejąco tablicę **`int`** $z[n+m]$, powstałą przez scalenie tablic x i y. Algorytm powinien mieć liniową (rzędu $n+m$) pesymistyczną złożoność czasową. Na przykład dla $x = (1, 4, 5)$ i $y = (0, 2, 3)$ powinniśmy otrzymać $z = (0, 1, 2, 3, 4, 5)$.

(**) Zadanie 4.78. Napisz program, który wyznaczy wartość iloczynu dwóch danych liczb naturalnych, odpowiednio n- i m-cyfrowych (gdzie z góry ustalone wartości n i m mogą być bardzo duże, np. równe 100).

Przykładowe wejście ($n = 35, m = 38$):

```
64839007276493230832760759032850932
6496439069034792375897238975897328764 3
```

Przykładowe wyjście:

```
42122266006844181320994996577016265801...
60969340605765334159528899176633276
```

(*) Zadanie 4.79. Tak zwany ciąg EKG[2] $(a_1, a_2, \dots)$ o elementach naturalnych jest zdefiniowany następująco: $a_1 = 1$, $a_2 = 2$ oraz a_{n+1} jest najmniejszą liczbą naturalną nie występującą wcześniej w ciągu, taką że $\mathrm{NWD}(a_n, a_{n+1}) > 1$, tzn. mającą nietrywialny wspólny dzielnik z a_n. Napisz program, który wyznaczy a_n dla zadanego z góry n.

(*) Zadanie 4.80. Dana jest tablica t o n elementach całkowitych, reprezentująca permutację zbioru $\{0, 1, \dots, n-1\}$. Napisz fragment kodu, który wyznaczy liczbę cykli w tej permutacji.

Na przykład tablica:

$$\begin{pmatrix} 2, & 1, & 5, & 4, & 3, & 0 \\ t[0] & t[1] & t[2] & t[3] & t[4] & t[5] \end{pmatrix}$$

reprezentuje permutację o trzech cyklach. Zawiera bowiem cykl o długości 3 ($t[0] = 2 \to t[2] = 5 \to t[5] = 0$), cykl o długości 2 ($t[3] = 4 \to t[4] = 3$) oraz cykl o długości 1 ($t[1] = 1$).

(*) Zadanie 4.81. Ciąg Golomba $(g_1, g_2, \dots)$ to jedyny niemalejący ciąg liczb naturalnych, w którym każda liczba i występuje dokładnie g_i razy. Oto kilka jego początkowych wyrazów:

i	1	2	3	4	5	6	7	8	9	10	11	12
g_i	1	2	2	3	3	4	4	4	5	5	5	6

Zachodzi $g_1 = 1$. Napisz program, który wyznaczy n-ty wyraz tego ciągu dla z góry ustalonego n.

4.4 Laboratoria

Zadanie 4.82. Dana jest tablica zadeklarowana jako `int tab[n]` (dla pewnego `n`). Napisz kod, który przesunie każdy element o indeksie większym od zera o jedną komórkę w lewo, a element pierwszy wstawi na miejsce ostatniego.

[2] Por. http://www.deltami.edu.pl/temat/matematyka/teoria_liczb/2011/05/01/Ciag_EKG/

Zadanie 4.83. Za pomocą tylko jednej pętli `for` znajdź w tablicy `double tab[n]` element najmniejszy i element największy.

Zadanie 4.84. Znajdź w danej n-elementowej ($n \geqslant 2$) tablicy `double tab[n]` drugi najmniejszy element, np. dla ciągu $(3.0, 5.0, 2.0, 4.0, 1.0)$ będzie to 2.0.

Zadanie 4.85. Napisz kod, który znajduje najmniejszy element w tablicy `int tab[n]`. Następnie wypełnij tym elementem wszystkie komórki o parzystych indeksach oraz elementem przeciwnym do niego komórki o indeksach nieparzystych.

Zadanie 4.86. Napisz fragment kodu, który zliczy, ile razy w tablicy `bool tab[n]` występuje wartość `true` oraz dokona negacji wszystkich elementów w `tab`.

Zadanie 4.87. Dana jest n-elementowa tablica t składająca się z liczb całkowitych od 0 do 19. Wyznacz jej dominantę (modę), czyli najczęściej pojawiającą się wartość.
Wskazówka. Skorzystaj z pomocniczej, 20-elementowej tablicy, za pomocą której zliczysz, ile razy w tablicy t występuje każda możliwa wartość. Tablica powinna być zainicjowana zerami. Następnie należy znaleźć jej maksimum.

Zadanie 4.88. Niech dane będą dwie n-elementowe (np. dla $n = 2$) tablice x i y o elementach rzeczywistych (wartości pobrane z klawiatury). Napisz program, który policzy wartość odległości $d_m(x, y)$ w metryce supremum, wg wzoru

$$d_m(x, y) = \max_{i=0,1,\ldots,n-1} |x[i] - y[i]|.$$

Zadanie 4.89. Niech dane będą dwie n-elementowe tablice x i y złożone tylko z zer i jedynek. Tablice te reprezentują pewne dwie liczby nieujemne w systemie binarnym (element o indeksie 0 oznacza najmłodszą cyfrę, element o indeksie $n - 1$ oznacza najstarszą cyfrę). Utwórz $(n+1)$-elementową tablicę z, która przechowywać będzie kolejne cyfry sumy tych dwóch liczb binarnych oraz wypisz ją na ekran.

Zapewnij ponadto poprawność danych wejściowych, tzn. sprawdź, czy w tablicach wejściowych rzeczywiście są tylko zera i jedynki.

Skorzystaj z następującego szablonu programu.

```
1  // ...
2  int main()
3  {
4      // Przykladowe dane wejsciowe:
5      const int n = 4;
6      int x[n] = { 1, 0, 1, 1 }; // liczba 1101_2
7      int y[n] = { 1, 1, 0, 0 }; // liczba 0011_2
8
9      // Sprawdz poprawnosc x
10     // ...
11     // Sprawdz poprawnosc y
12     // ...
```

```
13      // Utworz tablice z
14      // ...
15      // Wyznacz wartosc sumy, wynik zapisz w tablicy z
16      // ...
17      // Wypisz wartosc sumy (tablica z) na ekran
18      // ...
19
20      return 0;
21  }
```

Wskazówka. Zastosuj dodawanie liczb systemem „szkolnym” (słupkowym). Najpierw wyznacz sumę dwóch najmłodszych cyfr i zapisz w dodatkowej zmiennej, czy nastąpiło przeniesienie (tzn. czy mamy przypadek $1_2 + 1_2 = 10_2$). Kolejne odpowiadające sobie cyfry sumuj z dodatkowym uwzględnieniem ewentualnego przeniesienia.

Zadanie 4.90. Zmodyfikuj rozwiązanie powyższego zadania, tak by przy wypisywaniu wartości liczb reprezentowanych przez tablice x, y i z pomijane były wiodące zera (najstarsze bity równe 0), np. liczba reprezentowana przez tablicę $(1, 0, 1, 1, 0, 0)$ powinna być wypisana jako `1101`. Nie zapomnij o uwzględnieniu przypadku, gdy jedna z tablic składa się z samych zer (wynik na ekranie powinien być wtedy pokazany jako po prostu `0`).

Zadanie 4.91. Zmodyfikuj rozwiązanie powyższego zadania, tak by wyznaczać wartość różnicy dwóch liczb. Wykonuj obliczenia tylko w przypadku (należy sprawdzić dodatkowy warunek), gdy pierwsza liczba nie jest mniejsza od drugiej.

Funkcje

5

5.1 Funkcje — podstawy

5.1.1 Funkcje w matematyce

Niech X i Y będą dowolnymi niepustymi zbiorami. Powiemy, że f jest **funkcją**[1] odwzorowującą (przekształcającą) X w Y, ozn. $\mathsf{f} : X \to Y$, jeśli dla każdego $x \in X$ istnieje dokładnie jeden element $y \in Y$, taki że $\mathsf{f}(x) = y$.

Zbiór X, złożony z elementów, dla których funkcja f została zdefiniowana, nazywamy **dziedziną** (lub zbiorem argumentów). Z kolei zbiór Y, do którego należą wartości funkcji, nazywamy jej **przeciwdziedziną**.

Przyjrzyjmy się dwom przykładom.

Przykład 1. Niech

$$\overbrace{\mathsf{kwadrat}:}^{\text{nazwa funkcji}} \quad \overbrace{\mathbb{R}}^{\text{dziedzina}} \quad \to \quad \overbrace{\mathbb{R}}^{\text{przeciwdziedzina}}$$

będzie funkcją, taką że

$$\mathsf{kwadrat}(x) := x \cdot x.$$

Powyższa funkcja oblicza po prostu kwadrat danej liczby rzeczywistej. Zauważmy, że definicja ta składa się z dwóch części. Pierwsza, „niech…”, to tzw. **deklarator** określający, że od tej pory identyfikator $\mathsf{kwadrat}$ oznacza funkcję o danej dziedzinie i danej przeciwdziedzinie. Druga, „taka że”, to tzw. **definicja właściwa**, która prezentuje abstrakcyjny **algorytm**, przepis, za pomocą którego dla konkretnego elementu dziedziny[2] możemy uzyskać wartość wynikową, tj. $\mathsf{kwadrat}(x)$. ■

[1] Por. H. Rasiowa, *Wstęp do matematyki współczesnej*, Warszawa, PWN 2003.

[2] Element ten oznaczyliśmy symbolem x, choć równie dobrze mogłoby to być t, ζ albo $\widehat{\alpha}^*$.

Przykład 2. Niech

$$\mathsf{sindod}: \quad \mathbb{R} \quad \to \quad \mathbb{R}$$

będzie funkcją, taką że

$$\mathsf{sindod}(u) := \begin{cases} \mathsf{sin}(u) & \text{dla } \mathsf{sin}(u) \geqslant 0, \\ 0 & \text{w p.p..} \end{cases}$$

Tym razem widzimy, że w definicji wprowadzonej funkcji pojawia się odwołanie do innego odwzorowania, tj. do dobrze znanej funkcji sin. Oczywiście w tym przypadku niejawnie odwołujemy się do wiedzy Czytelnika, że sin jest przekształceniem $\mathbb{R}$ w np. $\mathbb{R}$, danym jakimś-tam wzorem. ■

W niniejszej części naszego podręcznika, przyjrzymy się sposobom deklaracji i definicji funkcji w języku **C++** oraz różnym przykładom ich użycia. Dowiemy się, czym różnią się funkcje w **C++** od ich matematycznych odpowiedników. Nauczymy się, jak tworzyć funkcje „dla potomnych" i jak korzystać z „dorobku naszych przodków". Poznamy także bardzo ciekawą i potężną technikę zwaną rekurencją, w której funkcja wywołuje samą siebie.

5.1.2 Definiowanie funkcji w języku C++

Na etapie projektowania programu często można wyróżnić wiele **modułów** (części, podprogramów), z których każdy jest odpowiedzialny za pewną logicznie wyodrębnioną, niezależną czynność. Fragmenty mogą cechować się dowolną złożonością. Mogą też same korzystać z innych podmodułów.

Jednym z zadań programisty jest wtedy powiązanie tych fragmentów w spójną całość, m.in. przez zorganizowanie odpowiedniego przepływu sterowania i wymiany informacji (proces taki może przypominać budowanie domku z klocków różnych kształtów — w odróżnieniu od rzeźbienia go z jednego, dużego kawałka drewna).

Rozważmy dwa przykładowe zalążki programów, obrazujące pewne sytuacje z tzw. życia.

Przykład 3. Najpierw przyjrzyjmy się czynnościom potrzebnym do wyruszenia samochodem na wycieczkę (albo do rozpoczęcia egzaminu praktycznego na prawo jazdy).

```
1 int main()
2 {
3     ustawFotel();
4     ustawLusterka();
5     zapnijPasy();
6     przekrecKluczyk();
7     sprawdzKontrolki();
```

```
    uruchomRozrusznik();
    return 0;
}
```

Programiście, który wyróżnił ciąg czynności potrzebnych do wykonania pewnego zadania, pozostaje tylko szczegółowe określenie (implementacja), na czym one polegają. Zauważmy, że dzięki takiemu sformułowaniu rozwiązania możliwe jest łatwiejsze zapanowanie nad złożonością kodu. ■

Przykład 4. Kolejny przykład dotyczy organizacji nauki pewnego pilnego studenta.

```
int main()
{
    // ...
    do
    {
        if (dzienTygodnia == niedziela)
            continue;

        pouczSieTroche();

        if (zmeczony) {
            if (godzinNaukiDzis > 5)
                bedeGralWGre();
            else
                odpocznijChwile();
        }
    } while (!nauczony);
    // ...
}
```

■

Najprostszym sposobem podziału programu w języku **C++** na (pod)moduły jest użycie funkcji.

Ważne

Funkcja (zwana też czasem procedurą, metodą, podprogramem) to odpowiednio wydzielony fragment kodu wykonujący pewne instrukcje. Do funkcji można odwoływać się z innych miejsc programu.

Informacja

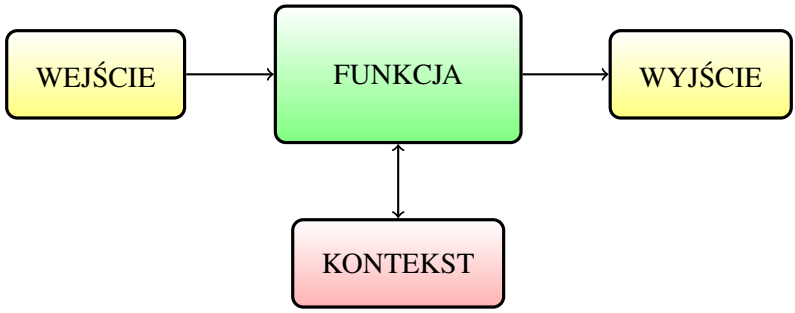
Funkcje służą do dzielenia kodu programu na mniejsze, łatwiejsze w tworzeniu, zarządzaniu i testowaniu fragmenty o ściśle określonym działaniu. Fragmenty te są względnie niezależne od innych części.

Ponadto dzięki funkcjom uzyskujemy m.in. możliwość wykorzystania tego samego kodu wielokrotnie.

Projektując każdą funkcję, należy wziąć pod uwagę, w jaki sposób ma ona wchodzić w interakcję z innymi funkcjami, to znaczy jakiego typu **dane wejściowe** powinna ona przyjmować i jakiego typu **dane wyjściowe** będą wynikiem jej działania.

Ważne

Bezpośrednim skutkiem działania funkcji jest uzyskanie jakiejś konkretnej wartości na wyjściu. **Skutkiem pośrednim** działania funkcji może być z kolei zmiana tzw. *kontekstu*, tj. stanu komputera (np. wypisanie czegoś na ekran, pobranie wartości z klawiatury itp.).



Rysunek 5.1. Schemat przepływu danych w funkcjach.

Powyższa idea została zobrazowana na rys. 5.1.

Przyjrzyjmy się wpierw najważniejszym różnicom między funkcjami w matematyce a funkcjami w języku `C++`.

a) **Funkcje w matematyce nie mają skutków pośrednich**. Jedynym efektem ich działania może być tylko przekształcenie konkretnego elementu dziedziny w ściśle określony element przeciwdziedziny. Funkcje w `C++` z kolei mogą, niejako „przy okazji”, wypisać coś na ekran, zapisać na dysk plik pobrany z internetu, albo też odegrać piosenkę w formacie MP3.

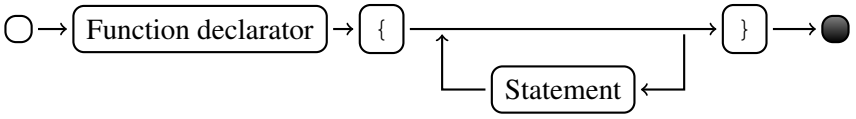
b) **Dziedzina i przeciwdziedzina funkcji matematycznej nie może być zbiorem pustym**. W języku `C++` został określony specjalny typ `void` (∅, od ang. *pusty*),

który pozwala na określenie funkcji „robiącej coś z niczego” (korzystającej tylko z kontekstu, np. danych pobranych z klawiatury), „niby-nic z czegoś” (tylko np. zmieniającej kontekst) lub nawet „niby-nic z niczego”.

c) **Przeciwdziedziną funkcji w języku C++ może być tylko jeden typ**[3]. Dziedziną jednak może być z powodzeniem iloczyn kartezjański innych typów.

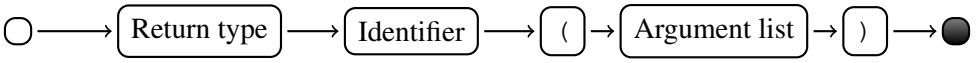
Ciekawostka
Funkcje w języku programowania mogłyby być postrzegane jako funkcje matematyczne, jeśliby ich dziedziną było zawsze $X \times \mathbb{K}$, a przeciwdziedziną $Y \times \mathbb{K}$, gdzie $\mathbb{K}$ oznacza kontekst działania komputera (bardzo trudny w opisie, na niego bowiem składają się wszystkie urządzenia wejściowe i wyjściowe komputera, w tym urządzenia wideo, audio, sieciowe, pamięć masowa itp.).

Przejdźmy do opisu składni **definicji funkcji** w języku C++. Możemy ją przedstawić następująco.

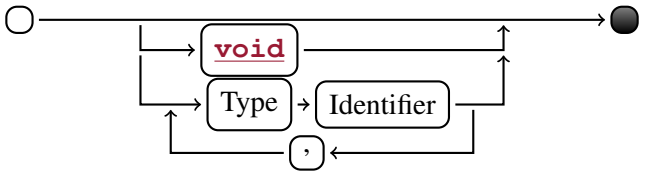


Blok instrukcji zawarty między nawiasami klamrowymi to tzw. **ciało funkcji** (definicja właściwa) — tutaj zawarta jest implementacja algorytmu realizowanego przez funkcję.

Z kolei Function declarator to **deklarator funkcji** postaci



gdzie Identifier jest pewnym identyfikatorem oznaczającym nazwę definiowanej funkcji, Return type jest typem określającym przeciwdziedzinę (może to być `void`), a Argument list to lista argumentów określająca dziedzinę



Zauważmy, że pusta lista argumentów jest tożsama z `void`. W ten sposób można określać funkcje pobierające dane tylko z tzw. kontekstu.

[3] W pewnym sensie da się ominąć to ograniczenie korzystając z argumentów przekazanych przez referencję (o czym dalej) albo korzystając z tzw. struktur, które poznamy dalej.

Informacja
Do zwracania wartości przez funkcję (należących do określonej przeciwdziedziny) służy instrukcja **return**.

Instrukcja **return** działa podobnie jak **break** w przypadku pętli, tj. natychmiast przerywa wykonywanie funkcji, w której aktualnie znajduje się sterowanie. Dzięki temu następuje od razu powrót do miejsca, w którym nastąpiło wywołanie danej funkcji.

W przypadku funkcji o przeciwdziedzinie **void**, instrukcja **return** (w takim przypadku nieprzyjmująca żadnych argumentów) może zostać pominięta. Po wykonaniu ostatniej instrukcji następuje automatyczny powrót do funkcji wywołującej.

Ważne
Funkcje są podstawowym „budulcem" programów pisanych w języku **C++**. Każda wykonywana instrukcja (np. pętla, instrukcja warunkowa, instrukcja przypisania) musi należeć do jakiejś funkcji.

Funkcji w języku **C++** nie można zagnieżdżać, tzn. nie można tworzyć funkcji w funkcji.

Przykład 5. Przypomnijmy, że najprostszy, pełny kod źródłowy działającego programu w języku **C++** można zapisać w sposób następujący.

```
1 int main()
2 {
3     return 0; // zakonczenie programu (,,sukces'')
4 }
```

Widzimy zatem (nareszcie!), że `main()` jest funkcją, która nie przyjmuje żadnych parametrów wejściowych, a na wyjściu zwraca wartość całkowitą. Robi zatem „coś" z „niczego"! ■

Ciekawostka
Wartość zwracaną przez funkcję `main()` odczytuje system operacyjny. Dzięki temu wiadomo, czy program zakończył się prawidłowo (wartość równa 0), czy też jego działanie nie powiodło się (wartość różna od 0). Z reguły informacja ta jest ignorowana i na jej podstawie nie jest podejmowane żadne specjalne działanie.

Informacja

Każdy program w języku **C++** musi zawierać definicję funkcji `main()`, inaczej proces kompilacji nie powiedzie się. `main()` (od ang. *główny*) stanowi punkt startowy każdego programu. Instrukcje zawarte w tej funkcji zostają wykonane jako pierwsze zaraz po załadowaniu programu do pamięci komputera.

5.1.3 Wywołanie funkcji

Rozpatrzmy jeszcze raz definicję funkcji kwadrat.

Przykład 1 (cd.). Niech

$$\underbrace{\text{kwadrat}:}_{\text{nazwa funkcji}} \quad \underbrace{\mathbb{R}}_{\text{dziedzina}} \quad \to \quad \underbrace{\mathbb{R}}_{\text{przeciwdziedzina}}$$

będzie funkcją, taką że

$$\text{kwadrat}(x) := x \cdot x.$$

Przyjrzyjmy się jej implementacji i jej przykładowemu zastosowaniu.

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

                              //  ,,Niech...''
double kwadrat(double x)
{                             //  ,,taka, ze...''
    return x*x;
}


int main() // funkcja bezargumentowa
{
    double y = 0.5;
    cout << kwadrat(y)    << endl; // wywolanie funkcji
    cout << kwadrat(2.0) << endl; // mozna i tak
    return 0;
}
```

W powyższej funkcji `main()` zapis `kwadrat(y)` oznacza **wywołanie** (nakaz ewaluacji) funkcji `kwadrat()` na argumencie równym wartości przechowywanej w zmiennej `y`.

W momencie wywołania działanie funkcji `main()` zostaje wstrzymane, a kontrolę nad działaniem programu przejmuje funkcja `kwadrat()`. Parametr `x` otrzymuje wartość `0.5` i od tego momentu można traktować go jak zwykłą zmienną. Po wywołaniu instrukcji **`return`** przebieg sterowania wraca do funkcji `main()`. ■

Ważne

Zapis `kwadrat(y)` jest więc traktowany jako wyrażenie o typie takim, jak przeciwdziedzina funkcji (czyli tutaj: **`double`**).

Obliczenie wartości takiego wyrażenia następuje przez wykonanie kodu funkcji na danej wartości przekazanej na wejściu.

Wywołanie funkcji można więc wyobrażać sobie jako „zlecenie" wykonania pewnej czynności przekazane jakiemuś „podwykonawcy". „Masz tu coś. Chcę uzyskać z tego to i to. Nie interesuje mnie, jak to zrobisz (to już twoja sprawa), dla mnie się liczy tylko wynik".

Z tych powodów w powyższej funkcji `main()` moglibyśmy także napisać

```
1 double y = 0.5;
2 y = kwadrat(y);
3 // albo
4 double z = kwadrat(4.0)-3.0*kwadrat(-1.5);
5 // itp.
```

Co więcej, należy pamiętać, że instrukcja

```
cout << kwadrat(kwadrat(0.5));
```

zostanie wykonana w sposób podobny do następującego

```
double zmPomocnicza1 = kwadrat(0.5);
double zmPomocnicza2 = kwadrat(zmPomocnicza1);
cout << zmPomocnicza2;
```

Wartości pośrednie są obliczanie i odkładane „na boku". Tym samym, podanie „zewnętrznej" funkcji `kwadrat()` argumentu „`kwadrat(0.5)`" jest tożsame z przekazaniem jej wartości `0.25`.

Jest to o tyle istotne, iż w przeciwnym przypadku „`kwadrat(0.5)`" musiałby być wyliczony dwukrotnie (mamy przecież **`return`** `x*x;` w definicji). Nic takiego na szczęście jednak się nie dzieje.

5.1.4 Definicja a deklaracja

Jako że program czytany jest (przez nas jak i przez kompilator) od góry do dołu, **definicja każdej funkcji musi się pojawić przed jej pierwszym użyciem**. W niektórych

przypadkach może się jednak zdarzyć, że lepiej będzie (np. dla czytelności) umieścić definicję funkcji w jakimś innym miejscu. Można to zrobić pod warunkiem, że obiektom z niej korzystającym zostanie udostępniona jej **deklaracja**.

Intuicyjnie, na etapie pisania programów, innym obiektom wystarcza informacja, że dana funkcja istnieje, nazywa się tak i tak, przyjmuje podane wartości i zwraca coś określonego typu. Dokładna specyfikacja jej działania nie jest im de facto potrzebna.

Informacja

Deklaracji funkcji dokonujemy w miejscu, w którym nie została ona jeszcze użyta (ale poza inną funkcją). Podajemy wtedy jej **deklarator** (Function declarator na schemacie) **zakończony średnikiem**. Co ważne, w deklaracji musimy użyć takiej samej nazwy funkcji oraz typów argumentów wejściowych i wyjściowych jak w definicji. Nazwy (identyfikatory) tych argumentów mogą być jednak inne, a nawet mogą zostać pominięte.

Definicja funkcji może wówczas pojawić się w dowolnym innym miejscu programu (także w innym pliku źródłowym, zob. dalej).

Przykład 1 (cd.). Nasz przykład możemy zatem „sparafrazować" następująco:

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

double kwadrat(double); // deklaracja - tu jest srednik

int main()
{
    double x = kwadrat(2.0);
    cout << x                          << endl;
    cout << kwadrat(x)+kwadrat(8.0)    << endl;
    cout << kwadrat(kwadrat(0.5))      << endl;
    return 0;
}

// uszczegolowienie:
double kwadrat(double x) // definicja - tu nie ma srednika
{
     return x*x;
}
```

■

Ważne
Kolejny raz mamy okazję zaobserwować, jak ważne jest, by nadawać obiektom (zmiennym, funkcjom) intuicyjne, samoopisujące się identyfikatory!

