Wydział Matematyki i Nauk Informacyjnych

Politechnika Warszawska

Algorytmy i podstawy programowania - wykłady

Marek Gągolewski

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

PROGRAM ROZWOJOWY POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

I. Etapy tworzenia oprogramowania. Algorytm

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Etapy tworzenia oprogramowania

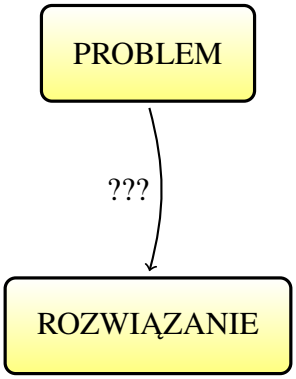
1.1 Sformułowanie i analiza problemu

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

- obliczenie wartości jakiegoś 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 przedmiotu Algorytmy i Podstawy Programowania,
- rozwiązanie dylematu filozoficznego, np. czym jest szczęście i jak być szczęśliwym?

Znalezienie **rozwiązania** danego problemu jest często dla nas bardzo istotne. Rzecz jasna, kluczowym pytaniem jest: jak to zrobić?

Sytuację tę obrazuje rys. 1.



Rysunek 1: Punkt wyjścia naszych rozważań.

Po dość abstrakcyjnym sformułowaniu problemu przechodzimy do jego **analizy**. Tutaj doprecyzowujemy, o co nam naprawdę chodzi, co rozumiemy pod pewnymi pojęciami, jakich wyników się spodziewamy i do czego ewentualnie mogą one się nam przydać.

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 wyników, które są jednoznaczne itd.

Jednakże inne zagadnienia mogą przytłaczać swoją złożonością. Na przykład „zaliczenie przedmiotu AiPP" wymaga określenia, jaki stan końcowy jest pożądany (ocena bardzo dobra?), jakie czynności są kluczowe do osiągnięcia rozwiązania (udział 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?).

1.1.1 Komputery

Istotną cechą niektórych problemów jest to, że nadają się do rozwiązania za pomocą *komputera*. Wbrew pozorom, jak pokazuje rozwój współczesnej informatyki, takich zagadnień jest wcale niemało. Często też mamy do czynienia z sytuacją, w której komputery mogą pomóc uzyskać **rozwiązanie częściowe** lub zgrubne przybliżenie rozwiązania, które może 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 od kilkudziesięciu lat opowiadania *science-fiction*, których autorzy przepowiadają, że te maszyny — kiedyś — będą potrafiły zrobić prawie wszystko, co jest do pomyślenia.

Niestety, oprócz tego, komputery podlegają trzem następującym ograniczeniom. By łatwiej można było je sobie uzmysłowić, posłużymy się analogią z dziedziny motoryzacji.

– Komputer ma ograniczoną moc obliczeniową. Każda instrukcja wykonuje się przez pewien czas. Im bardziej złożone zadanie, tym jego rozwiązywanie trwa dłużej. Wraz z rozwojem techniki, sytuacja ta jednak się poprawia. (Samochody mają np. ograniczoną prędkość, ograniczone przyspieszenie).

– 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 odstępstw lub toleruje tylko nieliczne. (W samochodach, aby zwiększyć obroty silnika, należy wykonać jedną ściśle określoną czynność — wcisnąć mocniej pedał gazu. Nic nie da uśmiechnięcie się bądź próby sympatycznej konwersacji na temat zalet jazdy z inną prędkością).

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

Celem głównym naszych wszystkich rozważań 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 po prostu oznacza *obliczać*. Jednak powiedzenie, że komputer zajmuje się tylko obliczaniem, to stanowczo za wąskie spojrzenie.

Definicja 1. 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 wykonywania licznych prób m.in. przez biologów molekularnych i fizyków kwantowych, współczesne komputery to w znakomitej większości maszyny **elektroniczne**. O implikacjach tego faktu dowiemy się więcej z drugiego wykładu poświęconego organizacji i działaniu tych urządzeń.

Dalej, jednostkę **informacji** rozumiemy jako ciąg liczb lub symboli. 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 naturalnych), 011100 (ciąg liczb binarnych), (”WARSZAWA”, 52°13’56”N, 21°00’30”E) (współrzędne GPS pewnego miejsca na mapie).

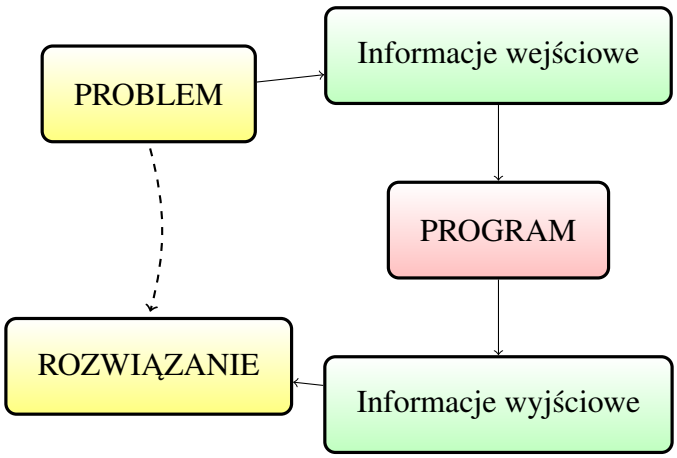
Wartym zanotowania faktem jest to, iż wiele obiektów spotykanych na co dzień może mieć swoje **reprezentacje** w postaci ciągów liczb lub symboli. Często te reprezentacje nie muszą odzwierciedlać ich wszystkich cech, lecz tylko te, które są potrzebne w danym zagadnieniu. Ciąg (”Jola Kowalska”, ”Matematyka 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 jest zbędnym szczegółem.

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

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

są przykładowymi operacjami przetwarzającymi, odpowiednio: liczby naturalne, wartości logiczne i napisy.

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



Rysunek 2: Rozwiązanie problemu za pomocą programu komputerowego.

Przyjrzyjmy się diagramowi na rys. 2. Otóż **problem, który da się rozwiązać za pomocą komputera** posiada dwie istotne cechy:

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 Projektowanie

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

Definicja 2. Algorytm to abstrakcyjny przepis (proces, metoda, „instrukcja obsługi") pozwalający na uzyskanie, za pomocą skończonej liczby działań, oczekiwanych danych wyjściowych na podstawie poprawnych danych wejściowych.

Przykładowym algorytmem „z życia" jest przepis na ciasto marchewkowe,[1] przedstawiony w tab. 1.

Widzimy, jak wygląda *zapisany* algorytm. Ważną umiejętnością, którą będziemy ćwiczyć, jest odpowiednie jego *przeczytanie*. Dobrze to opisał D. E. 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!

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;

– ściśle określone dane wejściowe pochodzące z odpowiednio określonych zbiorów;

– dane wyjściowe, czyli wartości powiązane z danymi wejściowymi, powinny odpowiadać specyfikacji oczekiwanego poprawnego rozwiązania;

– efektywność — w algorytmie powinno unikać się operacji niepotrzebnie wydłużających czas wykonania, w miarę możliwości należy zastępować je prostszymi bądź w ogóle je usuwać.

[1] Przepis ten pochodzi z serwisu Kwestia Smaku:
http://www.kwestiasmaku.com/desery/ciasta/ciasto_marchewkowe/przepis.html.

Tablica 1: Przepis na ciasto marchewkowe.

Dane wejściowe: Ciasto: 2 jaja, 200 g brązowego cukru, 150 ml oleju roślinnego, 200 g drobno startej marchewki, 50 g posiekanych orzechów włoskich, 75 g drobno pokrojonego ananasa (świeżego lub z puszki), 50 g wiórków kokosowych, 200 g mąki, po 1 łyżeczce: cynamonu, sody, soli, 1/2 łyżeczki proszku do pieczenia. Polewa: 400 g cukru pudru (można mniej), 100 g kremowego serka, np. Philadelphia, 50 g masła.

Dane wyjściowe: Ciasto marchewkowe.

Wykonanie:
Polewa:

a) Mikserem ucieram serek wraz z masłem.
b) Dodaję cukier puder, w 3 częściach, cały czas ucierając pomiędzy kolejnymi partiami.
c) Wstawiam do lodówki, aby polewa lekko stężała.

Ciasto:

a) Ubijam jajka do podwojenia objętości. Dodaję cukier i dalej ubijam aż masa będzie gładka i puszysta. Wciąż ubijając na wysokich obrotach, dolewam ciągłym, cieniutkim strumieniem olej.
b) Do powstałej masy dodaję marchewkę, ananasa, orzechy, wiórki kokosowe i delikatnie mieszam.
c) Dodaję przesianą mąkę, cynamon, sodę, proszek do pieczenia i sól, delikatnie łączę wszystkie składniki. Przekładam do małej formy 21 x 21 cm, wyłożonej papierem do pieczenia.
d) Piekę przez 1 godzinę, w piekarniku nagrzanym do 150 stopni.
e) Wystudzone ciasto przekrawam poziomo na 2 części. Spód smaruję 1/3 ilości polewy. Przykrywam górą ciasta i smaruję resztą polewy. Wstawiam do lodówki.

Wygląda na to, że nasz przepis na ciasto marchewkowe posiada wszystkie te cechy. Wykonując powyższe czynności ciasto to uda nam się kiedyś zjeść (a więc jest skończony). Wszystkie czynności są zrozumiałe nawet dla mało wprawionej gospodyni. Dane wejściowe i wyjściowe są dobrze wyspecyfikowane. Sam proces przygotowania jest efektywny (nie każe np. kucharce umalować się w trakcie mieszania składników bądź pojechać na zagraniczną wycieczkę).

Jeden program rozwiązujący rozpatrywany problem może zawierać realizację **wielu algorytmów**, np. gdy złożoność zagadnienia wymaga podzielenia go na kilka **podproblemów**. I tak zdolna kucharka wykonująca program „obiad" powinna podzielić swą pracę na podprogramy „rosół", „kotlet schabowy z frytkami" oraz „ciasto marchewkowe",

Ważne jest, że algorytm nie musi być wyrażony za pomocą języka zrozumiałego przez komputer. Taki sposób opisu algorytmów nazywa się często **pseudokodem**. Stanowi on etap pośredni między analizą problemu a implementacją, opisaną w kolejnym paragrafie. Ma on nam po prostu pomóc w bardziej formalnym podejściu do tworzenia programu.

Dla przykładu, w przytoczonym powyżej przepisie, nie jest dokładnie wytłumaczone — krok po kroku — co oznacza „piekę przez jedną godzinę". Wszak wymaga ono wielu czynności (włączenie piekarnika, rozgrzanie, pilnowanie ustawionego zegarka 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.

1.2.1 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 $a, b \in \mathbb{N}$ (oryginalnie było to $a = 1$ i $b = 100$). Zapewne myślał on, że tamten użyje algorytmu I i spędzi nad tym trochę czasu.

Algorytm I wyznaczania sumy kolejnych liczb naturalnych.

```
// Wejście: a,b ∈ ℕ (a < b)
// Wyjście: a+a+1+...+b ∈ ℕ

niech suma,i ∈ ℕ;
suma = 0;

dla (i = a,a+1,...,b)
{
    suma = suma + i;
}
zwróć suma jako wynik;
```

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

Algorytm II wyznaczania sumy kolejnych liczb naturalnych.

```
// Wejście: a, b ∈ ℕ (a < b)
// Wyjście: a + a + 1 + ... + b ∈ ℕ

niech suma ∈ ℕ;
suma = (a+b)/2 (b − a + 1);
zwróć suma jako wynik;
```

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

Tablica 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

Charakteryzacją efektywności algorytmów zajmuje się dziedzina zwana **analizą algorytmów**. Czerpie ona szeroko z wyników takich działów jak matematyka dyskretna czy rachunek prawdopodobieństwa. Z jej elementami zapoznamy się podczas innych wykładów.

1.3 Implementacja

Na kolejnym etapie, abstrakcyjne algorytmy zapisane często w postaci pseudokodu (np. za pomocą języka polskiego) należy zapisać w formie, która jest *zrozumiała* nie tylko przez nas, ale i *przez komputer*. Nadto, należy dokonać scalenia (powiązania) podrozwiązań w jeden spójny projekt.

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

Formalnie rzecz ujmując, zbiór zasad określających, jaki ciąg symboli tworzy kod źródłowy programu, nazywamy **językiem programowania**. **Reguły składniowe** (ang. *syntax*) 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*.

Podczas tego wykładu będziemy poznawać **język C++**, zaprojektowany ok. 1983 r. przez B. Stroustrupa (aktualny standard: ISO/IEC 14882:2003). Język ten powstał jako rozwinięcie języka C i jest przedstawicielem klasy języków cechujących się podobną doń składnią. Wśród innych podobnych można wymienić inne popularne narzędzia, takie jak Java, C#, PHP czy

JavaScript. Widzimy zatem, że C++ jest tylko jedną z wielu możliwości wydawania poleceń komputerowi, jednakże spośród innych wyróżnia się on zwięzłością, efektywnością i względną łatwością nauki.

Kod źródłowy zapisujemy zwykle w postaci kilku plików tekstowych (tzw. plików źródłowych, ang. *source files*). Pliki te można edytować za pomocą dowolnego edytora tekstowego, np. *Notatnik* systemu Windows. Mimo to wygodniejsze jest korzystanie z całych środowisk programistycznych. My podczas laboratoriów będziemy używać *Microsoft Visual C++*[2].

I wreszcie narzędziem, które pozwala przetworzyć pliki źródłowe (zrozumiałe przez człowieka i komputer) na kod maszynowy programu komputerowego (zrozumiały tylko przez komputer) nazywamy **kompilatorem** (ang. *compiler*).

1.4 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 pracy, a nie liczyć na to, że jakiś chochlik robi nam na złość. Jak powiedzieliśmy, komputer robi tylko to, co my mu każemy. Zatem jeśli mu nakazaliśmy wykonać instrukcję, której skutków ubocznych nie jesteśmy do końca pewni, jest to nasza odpowiedzialność, żeby te skutki opanować.

1.5 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 znów powrót do wcześniejszych etapów pracy.

Jak widać, nauka programowania komputerów może nam pomóc rozwiązać wiele problemów, których rozwiązanie bez nich byłoby często niedostępne. Poza tym jest wspaniałą rozrywką!

1.6 Podsumowanie

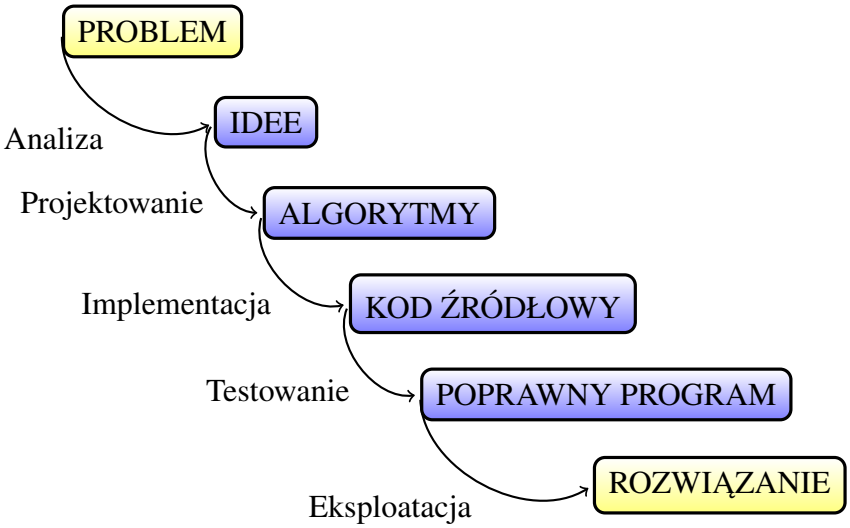
Omówiliśmy następujące etapy tworzenia oprogramowania, które są niezbędne do rozwiązania zagadnień za pomocą komputera:

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

[2]Do pobrania bezpłatnie ze strony http://www.microsoft.com/express/Downloads/.

Efekty pracy po każdym z etapów podsumowuje rys. 3.

Godne polecenia rozwinięcie omawianego tematu można znaleźć w książce Harela (2001, s. 9–31).



Rysunek 3: Efekty pracy po każdym z etapów tworzenia oprogramowania.