5.1.5 Własne biblioteki funkcji

Zbiór różnych funkcji, które warto mieć „pod ręką", możemy umieścić w osobnych plikach, tworząc tzw. **bibliotekę** funkcji. Dzięki temu program stanie się bardziej czytelny, łatwiej będziemy mogli zapanować nad jego złożonością, a ponadto otrzymamy sposobność wykorzystania pewnego kodu ponownie w przyszłości. Raz napisany i przetestowany zestaw funkcji może oszczędzić sporo pracy.

Aby stworzyć bibliotekę funkcji, tworzymy najczęściej dwa dodatkowe pliki.
- plik nagłówkowy (ang. *header file*), `nazwabiblioteki.h` — zawierający tylko deklaracje funkcji,
- plik źródłowy (ang. *source file*), `nazwabiblioteki.cpp` — zawierający definicje zadeklarowanych funkcji.

Plik nagłówkowy należy dołączyć do wszystkich plików, które korzystają z funkcji z danej biblioteki (również do pliku źródłowego biblioteki). Dokonujemy tego za pomocą dyrektywy:

```
#include "nazwabiblioteki.h"
```

Zwróćmy uwagę, że nazwa naszej biblioteki, znajdującej się wraz z innymi plikami tworzonego projektu, ujęta jest w cudzysłowy, a nie w nawiasy trójkątne (które służą do ładowania bibliotek systemowych).

Informacja
Najlepiej, gdy wszystkie pliki źródłowe i nagłówkowe znajdować się będą w tym samym katalogu. Pamiętajmy o tym, by uniknąć problemów.

Przykład 6. Dla ilustracji przyjrzymy się, jak wyglądałby program, korzystający z biblioteki zawierającej dwie następujące funkcje

Niech

$$\mathsf{min} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z},$$

$$\mathsf{max} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z},$$

będą funkcjami, takimi że

$$\mathsf{min}(n,m) := \begin{cases} n & \text{dla } n \leqslant m, \\ m & \text{dla } n > m, \end{cases}$$

oraz

$$\mathsf{max}(n,m) := \begin{cases} n & \text{dla } n \geqslant m, \\ m & \text{dla } n < m. \end{cases}$$

Nasz program będzie składał się z trzech plików: nagłówkowego `podreczna.h` oraz dwóch źródłowych `podreczna.cpp` i `program.cpp`.

1. Oto zawartość pliku `podreczna.h` — zawierającego tylko deklaracje funkcji pomocniczych.

```
#pragma once  // na poczatku kazdego pliku naglowkowego!

int min(int i, int j);
int max(int i, int j);
```

Ważne

Dyrektywa

```
#pragma once
```

powinna znaleźć się na początku każdego pliku nagłówkowego. Dzięki niej zapobiegniemy wielokrotnemu dołączaniu tego pliku w przypadku korzystania z niego przez inne biblioteki.

Ciekawostka

Dyrektywa `#pragma once` dostępna jest tylko w kompilatorze *Visual C++*. W innych kompilatorach uzyskanie tego samego efektu jest nieco bardziej złożone:

```
#ifndef __PODRECZNA_H
#define __PODRECZNA_H

// deklaracje....

#endif
```

2. Definicje funkcji z naszej biblioteki znajdą się w pliku `podreczna.cpp`.

```
#include "podreczna.h" /* dolacz plik z deklaracjami */

int min(int x, int y)  /* nazwy parametrow nie musza byc
    takie same jak w deklaracji */
{
    if (x <= y) return x;
    else        return y;
}

int max(int w, int z)
{
    if (w >= z)
       return w;
    return z;
}
```

3. Gdy biblioteka jest gotowa, można przystąpić do napisania funkcji głównej. Oto zawartość pliku `program.cpp`.

```
#include <iostream>     /* dolacz biblioteke systemowa */
using namespace std;

#include "podreczna.h" /* dolacz wlasna biblioteke */

int main(void)
{
    int x, y;

    cout << "Podaj dwie liczby. ";
    cin  >> x >> y; // wczytaj z klawiatury

    cout << "Max=" << max(x,y) << endl;
    cout << "Min=" << min(x,y) << endl;

    return 0;
}
```

■

5.1.6 Przegląd funkcji w bibliotece standardowej

Elementem standardu naszego języka jest też wiele przydatnych (gotowych do natychmiastowego użycia) funkcji zawartych w tzw. bibliotece standardowej. Poniżej omówimy te, które interesować nas będą najbardziej.

5.1.6.1 Funkcje matematyczne

Biblioteka <cmath> udostępnia wybrane funkcje matematyczne[4]. Ich przegląd zamieszczamy w tab. 5.1, 5.2 i 5.3.

Tabela 5.1. Funkcje trygonometryczne dostępne w bibliotece <cmath>.

Deklaracja	Znaczenie
`double cos(double);`	cosinus (argument w rad.)
`double sin(double);`	sinus (argument w rad.)
`double tan(double);`	tangens (argument w rad.)
`double acos(double);`	arcus cosinus (wynik w rad.)
`double asin(double);`	arcus sinus (wynik w rad.)
`double atan(double);`	arcus tangens (wynik w rad.)
`double atan2(double y, double x);`	arcus tangens y/x (jw.)

Tabela 5.2. Funkcja wykładnicza, logarytm, potęgowanie w bibliotece <cmath>.

Deklaracja	Znaczenie
`double exp(double);`	funkcja wykładnicza
`double log(double);`	logarytm naturalny
`double log10(double);`	logarytm dziesiętny
`double sqrt(double);`	pierwiastek kwadratowy
`double pow(double x, double y);`	x^y

Tabela 5.3. Funkcje dodatkowe w bibliotece <cmath>.

Deklaracja	Znaczenie
`double fabs(double);`	wartość bezwzględna
`double ceil(double);`	sufit
`double floor(double);`	podłoga

[4]Pełna dokumentacja biblioteki <cmath> w języku angielskim dostępna jest do pobrania ze strony https://cplusplus.com/reference/cmath/.

Dla przypomnienia, funkcja sufit określona jest wzorem

$$\lceil x \rceil = \min\{i \in \mathbb{Z} : i \geqslant x\},$$

a funkcja podłoga zaś jako

$$\lfloor x \rfloor = \max\{i \in \mathbb{Z} : i \leqslant x\}.$$

Przykład 2 (cd.). Przypomnijmy definicję funkcji sindod. Niech

$$\mathsf{sindod} : \quad \mathbb{R} \quad \to \quad \mathbb{R}$$

będzie funkcją, taką że

$$\mathsf{sindod}(u) := \begin{cases} \mathsf{sin}(u) & \text{dla } \mathsf{sin}(u) \geqslant 0, \\ 0 & \text{w p.p..} \end{cases}$$

Oto przykładowy program wyznaczający wartość tej funkcji w różnych punktach.

```
#include <iostream>
#include <cmath>
using namespace std;

double sindod(double);

int main()
{
    cout << sindod(-3.14159*0.5) << endl;
    cout << sindod(+3.14159*0.5) << endl;
    return 0;
}


double sindod(double u)
{
    double s = sin(u); // funkcja z biblioteki <cmath>
    if (s >= 0) return s;
    else        return 0.0;
}
```

■

5.1.6.2 Liczby pseudolosowe

W bibliotece <cstdlib> znajduje się funkcja służąca do generowania liczb pseudolosowych.

Generator należy zainicjować przed użyciem funkcją `void srand(int z)`, gdzie `z>1` to tzw. **ziarno**. Jedno ziarno generuje zawsze ten sam ciąg liczb. Można także użyć aktualnego czasu systemowego do zainicjowania generatora. Dzięki temu podczas każdego kolejnego uruchomienia programu otrzymamy inny ciąg.

```
#include <cstdlib>
#include <ctime> // tu znajduje sie funkcja time()

//...
srand(time(0)); // za kazdym razem inne liczby
//
```

Funkcja `int rand();` generuje (za pomocą czysto algebraicznych metod) pseudolosowe liczby całkowite z rozkładu dyskretnego jednostajnego określonego na zbiorze

$$\{0, 1, \ldots, RAND\_MAX - 1\}.$$

Jednostajność oznacza, że prawdopodobieństwo „wylosowania" każdej z liczb jest takie samo. *RAND_MAX* jest stałą zdefiniowaną w bibliotece <cstdlib>.

Jeśli chcemy uzyskać liczbę np. ze zbioru $\{1, 2, 3\}$, możemy napisać[5]:

```
cout << (rand() % 3) + 1;
```

Aby z kolei uzyskać liczbę rzeczywistą z przedziału $[0, 1)$, piszemy

```
cout << ((double)rand() / (double)RAND_MAX);
```

[5] Podana metoda nie cechuje się zbyt dobrymi własnościami statystycznymi.

Przykład 7. Korzystając z własnoręcznie napisanej funkcji generującej liczbę rzeczywistą z przedziału $[0, 1)$, łatwo napisać funkcję „losującą” liczbę ze zbioru $\{a, a+1, \ldots, b\}$, gdzie $a, b \in \mathbb{Z}$.

Oto przykładowy program.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>
using namespace std;

double los01()
{
    return ((double)rand() / (double)RAND_MAX);
}

int losAB(int a, int b)
{
    double ab = los01()*(b-a+1)+a; // liczba rzeczywista
                                   // z przedzialu [a,b+1)
    return (int)(floor(ab));       // ,,podloga'' z ab
}

int main()
{
    srand(time(0)); // zainicjuj generator
    for (int i=0; i<5; i++) // 5 razy...
       cout << losAB(1,10) << " ";
    cout << endl;
    return 0;
}
```

Przykładowy wynik na ekranie:

```
4 3 8 2 1
```

a po chwili:

```
4 6 10 4 2
```

■

Informacja
W bibliotece <cstdlib> znajduje się także funkcja `void exit(int status);`, której wywołanie powoduje natychmiastowe zakończenie programu. Działa ona podobnie jak wywołanie instrukcji `return` w `main()`, jednakże można z niej korzystać (z powodzeniem) w dowolnym miejscu programu.

5.1.6.3 Asercje

Biblioteka <cassert> udostępnia funkcję o następującej deklaracji

```
void assert(bool);
```

Funkcja `assert()` umożliwia sprawdzenie dowolnego warunku logicznego. Jeśli nie jest on spełniony, nastąpi zakończenie programu. W przeciwnym wypadku nic się nie stanie.

Taka funkcja może być szczególnie przydatna przy testowaniu programu. Zabezpiecza ona m.in. przed danymi, które teoretycznie nie powinny się w danym miejscu pojawić.

Przykład 8. Rozpatrzmy „bezpieczną” funkcję wyznaczającą pierwiastek z nieujemnej liczby rzeczywistej.

```
#include <cassert>
#include <cmath>

double pierwiastek(double x)
{
    \\ Dane: x ⩾ 0
    \\ Wynik: √x

    assert(x>=0); // jesli nie, to blad - zakonczenie
        programu
    return sqrt(x);
}
```

■

Ciekawostka
Sprawdzanie warunków przez wszystkie funkcje `assert()` można wyłączyć globalnie za pomocą dyrektywy

```
#define NDEBUG
```

umieszczonej na początku pliku źródłowego lub nagłówkowego.

5.2 Rozszerzenie wiadomości o funkcjach

W niniejszym podrozdziale rozszerzymy naszą wiedzę o funkcjach, poznając bardziej zaawansowane zagadnienia.

5.2.1 Zasięg zmiennych

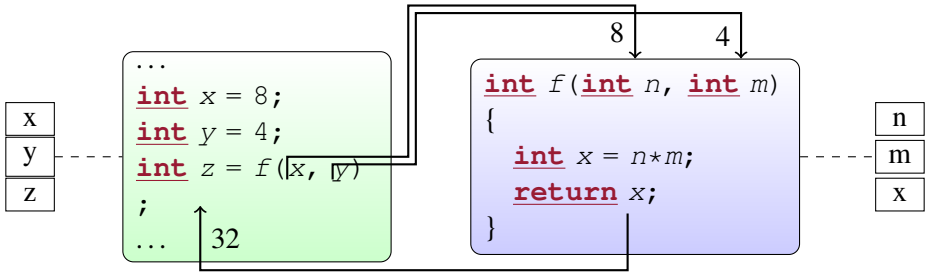
Przyjrzyjmy się **zasięgowi** definiowanych przez nas zmiennych. Zasięg określa, jak długo dany obiekt istnieje i w jaki sposób dane zmienne są widoczne z innych miejsc programu.

Zmienne które definiujemy wewnątrz każdej funkcji to tzw. **zmienne lokalne**. **Tworzone** są one, gdy następuje wywołanie funkcji, a **usuwane**, gdy następuje jej opuszczenie.

Zmienne lokalne nie są współdzielone między funkcjami. Zmienna `x` w `f()` to zmienna inna niż `x` w funkcji `g()`. Zatem jedynym sposobem wymiany danych między funkcjami jest zastosowanie parametrów wejściowych i wartości zwracanych.

Parametry funkcji spełniają **rolę zmiennych lokalnych**, których wartości są przypisywane automatycznie przy wywołaniu funkcji.

Przyjrzyjmy się rys. 6.1.



Rysunek 5.2. Zasięg zmiennych.

Jak powiedzieliśmy, `x` po lewej i `x` po prawej to dwie różne zmienne. Zmienne lokalne `n`, `m`, `x` są tworzone na użytek wewnętrzny aktualnego wywołania funkcji `f()`.

Mają one „pomóc” funkcji w spełnieniu swego zadania, jakim jest uzyskanie na wyjściu pewnej wartości, która jest konieczna do działania bloku kodu po lewej stronie. Zmienne te zostaną skasowane zaraz po tym, gdy funkcja zakończy swoje działanie (nie są one już do niczego potrzebne). W tym sensie można traktować taką funkcję jako **czarną skrzynkę**, gdyż to, jakie procesy wewnątrz niej zachodzą, nie ma bezpośredniego wpływu na obiekt, który posiłkuje się `f()` do osiągnięcia swoich celów.

Spójrzmy na blok kodu po lewej stronie. Zmienne `x` i `y` są przekazywane do funkcji `f()` **przez wartość**. Znaczy to, że wartości parametrów są obliczane przed wywołaniem funkcji (w tym przypadku pobierane są po prostu wartości przechowywane w tych zmiennych), a wyniki tych operacji są **kopiowane** do argumentów wejściowych. Widoczne są one w `f()` jako zmienne lokalne, odpowiednio, `n` i `m`.

Z wartością zwracaną za pomocą instrukcji **`return`** (prawa strona rysunku) kompilator postępuje w sposób analogiczny, tzn. nie przekazuje obiektu `x` jako takiego, lecz wartość, którą on przechowuje. Jeśli w tym miejscu stałoby np. złożone wyrażenie arytmetyczne albo stała, reguła ta byłaby oczywiście zachowana.

Ważne
Powtórzmy, zmienne przekazywane **przez wartość** są **kopiowane**.

Co więcej, zmienne lokalne nie są przypisane na stałe samym funkcjom, tylko ich wywołaniom. Obiekty utworzone dla jednego wywołania funkcji są **niezależne** od zmiennych dla innych wywołań. Jest to bardzo istotne w przypadku techniki zwanej rekurencją, która polega na tym, że funkcja wywołuje samą siebie (zob. dalej). Innymi słowy, wywołując tę samą funkcję `f()` raz po raz, za każdym razem tworzony jest nowy, niezależny zestaw zmiennych lokalnych.

Ciekawostka
W języku **C++** dostępne są także zmienne globalne. Jednakże stosowanie ich nie jest zalecane. Nie będziemy ich zatem omawiać.

5.2.2 Przekazywanie parametrów przez referencję

Wiemy już, że standardowo parametry wejściowe funkcji zachowują się jak zmienne lokalne – ich zmiana nie jest widoczna „na zewnątrz".

Przykład 9. Rozważmy następujący przykład — zamianę (ang. *swap*) dwóch zmiennych.

```
1  void zamien(int x, int y)
2  {
3      int t = x;
4      x = y;
5      y = t;
6  }
7
8  int main()
9  {
10     int n = 1, m = 2;
11     zamien(n, m); // przekazanie parametrow przez wartosc
12                   // (skopiowanie wartosci)
13     cout << n << ", " << m << endl;
14     return 0;
15 }
```

Wynikiem działania tego fragmentu programu jest, rzecz jasna, napis `1, 2` na ekranie. Funkcja `zamien()` z punktu widzenia `main()` nie robi zupełnie nic. ■

W pewnych szczególnych przypadkach uzasadnione jest zatem przekazywanie parametrów w inny sposób — **przez referencję** (odniesienie). Dokonuje się tego przez dodanie znaku `&` bezpośrednio po typie zmiennej na liście parametrów funkcji.

Przekazanie parametrów przez referencję umożliwia nadanie bezpośredniego dostępu (również do zapisu) do zmiennych lokalnych funkcji wywołującej (być może pod inną nazwą). Obiekty te **nie są kopiowane**; udostępniane są takimi, jakimi są.

Co ważne, w taki sposób można tylko przekazać zmienną. Próba przekazania wartości zwracanej przez jakąś inną funkcję, stałej albo złożonego wyrażenia arytmetycznego zakończy się niepowodzeniem.

Przykład 9 (cd.). Tym samym dopiero teraz możemy przedstawić prawidłową wersję funkcji `zamien()`.

```
void zamien(int& x, int& y)
{
    int t = x;
    x = y;
    y = t;
}

int main()
{
    int n = 1, m = 2;
    zamien(n, m); // przekazanie param. przez referencje
    cout << n << ", " << m << endl;
    return 0;
}
```

Zmienna `n` widoczna jest tutaj pod nazwą `x`. Jest to jednak de facto ta sama zmienna – są to dwa różne odniesienia do tej samej, współdzielonej komórki pamięci.

Uzyskujemy wreszcie oczekiwany wynik: `2, 1`. ■

Istnieje jeszcze jedno ważne zastosowanie zmiennych przekazywanych przez referencję. Może ono służyć do obejścia ograniczenia związanego z tym, że funkcja za pomocą instrukcji **`return`** może zwrócić co najwyżej jedną wartość.

Przykład 10. Rozpatrzmy poniższy przykład. Jest to funkcja zwracająca część całkowitą ilorazu *oraz* resztę z dzielenia dwóch liczb.

```
void iloreszta(int x, int y, int& iloraz, int& reszta)
{
    iloraz = x / y;
    reszta = x % y;
}

int main()
{
    int n = 7, m = 2;   // wejscie
    int i, r;   // to bedzie wynik
    iloreszta(n, m, i, r);
    cout << n << "=" << i << "*" << m << "+" << r;
    return 0;
}
```

Wynikiem tego programu będzie więc `7=3*2+1`. ■

5.2.3 Parametry domyślne funkcji

Parametry domyślne to argumenty, których jawne pominięcie przy wywołaniu funkcji powoduje, że zostaje im przypisana pewna z góry ustalona wartość.

Parametry z wartościami domyślnymi mogą być tylko przekazywane przez wartość (nie: referencję). Ponadto mogą się one pojawić jedynie na końcu listy parametrów (choć może być ich wiele).

Jeśli rozdziela się deklarację i definicję funkcji, parametry domyślne powinny pojawić się w tej pierwszej części.

Oto kilka przykładów prawidłowych deklaracji funkcji:

- `void f(int x=3);`
- `void f(int x=3, int y=2, int z=5);`
- `void f(int x, int y=3, int z=2);`
- `void f(int x, int y, int z=2);`
- `void f(int& x, int y=2);`

A oto nieprawidłowe deklaracje funkcji:

- `void f(int x, int y=3, int z);`
- `void f(int x=3, int y);`
- `void f(int x, int& y=2);`

Przykład 11. Dla ilustracji rozważmy funkcję wyznaczającą pierwiastek liczby rzeczywistej, domyślnie o podstawie 2.

```
#include <cmath>

double pierwiastek(double x, double p=2)
{ // pierwiastek, domyslnie kwadratowy
   assert(p >= 1);
   return pow(x, 1.0/p);
}

int main(void)
{
   cout << pierwiastek(10) << endl;       // 3.162278
   cout << pierwiastek(10, 2.0) << endl; // 3.162278
   cout << pierwiastek(10, 3.0) << endl; // 2.154435
   return 0;
}
```

■

5.2.4 Przeciążanie funkcji

W języku **C++** można nadawać te same nazwy (identyfikatory) wielu funkcjom pod warunkiem, że różnią się one co najmniej typem lub liczbą parametrów. Jest to tzw. **przeciążanie** funkcji (ang. *function overloading*). Funkcje takie mogą się różnić zwracanym typem, nie jest to jednak warunek dostateczny pozwalający na rozróżnienie funkcji przeciążonych.

Przeciążanie ma sens, gdy funkcje wykonują podobne (w sensie interpretacyjnym) czynności, jednakże na danych różnego typu.

Przykład poprawnych deklaracji:

```
int    modul(int x);
double modul(double x);
```

Przykłady niepoprawnych deklaracji:

```
void f();
int  f(int x, int y=2);
int  f(int x); // Blad: szczegolny przypadek powyzszego
```

oraz:

```
bool g(int x, int y);
char g(int x, int y); // Blad: nie roznia sie argumentami
```

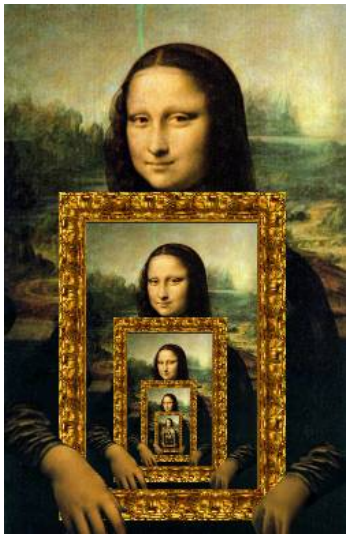
5.3 Rekurencja

Z **rekurencją** (lub rekursją, ang. *recursion*) mamy do czynienia wtedy, gdy w definicji pewnego obiektu pojawia się odniesienie do niego samego.

Rozważmy najpierw rys. 5.3. Przedstawia on pewną znaną damę, trzymającą obraz, który przedstawia ją samą trzymającą obraz, na którym znajduje się ona sama trzymająca obraz... Podobny efekt moglibyśmy uzyskać, filmując telefonem ekran pokazujący to, co właśnie nagrywa kamera.

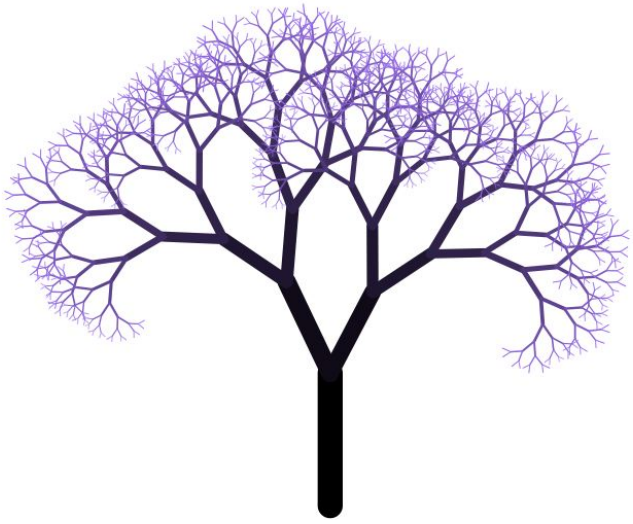
A teraz popatrzmy na rys. 5.4 przedstawiający prosty fraktal. Przypatrując się dłużej tej strukturze widzimy, że ma ona bardzo prostą konstrukcję. Wydaje się, że narysowanie kształtu takiego drzewa odbywa się następująco. Rysujemy odcinek o pewnej długości. Następnie obróciwszy się nieco w lewo/prawo, rysujemy nieco krótszy odcinek. W punktach, w których zakończyliśmy rysowanie, czynimy podobnie itd.

W przypadku języków programowania mówimy o rekurencji, **gdy funkcja wywołuje samą siebie**. Czyniąc tak jednak we „frywolny" sposób spowodowalibyśmy zawieszenie komputera. Istotną cechą rekurencji jest więc dodatkowo **warunek stopu**, czyli zdefiniowanie przypadku, w którym zagłębianie się rekurencyjne zostaje przerwane.



Rysunek 5.3. Rekurencyjna Mona Lisa.

Źródło: http://www.dominiek.eu/blog/wp-content/uploads/2007/11/megamonalisa_recursion.jpg.



Rysunek 5.4. Fraktalne drzewo.

Źródło: http://diffusedproductions.net/images/fractal-tree2.jpg.

Rekurencja, jak widzimy, jest bardzo prostą techniką, której opanowanie pozwoli nam tworzyć bardzo ciekawe i często łatwe do zaprogramowania algorytmy. Wiele bowiem obiektów (np. matematycznych) jest właśnie ze swej natury zdefiniowana rekurencyjnie. W poniższych paragrafach rozważymy kilka przykładów takich zagadnień.

5.3.1 Przykład: silnia

W poniższej definicji silni pojawia się odniesienie do... silni. Zauważmy jednak, że obiekt ten jest dobrze określony, gdyż podany został warunek stopu.

$$\begin{cases} 0! &= 1, \\ n! &= n\,(n-1)! \quad \text{dla } n > 0. \end{cases}$$

Funkcję służącą do obliczenia silni można napisać, bazując wprost na definicji.

```
int silnia(int n)
{
    assert(n >= 0); // upewnijmy sie...
    if (n == 0) return 1;
    else return n*silnia(n-1);
}
```

Ważne

Każde wywołanie rekurencyjne funkcji powoduje utworzenie nowego zestawu zmiennych lokalnych.

Dla porównania przyjrzyjmy się, jakby mogła wyglądać funkcja analogiczna, lecz niekorzystająca z dobrodziejstw rekurencji.

```
int silnia2(int n)
{
    assert(n >= 0);
    int w = 1;
    for (int i=1; i<=n; ++i)
        w *= i;
    return w;
}
```

To którą wersję uważamy za czytelniejszą, zależy oczywiście od nas samych.

5.3.2 Przykład: NWD

Poznaliśmy już algorytm wyznaczania największego wspólnego dzielnika dwóch liczb naturalnych. Okazuje się, że NWD można także określić poniższym równaniem rekurencyjnym. Niech $1 \leqslant n \leqslant m$.

$$\mathsf{NWD}(n,m) = \begin{cases} m & \text{dla } n = 0, \\ \mathsf{NWD}(m \bmod n, n) & \text{dla } n > 0. \end{cases}$$

Definicja ta przekłada się bezpośrednio na następujący kod w języku **C++**.

```
int nwd(int n, int m)
{
    assert(n >= m && n >= 0);
    if (n == 0) return m;
    else return nwd(m % n, n);
}
```

5.3.3 Przykład: wieże z Hanoi

A teraz zabawna łamigłówka. Załóżmy, że danych jest n **krążków** o różnych średnicach oraz trzy **słupki**. Początkowo wszystkie krążki znajdują się na słupku nr 1, w kolejności od największego (dół) do najmniejszego (góra). Sytuację wyjściową dla $n = 4$ przedstawia rys. 5.5.

Rysunek 5.5. Sytuacja wyjściowa dla 4 krążków w problemie wież z Hanoi.

Cel: przeniesienie wszystkich krążków na słupek nr 3.

Zasada #1: krążki przenosimy pojedynczo.

Zasada #2: krążek o większej średnicy nie może znaleźć się na krążku o mniejszej średnicy.

Zastanówmy się, jak by mógł wyglądać algorytm służący do rozwiązywania tego problemu. Niech będzie to funkcja `Hanoi(k, A, B, C)`, która przekłada k krążków ze słupka A na słupek C z wykorzystaniem słupka pomocniczego B, gdzie $A, B, C \in \{1, 2, 3\}$. Wykonywany ruch będzie wypisywany na ekranie.

Aby przenieść n klocków ze słupka A na C, należy przestawić $n - 1$ mniejszych elementów na słupek pomocniczy B, przenieść n-ty krążek na C i potem pozostałe $n - 1$ krążków przestawić z B na C. Jest to, rzecz jasna, sformułowanie zawierające elementy rekurencji.

Wywołanie początkowe: `Hanoi(n, 1, 2, 3).`

Rozwiązanie w języku **C++**:

```
void Hanoi(int k, int A, int B, int C)
{
    if (k>0) // warunek stopu
    {
        Hanoi(k-1, A, C, B);
        cout << A << "->" << "C" << endl;
        Hanoi(k-1, B, A, C);
    }
}
```

Ciekawostka

Interesującym zagadnieniem jest wyznaczenie liczby przestawień potrzebnych do rozwiązania łamigłówki. W zaproponowanym algorytmie jest to:

$$\begin{cases} L(1) &= 1, \\ L(n) &= L(n-1)+1+L(n-1) \text{ dla } n>0. \end{cases}$$

Można pokazać, że jest to optymalna liczba przestawień.

Rozwiążmy powyższe równanie rekurencyjne, aby uzyskać postać jawną rozwiązania:

$$\begin{aligned} L(n) &= 2L(n-1)+1, \\ L(n)+1 &= 2(L(n-1)+1). \end{aligned}$$

Zauważmy, że $L(n)+1$ tworzy ciąg geometryczny o ilorazie 2. Zatem

$$\begin{aligned} L(1)+1 &= 2, \\ L(2)+1 &= 4, \\ L(3)+1 &= 8, \\ &\vdots \\ L(n)+1 &= 2^n. \end{aligned}$$

Więc

$$L(n) = 2^n - 1.$$

Zagadka Wież z Hanoi stała się znana w XIX wieku dzięki matematykowi E. Lucasowi. Jak głosi tybetańska legenda, mnisi w świątyni Brahmy rozwiązują tę łamigłówkę, przesuwając 64 złote krążki. Gdy skończą oni swe zmagania, ma nastąpić koniec świata. Zakładając jednak, że nawet gdyby wykonanie jednego ruchu zajmowało tylko 1 sekundę, to na pełne rozwiązanie i tak potrzeba $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$

sekund, czyli około $584\,542$ miliardów lat. To ponad 400 razy dłużej niż szacowany wiek Wszechświata.

5.3.4 Przykład (niemądry): liczby Fibonacciego

Jako ostatni problem rozpatrzmy równanie, do którego doszedł Leonard z Pizy (1202), rozważając rozmnażanie się królików.

Sformułował on następujący uproszczony model szacowania liczby par w populacji. Na początku mamy daną jedną parę królików. Każda para osiąga płodność po upłynięciu jednej jednostki czasu od narodzenia. Z każdej płodnej pary rodzi się w kolejnej jednostce jedna para potomstwa (to tylko matematyczny model, nie wnikajmy zbytnio w szczegóły).

Liczbę par królików w populacji w chwili n można opisać za pomocą tzw. ciągu Fibonacciego.

$$\begin{cases} F_0 &= 1, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \text{ dla } n > 1. \end{cases}$$

W wyniku otrzymujemy zatem ciąg $(1, 1, 2, 3, 5, 8, 13, 21, 34, \dots)$.

Bezpośrednie przełożenie powyższego równania na kod w **C++** może wyglądać jak niżej.

```
1 int fibrek(int n)
2 {
3    if (n > 1) return fibrek(n-1)+fibrek(n-2);
4    else return 1;
5 }
```

Powyższy algorytm rekurencyjny jest jednak bardzo nieefektywny. Przypatrzmy się, jak przebiegają obliczenia dla wywołania `fibrek(5)`:

```
fibrek(5);
fibrek(4);
fibrek(3);
fibrek(2);
fibrek(1);
fibrek(0);
fibrek(1);
fibrek(2);
fibrek(1);
fibrek(0);
fibrek(3);
fibrek(2);
```

```
fibrek(1);
fibrek(0);
fibrek(1);
```

Większość wartości jest niepotrzebnie liczona wielokrotnie.

Rozważmy, ile potrzebnych jest operacji arytmetycznych (dodawanie) do znalezienia wartości F_n za pomocą powyższej funkcji. Liczbę tę można opisać równaniem:

$$\begin{cases} L(0) &= 0, \\ L(1) &= 0, \\ L(n) &= L(n-1) + L(n-2) + 1 \text{ dla } n > 1. \end{cases}$$

Rozpatrując kilka kolejnych wyrazów, zauważamy, że $L(n) = F_{n-1} - 1$ dla $n > 0$.

Okazuje się, że dla dużych n, $L(n)$ jest proporcjonalne do c^n dla pewnej stałej c. Zatem liczba kroków potrzebnych do uzyskania n-tej liczby Fibonacciego algorytmem rekurencyjnym rośnie wykładniczo, tj. bardzo (naprawdę bardzo) szybko. Czas potrzebny do policzenia F_{40} na starym komputerze autora tych refleksji to $1{,}4$ s, dla F_{45} to już $15{,}1$ s, a dla F_{50} to aż 2 minuty i 47 s.

Tym razem okazuje się, że rozwiązanie rekurencyjne jest nieefektywne. Zatem pozostaje jedynie użycie wersji iteracyjnej, które wykorzystuje tylko co najwyżej n operacji arytmetycznych.

Przykładowy pełny program wyznaczający n-tą liczbę Fibonacciego (dla danego z klawiatury n) mógłby wyglądać jak niżej. Zwróćmy uwagę na podział programu na czytelne moduły, z których każdy służy do przeprowadzenia ściśle określonej czynności.

```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  int pobierz_n()
6  {
7      int n;
8      cout << "Podaj n: ";
9      cin >> n;
10     if (n < 0) {
11         cout << "Niepoprawne dane!" << endl;
12         exit(1); // zakoncz natychmiast dzialanie programu
13     }
14
15     return n; // tutaj na pewno n >= 0
16 }
17
```

```
18 int fibiter(int n)
19 {
20     if (n <= 1) return 1; // tu wynik jest znany
21
22     int k = 1;
23     int Fk_poprz = 1;   // F_{k-1}
24     int Fk_teraz = 1;   // F_k
25
26     do {
27         int nast = Fk_teraz+Fk_poprz; // kolejny wyraz
28         k++;
29
30         Fk_poprz = Fk_teraz;   // F_{k-1} (juz dla nowego k)
31         Fk_teraz = nast;       // F_k
32     }
33     while (k <= n);
34
35     return Fk_teraz;
36 }
37
38
39 int main()
40 {
41     int n = pobierz_n();
42     cout << "F(" << n << ") = " << fibiter(n) << endl;
43     return 0;
44 }
```

5.4 Ćwiczenia

Zadania do samodzielnego rozwiązania

Zadanie 5.92. Napisz funkcję parzysta(), która sprawdza, czy dany argument typu `int` jest liczbą parzystą czy nieparzystą. Zwróć wynik typu `bool`.

Zadanie 5.93. Napisz funkcję silnia(), która dla danego całkowitego $n \geqslant 1$ zwraca wartość równą $1 \times 2 \times \cdots \times n$, dla $n = 0$ wartość 1, a dla $n < 0$ wartość 0.

Zadanie 5.94. Napisz funkcję max(), która zwraca maksimum danych argumentów a, b, c typu `int`.

Zadanie 5.95. Napisz funkcję med(), która znajduje medianę (wartość środkową) trzech liczb rzeczywistych, np. $\mathrm{med}(4, 2, 7) = 4$ i $\mathrm{med}(1, 2, 3) = 2$.

Napisz funkcję nwd(), zwracającą największy wspólny dzielnik dwóch liczb naturalnych albo wartość 0, jeśli chociaż jedna z podanych liczb jest mniejsza od 1.

Zadanie 5.97. Napisz funkcję nww() zwracającą najmniejszą wspólną wielokrotność dwóch liczb naturalnych albo wartość 0, jeśli chociaż jedna z podanych liczb jest mniejsza od 1.

Zadanie 5.98. Napisz funkcję o nazwie bmi(), która jako argument przyjmuje wzrost (w m) i wagę (w kg) pacjenta, a jako wynik zwraca jego wskaźnik masy ciała (BMI), określony jako BMI $=$ waga$/$wzrost^2.

Ciekawostka. Według WHO BMI od $18{,}5$ do $25{,}0$ jest uznawana za wartość prawidłową dla „typowego" (cokolwiek to znaczy) człowieka.

Zadanie 5.99. Napisz funkcję odle(), która przyjmuje współrzędne rzeczywiste dwóch punktów `x1`, `y1`, `x2`, `y2` i zwraca ich odległość euklidesową, która dana jest wzorem $\sqrt{(x1-y1)^2+(x2-y2)^2}$.

Wskazówka. Skorzystaj z funkcji `sqrt()` z biblioteki `<cmath>`.

Zadanie 5.100. Napisz funkcję odlsup(), która przyjmuje współrzędne dwóch punktów `x1,y1,x2,y2` i zwraca ich odległość w metryce supremum, tj. $\max\{|x1-y1|, |x2-y2|\}$.

Zadanie 5.101. Napisz funkcję odlLp(), która przyjmuje współrzędne rzeczywiste dwóch punktów `x1,y1,x2,y2` i zwraca ich odległość w metryce L^p, gdzie $p \in [1,\infty)$ jest także parametrem funkcji, wg wzoru $\sqrt[p]{|x1-y1|^p+|x2-y2|^p}$.

Wskazówka. Skorzystaj z funkcji `pow()` z biblioteki `<cmath>`.

Zadanie 5.102. Napisz funkcję zaokr, która dla liczby $x \in \mathbb{R}$ wyznacza jej zaokrąglenie dziesiętne — z dokładnością do podanej liczby cyfr dziesiętnych k — jako $\lfloor x\ 10^k + 0{,}5\rfloor/10^k$, gdzie $\lfloor u\rfloor$ jest funkcją „podłoga".

Zadanie 5.103. Napisz kilka przeciążonych wersji funkcji swap(), które przestawiają wartości dwóch argumentów wejściowych o typach **`int`**, **`double`** i **`bool`**.

Zadanie 5.104. Napisz nierekurencyjną funkcję wyznaczającą n-tą liczbę Fibonacciego, F_n. Stwórz także jej wersję rekurencyjną i zmierz stoperem, ile czasu potrzebuje każda z nich, aby policzyć F_n dla różnych n. Narysuj na kartce wykresy czasu działania (w sek.) w zależności od n.

Zadanie 5.105. Napisz funkcję swap(), która przy użyciu ciągu przypisań przestawi wartości czterech zmiennych rzeczywistych (a,b,c,d) tak, by na wyjściu otrzymać (c,b,d,a).

Zadanie 5.106. Napisz funkcję sort(), która porządkuje niemalejąco wartości trzech argumentów wejściowych (liczby całkowite).

5.5 Laboratoria

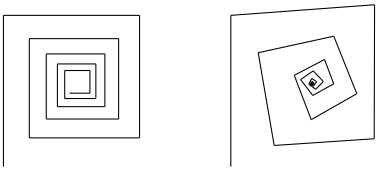
Zobacz: https://raw.githubusercontent.com/gagolews/aipp/master/zrodla/Logo.cpp

Zadanie 5.107. Napisz funkcję `void kwadrat(double x);`, która narysuje na planszy kwadrat o boku x (poczynając od aktualnej pozycji, w której znajduje się Żółw).

Zadanie 5.108. Napisz funkcję `void nkat(int n, double x);`, która narysuje na planszy n-kąt foremny o boku x.

(*) Zadanie 5.109. Napisz funkcję `void okrag(double r);`, która narysuje na planszy okrąg o promieniu r o środku w punkcie, w którym znajduje się Żółw. Pamiętaj o powrocie Żółwia do punktu wyjścia po narysowaniu figury.

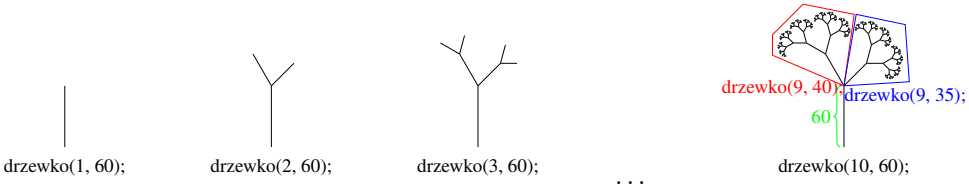
Zadanie 5.110. Napisz dwie wersje funkcji służącej do rysowania spirali w Logo — jedną korzystającą z pętli `for`, a drugą rekurencyjną. Deklaracja funkcji powinna wyglądać następująco: `void spirala(int n, double x);` gdzie n — liczba odcinków, z których złożona jest spirala oraz x — długość najdłuższego odcinka. Spróbuj poeksperymentować z losowymi obrotami Żółwia i skracaniem kolejnych odcinków spirali o losowe długości.



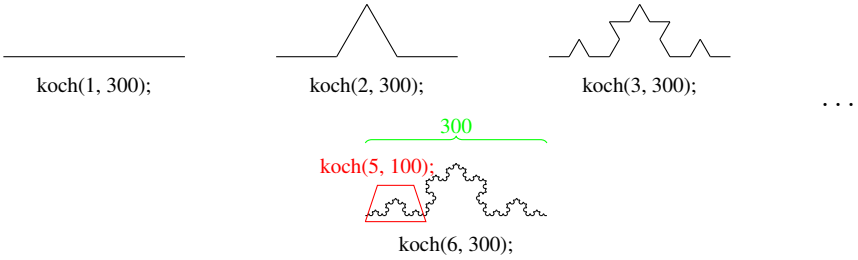
(*) Zadanie 5.111. Napisz rekurencyjną funkcję, rysującą fraktalne drzewko. Jej deklaracja może wyglądać na przykład tak:

```
void drzewko(int n, double x);
```

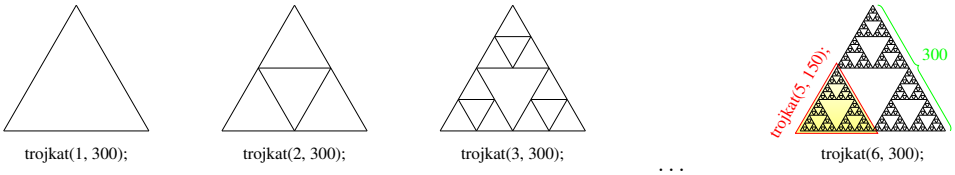
gdzie n — liczba poziomów poddrzewek do narysowania, a x — długość pnia aktualnie rysowanego poddrzewka. Poeksperymentuj z losowymi długościami pni poddrzewek i losowymi obrotami Żółwia przy rysowaniu zarówno lewego, jak i prawego poddrzewka.



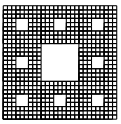
(*) Zadanie 5.112. Napisz funkcję `void koch(int n, double x);`, rysującą krzywą Kocha zgodnie z podanym niżej schematem.



(*) Zadanie 5.113. Napisz rekurencyjną funkcję `void trojkat(int n, double x);`, rysującą trójkąt Sierpińskiego zgodnie z podanym niżej schematem.



(*) Zadanie 5.114. Napisz rekurencyjną funkcję `void dywan(int n, double x);`, rysującą dywan Sierpińskiego zgodnie ze schematem podanym poniżej.



Dynamiczna alokacja pamięci. Napisy (łańcuchy znaków)

6

6.1 Dynamiczna alokacja pamięci

6.1.1 Organizacja pamięci komputera

W drugiej części skryptu dowiedzieliśmy się, iż w **pamięci operacyjnej** komputera przechowywane są zarówno programy jak i dane. Podstawową jednostką pamięci jest **komórka** o rozmiarze jednego bajta. Każda komórka pamięci posiada swój **adres**. Reprezentowany jest on we współczesnych komputerach jako 32- lub 64-bitowa liczba całkowita (bez znaku).

Z punktu widzenia każdego programu można wyróżnić następujący **podział puli adresowej** pamięci (w tzw. architekturze von Neumanna):

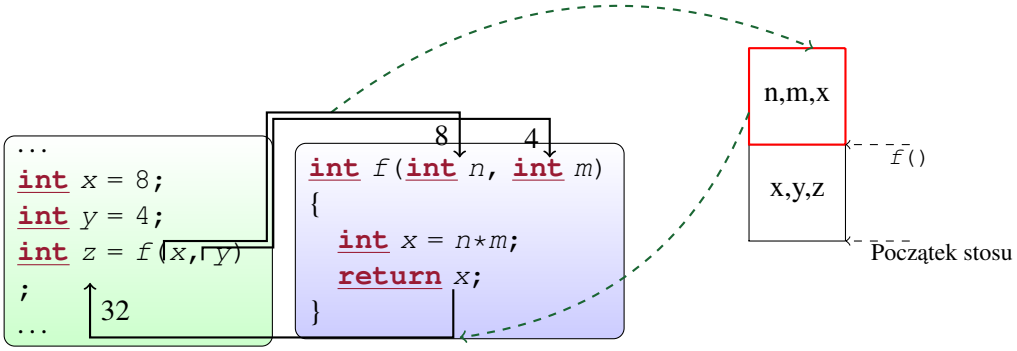
- kod programu — dane interpretowane są jako instrukcje procesora,
- stos (ang. *stack*) — przechowywane są wartości zmiennych lokalnych funkcji,
- sterta (ang. *heap*) — znajdują się dane dynamicznie przydzielane na prośbę programu (zob. dalej),
- część niedostępna — zarządzana przez system operacyjny (m.in. dane innych programów).

Zatem każdy program przechowuje informacje potrzebne do wykonywania swych czynności na **stosie** i **stercie**.

Stos jest częścią pamięci operacyjnej, na której dane są umieszczane i kasowane w porządku „ostatni na wejściu, pierwszy na wyjściu” (LIFO, ang. *last-in-first-out*). Umieszczanie i kasowanie danych na stosie odbywa się automatycznie. Każda wywoływana funkcja tworzy na stosie miejsce dla swoich zmiennych lokalnych. Gdy funkcja kończy działanie, usuwa z niego dane (to dlatego zmienne lokalne przestają wtedy istnieć).

Przyjrzyjmy się rozszerzonej wersji ilustracji z poprzedniej części skryptu (rys. 6.1). Po lewej stronie widzimy fragment funkcji `main()`, w której zostały zadeklarowane zmienne `x, y, z`. Umieszczone są one na „dole” stosu (jako pierwsze w programie).

Gdy funkcja ta wywołuje `f()`, na stosie tworzone jest miejsce dla zmiennych `n, m, x`. Gdy `f()` kończy swe działanie, `n, m, x` są ze stosu usuwane.



Rysunek 6.1. Zasięg zmiennych.

6.1.2 Wskaźniki

Każda zmienna ma przyporządkowaną komórkę (lub komórki) pamięci, w której przechowuje swoje dane, np. zmienna typu `int` zajmuje najczęściej 4 takie komórki (4 bajty). Fizyczny adres zmiennej (czyli numer komórki) można odczytać za pomocą operatora `&`.