2 Ćwiczenia

Zadanie 1.1. Dany jest algorytm Euklidesa znajdywania największego wspólnego dzielnika (NWD) dwóch liczb $a, b \in \mathbb{Z}$, $0 \leqslant a < b$.

```
// Wejście: 0 ⩽ a < b
// Wyjście: NWD(a, b)

niech c ∈ ℕ;
dopóki (a ≠ 0)
{
    c = reszta z dzielenia b przez a;
    b = a;
    a = c;
}
zwróć b jako wynik;
```

Prześledź działanie algorytmu Euklidesa (jakie wartości przyjmują zmienne a, b, c w każdym kroku) znajdywania największego wspólnego dzielnika dla następujących par liczb

a) 42, 56,
b) 192, 348,
c) 199, 544,
d) 2166, 6099.

Zadanie 1.2. Pokaż, w jaki sposób za pomocą ciągu przypisań można przestawić wartości dwóch zmiennych (a, b), by otrzymać (b, a).

Zadanie 1.3. Pokaż, w jaki sposób za pomocą ciągu przypisań można przestawić wartości trzech zmiennych (a, b, c), by otrzymać (c, a, b).

Zadanie 1.4. Pokaż, w jaki sposób za pomocą ciągu przypisań można przestawić wartości czterech zmiennych (a, b, c, d), by otrzymać (c, d, b, a).

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

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

Rozważmy następujący algorytm służący do jej wyznaczania.

```
// Wejście: n > 0 oraz x[0], x[1], ..., x[n − 1] ∈ ℝ
// Wyjście: średnia arytmetyczna elementów x[0], x[1], ..., x[n − 1])
niech sredniaarytm ∈ ℝ;
niech i ∈ ℕ;
sredniaarytm = 0;
dla (i=0,1,...,n−1)
    sredniaarytm = sredniaarytm + x[i];
sredniaarytm = sredniaarytm / n;
zwróć sredniaarytm jako wynik;
```

Wyznacz za pomocą powyższego algorytmu wartość średniej arytmetycznej dla ciągów $(1, -1, 2, 0, -2)$ oraz $(34, 2, -3, 4, 3{,}5)$.

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

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

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

$\star$ **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, określoną jako

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

Rozważmy następujący algorytm służący do wyznaczania SKO.

```
1  // Wejście: n > 0 oraz x[0], x[1], ..., x[n-1] ∈ ℝ
2  // Wyjście: SKO(x[0], x[1], ..., x[n-1])
3  niech sko, sredniaarytm ∈ ℝ;
4  niech i, j ∈ ℕ;
5
6  sko = 0;
7  dla (i=0, 1, ..., n-1)
8  {
9      sredniaarytm = 0;
10     dla (j=0, 1, ..., n-1)
11     {
12         sredniaarytm = sredniaarytm + x[j];
13     }
14     sredniaarytm = sredniaarytm / n;
15
16     sko = sko + (x[i] - sredniaarytm)*(x[i] - sredniaarytm);
17 }
18 zwróć sko jako wynik;
```

a) Wyznacz za pomocą powyższego algorytmu 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 teraz będzie potrzebnych operacji arytmetycznych?

3 Wskazówki i odpowiedzi do ćwiczeń

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.

```
1 // Wejście: (a,b)
2 // Wyjście: (a',b')=(b,a)
3 niech x — zmienna pomocnicza;
4 x = a; a = b; b = x;
```

Odpowiedź do zadania 1.3.

```
// Wejście: (a,b,c)
// Wyjście: (a',b',c')=(c,a,b)
niech x - zmienna pomocnicza;
x = b;
b = a;
a = c;
c = x;
```

Odpowiedź do zadania 1.4.

```
// Wejście: (a,b,c,d)
// Wyjście: (a',b',c',d')=(c,d,b,a)
niech x - zmienna pomocnicza;
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 $(+, -, *, /)$.

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

II. Podstawy organizacji i działania komputerów

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Historia i organizacja współczesnych komputerów

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 istotniejszych wydarzeń, które pomogą nam rzucić światło na przedmiot naszych zainteresowań.

- Ok. 2400 r. p.n.e. używany jest już w Babilonii 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 znany nam algorytm przypisywany jest Euklidesowi (ok. 400-300 r. p.n.e.). Opisał on operacje, których wykonanie krok po kroku pozwala wyznaczyć największy wspólny dzielnik dwóch liczb.

- Matematyk arabski Al Kwarizmi (IX w.) określa reguły podstawowych operacji arytmetycznych dla liczb dziesiętnych. Od jego nazwiska powstaje potem pojęcie algorytmu.

- 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 wykonywanie obliczeń.

- W. Schickard (1623 r.) oraz B. Pascal (1645 r.) tworzą pierwsze mechaniczne sumatory.

- 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 jednak 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 pierwszą programistkę.

- H. Hollerith (1890) — maszyna wspomagająca spis powszechny w USA.

- Niemiecki inżynier w 1918 r. patentuje maszynę szyfrującą Enigma. (Nota bene polski matematyk Marian Rejewski w 1934 r. łamie jej kod jako pierwszy).

– 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, która mówi o tym, że jeśli dla jakiegoś problemu istnieje efektywny algorytm korzystający z nieograniczonych zasobów, to da się go wykonać na stworzonym przez nich prostym 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").

– Claude E. Shannon w swojej pracy magisterskiej w 1937 r. opisuje możliwość użycia elektronicznych przełączników do wykonywania operacji logicznych (algebra 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. 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, np. Z3 wykonywał jedno mnożenie w 3 sekundy. Pierwsze zastosowanie: łamanie szyfrów, obliczanie trajektorii lotów balistycznych.

– 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).

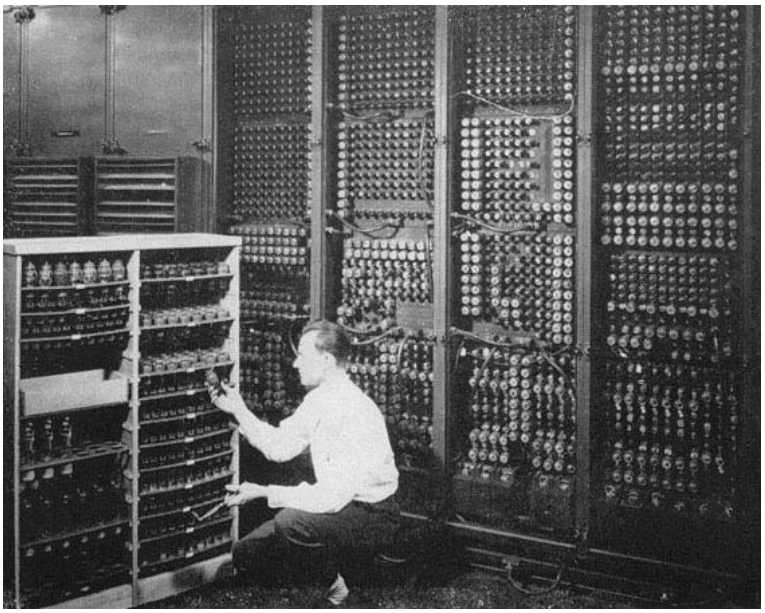
– Pierwszy kompilator języka Fortran zostaje uruchomiony w 1957 r. Kompilator języka LISP powstaje rok później.

– 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 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 osobisty (o mocy kilka miliardów razy większej niż ENIAC) podłączony do sieci Internet znajdziemy w prawie każdym domu, a elementy informatyki wykładane są już w szkołach podstawowych!



Rysunek 1: Komputer ENIAC, źródło: http://www.ifj.edu.pl/str/psk/img/eniac4.jpg.

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

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 znajdywania 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?)

c) Algorytmy i struktury danych

   i. Analiza algorytmów (ze względu na czas działania i potrzebne zasoby)
   ii. Projektowanie algorytmów
   iii. Struktury danych (organizacja informacji)
   iv. Algorytmy genetyczne (znajdywanie rozwiązań przybliżonych problemów optymalizacyjnych)

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

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, organizacja komputerów, także z punktu widzenia elektroniki)
   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. Uczenie się maszynowe (tworzenie zestawów reguł w oparciu o informacje wejściowe)

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)

Do tej listy można też dopisać wiele dziedzin z tzw. pogranicza informatyki, np. zbiory rozmyte, bioinformatykę, statystykę obliczeniową itd.

Na koniec tego paragrafu warto zwrócić uwagę (choć zdania na ten temat są podzielone), że to, co 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. „Informatyka jest nauką o komputerach w takim stopniu jak astronomia nauką o teleskopach" (Dijkstra).

1.2 Organizacja komputerów

Jak zdążyliśmy nadmienić, podstawy teoretyczne działania współczesnych elektronicznych komputerów zawarte zostały w pracy C. Shannona (1937 r.). Mogą być one pojmowane jako zestaw odpowiednio sterowanych **przełączników**, tzw. bitów (ang. *binary digits*). Stany tych przełączników mogą być dwa:

- prąd nie płynie (co oznaczamy jako 0),
- prąd płynie (co oznaczamy przez 1).

Na współczesny komputer osobisty (PC) składają się:

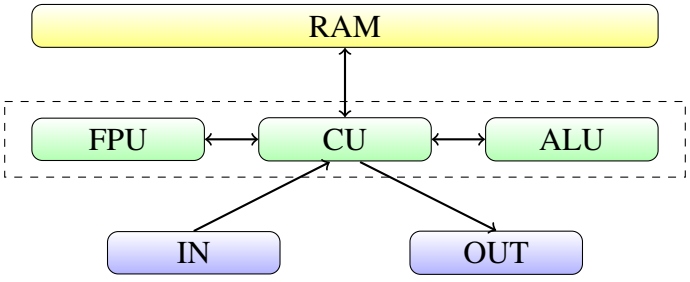
a) jednostka obliczeniowa (CPU, od ang. *central processing unit*):
   - jednostki arytmetyczno-logiczne (**ALU**, ang. *arithmetic and logical units*),
   - jednostka do obliczeń zmiennopozycyjnych (**FPU**, ang. *floating point unit*),
   - układy sterujące (**CU**, ang. *control units*),
   - rejestry (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). 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: Architektura współczesnych komputerów osobistych.

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.

2.1 System dziesiętny

System dziesiętny (decymalny, ang. *decimal*), przyjęty w Europie od 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 o liczbę w systemie dziesiętnym). Jak w każdym systemie pozycyjnym istotne jest tu, na której pozycji 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**	$\times 10^0$

Numerację pozycji cyfr zaczynamy od 0. Dalej oznaczamy je w kierunku od prawej do lewej, czyli od najmniej do najbardziej znaczącej.

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

Ogólnie rzecz biorąc, mając daną liczbę n cyfrową, zapisaną jako *ciąg* 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 wprowadzić systemy pozycyjne o innych podstawach. My szczególnie skupimy się na systemach o podstawie 2 i 16, które są przydatne z punktu widzenia informatyki.

2.2 System dwójkowy

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

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 tablica 1, przedstawiająca wybrane potęgi liczby 2.

Tablica 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
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

Konwersja liczb do systemu dziesiętnego jest stosunkowo prosta. Przyjrzyjmy się jak przekształcić liczbę dziesiętną na dwójkową. Listing 1 podaje algorytm, który może być w tym celu wykorzystany.

Dla przykładu, rozważmy liczbę 1945_{10}. Tablica 2 opisuje kolejno wszystkie kroki potrzebne do uzyskania wyniku w postaci dwójkowej. Możemy z niej odczytać, iż $1945_{10} = 11110011001_2$.

Listing 1: Konwersja liczby dziesiętnej na dwójkową.

```
// Wejscie: liczba k ∈ ℕ.
// Wyjście: postać tej liczby w systemie dwójkowym (kolejne
   cyfry będą ,,wypisywane'' w kolejności od lewej do prawej).
niech i – największa liczba naturalna taka, że k ⩾ 2^i;
dopóki (i ⩾ 0)
{
    jeśli (k ⩾ 2^i)
    {
        wypisz(1);
        k = k – 2^i;
    }
    w_przeciwnym_przypadku
        wypisz(0);

    i = i – 1;
}
// (wynik został ,,wypisany'')
```

Tablica 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			↑

2.3 System szesnastkowy

System szesnastkowy (heksadecymalny, ang. *hexadecimal*) jest systemem pozycyjnym o podstawie 16. Używanymi symbolami (cyframi) są: 0,1,...,9,A,B,C,D,E,F. Jak widzimy, wykorzystywać będziemy także kolejne litery alfabetu.

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

Tablica 3: Cyfry systemu szesnastkowego i ich wartości w systemie dwójkowym oraz dziesiętnym.

BIN	DEC	HEX	BIN	DEC	HEX
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

użyć jednego symbolu zamiast 4 w systemie dwójkowym. Fakt ten sprawia również, że konwersja z systemu binarnego na heksadecymalny i odwrotnie jest bardzo prosta.

Tablica 3 przedstawia cyfry systemu szesnastkowego i ich wartości w systemie dwójkowym oraz dziesiętnym.

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

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

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 znaczenia (wyjątek w § 2.4!).

Na koniec, konwersji na i z postaci dziesiętnej możemy dokonywać za pośrednictwem opanowanej już przez nas postaci binarnej.

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

W tej notacji najstarszy bit determinuje znak liczby. I tak:

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

Jeśli dana jest n cyfrowa liczba w systemie U2 postaci $b_{n-1}b_{n-2}\ldots b_1b_0$, gdzie $b_i \in \{0,1\}$ dla $i = 0, 1, \ldots, n-1$, 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}.
Dla przykładu, $0110_{U2} = 110_2$ 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 $0b_{n-2}\ldots b_{0\mathrm{U}2} \neq b_{n-2}\ldots b_{0\mathrm{U}2}$. Jednakże, $0b_{n-2}\ldots b_{0\mathrm{U}2} = 00\ldots 0b_{n-2}\ldots b_{0\mathrm{U}2}$. Z drugiej strony, można udowodnić, że również $1b_{n-2}\ldots b_{0\mathrm{U}2} = 11\ldots 1b_{n-2}\ldots b_{0\mathrm{U}2}$.

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

3 Reprezentacje liczb rzeczywistych

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

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 $b_{n-1}b_{n-2}\ldots b_k{,}b_{k-1}\ldots b_0$. 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 liczb o (relatywnie) dużej i małej wartości z zadowalającą dokładnością.

3.2 System zmiennoprzecinkowy

Współczesne komputery osobiste 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** liczb rzeczywistych (ang. *floating-point numbers*).

Reprezentowana liczba jest przechowywana w pamięci wg następującego schematu:

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

gdzie:

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

Liczba cyfr mantysy i wykładnika jest ustalona z góry, np. w komputerach osobistych jest to 23+8 (32 bitowa) bądź 52+11 (64 bitowa liczba zmiennoprzecinkowa).

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[2].

Warto jednak zapamiętać kilka problemów związanych z użyciem liczb zmiennopozycyjnych.

a) Nie da się reprezentować całego zbioru liczb rzeczywistych, a tylko jego podzbiór (a nawet właściwie jest to podzbiór liczb wymiernych!). Wszystkie liczby są zaokrąglane do najbliższej reprezentowalnej.
b) Arytmetyka zmiennoprzecinkowa nie spełnia w pewnych szczególnych przypadkach własności łączności i rozdzielności.
c) Dodatkowo, w tym systemie zdefiniowane są trzy specjalne elementy:
   – ∞ (`Inf`, ang. *infinity*),
   – $-\infty$ (`-Inf`),
   – nie-liczba (`NaN`, ang. *not a number*).

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

[2]Zainteresowanym osobom można polecić angielskojęzyczny artykuł: D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys* **21**(1), 1991, 5–48. Do pobrania ze strony http://dlc.sun.com/pdf/800-7895/800-7895.pdf.

4 Ćwiczenia

Zadanie 2.1. 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 2.2. Dana jest n cyfrowa liczba w systemie U2 zapisana jako ciąg cyfr $b_{n-1}b_{n-2}\ldots b_1b_0$, 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. $b_{n-1}b_{n-2}\ldots b_1b_0 = b_{n-1}b_{n-1}\ldots b_{n-1}b_{n-2}\ldots b_1b_0$.

Zadanie 2.3. Przedstaw następujące liczby dane w notacji U2 jako liczby dziesiętne: 10111100, 00111001, 1000000111001011.

Zadanie 2.4. Przedstaw następujące liczby dziesiętne w notacji U2 (do zapisu użyj 8, 16 lub 32 bitów): $-12, 54, -128, -129, 53263, -32000, -56321, -3263411$.

⋆ **Zadanie 2.5.** Rozważmy operacje dodawania i odejmowania liczb nieujemnych w reprezentacji binarnej. Oblicz wartość następujących wyrażeń korzystając z metody analogicznej do sposobu „szkolnego" (dodawania i odejmowanie słupkami). Pamiętaj jednak, że np. $1_2 + 1_2 = 10_2$.

a) $1000_2 + 111_2$,
b) $1000_2 + 1111_2$,
c) $1110110_2 + 11100111_2$,
d) $11111_2 + 11111111_2$,
e) $1111_2 - 0001_2$,
f) $1000_2 - 0001_2$,
g) $10101010_2 - 11101_2$,
h) $11001101_2 - 10010111_2$.

⋆ **Zadanie 2.6.** 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 2.7.** Niech dana będzie 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 za pomocą konwersji tych liczb do systemu dziesiętnego.

5 Wskazówki do ćwiczeń

Odpowiedź do zadania 2.1.

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 2.3.

a) $10111100_{\mathrm{U2}} = -68$.
b) $00111001_{\mathrm{U2}} = 57_{10}$.
c) $1000000111001011_{\mathrm{U2}} = -32309$.

Odpowiedź do zadania 2.4.

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) $53263_{10} = 0000000000000001101000000001111_{\mathrm{U2}}$.
f) $-32000_{10} = 11111111111111111000001100000000_{\mathrm{U2}}$.
g) $-56321_{10} = 11111111111111110010001111111111_{\mathrm{U2}}$.
h) $-3263411_{10} = 11111111110011100011010001001101_{\mathrm{U2}}$.

Odpowiedź do zadania 2.5.

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 2.6.

a) $1000_{\mathrm{U2}} + 0111_{\mathrm{U2}} = 1111_{\mathrm{U2}} = -1_{10} = -8_{10} + 7_{10}$.

b) $10110110_{\mathrm{U2}} + 11100111_{\mathrm{U2}} = 110011101_{\mathrm{U2}} = 10011101_{\mathrm{U2}} = -99_{10} = -74_{10} + (-25)_{10}$.

c) $10101010_{\mathrm{U2}} - 00011101_{\mathrm{U2}} = 10001101_{\mathrm{U2}} = -115_{10} = -86_{10} - 29_{10}$.

d) $11001101_{\mathrm{U2}} - 10010111_{\mathrm{U2}} = 00110110_{\mathrm{U2}} = 54_{10} = -51_{10} - (-105_{10})$.

e) $10001101_{\mathrm{U2}} - 01010111_{\mathrm{U2}} = 00110110_{\mathrm{U2}} = 54_{10} \neq -115_{10} - 87_{10} = -202_{10}$! 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:

$$1111111110001101_{\mathrm{U2}} - 0000000001010111_{\mathrm{U2}} = 1111111100110110_{\mathrm{U2}} = -202_{10}.$$

Aby lepiej zrozumieć to zjawisko, rozważ „kolejne” 4 bitowe liczby w systemie U2.

U2	DEC
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

III. Deklaracja zmiennych w języku C++. Operatory

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Deklaracja zmiennych w C++

W poprzednich rozdziałach, czytając pseudokody algorytmów, często napotykaliśmy na instrukcję podobną do następującej.