```
int x;
cout << "x znajduje sie pod adresem " << &x;
// np. 0xe3d30dbc
```

Przypomnijmy, że `0xe3d30dbc` oznacza liczbę w systemie szesnastkowym.

Każda zmienna ma zatem swoje „miejsce na mapie” (tzn. w pamięci komputera), znajdujące się pod pewnym adresem (np. na pl. Politechniki 1). Operator `&` pozwala więc uzyskać informację o pozycji danej zmiennej. Jeszcze inaczej: zmienna to „budynek magazynu” w którym można przechowywać towar określonego rodzaju, np. cukierki. Adres zmiennej to „współrzędne GPS” tegoż magazynu.

Specjalny typ danych do przechowywania informacji o adresach innych zmiennych („współrzędnych GPS”) określonego typu zwany jest **typem wskaźnikowym**. Określa się go za pomocą dodania symbolu $\star$ (gwiazdka) bezpośrednio po nazwie typu.

Na przykład, zadeklarowanie zmiennej typu `int`$\star$ oznacza stworzenie wskaźnika na zmienną typu `int`, czyli zmiennej przechowującej fizyczny adres w pamięci komputera pewnej liczby całkowitej (współrzędne GPS pewnego magazynu do przechowywania cukierków).

Ważne
Istnieje specjalne miejsce w pamięci o adresie 0 (*NULL*, „czarna dziura”), do którego odwołanie się podczas działania programu powoduje wystąpienie błędu. Często używa się tego adresu np. do zainicjowania wskaźników w celu oznaczenia, że początkowo nie wskazują one „nigdzie”.

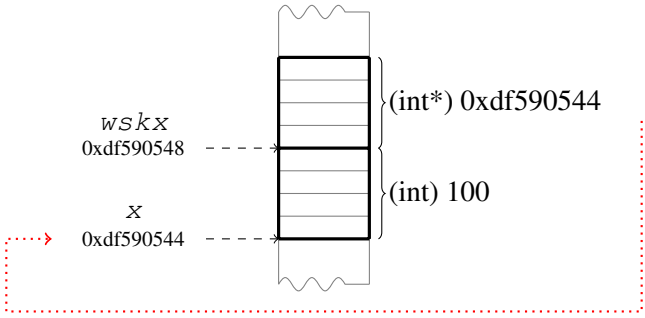
Na zmiennych wskaźnikowych został określony tzw. **operator wyłuskania**, ⋆ (nie mylić z „gwiazdką” modyfikującą znaczenie typu!), dzięki któremu możemy odczytać, co się znajduje pod danym adresem pamięci (co znajduje się w jakimś magazynie, którego znamy tylko współrzędne GPS).

Przyjrzyjmy się poniższemu przykładowi. Tworzone są dwie zmienne, jedna typu całkowitego, a druga wskaźnikowa. Ich rozmieszczenie w pamięci (na stosie, są to bowiem zmienne lokalne jakiejś funkcji) przedstawia rys. 6.2. Każda z tych zmiennych umieszczona jest pod jakimś adresem w pamięci RAM — można go odczytać za pomocą operatora `&`.

Listing 6.1. „Wyłuskanie” danych spod danego adresu.

```
1 int  x = 100;
2 int* wskx = &x;
3
4 cout <<  wskx << endl; // np. 0xdf590544
5 cout << *wskx << endl; // 100
6 cout <<     x << endl; // 100
```

Wypisanie wartości wskaźnika oznacza wypisanie adresu, na który wskazuje. Wypisanie zaś „wyłuskanego” wskaźnika powoduje wydrukowanie wartości komórki pamięci, na którą pokazuje wskaźnik. Jako że zmienna typu `int` ma rozmiar 4 bajtów, adres następnej zmiennej (*wskx*) jest o 4 jednostki większy od adresu *x*.



Rysunek 6.2. Zawartość pamięci w programie 6.1.

Aby jeszcze lepiej zrozumieć omawiane zagadnienie, rozważmy fragment kolejnego programu.

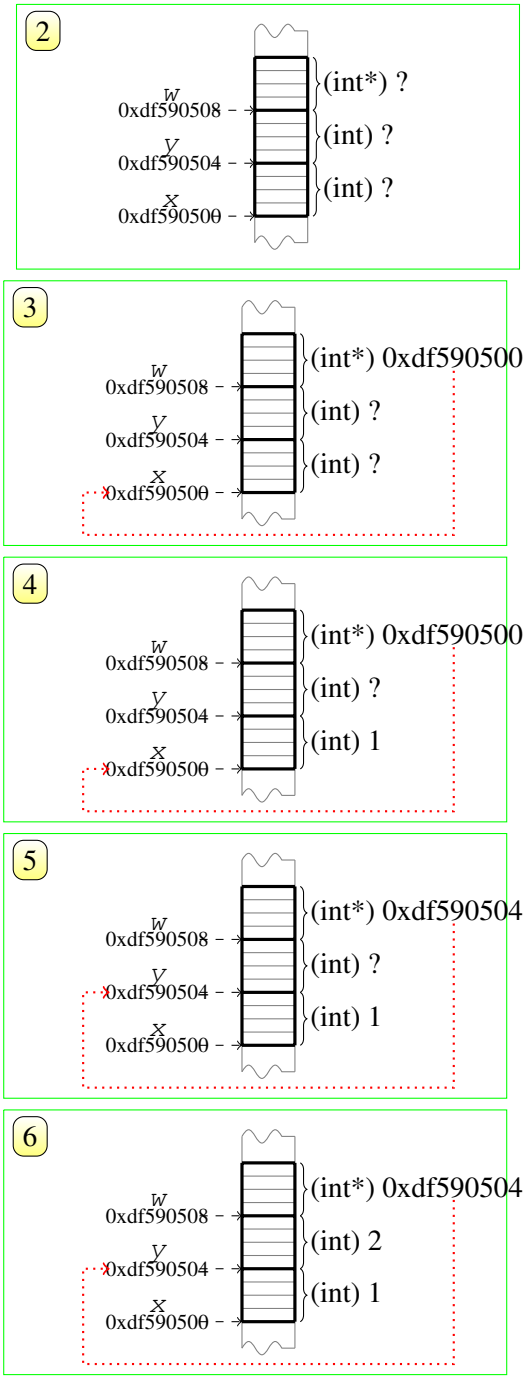
Listing 6.2. Proste operacje z użyciem wskaźników.

```
1 int  x, y;
2 int* w;
3 w  = &x;
4 *w = 1;
5 w  = &y;
6 *w = 2;
```

Zawartość pamięci po wykonaniu kolejnych linii kodu przedstawia rys. 6.3. Tym razem za pomocą operatora wyłuskania zapisujemy dane do komórek pamięci, na które pokazuje wskaźnik `w`.

Zadanie

Prześledź pokazane rysunki bardzo uważnie! Wskaźniki są niezmiernie istotnym elementem języka **C++**!



Rysunek 6.3. Zawartość pamięci po wykonaniu kolejnych linii kodu z przykładu 6.2.

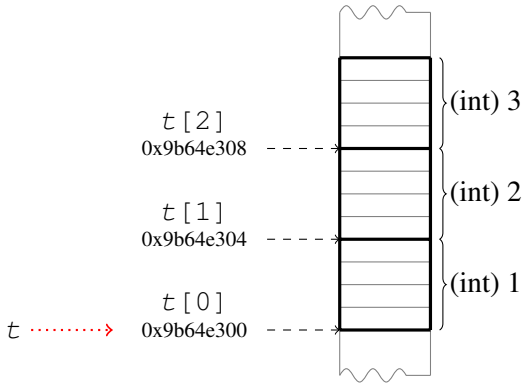
6.1.3 Tablice a wskaźniki

Zapewne ciekawi nas, w jaki sposób zorganizowana jest pamięć na stosie, gdy deklarowane są tablice jednowymiarowe. Możemy to sprawdzić w następujący sposób.

Listing 6.3. Organizacja pamięci dla tablic.

```
1 int t[3] = {1,2,3};
2 cout << &t[0];     // np. 0x9b64e300
3 cout << &t[1];     // np. 0x9b64e304
4 cout << &t[2];     // np. 0x9b64e308
```

Kolejne elementy tablicy w pamięci zawsze następują po sobie (tablica tworzy spójny ciąg bajtów), zob. rys. 6.4.



Rysunek 6.4. Organizacja pamięci w przypadku tablic w przykładzie 6.3.

Ponadto zwróćmy uwagę na to, co się dzieje, gdy wykonamy następujący kod.

```
1 cout << t;      //      0x9b64e300 - wskazuje tu
2 cout << &t[0]; //      0x9b64e300 - tu pierwszy element
```

Okazuje się, że zmienną tablicową (dotyczy to tablic o stałym rozmiarze) można traktować jako wskaźnik (ale nie odwrotnie).

Ważne

Typ `int[3]` (szczegółowy) jest sprowadzalny do typu `int*` (uniwersalny). Synonimem `int*` jest `int[]`, który oznacza „jakaś tablica", „wskaźnik na jakiś ciąg (być może jednoelementowy) danych typu `int`". Dlatego też zapis `*t` i `t[0]` jest równoważny. Co więcej, `*(t+k)` znaczy to samo, co `t[k]`.

Z powyższych uwag wynika, że tablicę (dowolnego) rozmiaru można przekazać funkcji właśnie za pomocą wskaźnika. Koniecznie jednak trzeba pamiętać, aby także

dostarczyć funkcji **rozmiar tablicy**, bowiem wskaźnik to tylko adres pierwszego elementu.

Przykład: funkcja wyznaczająca sumę elementów tablicy.

```
double suma(double* t, int n) // albo ,,double[] t''
{
    double s = 0.0;
    for (int i=0; i<n; ++i)
        s += t[i];
    return s;
}

int main(void)
{
    double punkty[4] = { 10.0, 11.0, 12.0, 9.5 } ;
    cout << suma(punkty, 4); // przekazanie tablicy do
        funkcji
    return 0;
}
```

Ciekawostka

Na marginesie, powyższa funkcja może być zapisana również w dwóch następujących postaciach. Nie jest to jednak zalecany sposób pisania kodu, gdyż trudno zrozumieć intencję jego autora.

```
double suma(double* t, int n)
{
    double s = 0.0;
    for (int i=0; i<n; ++i)
        s += *(t+i); // ROBI SIE GORACO!!! :)
    return s;
}
```

```
double suma(double* t, int n)
{
    double s = 0.0;
    for (int i=0; i<n; ++i)
        s += *(t++); // OLABOGA!!!
    return s;
}
```

6.1.4 Przydział i zwalnianie pamięci ze sterty

Oprócz ściśle określonej na etapie pisania programu ilości danych na stosie, można również dysponować pamięcią na **stercie**. Część tej pamięci jest przydzielana (alokowana) **dynamicznie** podczas działania programu za pomocą operatora `new`. Po użyciu należy ją zwolnić za pomocą operatora `delete`.

> **Ważne**
> Zaalokowany obiekt będzie istniał w pamięci nawet po wyjściu z funkcji, w której go stworzyliśmy! Dlatego należy pamiętać, aby go usunąć w pewnym miejscu kodu.

Oto składnia instrukcji służących do alokacji i dealokacji pamięci na jeden obiekt.

```
typ* obiekt = new typ; // przydzial (zwraca wskaznik)
// ...
delete obiekt; // zwolnienie
```

Tworzonemu pojedynczemu obiektowi można od razu przypisać wartość, np.

```
int* n = new int(7);
```

Można również przydzielić pamięć na wiele obiektów następujących po sobie, czyli na tablicę.

```
int n = 4; // tutaj juz nie musi byc stala
typ* obiekt = new typ[n]; // przydzial (zwraca wskaznik)
// ...
delete [] obiekt; // zwolnienie
```

Elementom tak tworzonej tablicy **nie można niestety przypisać od razu wartości**. Trzeba w tym celu skorzystać np. z operatora przypisania.

Oto fragment kodu, który napisał tata Jasia celem zmotywowania go do nauki.

```
1 int n;
2 double* godzinyNauki;
3
4 cout << "Ile dni uczyles sie do kolokwium?";
5 cin >> n;
6
7 godzinyNauki = new double[n]; // utworz tablice o n
      elementach
8
9 cout << "Ile godzin uczyles sie kazdego dnia?";
```

```
10 for (int i=0; i<n; i++)
11     cin >> godzinyNauki[i];
12
13 //....
14 cout << "I tak za malo :P"; // zly pan
15
16 delete [] godzinyNauki; // tablica juz nie jest dalej
       potrzebna
```

Zauważmy, że w przeciwieństwie do tablic z poprzedniego wykładu tym razem n nie jest stałą. Rozmiar tablicy zostaje ustalony na etapie działania programu. Pobierany jest on z klawiatury, uzależniając go od życzenia jego użytkownika.

6.2 Napisy (łańcuchy znaków)

Do tej pory nie zastanawialiśmy się wspólnie, w jaki sposób można reprezentować w naszych programach napisy. Często są one nam potrzebne, np. gdy chcemy zakomunikować coś ważnego użytkownikowi, czy też przechować lub przetworzyć informacje o nienumerycznym charakterze.

6.2.1 Kod ASCII

Pojedyncze znaki drukowane przechowywane są najczęściej jako typ **char** (ang. *character*). Oczywiście pamiętamy, że jest to po prostu typ, służący do przechowywania 8 bitowych **liczb** (sic!) całkowitych.

Istnieje ogólnie przyjęta umowa (standard), że liczbom z zakresu 0–127 *odpowiadają* ściśle określone znaki, tzw. kod ASCII (ang. *American Standard Code for Information Interchange*). Zestawiają je tablice 6.1–6.4.

Szczęśliwie w języku **C++** nie musimy pamiętać, która liczba odpowiada jakiemu symbolowi. Aby uzyskać wartość liczbową symbolu drukowanego, należy ująć go w **pojedyncze cudzysłowy**.

```
char c1 = 'A';
char c2 = '\n'; // znak nowej linii

cout << c1 << c2; //  "A" i przejscie do nowej linii
cout << (int)c1 << (char)59 << (int)c2; //  "65;13"
```

Jak widzimy, domyślnie wypisanie na ekran zmiennej typu **char** jest równoważne z wydrukowaniem symbolu. Można to zachowanie zmienić, rzutując ją na typ **int**.

Ponadto, przyglądając się uważniej tablicy ASCII, warto odnotować następujące prawidłowości.

Tabela 6.1. Kod ASCII cz. I — znaki kontrolne

DEC	HEX	Znaczenie	DEC	HEX	Znak
0	00	Null (`\0`)	16	10	Data Link Escape
1	01	Start Of Heading	17	11	Device Control 1
2	02	Start of Text	18	12	Device Control 2
3	03	End of Text	19	13	Device Control 3
4	04	End of Transmission	20	14	Device Control 4
5	05	Enquiry	21	15	Negative Acknowledge
6	06	Acknowledge	22	16	Synchronous Idle
7	07	Bell (`\a`)	23	17	End of Transmission Block
8	08	Backspace (`\b`)	24	18	Cancel
9	09	Horizontal Tab	25	19	End of Medium
10	0A	Line Feed (`\r`)	26	1A	Substitute
11	0B	Vertical Tab (`\t`)	27	1B	Escape
12	0C	Form Feed	28	1C	File Separator
13	0D	Carriage Return (`\n`)	29	1D	Group Separator
14	0E	Shift Out	30	1E	Record Separator
15	0F	Shift In	31	1F	Unit Separator

Tabela 6.2. Kod ASCII cz. II

DEC	HEX	Znaczenie	DEC	HEX	Znak
32	20	Spacja	48	30	0
33	21	!	49	31	1
34	22	"	50	32	2
35	23	#	51	33	3
36	24	$	52	34	4
37	25	%	53	35	5
38	26	&	54	36	6
39	27	’	55	37	7
40	28	(	56	38	8
41	29	)	57	39	9
42	2A	*	58	3A	:
43	2B	+	59	3B	;
44	2C	,	60	3C	<
45	2D	-	61	3D	=
46	2E	.	62	3E	>
47	2F	/	63	3F	?

Tabela 6.3. Kod ASCII cz. III

DEC	HEX	Znaczenie	DEC	HEX	Znak
64	40	@	80	50	P
65	41	A	81	51	Q
66	42	B	82	52	R
67	43	C	83	53	S
68	44	D	84	54	T
69	45	E	85	55	U
70	46	F	86	56	V
71	47	G	87	57	W
72	48	H	88	58	X
73	49	I	89	59	Y
74	4A	J	90	5A	Z
75	4B	K	91	5B	[
76	4C	L	92	5C	\
77	4D	M	93	5D	]
78	4E	N	94	5E	^
79	4F	O	95	5F	_

Tabela 6.4. Kod ASCII cz. IV

DEC	HEX	Znaczenie	DEC	HEX	Znak
96	60	`	112	70	p
97	61	a	113	71	q
98	62	b	114	72	r
99	63	c	115	73	s
100	64	d	116	74	t
101	65	e	117	75	u
102	66	f	118	76	v
103	67	g	119	77	w
104	68	h	120	78	x
105	69	i	121	79	y
106	6A	j	122	7A	z
107	6B	k	123	7B	{
108	6C	l	124	7C	|
109	6D	m	125	7D	}
110	6E	n	126	7E	∼
111	6F	o	127	7F	Delete

- Mamy następujący porządek leksykograficzny: `'A' < 'B' < ... < 'Z' < 'a' < 'b' < ... < 'z'`.
- Symbol cyfry `c` $\in \{0, 1, \ldots, 9\}$ można uzyskać za pomocą wyrażenia `'0'+c`.
- Kod ASCII `n`-tej wielkiej litery alfabetu łacińskiego to `'A'+n-1`.
- Kod ASCII `n`-tej małej litery alfabetu łacińskiego to `'a'+n-1`.
- Zamiana litery `l` na małą literę następuje za pomocą operacji `l+32 == l+0x20`.
- Zamiana litery `l` na wielką literę następuje za pomocą operacji `l-32 == l-0x20`.

Pozostałe symbole odpowiadające wartościom liczbowym (0x80–0xFF) nie są określone przez standard ASCII. Zdefiniowane są one przez inne kodowania, np. CP-1250 (Windows) lub ISO-8859-2 (Internet, Linux) zawierają polskie znaki diakrytyczne. Jak widzimy, sprawa polskich „ogonków" jest nieco skomplikowana. Zatem `na poczatkowym etapie programowania uzywajmy tylko liter alfabetu lacinskiego w programach, ktore przetwarzaja napisy.`

Ciekawostka
Współcześnie stosuje się coraz powszechnie inne standardy kodowania, zwane Unicode (UTF-8, UTF-16, ...), w których jednemu znakowi niekoniecznie musi odpowiadać jeden bajt. Obsługa ich jednak jest nieco skomplikowana, zatem nie będziemy się nimi zajmować w tym wykładzie.

6.2.2 Reprezentacja napisów (łańcuchów znaków)

Wiemy już, w jaki sposób wyświetlać pojedyncze znaki. Pora na coś więcej.

Ważne
Najprostszym sposobem reprezentowania ciągów symboli drukowanych, czyli **napisów**, są tablice elementów typu `char` **zakończone** umownie bajtem (liczbą całkowitą) o wartości **zero** (znak `'\0'`).

Napisy można utworzyć używając cudzysłowów (`"..."`). Są to jednak tablice tylko do odczytu. Nie wiadomo bowiem, w jakim miejscu w pamięci zostaną one umieszczone.

```
char* napis1 = "Pewien napis."; // 13 znakow + bajt 0
// *prawie* rownowaznie:
char napis2[14] =
    { 'P', 'e', 'w', 'i', 'e', 'n', ' ',
        'n', 'a', 'p', 'i', 's', '.', '\0' };

// Znaki w zmiennej napis1 sa tylko do odczytu!
```

```
napis1[1] = 'k'; // nie wiadomo, co sie stanie
napis2[1] = 'k'; // ok
```

6.2.3 Operacje na łańcuchach znaków

Jako że napisy są zwykłymi tablicami liczb całkowitych, implementacja podstawowych operacji na nich jest dość prosta. W niniejszym paragrafie rozważymy kilka z nich, resztę pozostawiając jako ćwiczenie.

Najpierw przyjrzymy się wypisywaniu.

```
char* napis = "Jakis napis."; // tablica znakow zakonczona
    zerem

// Zatem:
cout << napis;

// Jest rownowazne:
int i=0;
while (napis[i] != '\0') // dopoki nie koniec napisu
{
   cout << napis[i];
   ++i;
}
```

Ważne

Jeśli mamy zaalokowaną tablicę typu char o identyfikatorze `napis` mogącą przechowywać maksymalnie $n+1$ znaków (dodatkowe miejsce na bajt zerowy), to wywołanie

```
cin >> napis;
```

wczytuje tylko jeden wyraz (uwaga, by nie wprowadzić więcej niż n znaków!). Jeśli chcemy wczytać cały wiersz, musimy wywołać:

```
cin.getline(napis, n);
```

Długość napisu można sprawdzić w następujący sposób.

```
int dlug(char* napis)
{
   // Zwraca dlugosc napisu (bez bajtu zerowego). Zgodnie
```

```
    // z umowa, mimo ze jest to tablica, potrafimy jednak
    // sprawdzic, gdzie sie ona konczy

    int i=0;
    while (napis[i] != 0)
        ++i;

    return i;
}
```

Ostatnim przykładem będzie tworzenie dynamicznie alokowanej kopii napisu.

```
char* kopia(char* napis)
{
    int n = dlug(napis);
    char* nowy = new char[n+1]; // o jeden bajt wiecej

    // nie zapominamy o skopiowaniu bajtu zerowego:
    for (int i=0; i<n+1; ++i)
        nowy[i] = napis[i];

    return nowy; // dalej nie zapomnijmy o dealokacji
        pamieci
}
```

6.2.4 Biblioteka cstring

Biblioteka `<cstring>` definiuje wiele funkcji przetwarzających łańcuchy znaków [1]. Wybrane funkcje zestawia tab. 6.5.

6.3 Ćwiczenia

Zadanie 6.115. Napisz funkcję `char* napkopiuj(char* napis)`, która zwraca nowy, dynamicznie alokowany napis będący kopią napisu danego na wejściu.

Zadanie 6.116. Napisz funkcję `char* napzlacz(char* napis1, char* napis2)`, która zwraca nowy, dynamicznie alokowany napis będący złączeniem (konkatenacją) dwóch danych na wejściu napisów, np. dla `"ala"` i `"ola"` wynikiem powinno być `"alaola"`.

[1] Zobacz angielskojęzyczną dokumentację dostępną pod adresem http://www.cplusplus.com/reference/clibrary/cstring/.

Tabela 6.5. Wybrane funkcje biblioteki `<cstring>`.

Deklaracja	Opis
`int strlen(char* nap);`	Zwraca długość napisu.
`char* strcpy(char* cel, char* zrodlo);`	Kopiuje napisy. Uwaga: długość tablicy `cel` nie może być mniejsza niż długość napisu `zrodlo` +1.
`char* strcat(char* cel, char* zrodlo);`	Łączy napisy `cel` i `zrodlo`.
`int strcmp(char* nap1, char* nap2);`	Porównuje napisy. Zwraca 0, jeśli są identyczne. Zwraca wartość dodatnią, jeśli `nap1` jest większy (w porządku leksykograficznym) niż `nap2`.
`char* strchr(char* nap, char znak);`	Zwraca podnapis rozpoczynający się od pierwszego wystąpienia danego znaku.
`char* strrchr(char* nap, char znak);`	Zwraca podnapis rozpoczynający się od ostatniego wystąpienia danego znaku.
`char* strstr(char* nap1, char* nap2);`	Zwraca podnapis rozpoczynający się od pierwszego wystąpienia podnapisu `nap2` w napisie `nap1`.

Zadanie 6.117. Napisz funkcję, która tworzy i zwraca nową, dynamicznie stworzoną kopię danego napisu z zamienionymi wszystkimi małymi literami alfabetu łacińskiego (bez polskich znaków diakrytycznych) na wielkie.

Zadanie 6.118. Napisz funkcję, która usuwa wszystkie znaki odstępów (spacje) z końca danego napisu. Napis przekazany na wejściu powinien pozostać bez zmian: zwracany jest nowy, dynamicznie alokowany napis.

Zadanie 6.119. Napisz funkcję, która usuwa wszystkie znaki odstępów (spacje) z początku danego łańcucha znaków. Napis przekazany na wejściu powinien pozostać bez zmian, zwracany jest nowy, dynamicznie alokowany napis.

Zadanie 6.120. Napisz funkcję, która usuwa z danego napisu wszystkie znaki niebędące cyframi bądź kropką. Napis przekazany na wejściu powinien pozostać bez zmian: zwracany jest nowy, dynamicznie alokowany napis.

Zadanie 6.121. Napisz funkcję, która odwróci kolejność znaków w danym napisie. Na-

pis przekazany na wejściu powinien pozostać bez zmian: zwróć nowy napis.

Zadanie 6.122. Napisz funkcję, która oblicza, ile razy w danym napisie występuje dany znak.

Zadanie 6.123. Napisz funkcję, która oblicza, ile razy w danym napisie występuje dany inny napis, np. w `"ababbababa"` łańcuch `"aba"` występuje trzy razy.

Zadanie 6.124. Napisz funkcję, która dla danej nieujemnej liczby `int` zwraca dynamicznie alokowany łańcuch znaków, składający się z symboli 0 lub 1, przechowujący reprezentację argumentu wejściowego w systemie binarnym.

Zadanie 6.125. Napisz funkcję, która dla danego łańcucha znaków, składającego się z symboli 0 lub 1, reprezentującego pewną nieujemną liczbę w postaci binarnej, zwraca jej wartość w postaci zmiennej typu `int`.

Zadanie 6.126. Napisz funkcję, która dla danego łańcucha znaków, składającego się z symboli 0 lub 1, reprezentującego pewną liczbę w postaci binarnej, zwraca dynamicznie alokowany napis przechowujący jej reprezentację szesnastkową, np. dla `"10100001 "` powinniśmy otrzymać `"A1"`.

Zadanie 6.127. Palindrom to ciąg liter, które są takie same niezależnie od tego, czy czytamy je od przodu czy od tyłu, np. *kobyłamamałybok*, *możejutrotadamasamadatortujeżom*, *ikarłapałraki*. Napisz funkcję, która sprawdza, czy dany napis jest palindromem. Zwróć wartość typu `bool`.

Zadanie 6.128. Napisz funkcję `char* napscal(char* napis1, char* napis2)`, która przyjmuje na wejściu dwa napisy, których litery są posortowane (w porządku leksykograficznym). Funkcja zwraca nowy, dynamicznie alokowany napis będący scaleniem (z zachowaniem porządku) tych napisów, np. dla `"alz"` i `"lopxz"` wynikiem powinno być `"allopxzz"`.

Zadanie 6.129. Napisz funkcję `int napporownaj(char* napis1, char* napis2)`, która zwraca wartość równą 1, jeśli pierwszy napis występuje w porządku leksykograficznym przed napisem drugim, wartość -1, jeśli po napisie drugim albo 0 w przypadku, gdy napisy są identyczne. Na przykład, `napporownaj("ola", "ala")==-1` oraz `napporownaj("guziec", "guziec2")==1`.

Zadanie 6.130. Napisz funkcję `char* napsortlitery(char* napis)`, która zwraca nowy, dynamicznie alokowany napis składający się z posortowanych (dowolnym znanym Ci algorytmem) liter napisu wejściowego. Na przykład, `napsortlitery("abacdcd")` powinno dać w wyniku napis `"aabccdd"`.

(*) Zadanie 6.131. Napisz funkcję `void sortuj(char* napisy[], int n)`, która wykorzysta dowolny algorytm sortowania tablic oraz funkcję `napporownaj(...)` do leksykograficznego uporządkowania danej tablicy napisów.

Zadanie 6.132. Napisz funkcję `char losuj()`, która wykorzystuje funkcję `rand`

`()` z biblioteki `<cstdlib>`: do „wylosowania” wielkiej litery z alfabetu łacińskiego. Zakładamy, że funkcja *srand()* została wywołana np. w funkcji *main()*.

(*) Zadanie 6.133. Napisz funkcję **char*** *permutuj*(**char*** *napis*), która zwraca nowy, dynamicznie alokowany napis będący „losową” permutacją znaków występujących w napisie przekazanym jako argument wejściowy.

Zadanie 6.134. Zaimplementuj samodzielnie funkcje z biblioteki `<cstring>`: *strlen*(), *strcpy*(), *strcat*(), *strcmp*(), *strstr*(), *strchr*(), *strrchr*().

6.4 Laboratoria

Zadanie 6.135. Napisz program, który zawiera implementację funkcji z zadania 6.116 oraz implementację funkcji **bool** *czypusty*(**char*** *napis*), sprawdzającej, czy dany napis jest pusty (składa się tylko z tzw. białych znaków, czyli spacji i znaków nowej linii). Obydwie funkcje powinny się znaleźć w oddzielnej bibliotece (dodatkowy plik nagłówkowy i źródłowy).

W funkcji *main()* należy wczytywać kolejne napisy (w pętli) z klawiatury, póki użytkownik nie wprowadzi napisu pustego. Można założyć, że użytkownik nie wprowadzi na raz więcej niż 255 znaków (użyj tablicy o ustalonym rozmiarze do wczytywania danych). Po wprowadzeniu każdego napisu na ekranie powinien pojawiać się wynik złączenia wszystkich dotychczas wprowadzonych napisów.

Przykładowy wynik działania programu:

```
Wprowadz napis: ala
Wynik:          ala
Wprowadz napis: ma kota
Wynik:          alama kota
Wprowadz napis: <PUSTY>
Koniec.
```

Wskazówka. Użyj następującej funkcji *main()*:

```
1  int main() {
2     bool koniec = false;
3     char napis[256]; // wprowadzany napis
4     char* calosc = new char[1]; // aktualny caly "zlaczony"
          napis
5     calosc[0] = '\0';           // na razie pusty
6     while (!koniec) {
7        cout << "Wprowadz napis: ";
8        cin.getline(napis, 256); // wczytaj caly wiersz
9        if (czypusty(napis))
10          koniec = true;
```

```
        else {
            char* stary = calosc; // zapamietaj stary napis,
                by go nie zgubic
            calosc = napzlacz(calosc, napis);
            delete [] stary; // juz niepotrzebny
            cout << "Wynik: " << calosc << endl;
        }
    }
    delete [] calosc;
    return 0;
}
```

Zadanie 6.136. Napisz bibliotekę funkcji, która zawierać będzie implementację dwóch procedur. Pierwsza sprawdza, czy dany napis składa się tylko z małych i wielkich liter alfabetu łacińskiego. Druga zamienia wszystkie małe litery na wielkie i odwrotnie (funkcja powinna zwracać nowy, dynamicznie alokowany napis i pozostawiać napis podany na wejściu bez zmian).

W funkcji `main()` należy wczytywać kolejne napisy (w pętli) z klawiatury, póki użytkownik nie wprowadzi napisu składającego się ze znaków innych niż litery. Można założyć, że użytkownik nie wprowadzi na raz więcej niż 255 znaków (użyj tablicy o ustalonym rozmiarze do wczytywania danych). Po wprowadzeniu każdego napisu na ekranie powinien pojawiać się napis przekształcony za pomocą drugiej funkcji z biblioteki.

Przykładowy wynik działania programu:

```
Wprowadz napis: AlA
Wynik:          aLa
Wprowadz napis: ma kota
Napis niepoprawny.
```

Wskazówka. Użyj następującej funkcji `main()`:

```
int main() {
    bool koniec = false;
    char napis[256]; // wprowadzany napis
    while (!koniec) {
        cout << "Wprowadz napis: ";
        cin.getline(napis, 256); // wczytaj caly wiersz
        if (czylacinskie(napis))
            koniec = true;
        else {
            char* rezultat = zamienWielkoscLiter(napis);
            cout << "Wynik: " << rezultat << endl;
            delete [] rezultat;
```

```
13          }
14      }
15      cout << "Napis niepoprawny." << endl;
16      return 0;
17  }
```

Zadanie 6.137. Zaimplementuj wszystkie funkcje z tego zestawu zadań w postaci własnej biblioteki (oddzielny plik źródłowy i nagłówkowy). W funkcji `main()` przetestuj ich działanie.

(*) Zadanie 6.138. Napisz program, który wczytuje z klawiatury kolejne (w pętli) proste wyrażenia arytmetyczne w formie napisu, oblicza ich wynik i wypisuje go na ekran. Wyrażenia są postaci `"xoy"`, gdzie x, y — liczby całkowite nieujemne, a o — operacja `+`, `-`, `/` lub `*`, np. `32+823`. Jeśli wprowadzone zostanie niepoprawne wyrażenie, należy natychmiast zakończyć działanie programu. Jeśli nastąpi próba dzielenia przez zero, program powinien wypisać komunikat: nastąpiła próba dzielenia przez zero.

(*) Zadanie 6.139. Napisz program wczytujący z klawiatury (jeden po drugim) numery telefonów w dowolnym formacie i wypisujący ten sam numer w formacie standardowym.

Format standardowy: np. +48600100200. Dozwolone formaty na wejściu: (+48) 600-100-200, 600 100 200, 600100200, +48600100200.

Program powinien w pętli wczytywać numer telefonu; sprawdzać, czy jest poprawny tzn. czy zawiera jedynie cyfry, nawiasy, myślniki, znaki + i odstępy (jeśli nie — należy zakończyć działanie programu); a następnie usunąć z niego wszystkie spacje, myślniki i nawiasy oraz ewentualnie dodać na początku numer kierunkowy do Polski, czyli +48 i wypisać wynik na ekran.

7 Macierze

7.1 Reprezentacja macierzy

Dana jest macierz określona nad pewnym zbiorem

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix}$$

gdzie n — liczba wierszy, m — liczba kolumn.

Macierze mogą być wygodnie reprezentowane w języku **C++** na dwa sposoby:
– jako tablice tablic,
– jako tablice jednowymiarowe.

Będziemy tutaj najczęściej stosować sposób pierwszy. Pomimo nieco zawiłego tworzenia i usuwania tego typu obiektów, zapewnia on bardzo wygodny dostęp do poszczególnych elementów.

```
1  int n = ...; // liczba wierszy
2  int m = ...; // liczba kolumn
3  typ** A; // tablica o elementach typu "typ*", czyli
       tablica tablic
4
5  A = new typ*[n];
6  for (int i=0; i<n; ++i)
7      A[i] = new typ[m];
8
9  // A to n-elementowa tablica tablic m-elementowych
10
```

```
// teraz np. A[0][3] to element w I wierszu i IV kolumnie
    ....

for (int i=0; i<n; ++i)
    delete [] A[i];
delete [] A;
```

Z drugiej strony, macierze możemy reprezentować za pomocą jednowymiarowych tablic. Łatwo się je tworzy, jednak odwoływanie się do elementów jest dość skomplikowane.

```
int n = ...; // liczba wierszy
int m = ...; // liczba kolumn
typ* A; // jednowymiarowa tablica

A = new typ[n*m];

// teraz np. A[1*m+3] to element w II wierszu i IV
    kolumnie ....

delete [] A;
```

7.2 Przykładowe algorytmy z wykorzystaniem macierzy

Omówimy teraz następujące przykładowe algorytmy:

a) mnożenie dowolnego wiersza przez stałą,
b) odejmowanie pewnego wiersza przez inny, pomnożony przez stałą,
c) zamiana dwóch kolumn,
d) wyszukiwanie wiersza, który ma największy co do modułu element w danej kolumnie,
e) dodawanie macierzy,
f) mnożenie macierzy,
g) rozwiązywanie układów równań liniowych metodą eliminacji Gaussa.

Będziemy rozpatrywać macierze o wartościach rzeczywistych (reprezentowanych jako tablice tablic typu **double**). Zauważ, że żadna z funkcji (o ile nie zaznaczono inaczej) nie „psuje” macierzy wejściowej, w większości przypadków zwracana jest nowa, dynamicznie alokowana macierz (oczywiście wtedy, gdy tego się spodziewamy). Takie postępowanie zależy od naszych potrzeb i specyfiki implementowanego programu, czasem być może prościej i wydajniej będzie z niego po prostu zrezygnować.

Ważne

Powyższe algorytmy będziemy implementować w postaci funkcji. Za każdym razem jako argumenty przekazujemy:

a) macierz (wskaźnik),
b) rozmiar macierzy,
c) parametry wykonywanej operacji.

Przydadzą nam się funkcje, które wyręczą nas w tworzeniu, kasowaniu i (w celach testowych) wypisywaniu na ekran macierzy.

```
double** tworzmacierz(int n, int m)
{
    assert(n > 0 && m > 0);
    double** W = new double*[n]; // tablica tablic
    for (int i=0; i<n; ++i)
        W[i] = new double[m]; // tablica
    return W;
}
```

```
void kasujmacierz(double** W, int n)
{
    assert(n > 0);
    // liczba kolumn nie jest potrzebna

    for (int i=0; i<n; ++i)
        delete [] W[i];
    delete [] W;
}
```

```
void wypiszmacierz(double** W, int n, int m)
{
    assert(n > 0 && m > 0);

    // nie bedzie zbyt pieknie, ale...
    for (int i=0; i<n; ++i)
    {
        for (int j=0; j<m; ++j)
            cout << '\t' << W[i][j];
        cout << endl;
    }
}
```

7.2.1 Mnożenie wiersza macierzy przez stałą

Załóżmy, że dana jest macierz rzeczywista A typu $n \times m$. Chcemy pomnożyć k-ty wiersz przez stałą rzeczywistą c. Oto przykładowa funkcja, która realizuje tę operację.

```
double** mnozwiersz(double** A, int n, int m, int k,
    double c)
{
    assert(k >= 0 && k < n && m > 0);
    double** B = tworzmacierz(n, m);

    for (int i=0; i<n; ++i)
    {
        for (int j=0; j<m; ++j)
            if (i == k)
                B[i][j] = A[i][j]*c;
            else
                B[i][j] = A[i][j];
    }

    return B;
}
```

7.2.2 Odejmowanie wierszy element po elemencie

Dana jest macierz rzeczywista A typu $n \times m$. Załóżmy, że chcemy odjąć od k-tego wiersza wiersz l-ty pomnożony przez stałą rzeczywistą c. Oto stosowny kod.

```
double** odskalwiersz(double** A, int n, int m, int k,
    int l, double c)
{
    assert(k >= 0 && k < n && l >= 0 && l < n && m > 0);

    double** B = tworzmacierz(n, m);

    for (int i=0; i<n; ++i)
    {
        for (int j=0; j<m; ++j)
            if (i == k)
                B[i][j] = A[i][j] - c*A[l][j];
```

```
12            else
13                B[i][j] = A[i][j];
14        }
15
16        return B;
17    }
```

7.2.3 Zamiana dwóch kolumn

Dana jest macierz A typu $n \times m$. Załóżmy, że chcemy zamienić k-tą kolumnę z l-tą.

```
1  double** zamienkol(double** A, int n, int m, int k, int l)
2  {
3      assert(k >= 0 && k < m && l >= 0 && l < m && n > 0);
4
5      double** B = tworzmacierz(n, m);
6
7      for (int i=0; i<n; ++i)
8      {
9          for (int j=0; j<m; ++j)
10         {
11             if (j == k)
12                 B[i][j] = A[i][l];
13             else if (j == l)
14                 B[i][j] = A[i][k];
15             else
16                 B[i][j] = A[i][j];
17         }
18     }
19
20     return B;
21 }
```

Ciekawostka — wersja alternatywna (modyfikująca macierz wejściową):

```
1  void zamienkol(double** A, int n, int m, int k, int l)
2  {
3      assert(k >= 0 && k < m && l >= 0 && l < m && n > 0);
4
5      for (int i=0; i<n; ++i)
6      {
7          double t = A[i][k];
```

```
        A[i][k] = A[i][l];
        A[i][l] = t;
    }
}
```

7.2.4 Wyszukiwanie wiersza o pewnej własności

Dana jest macierz rzeczywista A typu $n \times m$. Załóżmy, że chcemy znaleźć indeks wiersza, który ma największy co do modułu element w kolumnie k.

```
int wyszukajnajwkol(double** A, int n, int m, int k)
{
    assert(k >= 0 && k < m);

    int m = 0;
    for (int i=1; i<n; ++i)
    {
        if (fabs(A[i][k]) >= fabs(A[m][k]))
            m = i;
    }

    return i;
}
```

Przypomnijmy, że funkcja `double fabs(double x)` z biblioteki `<cmath>` zwraca wartość bezwzględną swego argumentu.

7.2.5 Dodawanie macierzy

Dane są dwie macierze A, B typu $n \times m$. Wynikiem operacji dodawania $A + B$ jest macierz:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix} + \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,m-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,m-1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0} + b_{0,0} & a_{0,1} + b_{0,1} & \cdots & a_{0,m-1} + b_{0,m-1} \\ a_{1,0} + b_{1,0} & a_{1,1} + b_{1,1} & \cdots & a_{1,m-1} + b_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} + b_{n-1,0} & a_{n-1,1} + b_{n-1,1} & \cdots & a_{n-1,m-1} + b_{n-1,m-1} \end{bmatrix}.$$

Oto kod funkcji służącej do dodawania macierzy.

```
double** dodaj(double** A, double** B, int n, int m)
{
    assert(n > 0 && m > 0);
    C = tworzmacierz(n, m);

    for (int i=0; i<n; ++i)
    {
        for (int j=0; j<m; ++j)
            C[i][j] = A[i][j] + B[i][j];
    }

    return C; /* nie zapomnij o zwolnieniu pamieci dalej!
        */
}
```

Ciekawostka
Zamiana kolejności pętli wewnętrznej i zewnętrznej

```
for (int j=0; j<m; ++j)
    for (int i=0; i<n; ++i)
    {
        C[i][j] = A[i][j] + B[i][j];
    }
```

w przypadku macierzy dużych rozmiarów powoduje, że program może wykonywać się wolniej. Na komputerze skromnego autora niniejszego skryptu dla macierzy 10000×10000 czas wykonywania rośnie z 2.7 aż do 24.1 sekundy. Drugie rozwiązanie nie wykorzystuje bowiem w pełni szybkiej **pamięci podręcznej** komputera.

7.2.6 Mnożenie macierzy

Niech A — macierz typu $n \times m$ oraz B — macierz typu $m \times r$. Wynikiem mnożenia macierzy $A \cdot B$ jest macierz C typu $n \times r$, dla której

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj},$$

gdzie $0 \leqslant i < n$, $0 \leqslant j < r$ (por. rys. 7.1).

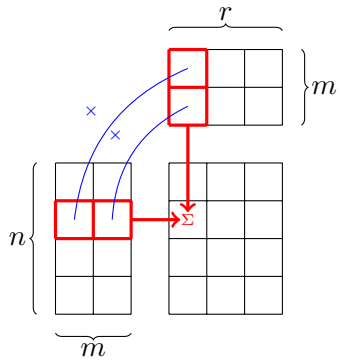
A oto kod funkcji służącej do wyznaczenia iloczynu dwóch macierzy.

```
double** mnoz(double** A, double** B, int n, int m, int r)
{
    assert(n > 0 && m > 0 && r > 0);

    C = tworzmacierz(n,r);

    for (int i=0; i<n; ++i)
    {
        for (int j=0; j<r; ++j)
        {
            C[i][j] = 0;
            for (int k=0; k<m; ++k)
                C[i][j] += A[i][k] * B[k][j];
        }
    }

  return C;   /* nie zapomnij o zwolnieniu pamieci dalej */
}
```



Rysunek 7.1. Ilustracja algorytmu mnożenia macierzy.

Ciekawostka

Podany algorytm wykonuje nmr operacji mnożenia, co dla macierzy kwadratowych o rozmiarach $n \times n$ daje n^3 operacji. Lepszy (i o wiele bardziej skomplikowany) algorytm, autorstwa Coppersmitha i Winograda, ma złożoność rzędu $n^{2,376}$.

7.2.7 Rozwiązywanie układów równań liniowych

Dany jest oznaczony układ równań postaci

$$A\mathbf{x} = \mathbf{b},$$

gdzie A jest macierzą nieosobliwą typu $n \times n$, $\mathbf{b}$ jest n-elementowym wektorem wyrazów wolnych oraz $\mathbf{x}$ jest n-elementowym wektorem niewiadomych.

Rozpatrywany układ równań można zapisać w następującej postaci.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

Uproszczona metoda **eliminacji Gaussa** polega na sprowadzeniu macierzy rozszerzonej $[A|\mathbf{b}]$ do postaci schodkowej $[A'|\mathbf{b}']$:

$$\left[\begin{array}{cccc|c} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} & b_0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} & b_{n-1} \end{array}\right]$$

Dokonuje się tego za pomocą następujących tzw. **operacji elementarnych** (wierszowych):

a) pomnożenie dowolnego wiersza macierzy rozszerzonej przez niezerową stałą,

b) dodanie do dowolnego wiersza kombinacji liniowej pozostałych wierszy.

Następnie uzyskuje się wartości wektora wynikowego $\mathbf{x}$ korzystając z tzw. **eliminacji wstecznej**:

$$x_i = \frac{1}{a'_{i,i}} \left(b'_i - \sum_{k=i+1}^{n-1} a'_{i,k} x_j \right).$$

dla kolejnych $i = n-1, n-2, \ldots, 0$.

Rozpatrzmy przykładową implementację tej metody w języku **C++**.