```
niech x ∈ ℝ;
```

Tym samym mieliśmy na myśli „niech **od tej pory** x będzie **zmienną** ze zbioru liczb rzeczywistych”.

Instrukcja tego typu to tzw. **deklaracja** zmiennej. Zmienna służy do przechowywania dowolnych wartości ze zbioru, z którego pochodzi. Taki zbiór nazywamy **typem** zmiennej (w przypadku powyższego x było to $\mathbb{R}$).

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

Dzięki zmiennym pisane przez nas programy (ale także np. twierdzenia matematyczne) nie opisują jednego, konkretnego przypadku szczególnego pewnego interesującego nas problemu, lecz wszystkie możliwe w danym kontekście. Dzięki takiemu mechanizmowi mamy więc możliwość **uogólniania** naszych wyników. Przykładowo, 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 (nawet przyszłego matematyka) byłby uznany, łagodnie rzecz ujmując, za nieudany żart. Jednakże uogólnienie powyższych wyników przez odwołanie się do de facto zadeklarowanej zmiennej sprawia, że wynik staje 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 formie. Większość autorów podchodzi do tego, rzecz jasna, w sposób elastyczny. 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.

Wiemy już, że komputery jednak nie są tak domyślne jak my. W języku C++ wszystkie zmienne muszą zostać **zadeklarowane przed** ich **pierwszym użyciem**. Składnia najprostszej instrukcji służącej do osiągnięcia tego celu jest ściśle określona:

```
typ ident;
```

gdzie **typ** jest typem zmiennej (§ 1.1), a *ident* jej identyfikatorem (§ 1.2).

Deklaracja zmiennej powoduje przydzielenie jej pewnego miejsca w pamięci RAM komputera (więcej szczegółów na ten temat na wykładzie o wskaźnikach). Warto zapamiętać, że zmienna nie jest inicjowana automatycznie. Dopóki nie przypiszemy jej jawnie jakiejś wartości, będzie przechowywać „śmieci”.

Uwaga

Każda instrukcja w C++ musi być zakończona **średnikiem**!

1.1 Typy liczbowe proste

W niniejszym paragrafie omówimy dostępne w języku C++ typy zmiennych liczbowych (są to tzw. **typy proste**).

1.1.1 Typy całkowite

Wśród **typów całkowitych** wyróżniamy:

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

My najczęściej będziemy korzystać z typu **int**. Jest to liczba 32-bitowa w systemie U2. Zatem może być ona traktowana jako zbiór **int** $= \{-2^{31}, -2^{31}+1, \ldots, 2^{31}-1\} \subset \mathbb{Z}$.

Co więcej, do każdego z ww. typów można zastosować modyfikator **unsigned** — wtedy otrzymujemy liczbę nieujemną (w zwykłym systemie binarnym). np. **unsigned int** $= \{0, 1, \ldots, 2^{32}-1\}$.

Stałe całkowite, np. 0, -5, 15210, są reprezentantami typu **int** danymi w postaci dziesiętnej. Co więcej, 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.

1.1.2 Typy zmiennoprzecinkowe

Z kolei wśród **typów zmiennoprzecinkowych** wyróżniamy:

Nazwa typu	Liczba bitów[1]
float	32 bity
double	64 bity

Mamy[2] **float** $\subset \bar{\mathbb{R}}$, **double** $\subset \bar{\mathbb{R}}$. W przypadku liczby 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 $1{,}80 \times 10^{308}$.

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 formie **notacji naukowej**, używając do tego celu separatora e, który oznacza „razy dziesięć do potęgi". I tak liczba $-2.32e-4$ jest stałą o wartości $-2{,}32 \times 10^{-4}$.

[1] Podana liczba bitów dotyczy Microsoft Visual C++.

[2] Przypomnijmy, że typy zmiennopozycyjne przechowują też wartości specjalne: `Inf`, `-Inf` oraz `NaN`.

1.1.3 Typ logiczny

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

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

1.2 Identyfikatory

Każda zmienna musi mieć jakąś nazwę, aby można się było do niej odwołać. Identyfikatory mogą się składać 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
```

Zatem przykładami poprawnych identyfikatorów są: *i*, *suma*, *wyrazenieZeWzoru32*, *_2m1a* oraz *zmienna_pomocnicza1*. Z kolei zgodnie z podaną regułą np. 3523*aaa*, *ala ma kota* oraz :−) nie mogą być identyfikatorami.

Identyfikatorami nie mogą być też **zarezerwowane słowa kluczowe** (np. **int**, **double**). Dlatego specjalnie wyróżniamy je w tym tekście pogrubioną czcionką.

Uwaga

W języku C++ wielkość liter ma znaczenie! Np. *I* oraz *i* to identyfikatory dwóch różnych zmiennych. Z tego też powodu identyfikator *Int* nie jest tożsamy z typem **int**.

Na koniec naszych rozważań o identyfikatorach pewna rada praktyczna. Dobrą praktyką jest nadawanie takich nazw zmiennym, by w pewnym sensie były samoobjaśniające się. Zmienne *poleKwadratu*, *delta*, *wspX* w pewnych kontekstach posiadają tę cechę. Często używamy też identyfikatorów *i*, *j*, *k* jako liczników w pętlach. Z drugiej strony, np. *zmienna* albo *ratunku* niezbyt jasno mówią do czego same mogą służyć.

2 Operatory

2.1 Operator przypisania

Operator przypisania, =, służy do nadania zmiennej wartości. Składnia:

```
ident = wyr;
```

gdzie *ident* to identyfikator zmiennej, której wartość chcemy ustalić, a *wyr* to **wyrażenie** (ang. *expression*), którego wykonanie (obliczenie, ewaluacja) zwraca pewną wartość odpowiedniego typu. Wyrażeniami mogą być m.in.

– stała całkowita, zmiennoprzecinkowa, logiczna (np. 7, −4.1*e*7, **false**),

– identyfikator zmiennej bądź stałej symbolicznej (wartość zostanie z niej pobrana),
– wartość zwracana przez funkcję (więcej o funkcjach w kolejnych wykładach),
– wynik zastosowania operatorów arytmetycznych, logicznych, relacyjnych lub bitowych (zob. § 2.3 i dalej) na innych wyrażeniach[3].

Prześledźmy poniższy kod.

```
int i;      // deklaracja zmiennej
int j;
i = 4;      // przypisanie wartości 4, czytaj: i staje się 4
j = i;      /* przypisanie wartości takiej,
               jaką aktualnie przechowuje zmienna i */
cout << j; // wypisze na ekran "4"
```

Uwaga

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. Jednakże opisywanie tworzonego kodu jest bardzo dobrym nawykiem, bo sprawia, że jest on bardziej zrozumiały dla jego czytelnika.

Uwaga

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

Istnieje jeszcze inny sposób przypisywania wartości zmiennym, dzięki któremu dane nie muszą być określone podczas pisania kodu. Informacje te można określić w trakcie działania programu. Można o nie bowiem poprosić użytkownika, aby je podał za pomocą klawiatury (za pośrednictwem obiektu *cin* i operatora >>).

```
int x;
cout << "Podaj wartość x: ";
cin >> x; /* wczytanie wartości x z tzw. standardowego wejścia
             (domyślnie jest to klawiatura komputera) */
cout << "Teraz x=" << x << endl;
```

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

```
const typ ident = wartość;
```

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

Przy tworzeniu stałej symbolicznej należy od razu przypisać jej wartość. Cechą zasadniczą stałych jest to, że raz zainicjowane, nie mogą być później zmienione (warto je 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; // błąd - pi jest "tylko do odczytu"
```

2.2 Operatory rzutowania. Hierarchia typów

Zastanówmy się, co się stanie, jeśli zechcemy przypisać zmiennej jednego typu wartość innego typu. Rozważmy inny przykład:

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

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

x = 3.14159;

j = x;        // błąd! (typ docelowy jest słabszy)
j = (int)x; // jawna konwersja (tzw. rzutowanie) - teraz 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**:

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

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* powiedzieć kompilatorowi, że postępujemy świadomie. Służy do tego tzw. **operator rzutowania**. Taki operator umieszczamy przed rzutowanym wyrażeniem. Oprócz zastosowanej w powyższym przykładzie składni **(typ)***wyrażenie*, dostępne są dodatkowo dwie równoważne: **typ**(*wyrażenie*) oraz **static_cast <typ>**(*wyrażenie*).

Warto zapamiętać, że w przypadku rzutowania typu zmiennopozycyjnego do całkowitoliczbowego następuje **obcięcie części ułamkowej**, np. (**int**)3.14 da w wyniku 3, a (**int**)−3.14 zaś wartość −3.

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

Uwaga

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

2.3 Operatory arytmetyczne

Wśród operatorów arytmetycznych można wyróżnić operatory binarne (wymagające dwóch operandów) oraz unarne (przyjmujące jeden operand). Można je (poza wyszczególnionymi wyjątkami) stosować dla wyrażeń typu całkowitego i zmiennoprzecinkowego.

Dostępne **operatory binarne** przedstawia poniższa tabela.

Operator	Znaczenie
+	dodawanie
−	odejmowanie
*	mnożenie
/	dzielenie
%	reszta z dzielenia (tylko całkowite)

Uwaga

W C++ nie ma 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 typów - to samo, co
    1.0/0.0 */
```

Jak widzimy, zastosowanie operatorów do wyrażeń dwóch różnych typów powoduje konwersję operandu o typie słabszym do typu silniejszego operandu. Wynikiem wykonania rozpatrywanej części 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 stałej 4
i = -i;     // zmiana znaku zmiennej i
cout << i;  // wynik: 4
```

Ponadto zdefiniowane są jeszcze 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.

Zdefiniowane są aż dwa warianty powyższych operatorów: przedrostkowy (ang. *prefix*) oraz przyrostkowy (ang. *suffix*), co — nie wiedzieć czemu — powoduje rokrocznie przerażenie studenckiej braci.

A przecież zasada jest bardzo prosta! W przypadku operatora inkrementacji/dekrementacji **przedrostkowego**, wartością całego wyrażenia jest wartość zmienionej zmiennej (operator ten działa tak, jakby najpierw 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).

Poświęćmy kilka chwil na przeanalizowanie następującego przykładu.

```
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; // błąd, 100 jest stałą
```

Zatem `j = ++i;` można 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.

2.4 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?

Nie należy mylić operatora porównania (==) i operatora przypisania (=) — jest to częsty błąd.

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!
   // Uwaga na porównywanie liczb zmiennoprzecinkowych - błędy!
```

2.5 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)
```

Uwaga

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 jest niepotrzebne. Podobnie jest dla alternatywy i jednego operandu prawdziwego. Na przykład:

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

2.6 Operatory bitowe

Operatory logiczne działają na poszczególnych bitach operandów typu całkowitego (jak już zdążyliśmy się przekonać, każda informacja, w tym i liczby, są przechowywane w pamięci komputera jako ciąg bitów). 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. 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. Dla przykładu, dokonajmy przesunięcia bitów liczby 8 bitowej o 3 pozycje w lewo.

$$\begin{array}{ccccc} 10101011 & & & & \overbrace{101\,01011\mathbf{000}}^{\text{wynik}} \\ 0xab & << 3 & == & & 0x58 \end{array}$$

Oto kolejne przykłady:

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

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).

Uwaga

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 zróżnicowania ich znaczenia w zależności od kontekstu. Więcej na ten temat dowiemy się na II semestrze.

2.7 Operatory łączone

Dla skrócenia zapisu, zostały też wprowadzone tzw. **operatory łączone**, łączące operacje arytmetyczne lub bitowe i przypisanie:

+=, −=, ∗=, /=, %=, |=, &=, ^=, <<=, >>=.

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

2.8 Priorytety operatorów

W niektórych z powyższych przykładów zaobserwowaliśmy, że w jednym wyrażeniu można używać wielu operatorów. Kolejność wykonywania działań jest jednak ściśle określona. Można ją wymusić za pomocą nawiasów okrągłych (...).

Domyślnie jednak wyrażenia obliczane są według priorytetów, zestawionych w kolejności od największego do najmniejszego w tab. 1. W przypadku operatorów o tym samym priorytecie, operacje wykonywane są w kolejności od lewej do prawej.

Przykłady:

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

Tablica 1: Priorytety operatorów.

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

3 Ćwiczenia

Zadanie 3.1. Wyraź następujące liczby zmiennoprzecinkowe dane w notacji naukowej w zwykłej postaci dziesiętnej.

a) $4.21e6$,
b) $1.95323e2$,
c) $2.314e{-}4$,
d) $4.235532e{-}2$.

Zadanie 3.2. 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 3.3. Dane są 3 zmienne zadeklarowane w sposób następujący.

```
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*+*b*)/**int**(*c*)),
g) **double**(**int**(*a*))/*c*.

Zadanie 3.4. Korzystając z przekształceń logicznych (np. praw De Morgana, praw rozdzielności) uprość następujące wyrażenia (zakładamy, że a,b,c,d są typu **double**, a p,q,r typu **bool**).

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*.

Zadanie 3.5. Jaki wynik dadzą następujące operacje bitowe wykonane na danych typu **short**?

a) 0*x0FCD* | 0*xFFFF*,
b) 364 & 0*x323*,
c) ~163,
d) 0*xFC93* ^ 0*x201D*,
e) 14 << 4,
f) 0*xf5a3* >> 8,
g) 0*x3a9f* >> 7.

Zadanie 3.6. Niech dana będzie zmienna typu **int**. W jaki sposób dokonać zmiany znaku wartości tej zmiennej na przeciwny nie używając operatora −?