```
1 void schodkowa(double** A, double* b, int n);
2 void eliminwst(double** A, double* b, int n);
3
4 double* gauss(double** A, double* b, int n)
5 {
6     assert(n > 0);
7     schodkowa(A, b, n);
8     return eliminwst(A, b, n);
9 }
```

Funkcja sprowadzająca macierz do postaci schodkowej:

```
void schodkowa(double** A, double* b, int n)
{
    for (int i=0; i<n; ++i)
    { /* dla kazdego wiersza */
       assert(fabs(A[i][i]) > 1e-10 );//jesli nie=>macierz
           osobliwa (dlaczego tak?)

       for (int j=i+1; j<n; ++j)
       { // wiersz j:=(wiersz j)-A[j][i]/A[i][i]*(wiersz i)

          for (int k=i; k<n; ++k)
             A[j][k] -= A[j][i]/A[i][i]*A[i][k];

          b[j] -= A[j][i]/A[i][i]*b[i];
       }
    }
}
```

Eliminacja wsteczna:

```
double* eliminwst(double** A, double* b, int n)
{
    double* x = new double[n];
    for (int i=n-1; i>=0; --i)
    {
       x[i] = b[i];
       for (int k=i+1; k<n; ++k)
       {
          x[i] -= A[i][k] * x[k];
       }
       x[i] /= A[i][i];
    }
    return x;
}
```

7.3 Ćwiczenia

Zadanie 7.140. Dana jest macierz A typu $n \times m$ o wartościach rzeczywistych oraz liczba $k \in \mathbb{R}$. Napisz funkcję, która wyznaczy wartość kA, czyli implementującą mnożenie macierzy przez skalar.

Dana jest macierz A typu $n \times m$ o wartościach rzeczywistych oraz wektor $\mathbf{b} \in \mathbb{R}^n$. Napisz funkcję, która zwróci macierz $[A|\mathbf{b}]$, czyli A rozszerzoną o nową kolumnę, której wartości pobrane są z $\mathbf{b}$.

Zadanie 7.142. Dana jest macierz A typu $n \times m$ o wartościach rzeczywistych oraz wektor $\mathbf{b} \in \mathbb{R}^m$. Napisz funkcję, która zwróci macierz A rozszerzoną o nowy wiersz, którego wartości pobrane są z $\mathbf{b}$.

Zadanie 7.143. Dana jest macierz A typu 2×2 o wartościach rzeczywistych. Napisz funkcję, która zwróci wyznacznik danej macierzy.

Zadanie 7.144. Dana jest macierz A typu 3×3 o wartościach rzeczywistych. Napisz funkcję, która zwróci wyznacznik danej macierzy.

(*) Zadanie 7.145. Dana jest macierz kwadratowa A o 4 wierszach i 4 kolumnach zawierająca wartości rzeczywiste. Napisz rekurencyjną funkcję, która zwróci wyznacznik danej macierzy. Skorzystaj wprost z definicji wyznacznika. Uwaga: taka metoda jest zbyt wolna, by korzystać z niej w praktyce.

Zadanie 7.146. Napisz funkcję, która rozwiązuje układ dwóch równań liniowych, korzystając z metody Cramera (dana jest macierz 2×2 i wektor). Poprawnie identyfikuj przypadki, w których dany układ nie jest oznaczony.

Zadanie 7.147. Napisz funkcję, która rozwiązuje układ 3 równań liniowych, korzystając z metody Cramera (dana jest macierz 3×3 i wektor). Poprawnie identyfikuj przypadki, w których dany układ nie jest oznaczony.

Zadanie 7.148. Dana jest macierz A o wartościach całkowitych. Napisz funkcję, która zwróci jej transpozycję.

Zadanie 7.149. Dana jest macierz A typu $n \times m$ o wartościach całkowitych. Napisz funkcję, która dla danego $0 \leqslant i < n$ i $0 \leqslant j < m$ zwróci podmacierz powstałą przez usunięcie z A i-tego wiersza i j-tej kolumny.

Zadanie 7.150. Dana jest kwadratowa macierz A o wartościach całkowitych. Napisz funkcję, która sprawdzi, czy macierz jest symetryczna. Zwróć wynik typu `bool`.

Zadanie 7.151. Dana jest macierz kwadratowa A o wartościach rzeczywistych typu $n \times n$. Napisz funkcję, która zwróci jej ślad, określony jako

$$\mathrm{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn} = \sum_{i=1}^{n} a_{ii}.$$

Zadanie 7.152. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej diagonalę w postaci tablicy jednowymiarowej.

Zadanie 7.153. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej macierz diagonalną, czyli macierz z wyzerowanymi wszystkimi elementami poza przekątną.

Kwadratem łacińskim stopnia n nazywamy macierz kwadratową typu $n \times n$ o elementach ze zbioru $\{1, 2, \ldots, n\}$, taką że żaden wiersz ani żadna kolumna nie zawierają dwóch takich samych wartości. Napisz funkcję, która sprawdza, czy dana macierz jest kwadratem łacińskim. Zwróć wynik typu `bool`.

Zadanie 7.155. Kwadratem magicznym stopnia n nazywamy macierz kwadratową typu $n \times n$ o elementach ze zbioru liczb naturalnych, taką że sumy elementów w każdym wierszu, w każdej kolumnie i na każdej z dwóch przekątnych są takie same. Napisz funkcję, która sprawdza, czy dana macierz jest kwadratem magicznym. Zwróć wynik typu `bool`.

Zadanie 7.156. Napisz funkcję

```
double** wstawWiersz(double** M, int n, int m, int i, double c)
```

(odpowiednio: wstawKolumnę), która wstawi do danej macierzy M nowy wiersz (nową kolumnę) pomiędzy wierszami (kolumnami) i oraz $i+1$, oraz wypełni go wartością c.

(*) Zadanie 7.157. Napisz funkcję, która sprawdzi czy dana macierz może być otrzymana z macierzy 1×1 przez ciąg operacji wstawWiersz i wstawKolumnę, zdefiniowanych w zadaniu 7.156.

(*) Zadanie 7.158. Napisz nierekurencyjną funkcję wyznaczającą wartość wyrażenia:

- $f(n,m) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (f(i,j) - i - j)^2$,
- $f(n,m) = \begin{cases} 1 & \text{dla } n = 0 \text{ lub } m = 0, \\ \|\{(i,j) \in \mathbb{N}^2 : i < n, j < m, f(i,j) < (i-j)^2\}\| & \text{dla } n, m > 0, \end{cases}$

gdzie $n, m \in \mathbb{N}$.

7.4 Laboratoria

Zadanie 7.159. Na rozgrzewkę:

a) napisz funkcję, która tworzy nową, dynamicznie alokowaną macierz rozmiaru $n \times m$ (dla danych n, m),

b) napisz funkcję, która wypełnia daną macierz rozmiaru $n \times m$ losowymi wartościami z przedziału $[0, i+1)$, gdzie $i = 0, 1, \ldots, n-1$ — numer wiersza (użyj funkcji `rand()`, zakładając, że funkcja `srand()` została wywołana wcześniej, np. w `main()`),

c) napisz funkcję, która wypisuje na ekran elementy danej macierzy rzeczywistej rozmiaru $n \times m$,

d) napisz funkcję, która usuwa (dealokuje przydzieloną pamięć) daną macierz o n kolumnach.

Zadanie 7.160. Napisz pełny program, który zawiera implementacje powyższych funkcji w osobnym pliku źródłowym i nagłówkowym. W funkcji `main()` inicjujemy ge-

nerator liczb pseudolosowych za pomocą wywołania `srand(time(0))`, wczytujemy wartości `n` i `m` z klawiatury (sprawdzając, czy są dodatnie) oraz wywołujemy kolejno operacje tworzenia, wypełniania, wypisywania i kasowania macierzy.

Zadanie 7.161. Napisz funkcję, która tworzy nową macierz całkowitą A o rozmiarze $n \times m$ o elementach wg wzoru:

$$a_{i,j} = \begin{cases} i + j & \text{dla } i = 0 \text{ lub } j = 0, \\ a_{i-1,j} + a_{i,j-1} & \text{dla } i, j > 0, \end{cases}$$

gdzie $i = 0, 1, \ldots, n-1$, $j = 0, 1, \ldots, m-1$.

Podstawowe abstrakcyjne struktury danych

8

8.1 Struktury w języku C++

Do tej pory omawialiśmy następujące ogólne typy zmiennych: zmienne skalarne, tablicowe i wskaźnikowe.

W języku **C++** możemy tworzyć własne **typy złożone** będące reprezentacją **iloczynu kartezjańskiego** różnych innych zbiorów (typów). Są to tzw. **struktury**. Oto składnia ich definicji:

```
struct NazwaNowegoTypu
{
    typ1 nazwaPola1;
    typ2 nazwaPola2;
    //....
    typN nazwaPolaN;
}; // konieczny srednik!
```

Zmienne typu złożonego deklarujemy w standardowy sposób, czyli np.

```
nazwaStruktury identyfikator;
```

Do poszczególnych pól struktury możemy odwołać się za pomocą kropki, np.

```
identyfikator.nazwaPola
```

Pola zmiennej typu złożonego traktujemy jak zwykłe zmienne odpowiednich typów.

Przykład: struktura reprezentująca zbiór $\mathbb{N} \times \mathbb{R}$.

```
1 struct NR
2 {
3     int polen;
```

```
    double poler;
};

int main()
{
    NR x;          // tzn. x ∈ ℕ × ℝ
    x.polen = 1;
    x.poler = 3.14;

    cout << x; // Blad: operacja niezdefiniowana
    cout << x.polen; // OK

    return 0;
}
```

Oczywiście można też tworzyć tablice i wskaźniki do obiektów tego typu:

```
int main()
{
    NR x[3];          // tablica
    x[0].polen = 100;
    x[0].poler = 100.0;
    // ....
    NR* wska = &x[1];
    // .... itp.
    return 0;
}
```

Rozważmy teraz następujące funkcje:

```
void f1(NR a)
{
    // a przekazana przez wartosc - kopiowana
    cout << a.polen << " " << a.poler;
}

void f2(NR& a)
{
    // a przekazana przez referencje - niekopiowana
    cout << a.polen << " " << a.poler;
}
```

Jak już wiemy, przekazanie zmiennej przez referencję jest wydajniejsze niż podanie jej przez wartość. W pierwszym przypadku nie zostanie utworzona jej kopia. Jednakże zmiany wprowadzone w przekazanych obiektach będą widoczne na zewnątrz funkcji.

Idąc dalej tym śladem, rozpatrzmy dwie kolejne funkcje.

```
void f3(const NR& a)
{
    // a przekazana przez referencje - niekopiowana
    // + zabezpieczona przed zmiana
    cout << a.polen << " " << a.poler;
}

void f4(NR* a)
{
    // a przekazana przez wskaznik - niekopiowana
    cout << (*a).polen << " " << (*a).poler;
    // rownowazny zapis (!)
    cout << a->polen << " " << a->poler;
}
```

W funkcji `f3()` dodatkowo zabezpieczyliśmy obiekt przed przypadkową zmianą za pomocą modyfikatora **const**.

Jak widzimy, przekazanie zmiennej przez wskaźnik w tym kontekście jest równoważne przekazaniu jej przez referencję. Dostęp do pól struktury przekazanej przez wskaźnik możemy uzyskać za pomocą operatora −>, który jest bardzo wygodny.

8.2 Podstawowe abstrakcyjne struktury danych

W niniejszym paragrafie omówimy następujące **dynamiczne abstrakcyjne struktury danych**:

a) listy jednokierunkowe,
b) stosy,
c) kolejki,
d) kolejki priorytetowe,
e) listy dwukierunkowe,
f) drzewa binarne.

Służą one do przechowywania różnego rodzaju danych. Każda z nich charakteryzuje się innymi właściwościami. Na przykład lista jednokierunkowa pozwala bardzo szybko dodawać nowe elementy do zbioru, a w kolejce priorytetowej mamy łatwy dostęp do danych uporządkowanych.

Dla uproszczenia w omawianych strukturach danych będziemy przechowywali jeden element typu **int**.

8.2.1 Lista jednokierunkowa

Lista kierunkowa (ang. *linked list*), podobnie jak tablica, służy do przechowywania elementów tego samego typu. W odróżnieniu jednak od tablicy nie ma ona z góry ustalonego rozmiaru. Pozwala na efektywne wstawianie i usuwanie elementów. Odbywa się to kosztem czasu ich wyszukiwania.

Dane przechowywane są w **węzłach** następującej postaci:

```
struct wezel
{
    int elem; // element(y), ktory przechowujemy w wezle
    wezel* nast; // wskaznik na nastepny element
};
```

Węzły umieszczone są w różnych (dowolnych) miejscach w pamięci. Każdy z nich jest osobno przydzielany dynamicznie.

Lista składa się z węzłów połączonych ze sobą w kierunku od pierwszego do ostatniego elementu. Zatem dodatkowo należy zapamiętać, gdzie leży pierwszy element:

```
1 wezel* pocz; // tzw. glowa listy
```

Schemat przykładowej listy jednokierunkowej przechowującej elementy 3, 5 oraz 7 przedstawia rys. 8.1. Zwróćmy uwagę na to, w jaki sposób węzły mogą być rozlokowane w pamięci.

Przyjrzymy się implementacji następujących operacji:
- sprawdzenie, czy dany element występuje na liście,
- wstawienie elementu na początek listy,
- wstawienie elementu na koniec listy,
- usunięcie elementu z początku listy,
- usunięcie elementu z końca listy.

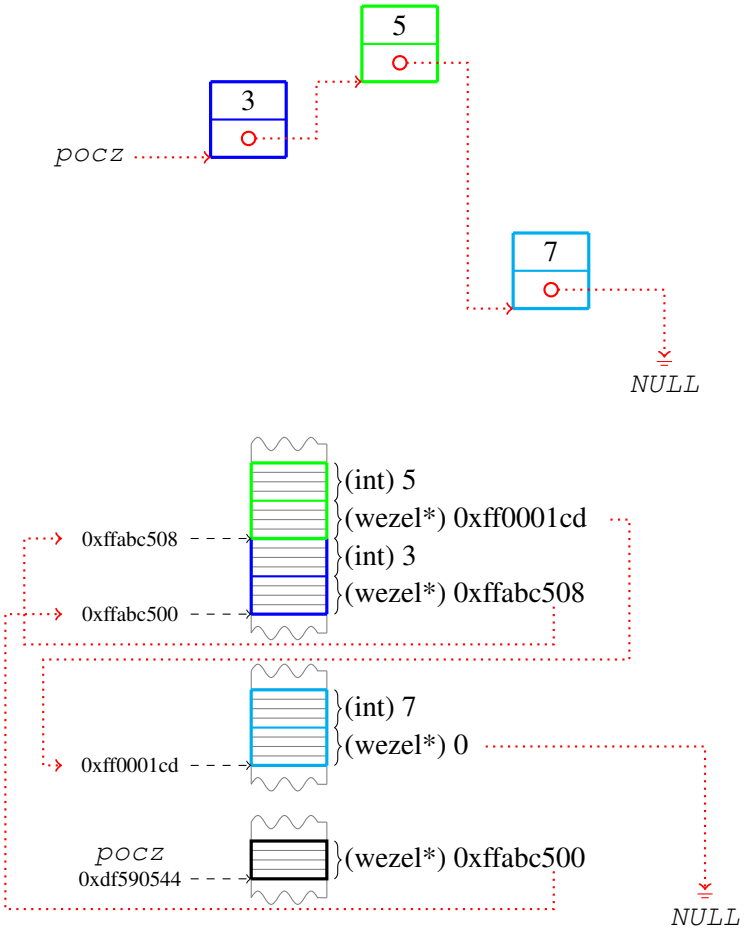
Ważne

Należy zwrócić szczególną uwagę na przypadek, gdy lista początkowo jest pusta (gdy `pocz == NULL`)!

8.2.1.1 Wyszukiwanie elementu

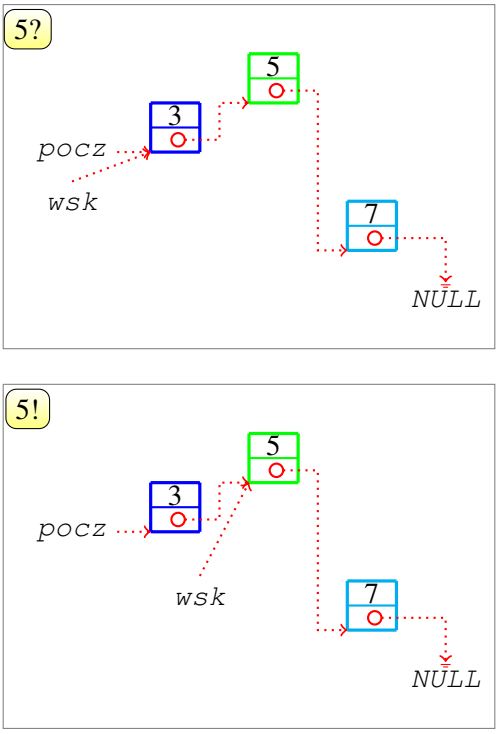
Na rys. 8.2 i 8.3 znajdziemy ilustrację, krok po kroku, w jaki sposób przebiega wyszukiwanie elementów 5 oraz 2 na liście zawierającej (3,5,7).

W pierwszym przypadku element zostaje odnaleziony już w drugim kroku. W kolejnym stwierdzamy, że nie ma go wcale na danej liście.

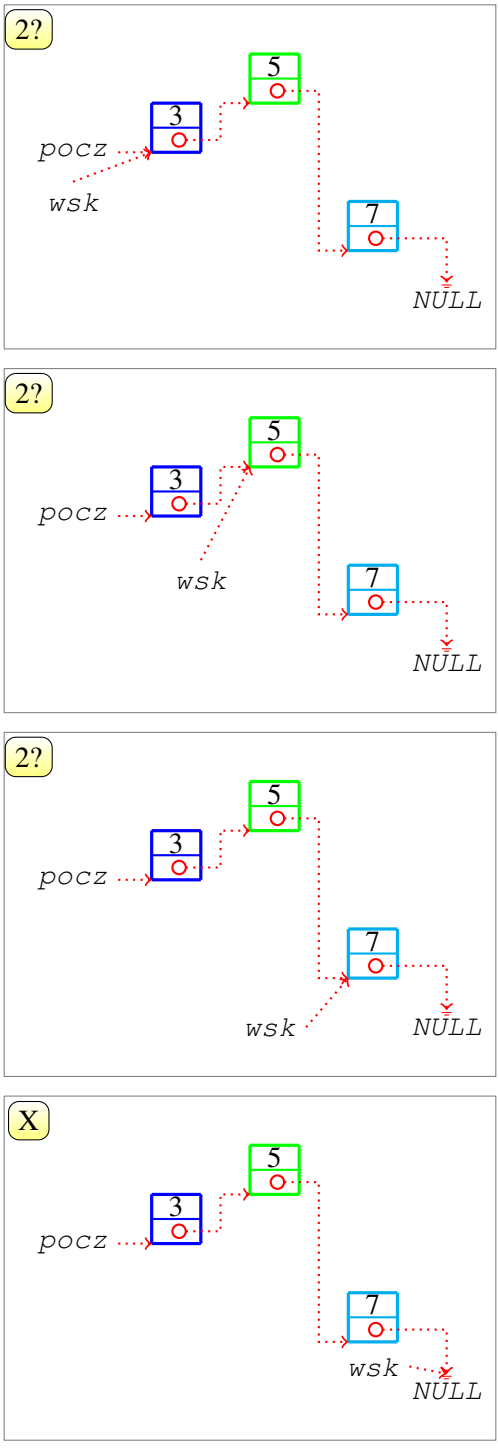


Rysunek 8.1. Przykładowa lista jednokierunkowa przechowująca elementy 3,5,7. Schemat graficzny i organizacja pamięci.

Zauważmy, że korzystamy tutaj z dodatkowego wskaźnika, za pomocą którego poruszamy się po odwiedzanych węzłach. Wskaźnik ten rozpoczyna swoje poszukiwania od głowy listy. Jest to bowiem jedyny dostępny bezpośrednio element. Do każdego kolejnego dostajemy się za pomocą pola `nast`.



Rysunek 8.2. Wyszukiwanie elementu 5 w liście jednokierunkowej.



Rysunek 8.3. Wyszukiwanie elementu 2 w liście jednokierunkowej.

Oto iteracyjna wersja funkcji formalizującej powyższe kroki.

```
bool szukaj(wezel* pocz, int x)
{
    wezel* wsk = pocz;
    while (wsk != NULL)
    {
        if (wsk->elem == x)
            return true; // znaleziony
        else
            wsk = wsk->nast; // przejdz do nastepnego
                wezla
    }

    return false; // doszlismy do konca listy
}
```

Wywołanie:

```
... szukaj(pocz, x);
```

Podany wyżej algorytm można również zapisać równoważnie w formie rekurencyjnej.

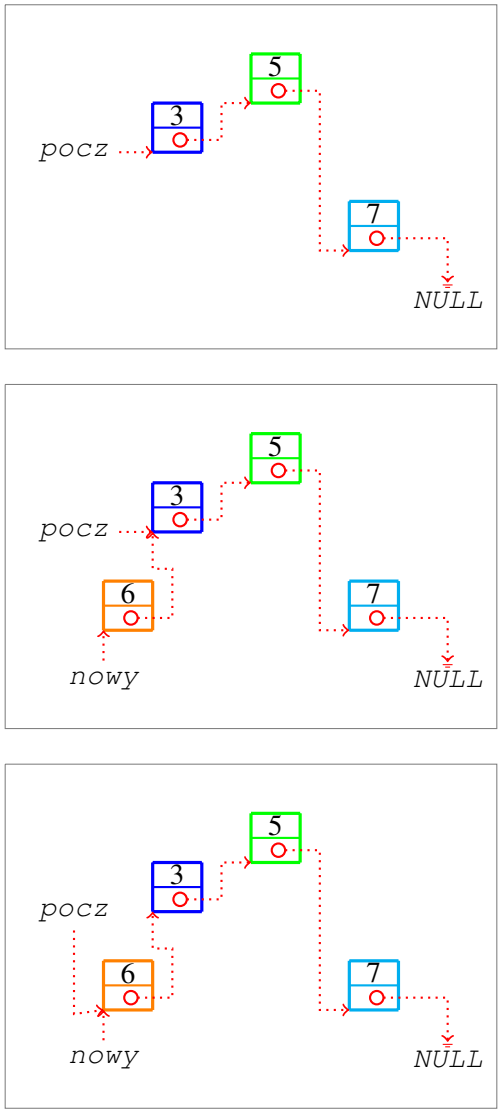
```
bool szukaj2(wezel* wsk, int x)
{
    if (wsk == NULL)
        return false; // doszlismy do konca listy
    else if (wsk->elem == x)
        return true; // znaleziony
    else
        return szukaj2(wsk->nast, x); // szukaj dalej
}
```

Wywołanie:

```
... szukaj2(pocz, x);
```

8.2.1.2 Wstawianie elementu

Najpierw zajmijmy się wstawianiem elementu na początek listy. Rys. 8.4 ilustruje kolejne kroki potrzebne do utworzenia listy (6,3,5,7) z listy (3,5,7). Zwróćmy uwagę, że po dokonaniu tej operacji zmienia się głowa listy.



Rysunek 8.4. Wstawianie elementu 6 na początek listy jednokierunkowej.

Oto kod stosownej funkcji w języku **C++**. Zauważmy, że pierwszym parametrem jest referencja na zmienną wskaźnikową. Przekazujemy tutaj głowę listy, która będzie zmieniona przez tę funkcję.

```
void wstawpocz(wezel*& pocz, int x)
{
     wezel* nowy = new wezel;
     nowy->elem = x;

     nowy->nast = pocz; /* wskazuj na to, na co wskazuje
         pocz; moze byc NULL */

     pocz = nowy;        /* teraz lista zaczyna sie od
         nowego wezla */
}
```

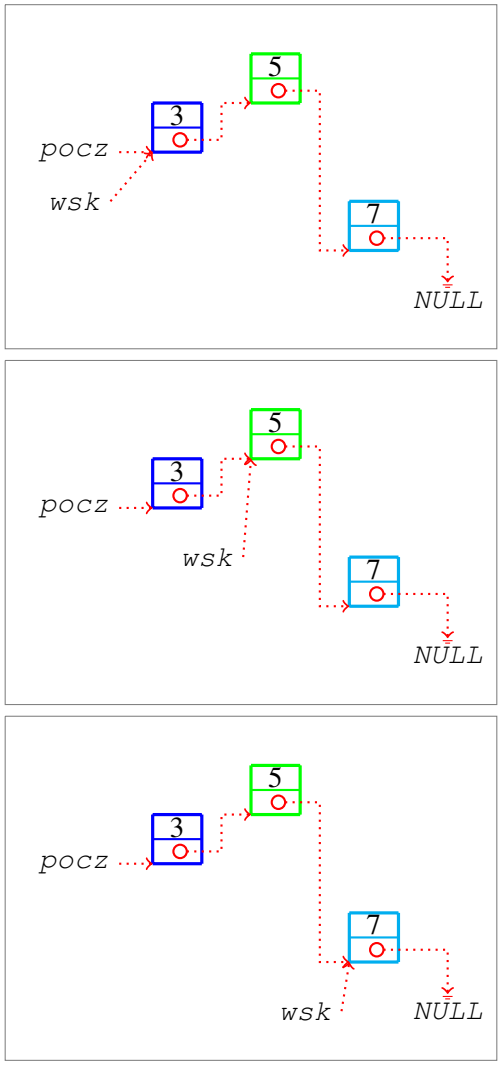
Rozważmy teraz, w jaki sposób wstawić element na koniec listy. Rys. 8.5 i 8.6 ilustrują, w jaki sposób utworzyć listę (3,5,7,1), mając daną listę (3,5,7). Zauważmy, że pierwszym krokiem, jaki należy wykonać, jest przejście na koniec listy. Tutaj korzystamy z dwóch dodatkowych wskaźników.

Poniższa funkcja implementuje interesującą nas operację. Zauważmy, że pierwszym parametrem jest referencja do wskaźnika. Jeżeli lista jest pusta, przekazana głowa może zostać zmieniona.

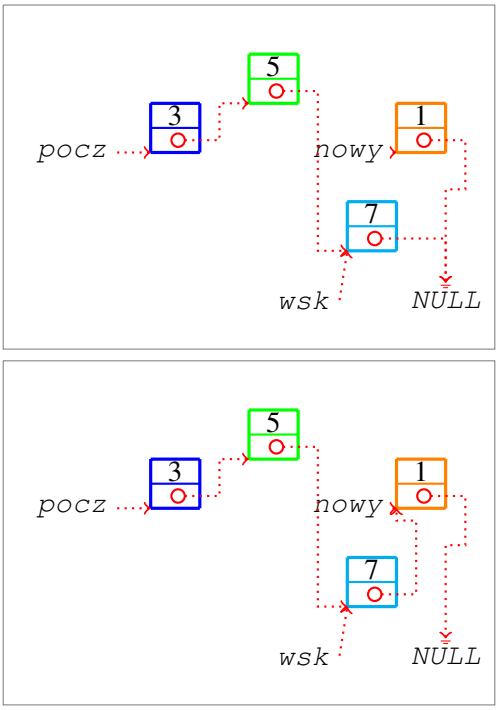
```
void wstawkon(wezel*& pocz, int x)
{
     wezel* nowy = new wezel;
     nowy->elem = x;
     nowy->nast = NULL;

     if (pocz == NULL) // czy lista pusta?
          pocz = nowy;
     else
     {
          wezel* wsk = pocz;
          while (wsk->nast != NULL)
               wsk = wsk->nast; // przejdz na ostatni element

          wsk->nast = nowy; // nowy ostatni element
     }
}
```



Rysunek 8.5. Wstawianie elementu 1 na koniec listy jednokierunkowej cz. I.



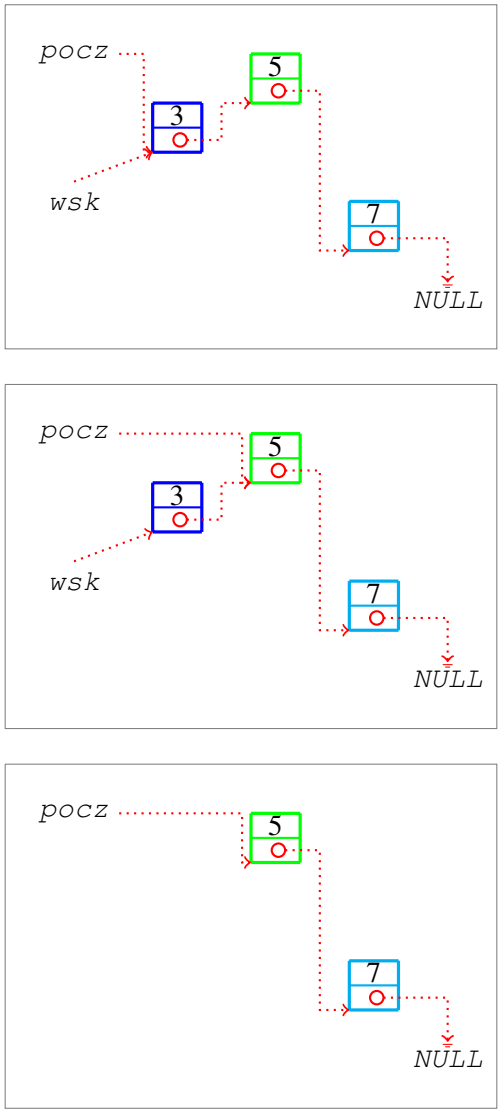
Rysunek 8.6. Wstawianie elementu 1 na koniec listy jednokierunkowej cz. II.

A oto równoważna wersja rekurencyjna.

```
void wstawkon2(wezel*& wsk, int x)
{
     if (wsk != NULL)
          wstawkon2(wsk->nast, x); // idz dalej
     else
     {
          wsk = new wezel;
          wsk->elem = x;
          wsk->nast = NULL;
     }
}
```

8.2.1.3 Usuwanie elementu

Podobnie jak wyżej, i tutaj rozpatrzymy dwa przypadki, w których usuwany element znajduje się na początku lub na końcu listy.



Rysunek 8.7. Usunięcie elementu z początku listy jednokierunkowej.

Rys. 8.7 przedstawia możliwy sposób usuwania elementów z początku listy jednokierunkowej. Implementację tej operacji w **C++** przedstawia poniższy kod. Zauważmy, że po jej dokonaniu możliwe jest osiągnięcie stanu *glowa == NULL*, co oznacza, że usunięty element był jedynym.

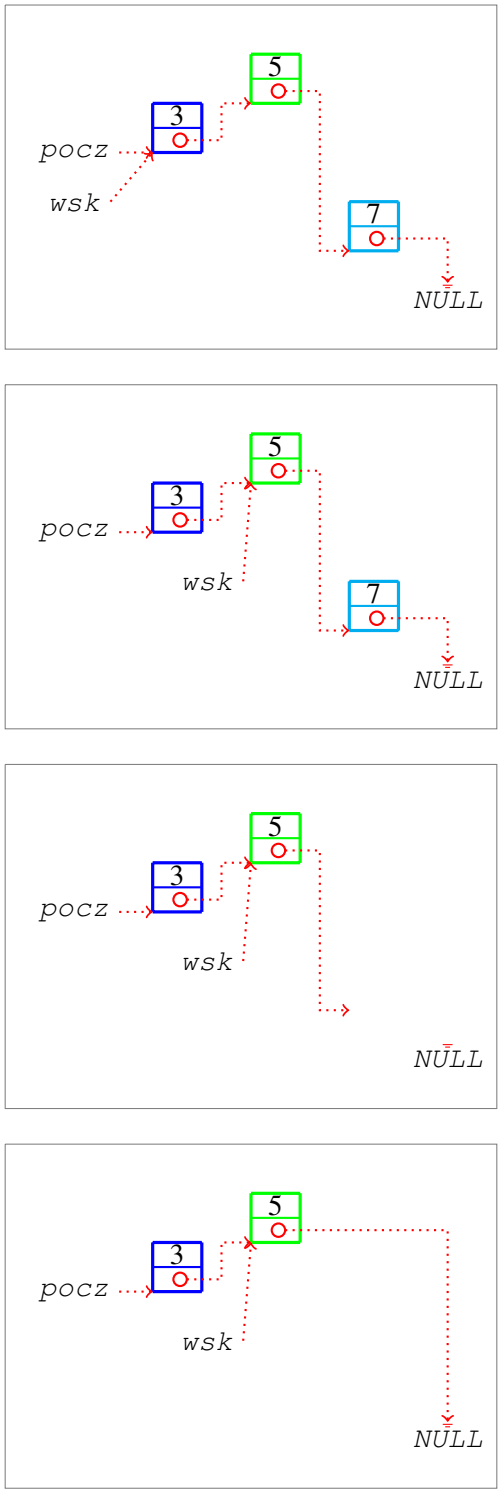
```
void usunpocz(wezel*& pocz)
{
    if (pocz != NULL)
    {
        wezel* wsk = pocz;
        pocz = pocz->nast;
        delete wsk;
    }
}
```

Zastanówmy się teraz, w jaki sposób dokonać wykasowania ostatniego elementu.

Rys. 8.8 przedstawia krok po kroku kolejne działania. Formalizuje je następujący kod w **C++**.

```
void usunkon(wezel*& pocz)
{
    if (pocz == NULL)
        return;        // lista pusta
    else if (pocz->nast == NULL)
    { // tylko jeden element
        delete pocz;
        pocz = NULL;
    }
    else
    {                              // > 1 element
        wezel* wsk = pocz;
        // przejdz na przedostatni element
        while (wsk->nast->nast != NULL)
            wsk = wsk->nast;

        delete wsk->nast;
        wsk->nast = NULL;
    }
}
```



Rysunek 8.8. Usunięcie elementu z końca listy jednokierunkowej.

Omawianą procedurę można zapisać również w postaci rekurencyjnej.

```
void usunkon2(wezel*& wsk)
{
     if (wsk == NULL) return; // lista byla pusta
     else if (wsk->nast == NULL) // jestesmy na koncu
     {
          delete wsk;
          wsk = NULL;
     }
     else // idziemy dalej
          usunkon2(wsk->nast);
}
```

8.2.1.4 Uwaga na temat wydajności

Porównajmy na koniec dwie struktury danych: listę jednokierunkową oraz zwykłą tablicę jednowymiarową, przechowujące n elementów. Poniższa tabela zestawia liczbę elementów, które należy rozpatrzeć, aby móc zrealizować podstawowe operacje.

Operacja	Tablica	Lista
Dostęp do i-tego elementu	1	i
Wyszukiwanie	$\leqslant n$	$\leqslant n$
Wstawianie na początek	n	1
Wstawianie na koniec	n	n (*)
Kasowanie z początku	n	1
Kasowanie z końca	n	n

(*) Liczbę operacji potrzebnych do wstawienia elementu na koniec listy można zredukować do 1 jeśli dodatkowo będziemy przechowywać wskaźnik na koniec listy. Wymaga to jednak przerobienia wszystkich funkcji.

Zauważmy, że dostęp do poszczególnych elementów w tablicy jest natychmiastowy. Z kolei lista ma tę zaletę, iż szybko pozwala zwiększać lub zmniejszać swój rozmiar. Nie wymaga ona bowiem kopiowania wszystkich elementów za każdym razem.

8.2.2 Stos

Stos (ang. *stack*) jest strukturą danych typu LIFO (ang. *last-in-first-out*), tzn. elementy są zdejmowane z niej w kolejności odwrotnej niż były na niej umieszczane.

Stos udostępnia tylko dwie operacje:

– umieść (ang. *push*) — wstawia element na początek stosu,

– wyjmij (ang. *pop*) — usuwa i zwraca element z początku stosu.

Bardzo łatwo jest zaimplementować stos z użyciem już poznanych algorytmów dla listy jednokierunkowej. Dlatego pozostawiamy go jako ćwiczenie.

8.2.3 Kolejka

Kolejka (ang. *queue*) jest strukturą danych typu FIFO (ang. *first-in-first-out*), czyli elementy są udostępniane w tej samej kolejności, w jakiej były w niej umieszczane. Udostępnia ona dwie następujące operacje:

– umieść (ang. *enqueue*) — wstawia element na koniec kolejki,

– wyjmij (ang. *dequeue*) — usuwa i zwraca element z początku kolejki.

Ze względów wydajnościowych kolejkę dobrze jest implementować jako listę jednokierunkową, w której przechowuje się dodatkowo wskaźnik na ostatni element. Jej implementację pozostawiamy jako ćwiczenie.

8.2.4 Kolejka priorytetowa

Jako kolejne ćwiczenie pozostawiamy także implementację tzw. **kolejki priorytetowej** (ang. *priority queue*), która jest modyfikacją listy jednokierunkowej. Przechowuje ona elementy wg porządku <= (bez duplikatów). Udostępnia następujące operacje:

– wstawienie elementu wg porządku <=,

– pobranie elementu najmniejszego,

– usunięcie elementu najmniejszego.

8.2.5 Lista dwukierunkowa

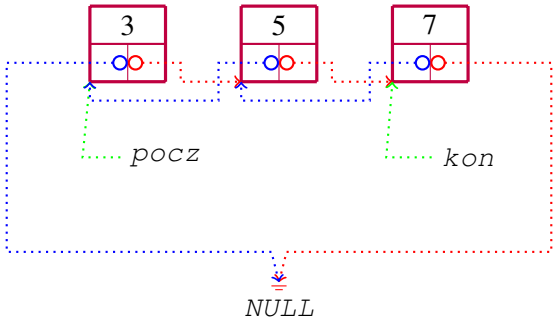
Lista dwukierunkowa pozwala na bardzo szybkie wstawianie i usuwanie elementów zarówno z końca, jak i z początku listy. Dodatkowo pozwala na przeglądanie elementów w kierunku przeciwnym. Dane przechowywane są w **węzłach** następującej postaci:

```
struct wezel
{
    int elem; // element(y), ktory przechowujemy w wezle
    wezel* nast; // wskaznik na nastepny element
    wezel* poprz; // wskaznik na poprzedni element
};
```

Należy rzecz jasna zapamiętać wskaźnik na pierwszy (`pocz`) i wskaźnik na ostatni (`kon`) element listy dwukierunkowej.

Przykładowa lista dwukierunkowa (3,5,7) przedstawiona jest na rys. 8.9.

Jako że jej implementacja jest dość podobna do omawianych już konstrukcji, pozostawiamy ją jako ćwiczenie.



Rysunek 8.9. Przykładowa lista dwukierunkowa.

8.2.6 Drzewo binarne

Drzewo poszukiwań binarnych (BST, ang. *binary search tree*) jest dynamiczną abstrakcyjną strukturą danych, która przechowuje dane uporządkowane względem relacji <=. Umożliwia ono często dość szybkie wyszukiwanie elementów. Dla danych losowych oczekiwana liczba operacji koniecznych do znalezienia danej wartości jest rzędu $\log_2 n$ (logarytm jest funkcją, która dość wolno zwiększa swe wartości wraz ze wzrostem wartości argumentu, np. $\log_2 1000 \simeq 10$, a $\log_2 10000 \simeq 13$). Niestety, w przypadku pesymistycznym liczba ta rośnie do n, gdzie n to rozmiar przechowywanego zbioru danych. Na zajęciach w III semestrze dowiemy się, jak równoważyć drzewa w taki sposób, aby wyszukiwanie było zawsze efektywne.

Dane w drzewie binarnym przechowywane są w węzłach zdefiniowanych następująco:

```
struct wezel
{
    int elem; // element(y), ktory przechowujemy w wezle
    wezel* lewy; // wskaznik na lewe poddrzewo (lewego
        potomka)
    wezel* prawy; // wskaznik na prawe poddrzewo (prawego
        potomka)
};
```

Zatem każdy węzeł ma co najwyżej dwóch potomków.

Drzewo jest strukturą **acykliczną**, tzn. wychodząc z dowolnego węzła za pomocą wskaźników, nie da się do niego wrócić. Co więcej, jeśli istnieje ścieżka pomiędzy węzłami v i w, to jest ona określona jednoznacznie.

Początek drzewa określa węzeł zwany **korzeniem** (ang. *root*). Nie da się do niego dojść z żadnego innego węzła (inaczej: nie ma on rodzica). Ponadto drzewo jest **spójne**, tzn. z korzenia można dojść do każdego innego węzła. Węzły, które nie mają potomków, zwane są inaczej **liśćmi**.

Rozpatrzmy dowolny węzeł v. Ważną cechą drzewa jest to, że

a) wszystkie elementy w jego lewym poddrzewie są nie większe niż element przechowywany w węźle v,
b) wszystkie elementy w jego prawym poddrzewie są większe niż element przechowywany w v.

Często rozpatruje się drzewa (i tak czynimy w tym przypadku), w których zabronione jest przechowywanie duplikatów elementów.

Rozważymy następujące operacje na drzewie binarnym.

– wyszukiwanie zadanego elementu,
– wypisywanie elementów wg porządku <=,
– wstawianie,
– usuwanie.

8.2.6.1 Wyszukiwanie elementu

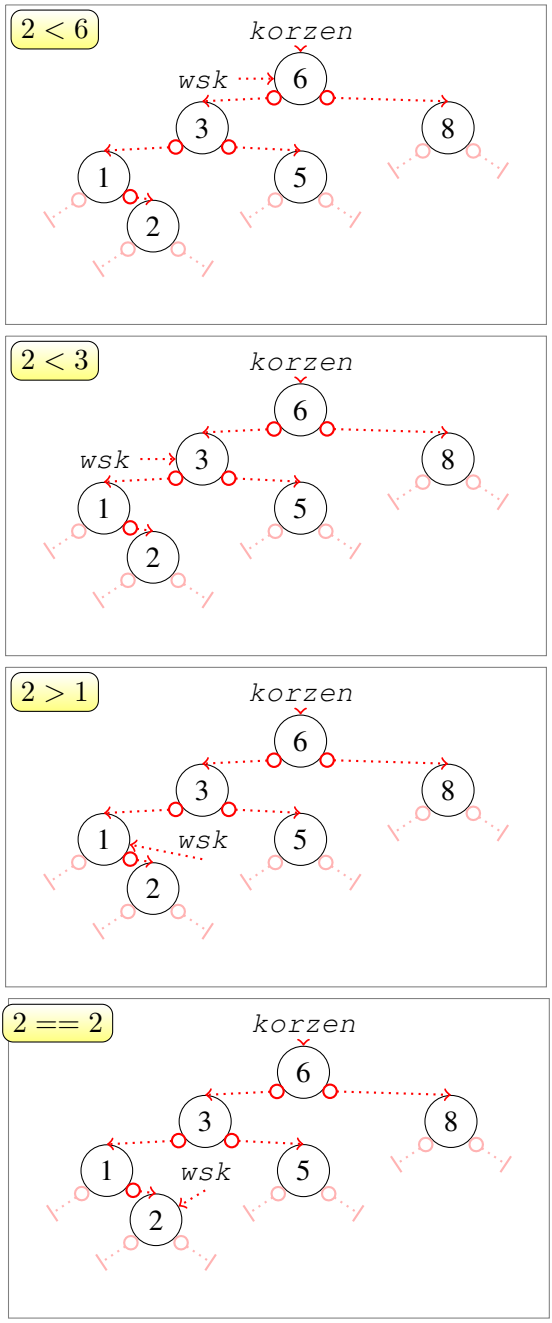
Rys. 8.10 ilustruje krok po kroku wyszukiwanie elementu 2 w drzewie składającym się z elementów $\{1, 2, 3, 5, 6, 8\}$.

Ważne

Zwróćmy uwagę, że istnieje wiele postaci drzew binarnych, które mogłyby służyć do przechowywania powyższego zbioru wartości. W szczególności drzewo takie mogłoby się zredukować do listy, np. jeśli żaden z węzłów nie miałby lewego lub prawego poddrzewa. W tym przypadku wyszukiwanie może wymagać przejrzenia wszystkich elementów.

Oto wersja iteracyjna funkcji służącej do wyszukiwania elementów o zadanej wartości.

```
1  bool szukaj(wezel* korzen, int x)
2  {
3      wezel* wsk = korzen;
4      while (wsk != NULL)
5      {
6          if (x == wsk->elem) return true;
7          else if (x < wsk->elem) wsk = wsk->lewy;
8          else                    wsk = wsk->prawy;
9      }
10
11     return false; // nie znaleziono
12 }
```



Rysunek 8.10. Wyszukiwanie elementu 2 w przykładowym drzewie binarnym.

Poniżej znajduje się równoważna funkcja rekurencyjna.

```
bool szukaj2(wezel* wsk, int x)
{
    if (wsk == NULL)          return false;
    else if (x == wsk->elem) return true;
    else if (x < wsk->elem)  return szukaj2(wsk->lewy,x);
    else                      return szukaj2(wsk->prawy,x);
}
```

8.2.6.2 Wypisywanie wszystkich elementów wg porządku

Procedura wypisująca wszystkie elementy przechowywane w drzewie uwzględniająca porządek <= jest ze swej natury rekurencyjna. Skoro wartości w lewym poddrzewie danego węzła są mniejsze (przypominany, omijamy duplikaty) od wartości w danym węźle, dajmy na to, v, przed wypisaniem wartości w v należy wypisać wartości w lewym poddrzewie v.

Implementacja wyszukiwania jest dość prosta.

```
void wypisz(wezel* wsk)
{
    if (wsk == NULL) return; // tu nie ma nic do roboty

    wypisz(wsk->lewy);   // najpierw lewe poddrzewo
    cout << wsk->elem;   // teraz wypisywanie
    wypisz(wsk->prawy); // a potem prawe poddrzewo
}
```

8.2.6.3 Wstawianie elementu

Wstawianie elementu 4 do przykładowego drzewa binarnego ilustruje rys. 8.11.

Najprościej wstawić element jako liść. Zatem najpierw należy znaleźć węzeł, do którego nowy element będziemy doczepiać jako potomka. Oto wersja rekurencyjna stosownej funkcji:

```
void wstaw(wezel*& wsk, int x)
{
    if (wsk == NULL)
    { // tutaj bedzie nowy lisc
        wsk = new wezel;
        wsk->elem = x;
        wsk->lewy = NULL;
```

```
8           wsk->prawy = NULL;
9       }
10      else if (x == wsk->elem) return; // nic nie rob
11      else if (x < wsk->elem)  return wstaw(wsk->lewy,  x);
12      else                     return wstaw(wsk->prawy, x);
13  }
```

8.2.6.4 Usuwanie elementu (*)

Usuwanie elementu z drzewa jest dość skomplikowane. Musimy rozpatrzeć trzy przypadki:

a) liść — po prostu usunąć,
b) węzeł z jednym potomkiem — zastąpić potomkiem,
c) węzeł v z dwoma potomkami — zastąpić lewym (prawym) dzieckiem (ozn. w). Lewe (prawe) poddrzewo w pozostaje bez zmian. Prawe (lewe) poddrzewo v staje się prawym (lewym) poddrzewem w. Prawe (lewe) poddrzewo w zostaje doczepione za elementem najmniejszym (największym) w prawym (lewym) poddrzewie v.

Zainteresowanym przedstawiamy przykładową implementację.