Zadanie 3.7. Policyjny fotoradar emituje fale elektromagnetyczne o częstotliwości f_e Hz. Wiązka tych fal jest odbijana od nadjeżdżającego z prędkością v samochodu i, jako że auto jest w ruchu, powraca do urządzenia ze zmienioną częstotliwością f_o Hz. Polscy funkcjonariusze testują właśnie najnowszy model brytyjskiej „suszarki". Związek pomiędzy omawianymi zmiennymi można wyrazić równaniem

$$v = 6{,}685 \times 10^8 \, \frac{f_o - f_e}{f_o + f_e}.$$

Prędkość jednak jest podawana w milach na godzinę. Wiedząc, że 1 mila to 1609,344 m, napisz fragment kodu w języku C++, który dla danego f_e i f_o poda prędkość nadjeżdżającego samochodu w km/h. Ponadto wypisz na ekran wartość logiczną mówiącą, czy została przekroczona dopuszczalna prędkość, wynosząca w tym miejscu 50 km/h.

Jaki będzie wynik działania tej procedury dla $f_e = 2 \times 10^{10}$ Hz i $f_o = 2.0000004 \times 10^{10}$ Hz?

⋆ **Zadanie 3.8.** Danych jest 6 zmiennych *ax*,*ay*,*bx*,*by*,*cx*,*cy* typu **double**, reprezentujących współrzędne 3 punktów w $\mathbb{R}^2$: $\mathbf{a} = (ax, ay)$, $\mathbf{b} = (bx, by)$, $\mathbf{c} = (cx, cy)$. Napisz fragment kodu, który wyznaczy kwadrat promienia okręgu przechodzącego przez $\mathbf{a}$, $\mathbf{b}$ i $\mathbf{c}$. Dany jest on 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 np. $|\mathbf{a} - \mathbf{b}| = \sqrt{(\mathrm{ax} - \mathrm{bx})^2 + (\mathrm{ay} - \mathrm{by})^2}$ oraz $\mathbf{a} \times \mathbf{b} = \mathrm{axby} - \mathrm{aybx}$.

Zadanie 3.9. Dane są $a, b, c, d, e, f \in \mathbb{R}$ takie, że układ dwu równań liniowych względem niewiadomych $x, y \in \mathbb{R}$:

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

jest oznaczony. Zaproponuj fragment kodu w języku C++, który wyznaczy jego rozwiązanie, tzn. obliczy wartość zmiennych *x*,*y* na podstawie pewnych wartości zmiennych *a*,*b*,*c*,*d*,*e*,*f*. Do reprezentacji liczb rzeczywistych użyj typu **double**.

4 Wskazówki do ćwiczeń

Odpowiedź do zadania 3.2.

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 ≃ 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 3.5.

a) $0x0FCD \mid 0xFFFF == -1$ ($0xFFFF$).
b) $364 \;\&\; 0x323 == 288$ ($0x0120$).
c) $\sim 163 == -164$ ($0xFF5C$) (dlaczego -164?).
d) $0xFC93 \;\hat{}\; 0x201D == -9074$ ($0xDC8E$).
e) $14 << 4 == 224$ ($0x00E0$).
f) $0xf5a3 >> 8 == 0xFFF5$ (bit znaku == 1).
g) $0x3a9f >> 7 == 117$ ($0x0075$).

Wskazówka do zadania 3.6. Zmienna typu **int** jest reprezentowana w kodzie U2.

Odpowiedź do zadania 3.8.

```
1  // Dane wejściowe:
2  double ax = ..., ay = ...; // współrzędne punktu a.
3  double bx = ..., by = ...; // współrzędne punktu b.
4  double cx = ..., cy = ...; // współrzędne punktu c.
5  // Dane wyjściowe:
6  double r2; // kwadrat promienia okręgu przech. przez a,b,c.
7
8  double amcx = ax-cx; //a - c --- wsp. x
9  double amcy = ay-cy; //a - c --- wsp. y
10 double bmcx = bx-cx; //b - c --- wsp. x
11 double bmcy = by-cy; //b - c --- wsp. y
12 double a_c  = amcx*amcx+amcy*amcy; //|a - c|^2
13 double b_c  = bmcx*bmcx+bmcy*bmcy; //|b - c|^2
14 double a_b  = (ax-bx)*(ax-bx)+(ay-by)*(ay-by); //|a - b|^2
15 double ilwekt = amcx*bmcy - amcy*bmcx;
16
17 r2 = a_c*b_c*a_b/4.0/ilwekt/ilwekt;
```

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

IV. Instrukcja warunkowa i pętle

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

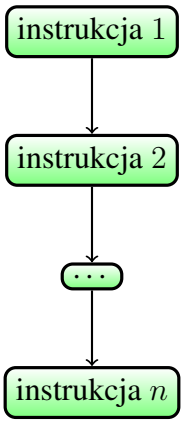
Spis treści

1 Bezpośrednie następstwo instrukcji

Poznaliśmy już następujące zagadnienia:

- deklaracja zmiennych i stałych nazwanych,
- przypisywanie wartości,
- obliczanie wartości wyrażeń z użyciem operatorów,
- wypisywanie wartości zmiennych na ekran i wczytywanie ich wartości z klawiatury.

Zauważmy, że do tej pory nasz kod w języku C++ nie miał żadnych rozgałęzień. Wszystkie instrukcje były wykonywane jedna po drugiej. Owo tzw. **bezpośrednie następstwo** można zobrazować jak na rys. 1. Jest to tak zwany schemat blokowy algorytmu (ang. *control flow diagram*, czyli schemat przepływu sterowania).



Rysunek 1: Schemat blokowy bezpośredniego następstwa instrukcji.

W kolejnych paragrafach poznamy konstrukcje, które pozwolą nam zmieniać przebieg programu. Dzięki temu będziemy mogli dostosowywać jego działanie do różnych szczególnych przypadków w zależności od wartości przetwarzanych danych.

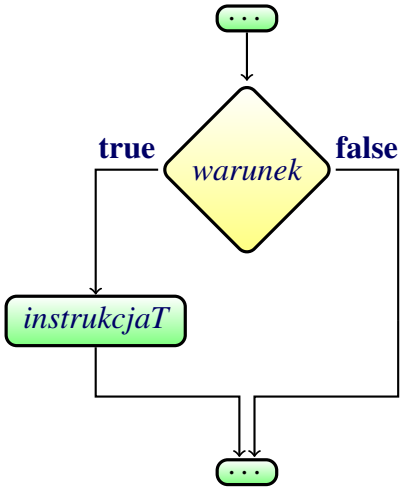
2 Instrukcja warunkowa

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 jedzie prawidłowo. Student przed sesją głowi się, czy 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++ takim mechanizmem używanym do dokonywania wyborów jest **instrukcja warunkowa if** (od ang. *jeśli*). Wykonuje ona pewien fragment kodu **wtedy i tylko wtedy**, gdy pewien dany warunek logiczny jest spełniony. Jej składnia jest następująca.

```
if (warunek) instrukcjaT;
```

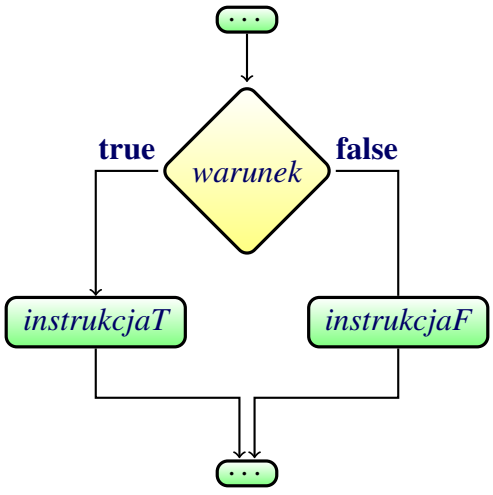
gdzie *warunek* (koniecznie ujęty w nawiasy!) jest pewnym wyrażeniem sprowadzalnym do typu **bool**. Stosowny schemat blokowy przedstawia rys. 2.



Rysunek 2: Schemat blokowy instrukcji warunkowej **if**.

Jeśli *warunek* nie jest spełniony, możliwe jest wykonanie innej instrukcji poprzez dodanie słowa kluczowego **else** (od ang. *w przeciwnym przypadku*) według składni (por. rys. 3):

```
if (warunek) instrukcjaT;
else instrukcjaF;
```



Rysunek 3: Schemat blokowy instrukcji warunkowej **if**... **else**.

Jeżeli chcemy wykonać warunkowo **kilka instrukcji**, tworzymy w tym celu tzw. **blok instrukcji**, ograniczony nawiasami klamrowymi {...}. 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;   // wprowadź x i y z klawiatury

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

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

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

if (x < y) { // klamerka może się znaleźć też w osobnym wierszu
   if (z < x)
      cout << z;
   else
      cout << x;
}
else
{
   if (z < y)
      cout << z;
   else
      cout << y;
}
```

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 widocznej intencji programisty, należałoby objąć fragment

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

nawiasami klamrowymi.

3 Pętle

Oprócz instrukcji warunkowej **if**, rozgałęziającej przebieg sterowania „w dół”, możemy skorzystać z tzw. **pętli**. Umożliwiają one wykonywanie tej samej instrukcji bądź 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 dzień pierwszy, potem dzień drugi, a potem dzień trzeci, a potem itd., czyli tak naprawdę dzień i-ty, gdzie $i = 1, 2, \ldots, n$. Dalej, a 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** oraz
- **do** ... **while**.

Przyjrzymy się im dokładnie w poniższych podrozdziałach.

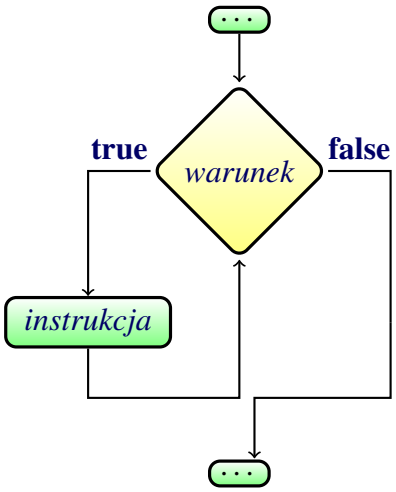
3.1 Pętla while

Najprostszą konstrukcją realizującą pętlę jest instrukcja **while**. Składnia:

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

gdzie *warunek* jest pewnym wyrażeniem sprowadzalnym do typu **bool**. Najlepiej będzie, gdy istnieje możliwość jego zmiany na skutek wykonywania podanej *instrukcji* . W przeciwnym wypadku stworzymy program, który nigdy się nie zatrzyma.

Schemat blokowy przedstawionej pętli zobrazowany jest na rys. 4.



Rysunek 4: Schemat blokowy pętli **while**.

Tak samo jak w przypadku instrukcji **if**, zamiast pojedynczej instrukcji we wszystkich omawianych pętlach może pojawić się cały ich blok, ujęty w nawiasy klamrowe.

Uwaga

Sam średnik (;) jest instrukcją pustą, dlatego pętla

```
while (true) ;
```

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

Oto sposób na wypisanie na ekran kolejno liczb 1,2,...,100 i zsumowanie ich wartości (przypomnijmy sobie w tym miejscu problem młodego Gaussa z pierwszego wykładu).

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

while (i<=100) // ten warunek nie będzie kiedyś spełniony...
{
    cout << i << endl;
    suma += i;
    i++;          /* ...gdyż jest zależny od wartości zmiennej i,
                     która się w tym miejscu zmienia */
}

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

3.2 Pętla for

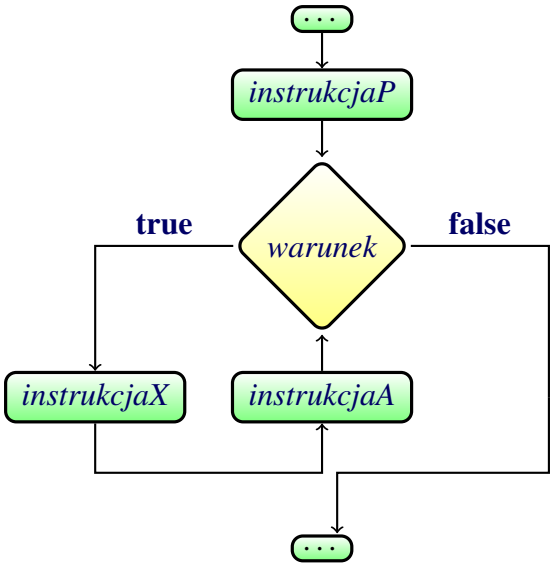
W pewnych zastosowaniach mamy czasem do czynienia z następującym schematem.

```
instrukcjaP; // instrukcja inicjująca
while (warunek)
{
    instrukcjaX;
    instrukcjaA; // instrukcja aktualizująca
}
```

Zauważmy, że przebiega on według następującego schematu. Najpierw przygotuj się przed wykonaniem pewnej pracy, wykonując **inicjującą** *instrukcjęP*. Potem, dopóki *warunek* pozostaje spełniony, wykonuj *instrukcjęX*, jednakże **aktualizując** za pomocą *instrukcjiA* pewne dane przed każdą kolejną iteracją.

Taką konstrukcję można zapisać równoważnie, korzystając z pętli **for**, której schemat blokowy przedstawia rys. 5. A oto jej składnia.

```
for (instrukcjaP; warunek; instrukcjaA)
    instrukcjaX;
```



Rysunek 5: Schemat blokowy pętli **for**.

Wróćmy na chwilę do przykładu omówionego w poprzednim paragrafie. 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;
```

Jeśli chcemy umieścić więcej niż jedną instrukcję inicjującą bądź aktualizującą, należy każdą z nich oddzielić przecinkiem (nie średnikiem, ani nie tworzyć dlań bloku). Zatem ten kod jest równoważny poniższemu, chociaż traci on znacznie na czytelności.

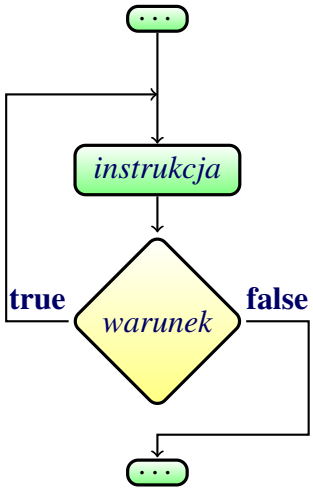
```
int suma = 0; /* chcemy, by ta zmienna była dostępna również
                 po zakończeniu działania pętli for... */

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

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

3.3 Pętla do...while

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

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

Jak widać na schemacie blokowym (rys. 6), używamy jej, gdy chcemy zapewnić, że *instrukcja* zostanie wykonana **co najmniej** jeden raz.



Rysunek 6: Schemat blokowy pętli **do**...**while**.

A oto kolejna wersja rozpatrywanego przez nas przykładu.

```
1 int i = 1, suma = 0;
2 do
3 {
4     suma += i;
5     cout << i << endl;
6     ++i;
7 }
8 while (i<=100);
```

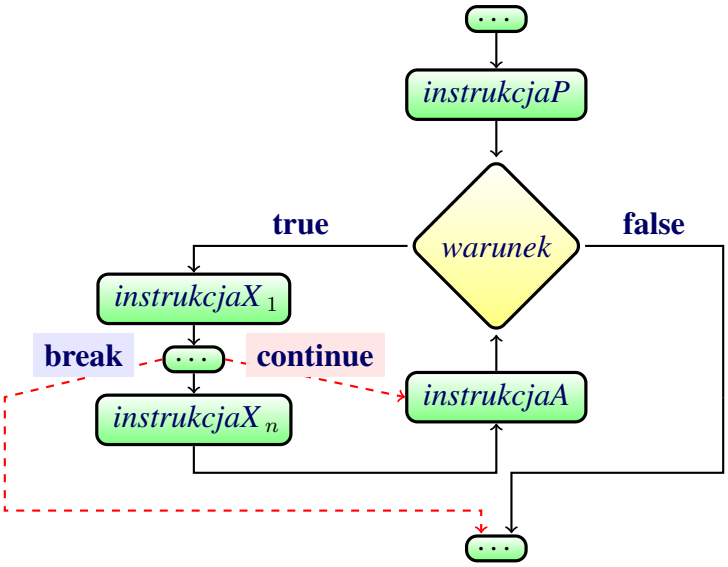
3.4 break i continue

W pewnych (dość rzadkich) szczególnych przypadkach zachodzi konieczność natychmiastowej zmiany przebiegu wykonywania pętli.

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. 7 przedstawia zmianę przebiegu programu za pomocą omawianych instrukcji na przykładzie pętli **for**.



Rysunek 7: Instrukcje **break** i **continue** a pętla **for**.

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

4 Ćwiczenia

Zadanie 4.1. Wyraź następujące pętle dane w sposób opisowy za pomocą instrukcji **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 4.2. Zaprogramuj w języku C++ algorytm Euklidesa do wyznaczania największego wspólnego dzielnika dwóch liczb (zob. zestaw zadań nr 1).

Zadanie 4.3. Spośród liczb $1, 2, \ldots, 100$ wypisz na ekran wszystkie podzielne przez 7, tzn. 7,14,21,....

Zadanie 4.4. Spośród liczb $1, 2, \ldots, 100$ wypisz na ekran wszystkie podzielne przez 2 lecz niepodzielne przez 5, tzn. 2,4,6,8,12, ...

Zadanie 4.5. Spośród liczb $1, 2, \ldots, 100$ wypisz na ekran co drugą podzielną przez 5 lub podzielną przez 7, tzn. 5, 10, 15, 21, 28, ...

$\star$ **Zadanie 4.6.** Napisz fragment kodu, który sprawdzi, czy następujące liczby są pierwsze: 7,93,97,6687,6689, 6691.

Zadanie 4.7. Napisz fragment kodu, który znajduje minimum z danych liczb całkowitych dodatnich. Liczby odczytuj z klawiatury, póki użytkownik nie wprowadzi 0.

Zadanie 4.8. 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 4.9. Napisz fragment kodu, który aproksymuje wartość liczby π za pomocą 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 4.10. Napisz fragment kodu, który aproksymuje wartość liczby e za pomocą 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 pomiędzy kolejnymi wyrazami szeregu będzie mniejsza niż 10^{-9}.

Zadanie 4.11. 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 $x \in (-1, 1)$.

Przypomnijmy, że zgodnie z umową np. $\sum_{i=1}^{10} i = 1 + 2 + \cdots + 10$ oraz np. $\prod_{i=1}^{5} \frac{i}{i+1} = \frac{1}{2} \times \frac{2}{3} \times \cdots \times \frac{5}{6}$.

Zadanie 4.12. Korzystając ze wzoru na przybliżoną wartość funkcji $\sin$ podanego w zad. 4.11, utwórz kod, 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 postaci podobnej do poniższej.

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

Zadanie 4.13. Pobierz wartości zmiennych *a*,*b* typu **double** z klawiatury. Będą one definiować równanie względem niewiadomej $x \in \mathbb{R}$ postaci *a* x + *b* = 0. Zaproponuj fragment kodu w języku C++, który wyznaczy jego rozwiązanie. Poprawnie identyfikuj przypadek, gdy dane równanie nie jest równaniem liniowym.

Zadanie 4.14. Dla danych $a, b, c, d, e, f \in \mathbb{R}$ zaproponuj kod w języku C++ do rozwiązywania układu dwóch równań liniowych względem niewiadomych $x, y \in \mathbb{R}$:

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

Algorytm ten powinien poprawnie identyfikować przypadki (np. wypisując stosowny komunikat na ekranie), w których dany układ jest sprzeczny bądź nieoznaczony. Współczynniki układu pobierz z klawiatury. Do reprezentacji zbioru $\mathbb{R}$ użyj typu **double**.

⋆ **Zadanie 4.15.** Napisz fragmenty kodu, które sprawdzą, 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 4.16.** Dla danej zmiennej typu **int** napisz program, który wypisze na ekran jej wartość w postaci binarnej.

5 Wskazówki do ćwiczeń

Wskazówka do zadania 4.6. 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 4.15. Użyj 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...
```

Tutaj może się przydać (ale nie musi) instrukcja **break**.

Wskazówka do zadania 4.16. Liczba typu **int** jest 32 bitowa. Należy użyć pętli, która będzie w każdej iteracji wypisywać wartość kolejnego bitu (0 lub 1). Wyrażenie x & (1<<k) przyjmuje wartość 0, gdy k-ty bit jest równy 0 oraz wartość różną od 0, gdy k-ty bit jest równy 1 (dlaczego?).

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

V. Tablice jednowymiarowe. Sortowanie

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Tablice jednowymiarowe

Do tej pory przechowywaliśmy dane używając pojedynczych zmiennych. Były to tzw. **zmienne skalarne** (atomowe). Jedna jednostka danych odpowiadała pojedynczej zmiennej.

Często jednak mamy dany cały ciąg zmiennych tego samego typu. Dla przykładu, rozważmy fragment programu dokonujący podsumowania rocznych zarobków pewnej doktorantki.

```
double zarobki1, zarobki2, /*...*/, zarobki12;
                                    // deklaracja 12 zmiennych
zarobki1 = 1399.0;  // styczeń
zarobki2  = 1493.0; // luty
// ...
zarobki12 = 999.99; // grudzień

double suma = 0.0;
suma += zarobki1;
suma += zarobki2;
//...
suma += zarobki12;

cout << "Zarobiłam w 2010 r. " << suma << " zł.";
```

Dochód z każdego miesiąca przechowywany jest w oddzielnej zmiennej. Wszystkie z nich są tego samego typu. Łatwo zauważyć, że nie jest to 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** bądź **wektorów**.

Deklaracja tablicy jednowymiarowej następuje według poniższej składni:

```
typ identyfikator[rozmiar];
```

gdzie *rozmiar* $\in \mathbb{N}$, *rozmiar*>0, jest **stałą** (!).

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

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

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

Operator indeksowania przyjmuje jako parametr dowolną wartość całkowitą (np. stałą bądź wyrażenie arytmetyczne). Każdy element tablicy traktujemy tak, jakby był zwykłą zmienną, z którą do tej pory mieliśmy do czynienia.

Uwaga

W języku C++ nie ma mechanizmów sprawdzania poprawności indeksów! Następujący kod najczęściej nie spowoduje natychmiastowego wystąpienia błędu.

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

Jednakże, zmieni on wartość 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 w wykładzie dotyczącym dynamicznego przydziału pamięci.

Wróćmy do przykładu dotyczącego naszej zaradnej koleżanki. Oto w jaki sposób można wykorzystać tablice do podsumowania jej zarobków. Zauważmy analogię pomiędzy tym a poprzednim fragmentem kodu.

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

zarobki[0] = 1399.0; // styczeń
zarobki[1] = 1493.0; // luty
// ...
zarobki[11] = 999.99; // grudzień

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

cout << "Zarobiłam w 2010 r. " << suma << " zł.";
```

Przetworzenie całej tablicy za pomocą pętli **for** i dokonanie na każdym elemencie tej samej operacji arytmetycznej sprawia, że kod staje się bardziej zwięzły i czytelny. Z drugiej strony, nietrudno go teraz byłoby poprawić tak, żeby np. uwzględniał zarobki z całego okresu jej studiów.

Ponadto, dla wygody, przy tworzeniu tablicy można przypisać wartości jej elementom w następujący sposób:

```
typ identyfikator[n] = {
   wartość0, wartość1, /*...*/, wartośćn
};
```

Pozwala to jeszcze bardziej uprościć program dla naszej doktorantki:

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

double suma = 0.0;
```

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

cout << "Zarobiłam w 2010 r. " << suma << " zł.";
```

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, można użyć następującego fragmentu kodu:

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

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

Z drugiej strony, aby wczytać elementy tablicy z klawiatury, można napisać:

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

cout << "Podaj wartości " << n << " elementów." << endl;
for (int i=0; i<n; ++i)
    cin >> t[i];
```

2 Sortowanie tablic

Rozpatrzmy problem **sortowania tablic jednowymiarowych**. Jest to jedno z ciekawszych zagadnień, które pojawiło się na początku rozwoju informatyki teoretycznej, a w którym zastosowanie znajdują właśnie tablice. Można je sformalizować w sposób następujący.

Dana jest tablica t rozmiaru n zawierająca elementy, które można porównywać za pomocą operatora relacyjnego <=. Należy zmienić kolejność (tj. dokonać permutacji) elementów t tak, by zachodziło

$$t[0] <= t[1], \quad t[1] <= t[2], \quad \ldots, \quad t[n-2] <= t[n-1].$$

Oczywiście rozwiązanie takiego problemu 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 $t[j] <= t[i]$, może istnieć wiele rozwiązań spełniających powyższy warunek.

W niniejszym paragrafie omówimy trzy elementarne algorytmy:

a) sortowanie przez wybór,
b) sortowanie przez wstawianie,

c) sortowanie bąbelkowe.

Cechują się one tym, że w **pesymistycznym** („najgorszym") przypadku liczba operacji porównań elementów tablicy jest **proporcjonalna** do n^2. Inną ważną cechą, którą poddamy analizie, będzie liczba potrzebnych przestawień tych elementów.

Bardziej wydajne (i bardziej złożone, np. sortowanie szybkie, przez łączenie, przez kopcowanie) algorytmy będą omówione w semestrze III. Niektóre z nich wymagają co najwyżej $kn \log_n$ porównań dla pewnego k.

2.1 Sortowanie przez wybór

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

Ideę tę formalizuje następujący pseudokod:

```
dla i=0,1,...,n-2
{
    // t[0], ..., t[i-1] są uporządkowane względem <=
    // (nadto, są już na swoich ostatecznych miejscach)
    j = indeks najmniejszego elementu
          spośród t[i], ..., t[n-1];
    zamień 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 działania tego algorytmu ilustrują rys. 1–5.

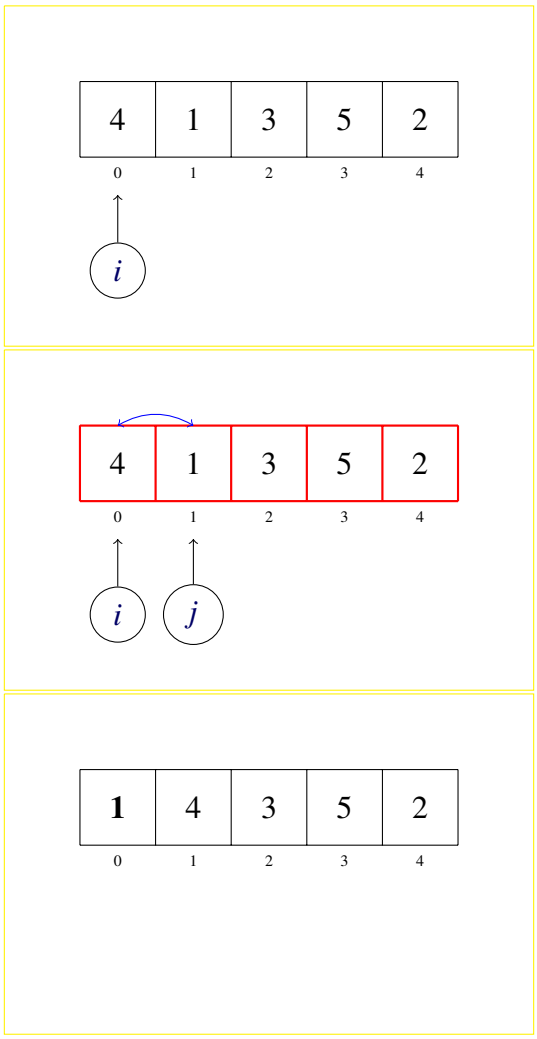
W kroku I (rys. 1) mamy i==0. Dokonując wyboru elementu najmniejszego spośród $t[0]$, ..., $t[4]$ otrzymujemy j==1. Zamieniamy więc elementy $t[0]$ i $t[1]$ miejscami.

W kroku II (rys. 2) mamy i==1. Wybór najmniejszego elementu wśród $t[1], \ldots, t[4]$ daje j==4 Zamieniamy miejscami zatem $t[1]$ i $t[4]$.

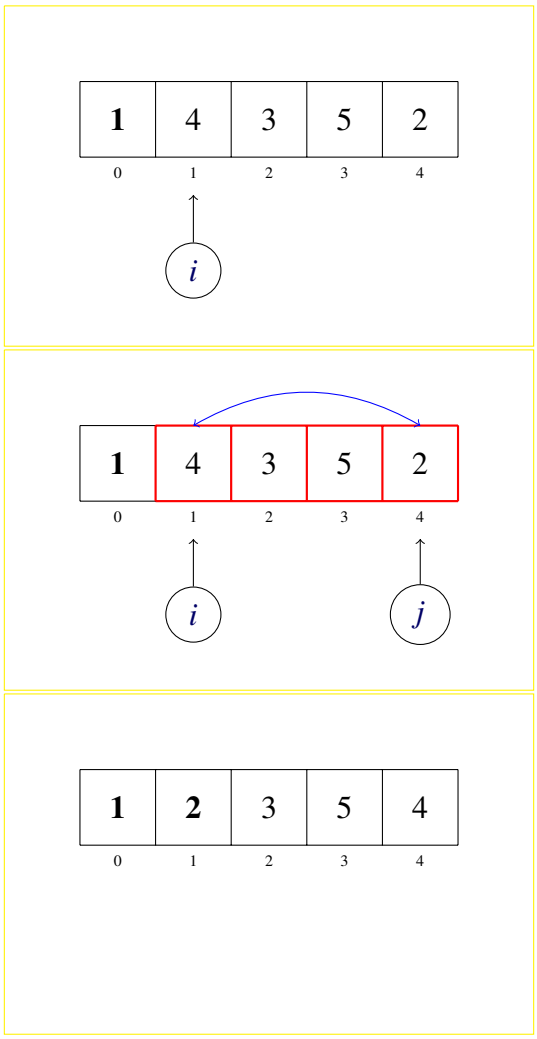
Dalej (rys. 3), i==2. Elementem najmniejszym spośród $t[2], \ldots, t[4]$ jest $t[j]$ dla j==2 Zamieniamy miejscami zatem niezbyt sensownie $t[2]$ i $t[2]$. Komputer, na szczęście, zrobi to bez grymasu.

W ostatnim kroku (rys. 4) i==3 i j==4, dzięki czemu możemy uzyskać ostateczne rozwiązanie (rys. 5).

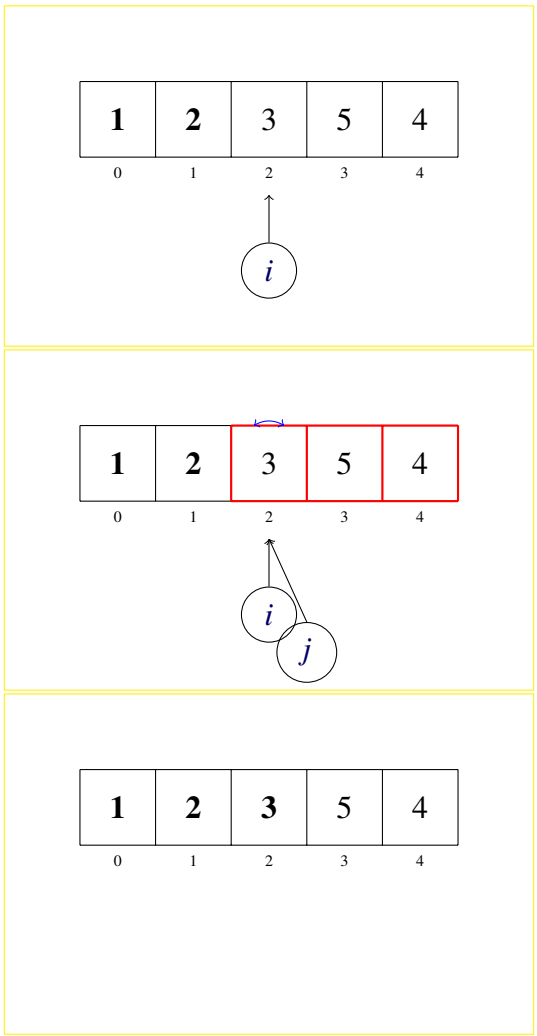
Implementację sortowania przez wybór w języku C++ i analizę liczby porównań oraz przestawień elementów pozostawiamy jako ćwiczenie.



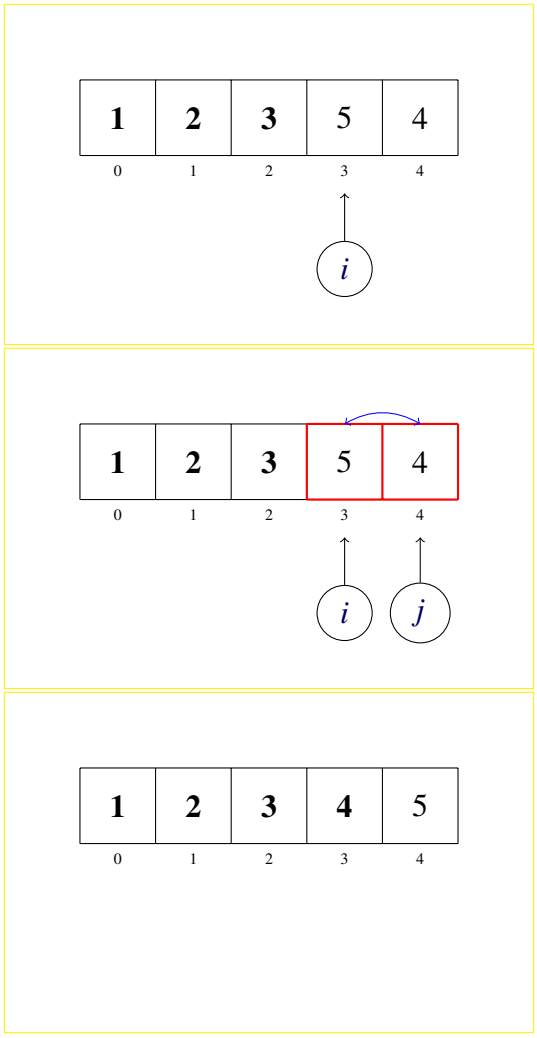
Rysunek 1: Sortowanie przez wybór — przykład — iteracja I.



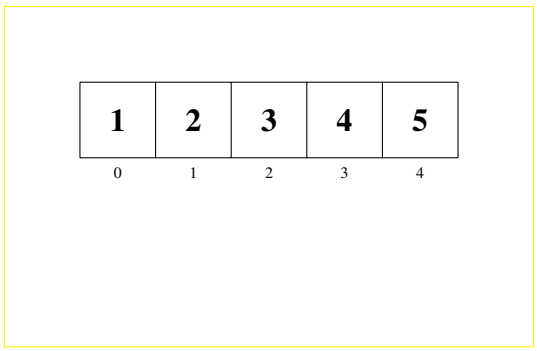
Rysunek 2: Sortowanie przez wybór — przykład — iteracja II.



Rysunek 3: Sortowanie przez wybór — przykład — iteracja III.



Rysunek 4: Sortowanie przez wybór — przykład — iteracja IV.



Rysunek 5: Sortowanie przez wybór — przykład — rozwiązanie.

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 <=. Pomiędzy nie wstawiamy $t[i]$ tak, by nie zaburzyć porządku.

Formalnie rzecz ujmując, idea ta może być wyrażona za pomocą pseudokodu:

```
dla i=1,2,...,n-1
{
   // t[0], ..., t[i-1] są uporządkowane względem <=
   // (ale niekoniecznie jest to ich ostateczne miejsce)
   j = indeks takiego elementu spośród t[0],...,t[i], że
      t[k] <= t[i] dla każdego k < j oraz
      t[l] >  t[i] dla każdego l ⩾ j;
   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:

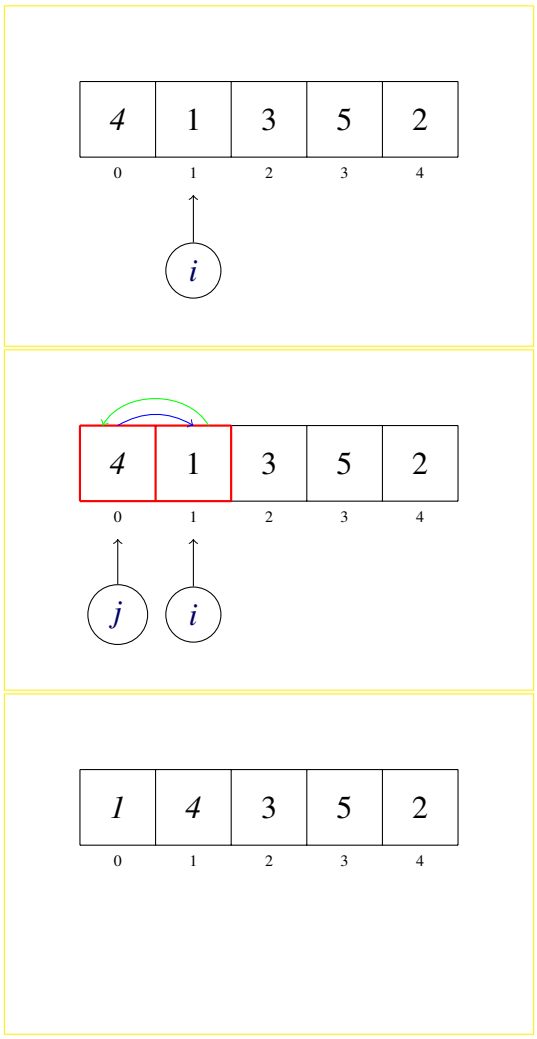
$t[0]$	...	$t[j-1]$	$t[i]$	$t[j]$	...	$t[i-1]$	$t[i+1]$	...	$t[n-1]$

Powyższy pseudokod może być wyrażony w następującej równoważnej formie:

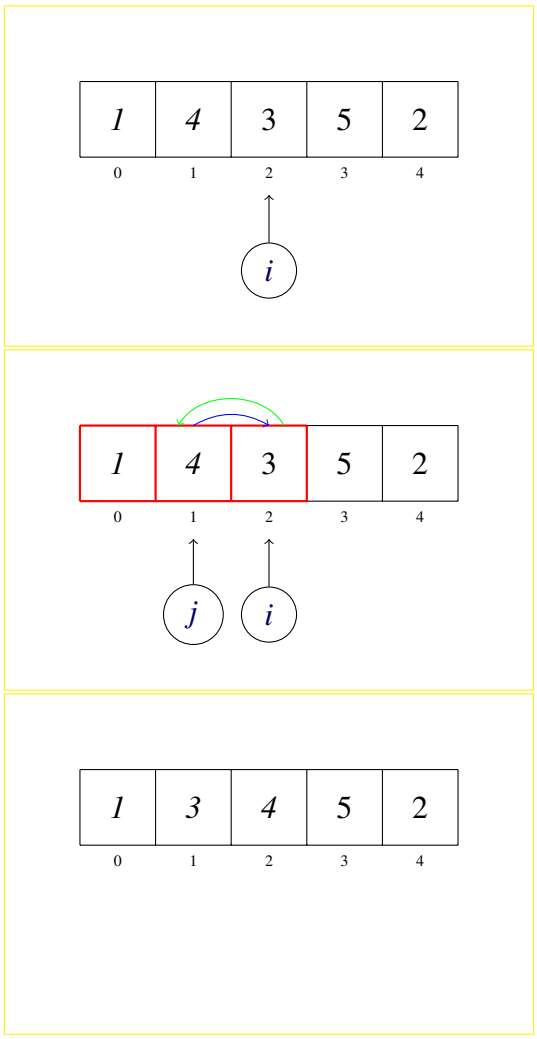
```
dla i=1,2,...,n-1
{
   // t[0], ..., t[i-1] są uporządkowane względem <=
   // (ale niekoniecznie jest to ich ostateczne miejsce)
   znajdź największe j ze zbioru {0,...,i} takie, że
      j == 0 lub t[j-1] <= t[i];
   wstaw t[i] przed t[j];
}
```

Jako przykład rozpatrzmy znów tablicę **int** $t[5] = \{4,1,3,5,2\}$;
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 6–9.

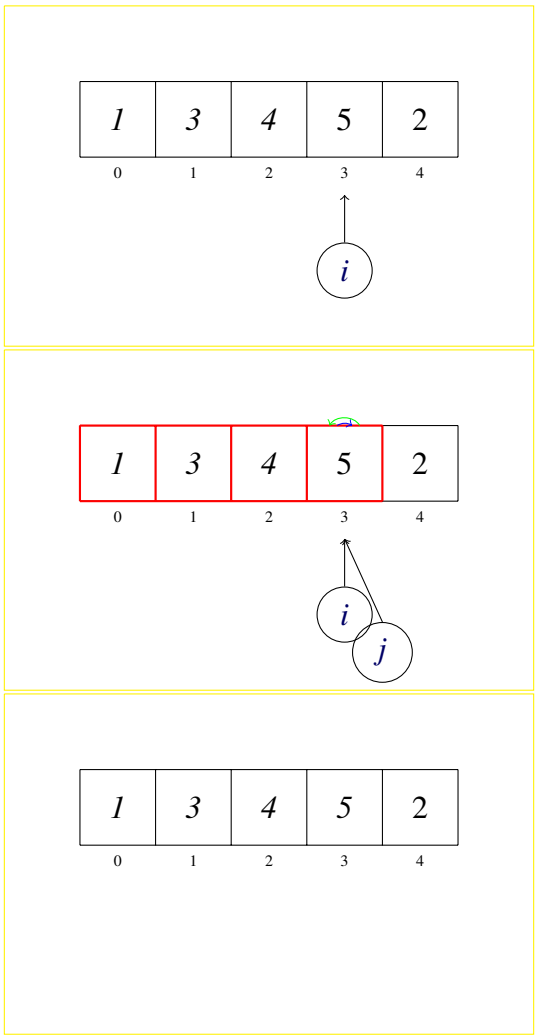
Implementację sortowania przez wstawianie i analizę liczby porównań oraz przestawień elementów pozostawiamy jako ćwiczenie.



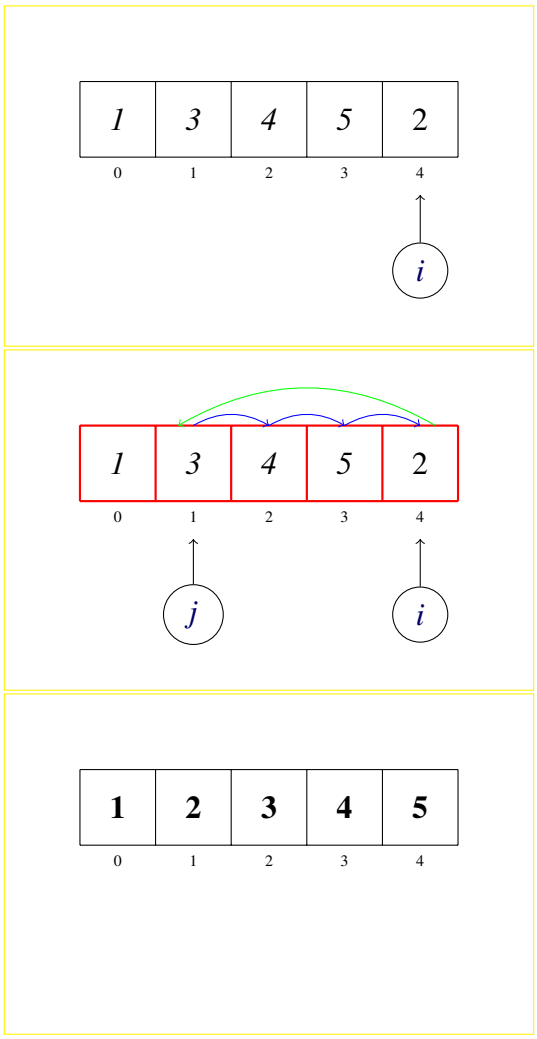
Rysunek 6: Sortowanie przez wstawianie — przykład — iteracja I.



Rysunek 7: Sortowanie przez wstawianie — przykład — iteracja II.



Rysunek 8: Sortowanie przez wstawianie — przykład — iteracja III.



Rysunek 9: Sortowanie przez wstawianie — przykład — iteracja IV.

2.3 Sortowanie bąbelkowe

Sortowanie bąbelkowe (ang. *bubble sort*) jest interesującym przykładem algorytmu pojawiającego się w większości podręczników akademickich dotyczących podstawowych sposobów sortowania tablic, który jednakże prawie wcale nie jest stosowany 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.

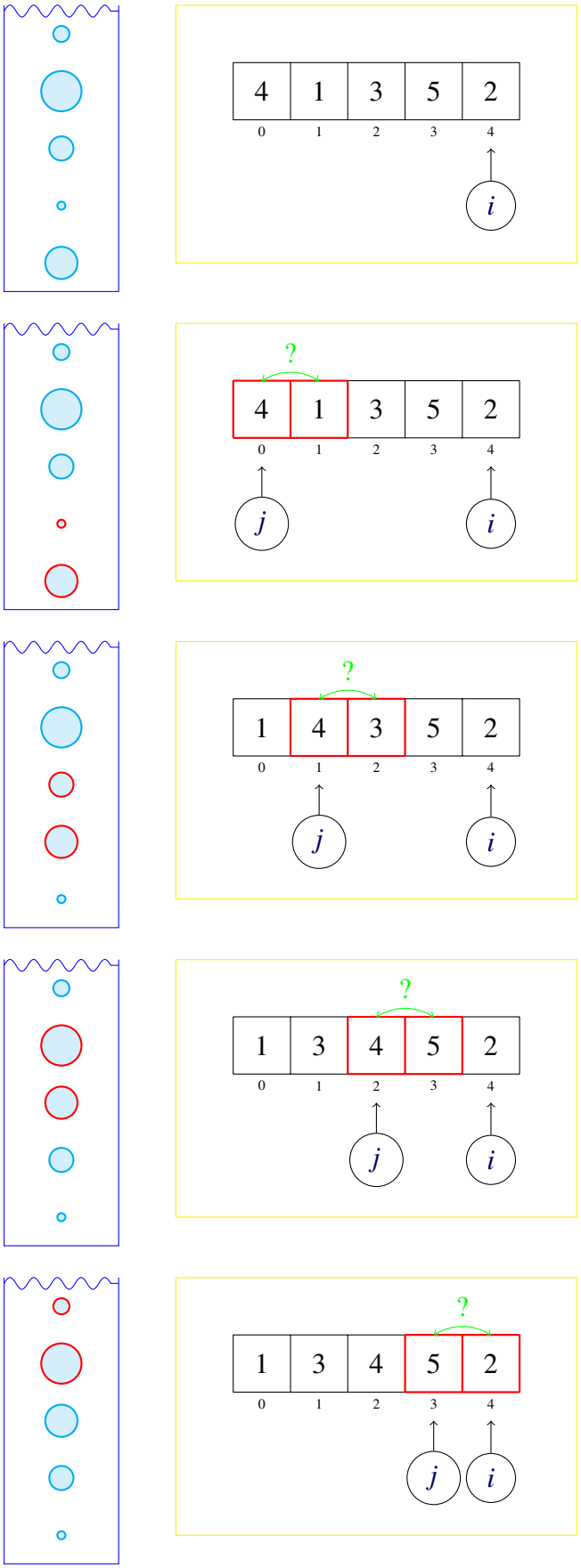
W tym algorytmie porównywane są elementy tylko ze sobą bezpośrednio sąsiadujące. Jeśli okaże się, że nie zachowują one odpowiedniej kolejności względem relacji <=, element „cięższy" wypychany jest „w górę", niczym pęcherzyk powietrza (tytułowy bąbelek) pod powierzchnią wody.