```
1  void usun(wezel*& wsk, int x) // rekurencyjna
2  {
3       if (wsk == NULL) return; // danego elementu brak
4       else if (x < wsk->elem) usun(wsk->lewy, x);
5       else if (x > wsk->elem) usun(wsk->prawy, x);
6       else {
7            if (wsk->lewy == NULL) {
8                 wezel* usuwany = wsk;
9                 wsk = wsk->prawy; // zastap prawym, nawet
                       jesli NULL
10                delete usuwany;
11           }
12           else if (wsk->prawy == NULL) {
13                wezel* usuwany = wsk;
14                wsk = wsk->lewy; // lewy nie jest NULL;
15                delete usuwany;
16           }
17           else
18           { // ma obydwu potomkow
19
```

```
20                // zastapimy "sprytnie" usuwany wezel jego
                     prawym potomkiem
21
22                // znajdz najwiekszy element mniejszy niz
                     usuwany->elem
23                wezel* wsk2 = wsk->lewy;
24                while (wsk2->prawy != NULL)
25                     wsk2 = wsk2->prawy;
26
27                // elementy wieksze od wsk2->elem, ale
                     mniejsze niz
28                // wsk->prawy->elem beda "adoptowane" przez
                     wsk2,
29                // ktore bedzie znajdowac sie gdzies w lewym
                     podrzewie wsk
30                wsk2->prawy = wsk->prawy->lewy;
31
32                // powiekszone lewe wsk podrzewo przenosimy
                     jako lewe poddrzewo
33                // prawego potomka wsk, zeby wlasnosci drzewa
34                // binarnego zostaly zachowane
35                wsk->prawy->lewy = wsk->lewy;
36
37
38                // zapamietaj element usuwany
39                wezel* usuwany = wsk;
40
41                // prawy potomek usuwanego wskakuje na miejsce
                     swego rodzica
42                wsk = wsk->prawy;
43
44                delete usuwany;
45           }
46      }
47 }
```

8.3 Ćwiczenia

Zadanie 8.162. Napisz program, który implementuje i testuje (w funkcji `main()`) następujące operacje na liście jednokierunkowej przechowującej wartości typu **`double`**:

a) Wyznaczenie sumy wartości wszystkich elementów.

b) Wyznaczenie sumy wartości co drugiego elementu.
c) Wyszukiwanie danego elementu.
d) Wstawienie elementu na początek listy.
e) Wstawienie elementu na koniec listy.
f) Wstawienie elementu na i-tą pozycję listy.
g) Usuwanie elementu z początku listy. Usuwany element jest zwracany przez funkcję.
h) Usuwanie elementu z końca listy. Usuwany element jest zwracany przez funkcję.
i) Usuwanie elementu o zadanej wartości. Zwracana jest wartość logiczna w zależności od tego, czy element znajdował się na liście, czy nie.
j) Usuwanie i-tego w kolejności elementu. Usuwany element jest zwracany przez funkcję.

Zadanie 8.163. Rozwiąż powyższe zadanie, implementując listę jednokierunkową, która dodatkowo przechowuje wskaźnik na ostatni element.

Zadanie 8.164. Rozwiąż powyższe zadanie, implementując listę dwukierunkową.

Zadanie 8.165. Dla danych dwóch list jednokierunkowych napisz funkcję, która je połączy, np. dla $(1, 2, 5, 4)$ oraz $(3, 2, 5)$ sprawi, że pierwsza lista będzie postaci $(1, 2, 5, 4, 3, 2, 5)$, a druga zostanie skasowana.

Zadanie 8.166. Napisz program, który implementuje i testuje (w funkcji `main()`) stos (LIFO) zawierający dane typu `int`.

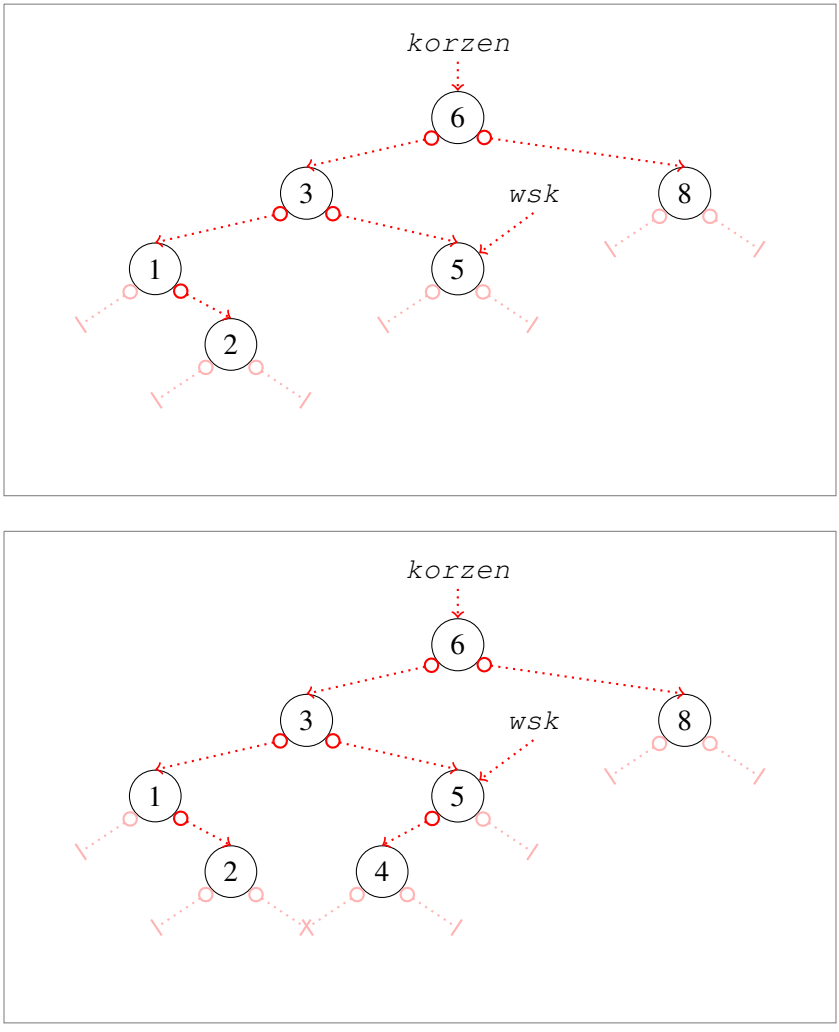
Zadanie 8.167. Napisz program, który implementuje i testuje (w funkcji `main()`) zwykłą kolejkę (FIFO) zawierającą dane typu `char*` (napisy).

(*) Zadanie 8.168. Zaimplementuj operacje `enqueue()` i `dequeue()` zwykłej kolejki (FIFO) typu `int`, korzystając tylko z dwóch gotowych stosów.

Zadanie 8.169. Napisz program, który implementuje i testuje (w funkcji `main()`) kolejkę priorytetową zawierającą dane typu `int`.

Zadanie 8.170. Napisz funkcję, która wykorzysta kolejkę priorytetową do posortowania danej tablicy o elementach typu `int`.

Zadanie 8.171. Napisz program, który implementuje i testuje (w funkcji `main()`) następujące operacje na drzewie binarnym przechowującym wartości typu `double`:
a) Wyszukiwanie danego elementu.
b) Zwrócenie elementu najmniejszego.
c) Zwrócenie elementu największego.
d) Wypisanie wszystkich elementów w kolejności od najmniejszego do największego.
e) Wstawianie danego elementu. Jeśli wstawiany element znajduje się już w drzewie, nie należy wstawiać jego duplikatu.
f) Usuwanie danego elementu. Usuwany element jest zwracany przez funkcję.



Rysunek 8.11. Wstawianie elementu do przykładowego drzewa binarnego.

Wskazówki i odpowiedzi do ćwiczeń

A

Odpowiedź do zadania 1.1.

NWD(42,56) = 14.

```
5: a=42, b=56, c=0
7: a=42, b=56, c=14
8: a=42, b=42, c=14
9: a=14, b=42, c=14
5: a=14, b=42, c=14
7: a=14, b=42, c=0
8: a=14, b=14, c=0
9: a=0, b=14, c=0
5: a=0, b=14, c=0
Wynik: 14
```

NWD(192,348) = 12.

```
5: a=192, b=348, c=0
7: a=192, b=348, c=156
8: a=192, b=192, c=156
9: a=156, b=192, c=156
5: a=156, b=192, c=156
7: a=156, b=192, c=36
8: a=156, b=156, c=36
9: a=36, b=156, c=36
5: a=36, b=156, c=36
7: a=36, b=156, c=12
8: a=36, b=36, c=12
9: a=12, b=36, c=12
5: a=12, b=36, c=12
```

```
7: a=12, b=36, c=0
8: a=12, b=12, c=0
9: a=0, b=12, c=0
5: a=0, b=12, c=0
Wynik: 12
```

NWD(199, 544) = 1.
NWD(2166, 6099) = 57. □

Odpowiedź do zadania 1.2.

```
// Input: (a,b)
// Output: (a',b')=(b,a)
let x be an auxiliary variable;  // zmienna pomocnicza
x = a; a = b; b = x;
```

□

Odpowiedź do zadania 1.3.

```
// Input: (a,b,c)
// Output: (a',b',c')=(c,a,b)
let x be an auxiliary variable;
x = b;
b = a;
a = c;
c = x;
```

□

Odpowiedź do zadania 1.4.

```
// Input: (a,b,c,d)
// Output: (a',b',c',d')=(c,d,b,a)
let x be an auxiliary variable;
x = a;
a = c;
c = b;
b = d;
d = x;
```

□

Odpowiedź do zadania 1.5.
Wynik dla $(1, -1, 2, 0, -2)$: 0.

Wynik dla $(34, 2, -3, 4, 3.5)$: 8,1. □

Odpowiedź do zadania 1.6.
Wynik dla $(1, 4, 2, 3, 1)$: $\frac{60}{37}$.
Wynik dla $(10, 2, 3, 4)$: $\frac{240}{71}$. □

Odpowiedź do zadania 1.7. $\mathrm{SKO}(5, 3, -1, 7, -2) = 59{,}2$.

Podany algorytm wymaga $n^2 + 5n$ operacji arytmetycznych. Nie jest on efektywny, gdyż można go łatwo usprawnić, tak by potrzebnych było $5n + 1$ działań $(+, -, *, /)$. □

Odpowiedź do zadania 2.14.
a) $10_{10} = 1010_2 = A_{16}$.
b) $18_{10} = 10010_2 = 12_{16}$.
c) $18_{16} = 11000_2 = 24_{10}$.
d) $101_2 = 5_{10} = 5_{16}$.
e) $101_{10} = 1100101_2 = 65_{16}$.
f) $101_{16} = 100000001_2 = 257_{10}$.
g) $\mathrm{ABCDEF}_{16} = 101010111100110111101111_2 = 11259375_{10}$.
h) $64135312_{10} = 11110100101010000010010000_2 = 3D2A090_{16}$.
i) $110101111001_2 = D79_{16} = 3449_{10}$.
j) $\mathrm{FFFFF0C2}_{16} = 11111111111111111111000011000010_2 = 4294963394_{10}$.

□

Odpowiedź do zadania 2.16.
a) $10111100_{\mathrm{U2}} = -68$.
b) $00111001_{\mathrm{U2}} = 57_{10}$.
c) $1000000111001011_{\mathrm{U2}} = -32309$.

□

Odpowiedź do zadania 2.17.
a) $-12_{10} = 11110100_{\mathrm{U2}}$.
b) $54_{10} = 00110110_{\mathrm{U2}}$.
c) $-128_{10} = 10000000_{\mathrm{U2}}$.
d) $-129_{10} = 1111111101111111_{\mathrm{U2}}$.
e) $53263_{10} = 00000000000000001101000000001111_{\mathrm{U2}}$.
f) $-32000_{10} = 11111111111111111000001100000000_{\mathrm{U2}}$.
g) $-56321_{10} = 11111111111111110010001111111111_{\mathrm{U2}}$.
h) $-3263411_{10} = 11111111110011100011010001001101_{\mathrm{U2}}$.

□

Odpowiedź do zadania 2.19.
a) $1000_2 + 111_2 = 1111_2$.
b) $1000_2 + 1111_2 = 10111_2$.

c) $1110110_2 + 11100111_2 = 101011101_2$, gdyż

	1	1	1			1	1		
			1	1	1	0	1	1	0
+		1	1	1	0	0	1	1	1
=	1	0	1	0	1	1	1	0	1

d) $11111_2 + 11111111_2 = 100011110_2$.
e) $1111_2 - 0001_2 = 1110_2$.
f) $1000_2 - 0001_2 = 111_2$.
g) $10101010_2 - 11101_2 = 10001101_2$, gdyż

				-1	-1	-1		-1	
	1	0	1	0	1	0	1	0	
-					1	1	1	0	1
=	1	0	0	0	1	1	0	1	

h) $11001101_2 - 10010111_2 = 110110_2$, gdyż

			-1	-1		-1	-1		
	1	1	0	0	1	1	0	1	
-	1	0	0	1	0	1	1	1	
=	0	0	1	1	0	1	1	0	

□

Odpowiedź do zadania 2.20.

a) $1000_{U2} + 0111_{U2} = 1111_{U2} = -1_{10} = -8_{10} + 7_{10}$.
b) $10110110_{U2} + 11100111_{U2} = 110011101_{U2} = 10011101_{U2} = -99_{10} = -74_{10} + (-25)_{10}$.
c) $10101010_{U2} - 00011101_{U2} = 10001101_{U2} = -115_{10} = -86_{10} - 29_{10}$.
d) $11001101_{U2} - 10010111_{U2} = 00110110_{U2} = 54_{10} = -51_{10} - (-105_{10})$.
e) $10001101_{U2} - 01010111_{U2} = 00110110_{U2} = 54_{10} \neq -115_{10} - 87_{10} = -202_{10}$! Wystąpiło tzw. **przepełnienie**, czyli przekroczenie zakresu wartości dla liczby 8-bitowej (ale mamy $2^8 - 202_{10} = 54_{10}$). Można zapisać operandy na większej liczbie bitów, np. szesnastu:

$$1111111110001101_{U2} - 0000000001010111_{U2} = 1111111100110110_{U2} = -202_{10}.$$

□

Wskazówka do zadania 2.22.

```
1 let x ∈ ℕ;
2 return ((x & (x - 1)) == 0);
```

Dlaczego tak jest? Udowodnij poprawność. □

Odpowiedź do zadania 2.23.

```
// Input: n > 0, x[0], x[1], ..., x[n − 1] ∈ {0, 1}
// Output: x w notacji dziesietnej
let ret, pot ∈ ℕ;

ret = 0;
pot = 1;
for (i = 1, 2, ..., n-1)     // to jest pseudokod, nie C++
{
    ret = ret + pot*x[n-i];
    pot = pot * 2;
}
ret = ret - pot * x[0];
return ret;
```

□

Odpowiedź do zadania 2.24.

```
// Input: n > 0 oraz
// Output: x[0], x[1], ..., x[k − 1] ∈ {0, 1}
let k = ⌈log2(n)⌉
let r ∈ ℕ;
let x[0], x[1], ..., x[k − 1] ∈ {0, 1}

for (i = 1, 2, ..., k)
{
    x[n-i] = n modulo 2;
    n = n / 2;
}
ret = ret - pot * x[0];
return x[0], ..., x[k-1];
```

□

Odpowiedź do zadania 3.31.

a) `10.0+15.0/2+4.3 == 21.8` (**double**).
b) `10.0+15/2+4.3 == 21.3` (**double**).
c) `3.0*4/6+6 == 8.0` (**double**).
d) `20.0-2/6+3 == 23.0` (**double**).
e) `10+17*3+4 == 65` (**int**).

f) `10+17/3.0+4` $\simeq$ `19.6667` (**double**).
g) `3*4%6+6 == 6` (**int**).
h) `3.0*4%6+6` — błąd kompilacji.
i) `10+17%3+4 == 16` (**int**).

□

Odpowiedź do zadania 3.34.

a) `0x0FCD | 0xFFFF == -1 (0xFFFF)`.
b) `364 & 0x323 == 288 (0x0120)`.
c) `~163 == -164 (0xFF5C)` (dlaczego -164?).
d) `0xFC93 ^ 0x201D == -9074 (0xDC8E)`.
e) `14 << 4 == 224 (0x00E0)`.
f) `0xf5a3 >> 8 == 0xFFF5` (bit znaku == 1).

□

Odpowiedź do zadania 3.35.

```
1  // Input:
2  double ax = ..., ay = ...; // wspolrzedne punktu a.
3  double bx = ..., by = ...; // wspolrzedne punktu b.
4  double cx = ..., cy = ...; // wspolrzedne punktu c.
5  // Output:
6  double r2; /* kwadrat promienia okregu przechodzacego
7      przez a,b,c. */
8
9  double amcx = ax-cx; //a − c --- wsp. x
10 double amcy = ay-cy; //a − c --- wsp. y
11 double bmcx = bx-cx; //b − c --- wsp. x
12 double bmcy = by-cy; //b − c --- wsp. y
13 double a_c  = amcx*amcx+amcy*amcy; //|a − c|^2
14 double b_c  = bmcx*bmcx+bmcy*bmcy; //|b − c|^2
15 double a_b  = (ax-bx)*(ax-bx)+(ay-by)*(ay-by); //|a − b|^2
16 double ilwekt = amcx*bmcy - amcy*bmcx;
17
18 r2 = a_c*b_c*a_b/4.0/ilwekt/ilwekt;
```

□

Wskazówka do zadania 3.37. Spróbuj zrobić to zadanie nie wprost, czyli sprawdzić, czy prostokąty nie mają części wspólnej. □

Wskazówka do zadania 3.42. Dla liczby k co najwyżej należy sprawdzić, czy dzieli się ona bez reszty przez każdą z liczb $2, 3, \ldots, \sqrt{k}$. □

Wskazówka do zadania 3.51. Do estetycznego sformatowania „tabelki” użyj narzędzi z biblioteki <iomanip>. □

Wskazówka do zadania 3.52. Możesz użyć zagnieżdżonych pętli `for` do sprawdzenia wszystkich możliwych podstawień wartości logicznych pod zmienne p, q, r:

```
for (int p=0; p<=1; ++p)    // bool(0) to false, bool(1) to
    true
    for (int q=0; q<=1; ++q)
        for (int r=0; r<=1; ++r)
            // sprawdzenie warunku...
```

Ponadto przydatną może (ale nie musi) okazać się instrukcja `break`. □

Odpowiedź do zadania 3.53.

a) Jest to funkcja identycznościowa, tzn. $f(n) = n$.
b) Program będzie zerował ostatni bit podanej liczby.
c) Jeżeli drugi bit liczby `n` jest równy `1`, wartość `r` nigdy nie przekroczy wartości `n` i program wpadnie w nieskończoną pętlę.

□

Odpowiedź do zadania 3.54.

a) Program liczy zera znaczące.
b) Nieskończona pętla (bo `-1>>i` jest równe `-1`), błędne liczenie `w` dla bitu znaku.

□

Odpowiedź do zadania 4.74. $u(x) = x^8+x^7+4x^6+3x^5+11x^4+2x^3+40x^2-10x+20$. □

Wskazówka do zadania 4.75. Należy wyprowadzić wzory (rekurencyjne) na średnią arytmetyczną i wariancję k początkowych elementów ciągu, zależne od elementu x_k oraz średniej arytmetycznej i wariancji elementów $(x_1, \ldots, x_{k-1})$. □

Wskazówka do zadania 4.76. Jeśli mamy $a \leqslant b \leqslant c$, to warunkiem koniecznym i dostatecznym tego, by dało się skonstruować trójkąt o bokach długości a, b, c, jest $c < a+b$.

Podpowiedź: nie trzeba sprawdzać wszystkich możliwych trójek! :-) □

Wskazówka do zadania 4.81. Skorzystaj z pomocniczej n-elementowej tablicy liczb całkowitych. □

Odpowiedź do zadania 6.134.

```
1 int strlen(char *s)
2 {
```

```
3    int i = 0;
4    while (s[i]!=0) i++;
5    return i;
6  }
```

```
1  char* strcpy(char *cel, char *zrodlo)
2  {
3    // zalozenie - pamiec dla tablicy przydzielona przed
       wywolaniem funkcji
4    // (cel = new char[strlen(zrodlo)+1];)
5
6    int i = 0;
7    while (zrodlo[i]!=0)
8    {
9       cel[i] = zrodlo[i];
10      i++;
11   }
12   cel[i] = '\0'; // wstaw "wartownika"
13
14   return cel;
15 }
```

```
1  int strcmp(char *s1, char *s2)
2  {
3    int i = 0;
4    int j = 0;
5    while (s1[i]!=0 && s1[i] == s2[j])
6    {
7       i++;
8       j++;
9    }
10
11   return s1[i] - s2[j];
12 }
```

```
1  char* strcat(char* cel, char* zrodlo)
2  {
3    // zalozenie - napis w tablicy cel ma n znakow,
4    // napis w tablicy zrodlo ma m znakow,
5    // pamiec przydzielona na tablice cel wystarczy na n+m+1
       znakow
```

```
6     // (bajt zerowy w cel nie musi wystepowac przeciez na
          koncu tablicy!)
7     int n = strlen(cel); // cel[n] == '\0' (!)
8     int j = 0;
9     while (zrodlo[j]!=0)
10    {
11        cel[n] = zrodlo[j];
12        n++;
13        j++;
14    }
15    cel[n] = '\0';
16
17    return cel;
18 }
```

```
1  char* strchr(char* nap, char znak)
2  {
3     int i=0;
4     while (nap[i] != 0)
5     {
6        if (nap[i] == znak)
7           return &(nap[i]); // znaleziony - zwracamy
                podnapis
8
9        i++;
10    }
11
12    // tutaj stwierdzamy, ze znak wcale nie wystepuje w
          napisie
13    return NULL; // albo return 0; - adres zerowy to "
          nigdzie"
14 }
```

```
1  char* strrchr(char* nap, char znak)
2  {
3     int i=strlen(nap)-1; // zaczynamy od ostatniego znaku
          drukowanego
4     while (i>=0)
5     {
6        if (nap[i] == znak)
7           return &(nap[i]); // znaleziony - zwracamy
                podnapis
```

```
8
9       i--;
10    }
11
12    return NULL;
13 }
```

```
1  char* strstr(char* nap1, char* nap2)
2  {
3     int n = strlen(nap1);
4     int m = strlen(nap2);
5
6     assert(n>=m);
7
8     int i, j;
9     for (i=0; i<n-m+1; ++i)
10    {
11       for (j=0; j<m; ++j)
12       {
13          if (nap1[i+j] != nap2[j]) break; // przerywamy
                petle
14       }
15
16       if (j == m) // doszlismy do konca petli => znaleziony
             !
17          return &(nap1[i]);
18    }
19
20    return NULL;
21 }
```

□

Wskazówka do zadania 7.145. Wyznacznik macierzy 4×4 można policzyć ze wzoru

$$\det A = \sum_{i=1}^{4} (-1)^{i+j} a_{ij} \det A_{i,j},$$

gdzie j jest dowolną liczbą ze zbioru $\{1, 2, 3, 4\}$, a $A_{i,j}$ jest podmacierzą 3×3 powstałą przez opuszczenie i-tego wiersza i j-tej kolumny. Do wyznaczenia $\det A_{i,j}$ można skorzystać z nieco zmodyfikowanej funkcji z poprzedniego zadania, której należy przekazać A, i oraz j. □

Wskazówka do zadania 8.168. Dane są stosy `A` oraz `B`.

Operacja `enqueue()`: odłóż element na stos `A`.

Operacja `dequeue()`: zdejmij element ze stosu `B`. Jeśli stos jest pusty, przerzuć na niego wszystkie elementy ze stosu `A`, korzystając z metod `pop()` i `push()`. □

Literatura

B

Algorytmika

- Harel. D (2008). *Rzecz o istocie informatyki. Algorytmika*, WNT, Warszawa.
- Wirth N. (2004). *Algorytmy + struktury danych = programy*, WNT, Warszawa.

Język C++ (propozycje)

- Deitel H. M., Deitel P. J. (1998). *ARKANA. C++ programowanie*, RM, Warszawa.
- Grębosz J. (2008). *Symfonia C++ Standard*, Editions 2000, Kraków.
- Prata S. (2006). *Język C++. Szkoła programowania*, Helion, Gliwice.
- Snaith P. (2000). *C++ nie tylko dla orłów*, Intersoftland, Warszawa.
- Schildt H. (2002). *Programowanie: C++*, RM, Warszawa.
- Liberty J. (2002). *C++ dla każdego*, Helion, Warszawa.
- Struzińska-Walczak A., Walczak K. (2001). *Nauka programowania dla początkujących. C++*, W&W, Warszawa.

Literatura uzupełniająca

- Knuth D. E. (2002). *Sztuka programowania. Tom I. Algorytmy podstawowe*, WNT, Warszawa.
- Graham R. L., Knuth D. E., Patashnik O. (2006). *Matematyka konkretna*, WN PWN, Warszawa.
- Bronson G. J. (2010). *C++ for Engineers and Scientists*. Course Tech., Boston.