A oto pseudokod:

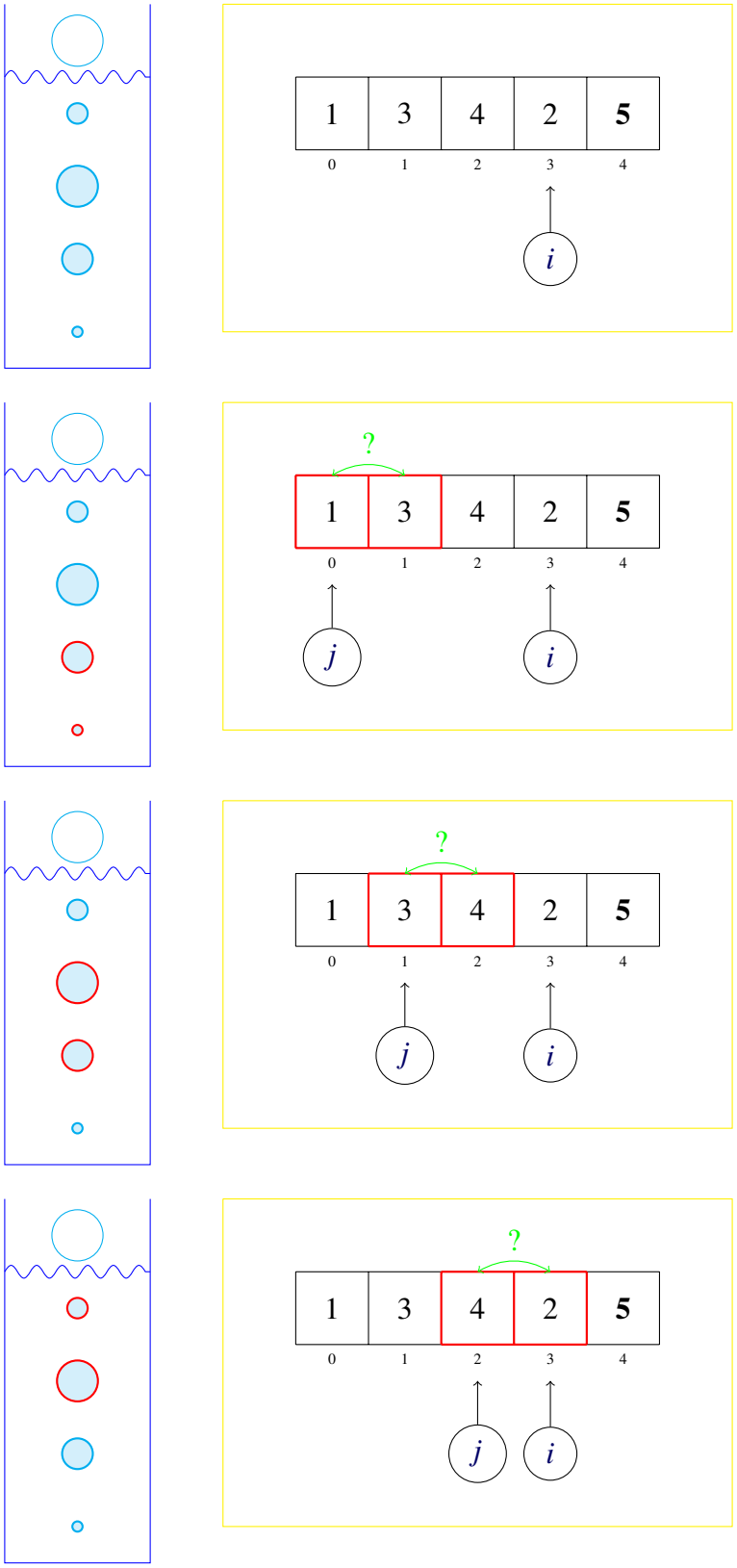
```
dla i=n-1,...,1
{
    dla j=0,...,i-1
    {
        // porównuj elementy parami
        jeśli (t[j] > t[j+1])
            zamień t[j] i t[j+1];
            // tzn. "wypchnij" cięższego "bąbelka" w górę
    }

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

Dla przykładu rozpatrzmy ponownie tablicę **int** t[5] = {4,1,3,5,2};
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 10–14.

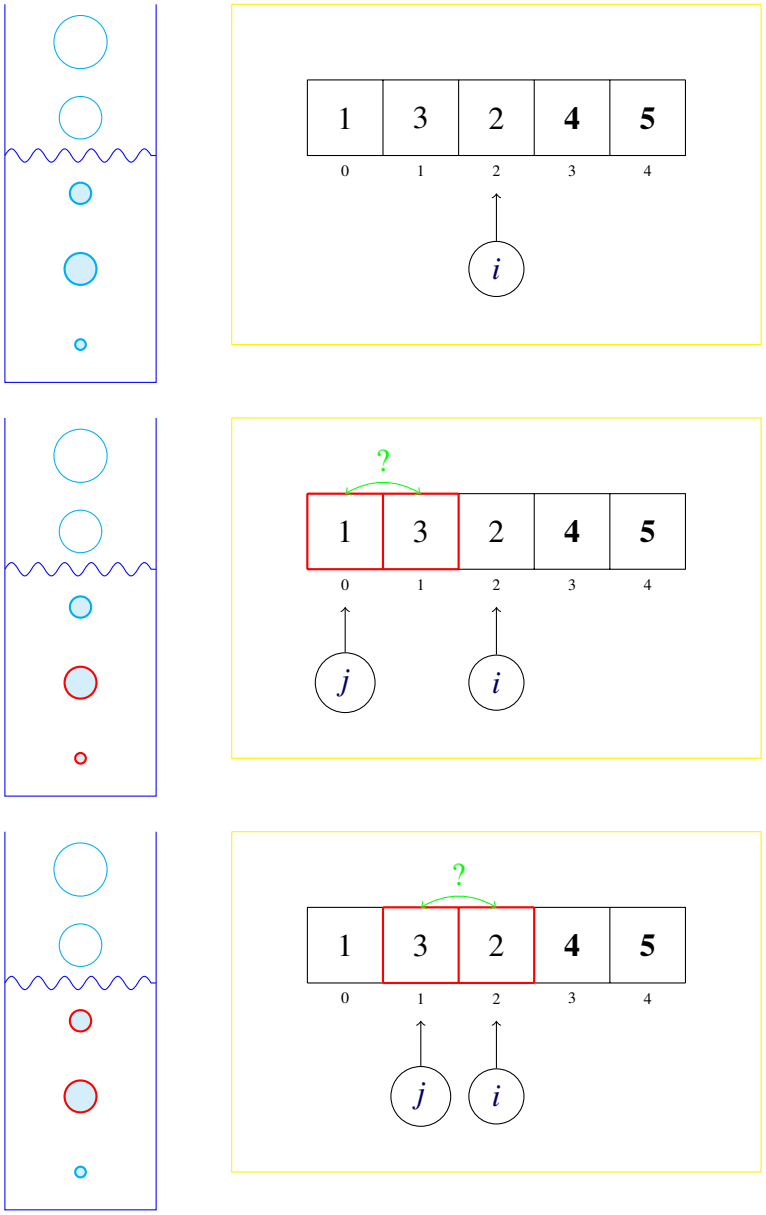
Implementację sortowania bąbelkowego w języku C++ i analizę liczby porównań oraz przestawień elementów pozostawiamy jako ćwiczenie.



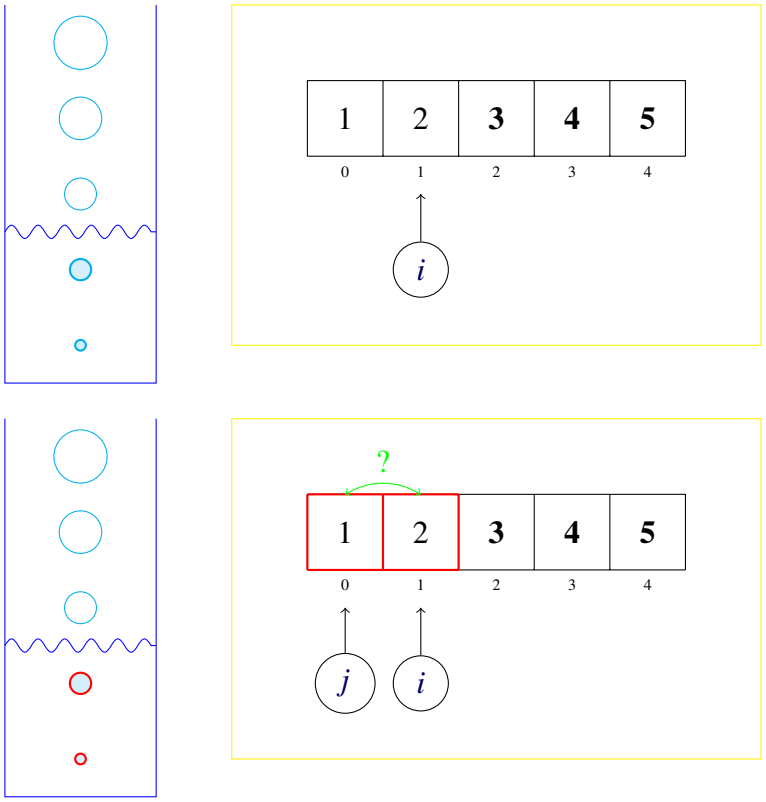
Rysunek 10: Sortowanie bąbelkowe — przykład — krok I.



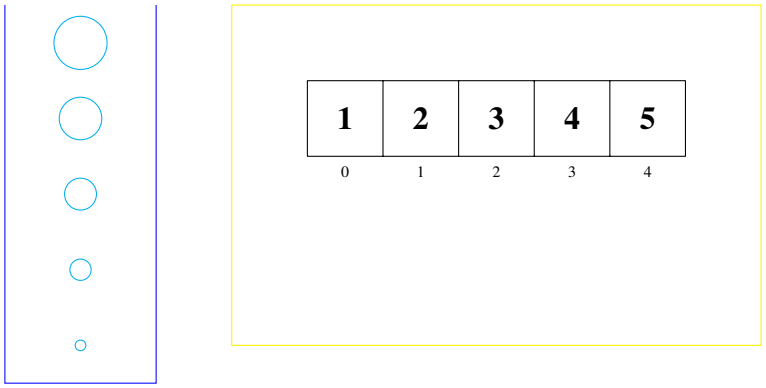
Rysunek 11: Sortowanie bąbelkowe — przykład — krok II.



Rysunek 12: Sortowanie bąbelkowe — przykład — krok III.



Rysunek 13: Sortowanie bąbelkowe — przykład — krok IV.



Rysunek 14: Sortowanie bąbelkowe — przykład — rozwiązanie.

3 Ćwiczenia

Zadanie 5.1. Niech dana będzie tablica zadeklarowana jako **int** *tab*[*n*], dla pewnego *n*. Napisz kod, który przesunie każdy element o indeksie > 0 o jedną komórkę w lewo, a element pierwszy wstawi na miejsce ostatniego.

Zadanie 5.2. Za pomocą tylko jednej pętli **for** znajdź w tablicy **double** *tab*[*n*] element najmniejszy i największy.

Zadanie 5.3. Napisz fragment kodu, który w tablicy **bool** *tab*[*n*] zliczy, ile razy występuje wartość **true** oraz dokona negacji wszystkich elementów.

Zadanie 5.4. 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 5.5. Dla danej tablicy liczb rzeczywistych *t* rozmiaru *n* napisz kod, który wyznaczy wartość średniej arytmetycznej wszystkich elementów, danej wzorem $\frac{1}{n}\sum_{i=0}^{n-1} t[i]$.

Zadanie 5.6. Niech dany będzie n-elementowy ciąg liczb rzeczywistych $\mathbf{a} = (a_1, a_2, \ldots, a_n)$. Średnią geometryczną nazywamy wartość

$$\mathsf{GM}(\mathbf{a}) = \sqrt[n]{\prod_{i=1}^{n} a_i}.$$

Napisz program, który wczyta do tablicy $n = 8$ wartości z klawiatury oraz następnie policzy wartość ich średniej geometrycznej.

Zadanie 5.7. Niech dany będzie n-elementowy ciąg liczb rzeczywistych $\mathbf{a} = (a_1, a_2, \ldots, a_n)$. Średnią ważoną względem wektora wag $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ o elementach nieujemnych oraz 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 dla $n = 5$ wczyta z klawiatury wektor wag $\mathbf{w}$ i sprawdzi, czy spełnia on postawione wyżej założenia oraz wyznaczy wartość średniej ważonej ciągu $(-2, -1, 0, 1, 2)$.

Zadanie 5.8. Niech dany będzie n-elementowy ciąg liczb rzeczywistych $\mathbf{a} = (a_1, a_2, \ldots, a_n)$. Operatorem maks-min względem wektora wag $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ składającego się z wartości rzeczywistych, nazywamy wartość

$$\mathsf{MaxMin}_{\mathbf{w}}(\mathbf{a}) = \max_{i=1,2,\ldots,n} \left(\min\{a_i, w_i\}\right).$$

Napisz program, który dla $n = 5$ wczyta z klawiatury ciąg $\mathbf{a}$, następnie wyznaczy wartość operatora maks-min względem wektora wag $(1, 2, \ldots, n)$.

Zadanie 5.9. Niech dany będzie wektor o elementach rzeczywistych $\mathbf{x} = (x_1, \ldots, x_n)$. Napisz program, który wczyta wartości jego elementów z klawiatury (dla $n = 9$) oraz policzy wartość jego normy euklidesowej, wg wzoru

$$|\mathbf{x}| = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Zadanie 5.10. Niech dane będą dwa n-elementowe wektory o elementach rzeczywistych $\mathbf{x} = (x_1, \ldots, x_n)$ i $\mathbf{y} = (y_1, \ldots, y_n)$. Napisz program, który wczyta wartości ich elementów z klawiatury (dla $n = 5$) oraz policzy wartość ich odległości w metryce supremum, wg wzoru

$$d_m(\mathbf{x}, \mathbf{y}) = \max_{i=1,2,\ldots,n} |x_i - y_i|.$$

Zadanie 5.11. Niech dany będzie wielomian rzeczywisty stopnia n, $w(x) = \mathrm{w[0]}x^0 + \mathrm{w[1]}x^1 + \ldots \mathrm{w[n]}x^n$, $w[n] \neq 0$, którego współczynniki przechowywane są w $n + 1$ wymiarowej tablicy o elementach typu **double**. Napisz program, który wczytuje wartość współczynników dla $n = 3$ oraz wartość x i wyznacza wartość $w(x)$ wg powyższego wzoru.

⋆ **Zadanie 5.12.** Zmodyfikuj program z zad. 5.11 tak, by korzystał z bardziej efektywnego obliczeniowo wzoru na wartość $w(x)$, zwanego schematem Hornera:

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

⋆ **Zadanie 5.13.** 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 5.14. Zaimplementuj algorytm sortowania przez wybór dla danej tablicy o *n* elementach typu **int**. Oblicz, ile jest potrzebnych operacji porównań oraz przestawień elementów w zależności od *n*.

Zadanie 5.15. Zaimplementuj algorytm sortowania przez wstawianie dla 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 5.16. Zaimplementuj algorytm sortowania bąbelkowego dla 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 5.17.** Dana jest tablica o *n*>1 elementach typu **double**. Napisz funkcję, która obliczy wariancję jej elementów używając tylko jednej pętli **for**. Wariancja elementów ciągu $\mathbf{x} = (x_1, \ldots, x_n)$ dana jest wzorem

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

gdzie $\overline{\mathbf{x}}$ jest średnią arytmetyczną ciągu $\mathbf{x}$.

4 Wskazówki do ćwiczeń

Odpowiedź do zadania 5.13.

$u(x) = x^8 + x^7 + 4x^6 + 3x^5 + 11x^4 + 2x^3 + 40x^2 - 10x + 20.$

Odpowiedź do zadania 5.14.

```
void zamien(int& x, int& y)
{
    int t = x;
    x = y;
    y = t;
}

int main()
{
    // ...
    const int n = ...;
    int t[n] = { ... };

    for (int i=0; i<n-1; ++i)
    {
        // znajdź indeks najmn. el. spośród t[i], ..., t[n-1];
        int j = i;
        for (int k=i+1; k<n; ++k)
            if (t[k] < t[j])          // porównanie elementów
                j = k;

        zamien(t[i], t[j]);           // zamiana elementów
    }
    // ...
}
```

Analiza liczby porównań elementów. Liczba ta jest ta sama niezależnie od kolejności danych wejściowych i wynosi $(n-1) + (n-2) + \cdots + 1 = n(n-1)/2 = n^2/2 - n/2$.

Liczba zamian elementów w powyższej implementacji jest również niezależna od permutacji danych wejściowych i wynosi $n - 1$. W prosty sposób można zmniejszyć tę liczbę tak, by w przypadku ciągu już posortowanego nie wykonywać żadnych przestawień.

Odpowiedź do zadania 5.15.

Przykładowa implementacja:

```
int main()
{
    // ...
    const int n = ...;
```

```
    int t[n] = { ... };

    for (int i=1; i<n; ++i)
    {
        // znajdź największe j ze zbioru {0,...,i} takie, że
        // j == 0 lub t[i] > t[j-1];
        int j;
        for (j=i; j>0; j--)
            if (t[j-1] <= t[i])            // porównanie
                break;

        // wstaw t[i] przed t[j];
        for (int k = i-1; k>=j; k--)
            zamien(t[k], t[k+1]);          // zamiana
    }
    // ...
}
```

Dla danych już posortowanych liczba porównań wykonywanych przez powyższy kod wynosi $n-1$, a liczba przestawień 0.

Dla danych odwrotnie posortowanych liczba przestawień i porównań równa jest $1+2+\cdots+(n-1)=n(n-1)/2$.

Odpowiedź do zadania 5.16.

```
int main()
{
    // ...
    const int n = ...;
    int t[n] = { ... };

    for (int i=n-1; i>0; --i)
    {
        for (int j=0; j<i; ++j)
        {
            // porównuj elementy parami
            if (t[j] > t[j+1])             // porownanie
                zamien(t[j], t[j+1]);      // zamiana
        }
    }

    // ...
}
```

Liczba porównań jest niezależna od permutacji ciągu wejściowego i wynosi $(n-1)+(n-2)+\cdots+1=n(n-1)/2$.

Liczba przestawień dla danych posortowanych wynosi 0. Dla danych odwrotnie posortowanych równa jest $n(n-1)/2$.

Wskazówka do zadania 5.17. Należy znaleźć wzory (tzw. rekurencyjne) na średnią arytmetyczną i wariancję k początkowych elementów ciągu, zależne od elementu x_k oraz średniej arytmetycznej i wariancji elementów $(x_1, \ldots, x_{k-1})$.

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

VI. Funkcje cz. I

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Funkcje

1.1 main()

Najprostszy pełny kod źródłowy działającego programu w języku C++ można zapisać w sposób następujący.

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

Każdy program w języku C++ musi zawierać definicję obiektu o nazwie *main*(), inaczej jego kompilacja nie zakończy się powodzeniem. *main*() (od ang. *główny*) stanowi punkt startowy każdego programu. Instrukcje w nim zawarte zostają wykonane jako pierwsze.

De facto, obiekt ten jest **funkcją**.

W niniejszej części materiałów przyjrzymy się sposobom deklaracji i użycia funkcji, które stosuje się, aby uprościć *podprogram* główny i zwiększyć czytelność kodu.

Uwaga

Znakomita większość pisanych przez nas programów będzie jednak wpisywać się w poniższy schemat.

```
1  #include <iostream>   // Wczytanie biblioteki
2                        // systemowej iostream
3  using namespace std;  // Udostępnienie zawartych
4                        // w niej narzędzi
5
6  int main()
7  {
8      // kod
9      // ...
10
11     return 0; // zakończenie programu (,,sukces'')
12 }
```

Zwróćmy uwagę na dwie pierwsze instrukcje. Służą one do wczytania i udostępnienia biblioteki `iostream`, w której zawarte są m.in. definicje obiektów *cout* i *cin*. Dzięki nim możemy wypisywać komunikaty na ekranie oraz pobierać dane z klawiatury.

1.2 Definiowanie funkcji

Na etapie projektowania programu często można wyróżnić wiele **modułów** (części, podprogramów), z których każda jest odpowiedzialna za pewną logicznie wyodrębnioną, niezależną czynność. Fragmenty mogą cechować się dowolną złożonością. Mogą same np. składać się z kolejnych podmodułów.

Zadaniem programisty jest powiązanie tych fragmentów w spójną całość, m.in. poprzez zorganizowanie odpowiedniego przepływu sterowania i wymiany informacji (proces taki może przypominać budowanie domku z klocków różnych kształtów).

Rozważmy dwa przykładowe zalążki programów, obrazujące pewne sytuacje z tzw. życia. Najpierw przyjrzyjmy się czynnościom potrzebnym do uruchomienia samochodu.

```
int main()
{
    ustawFotel();
    ustawLusterka();
    zapnijPasy();
    przekręćKluczyk();
    sprawdźKontrolki();
    włączŚwiatła();
    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.

Drugi przykład dotyczy organizacji nauki w semestrze pewnego pilnego studenta.

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

        pouczSięTrochę();

        if (zmęczony) {
            if (godzinNaukiDziś > 5)
                możeszWreszcieIśćNaRandkę();
            else
                odpocznijChwilę();
        }
    } while (!nauczony);
    // ...
}
```

Najprostszym praktycznym sposobem podziału programu na moduły jest użycie funkcji[1].

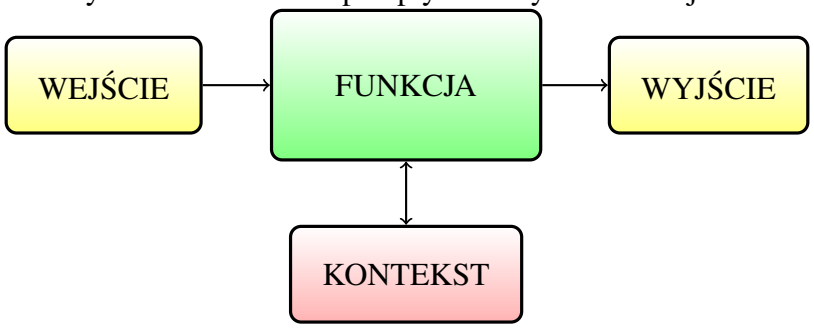
[1] W semestrze II poznamy jeszcze inny sposób, pozwalający na pisanie naprawdę dużych programów: tzw. programowanie obiektowe.

Funkcja to odpowiednio wydzielony fragment kodu, wykonujący pewne instrukcje, do którego można się odwoływać z innych miejsc programu.

Projektując funkcję należy wziąć pod uwagę, jak ma ona wchodzić w interakcję z innymi funkcjami (np. *main*()), to znaczy jakiego typu **dane wejściowe** powinna ona przyjąć i jakiego typu **dane wyjściowe** będą wynikiem jej działania.

Bezpośrednim skutkiem działania funkcji jest uzyskanie jakiejś konkretnej wartości na wyjściu (może to też być „nic", które jest reprezentowane przez typ **void**). **Skutkiem pośrednim** działania funkcji może być zmiana kontekstu, tj. stanu komputera (np. wypisanie czegoś na ekran). Idea ta została zobrazowana na rys. 1.

Rysunek 1: Schemat przepływu danych w funkcjach.



Składnia definicji **funkcji**:

```
typZwracany nazwaFunkcji(listaArgumentów)   // (nagłówek)
{
    // kod funkcji (ciało)
}
```

gdzie:

- *typZwracany* to przeciwdziedzina, np. **int**, **bool**, **void** itp.,
- *nazwaFunkcji* jest identyfikatorem (zob. wykład o deklaracji zmiennych),
- a *listaArgumentów* deklaruje dziedzinę, tj. parametry wejściowe (tak jak deklaruje się zmienne), oddzielając je przecinkami.

Do zwracania wartości przez funkcję (należących do określonej przeciwdziedziny) służy instrukcja **return**. Działa ona podobnie jak **break** w przypadku pętli, tj. natychmiast przerywa wykonywanie funkcji, w której aktualnie znajduje się sterowanie. Dzięki temu następuje powrót do miejsca, w którym nastąpiło wywołanie danej funkcji.

Jak już wspomnieliśmy, za *typZwracany* można przyjąć też **void**, czyli „nic" (∅). Jedynym skutkiem działania takiej funkcji jest zmiana kontekstu. W tym przypadku instrukcja **return** może zostać pominięta. Po wykonaniu ostatniej instrukcji następuje automatyczny powrót do funkcji wywołującej.

1.3 Definicja a deklaracja

Rozpatrzmy definicję funkcji służącej do wyznaczania kwadratu liczby rzeczywistej. Wyraźmy ją za pomocą notacji matematycznej.

Niech

$$\underbrace{\mathsf{kwadrat}:}_{\text{nazwa funkcji}}\ \underbrace{\mathbb{R}}_{\text{dziedzina}}\ \rightarrow\ \underbrace{\mathbb{R}}_{\text{przeciwdziedzina}}$$

taka, że

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

Pierwsza część — „Niech…” — powyższej definicji nazywa się **deklaracją** (prototypem). Za jego pomocą orzekamy, że mamy zamiar określić funkcję o danej nazwie, posiadającą pewną dziedzinę i pewną przeciwdziedzinę. W tym momencie nie wiadomo jednak jeszcze, *co i jak* dokładnie taka funkcja ma robić.

Dlatego dalej — „taka, że …” — następuje **właściwa część definicji**, która podaje dokładny sposób (algorytm) przetworzenia danych wejściowych celem uzyskania spodziewanego wyniku.

Powyższa definicja określa to postępowanie w następujący sposób.

a) Weź argument wejściowy (będący liczbą rzeczywistą). Niech w tym kontekście będzie on identyfikowany jako x (równie dobrze mógłby to być każdy inny symbol, np. y, α, Ω_1).
b) Wyznacz wartość iloczynu $x \times x$.
c) Rezultat obliczeń z p. b) (liczbę rzeczywistą) zwróć jako wynik.

Przyjrzyjmy się implementacji rozpatrywanej funkcji i jej przykładowemu zastosowaniu.

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

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

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

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.

Jednakże czasem dobrze jest (np. dla czytelności) umieścić ją w innym miejscu (poniżej, w innymi pliku itp.). Można to zrobić, pod warunkiem, że obiektom z niej korzystającym udostępnimy jej deklarację. Intuicyjnie, innym obiektom wystarczy przekazać informację, że dana funkcja istnieje. Dokładna specyfikacja jej działania nie jest im de facto potrzebna.

Deklaracji funkcji dokonujemy przez podanie jej nagłówka zakończonego średnikiem w miejscu, w którym nie została ona jeszcze użyta (ale poza inną funkcją), według składni:

```
typZwracany nazwaFunkcji(listaArgumentów);
```

Co ważne, w deklaracji musimy użyć takiej samej nazwy funkcji oraz typów argumentów wejściowych i wyjściowych jak w definicji (która ostatecznie musi się gdzieś pojawić). Nazwy tych argumentów mogą być jednak inne, a nawet mogą zostać pominięte.

Dlatego nasz przykład również mógłby być zapisany tak:

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

double kwadrat(double); // deklaracja

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;
}

double kwadrat(double x) // definicja
{
    return x*x;
}
```

Uwaga

Warto mieć na uwadze, że instrukcja

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

zostanie wykonana w sposób podobny do następującego:

```
double zmiennaPomocnicza1 = kwadrat(0.5);
double zmiennaPomocnicza2 = kwadrat(zmiennaPomocnicza1);
cout << zmiennaPomocnicza2;
```

Wartości pośrednie są obliczanie i odkładane „na boku”. Tym samym, podanie funkcji *kwadrat* (czy dowolnej innej przyjmującej jako parametr typ **double**) jako 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 się jednak nie dzieje.

1.4 Biblioteki funkcji

Zbiór funkcji podręcznych można umieścić w osobnych plikach, tworząc tzw. **bibliotekę** funkcji. Dzięki temu program staje się bardziej czytelny, można łatwiej zapanować nad jego złożonością, a ponadto otrzymujemy sposobność wykorzystania pewnego kodu ponownie w przyszłości. Jest to ważne, gdyż raz napisany i przetestowany zestaw funkcji oszczędza nam sporo pracy.

Aby zatem stworzyć bibliotekę, tworzymy najczęściej dwa dodatkowe pliki.

- plik nagłówkowy (ang. *header file*), `nazwabiblioteki.h` — zawierający tylko deklaracje funkcji,
- plik źródłowy (ang. *source file*), `nazwabiblioteki.cpp` — zawierający tylko definicje funkcji.

Plik nagłówkowy należy dołączyć do wszystkich plików, które wykorzystują funkcje z danej biblioteki (również do pliku źródłowego biblioteki) 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 ładują biblioteki systemowe).

Zauważmy, że podczas laboratoriów już korzystaliśmy z jednej biblioteki. Był to zestaw funkcji służący do rysowania. W funkcji *main*() wywoływaliśmy je celem stworzenia cieszących oczy obrazków. Z pomocą jednej biblioteki można by było napisać bardzo dużo programów, jeden rysujący np. domek, drugi kotka itd.

Dla ilustracji przyjrzymy się, jak wyglądałby program składający się z biblioteki zawierającej dwie następujące funkcje dane w notacji matematycznej.

Niech

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

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

takie, że

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

oraz

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

Oto zawartość pliku `podreczna.h`.

```
1 #pragma once   // na początku każdego pliku nagłówkowego!
2
3 int min(int i, int j);
4 int max(int i, int j);
```

Uwaga

Dyrektywa

```
#pragma once
```

powinna się znaleźć na początku każdego pliku nagłówkowego. Zapobiega to ponownemu dołączeniu tego pliku, jeśli i inne biblioteki korzystają z danej.

Ta dyrektywa jest dostępna tylko w kompilatorze Visual C++. W innych narzędziach uzyskanie tego efektu jest nieco bardziej skomplikowane:

```
#ifndef __PODRECZNA_H
#define __PODRECZNA_H

// deklaracje....

#endif
```

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

```
1  #include "podreczna.h" /* dołącz plik z deklaracjami */
2
3  int min(int x, int y)  /* nazwy parametrów nie muszą być takie
       same jak w deklaracji */
4  {
5      if (x <= y) return x;
6      else        return y;
7  }
8
9  int max(int w, int z)
10 {
11     if (w >= z)
12         return w;
13     return z;
14 }
```

Gdy biblioteka jest gotowa, można przystąpić do napisania funkcji głównej, która będzie posiłkować się nimi do własnych celów. Oto zawartość pliku `program.cpp`.

```
1  #include <iostream>     /* dołącz bibliotekę systemową */
2  using namespace std;
3
4  #include "podreczna.h" /* dołącz własną bibliotekę */
5
6  int main(void) // tak też można zadeklarować
7                 // funkcję bezargumentową
```

```
{
    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;
}
```

2 Przegląd funkcji w bibliotekach systemowych

Elementem standardu języka C++ jest też wiele przydatnych bibliotek systemowych zawierających często wykorzystywane funkcje. Poniżej omówimy te, które interesować nas będą najbardziej.

2.1 Funkcje matematyczne

Biblioteka <*cmath*> udostępnia wybrane funkcje matematyczne[2]. Ich przegląd zamieszczamy w tab. 1, 2 i 3.

Tablica 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 (wynik w rad.)

Dla przypomnienia, funkcja „sufit” określona jest jako

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

a funkcja „podłoga” zaś jako

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

[2]Pełna dokumentacja biblioteki <*cmath*> w języku angielskim dostępna jest do pobrania ze strony http://www.cplusplus.com/reference/clibrary/cmath/.

Tablica 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

Tablica 3: Funkcje dodatkowe w bibliotece <*cmath*>.

Deklaracja	Znaczenie
double *fabs*(**double**);	wartość bezwzględna
double *ceil*(**double**);	„sufit”
double *floor*(**double**);	„podłoga”

2.2 Liczby pseudolosowe

W bibliotece <*cstdlib*> znajdują się funkcje służące do generowania liczb pseudolosowych. Zawarta jest tam funkcja, dająca za pomocą czysto algebraicznych metod w wyniku liczby, które można traktować jako (które wyglądają jak) liczby losowe.

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 się funkcja time()

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

Funkcja

```
int rand();
```

generuje całkowite liczby losowe z rozkładu dyskretnego jednostajnego określonego na zbiorze

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

Jednostajność oznacza, że prawdopodobieństwo „wylosowania” każdej z liczb jest takie samo.

Uwaga

RAND_MAX jest stałą zdefiniowaną w bibliotece <*cstdlib*>.

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

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

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

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

Uwaga

Korzystając z własnoręcznie napisanej funkcji generującej liczbę rzeczywistą $\in [0, 1]$

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

łatwo napisać funkcję „losującą” liczbę ze zbioru $\{a, a+1, \ldots, b\}$, gdzie $a, b \in \mathbb{Z}$:

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

2.3 Asercje

Biblioteka *cassert* udostępnia funkcję o następującej deklaracji:

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

Umożliwia ona 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.

Dla przykładu rozpatrzmy „bezpieczną” funkcję wyznaczającą pierwiastek z nieujemnej liczby rzeczywistej.

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

double pierwiastek(double x)
{
```

[3]Podana metoda nie cechuje się zbyt dobrymi własnościami statystycznymi. Szczegóły poznamy jednak dopiero na laboratoriach ze statystyki matematycznej w semestrze VI.

```
6       \\ Dane: x ⩾ 0
7       \\ Wynik: √x
8
9       assert(x>=0); // jeśli nie, to błąd – zakończenie programu
10      return sqrt(x);
11  }
```

Uwaga

Sprawdzanie wszelkich warunków za pomocą *assert* () można wyłączyć globalnie za pomocą dyrektywy

```
#define NDEBUG
```

umieszczonej na początku pliku źródłowego bądź nagłówkowego.

3 Ćwiczenia

Zadanie 6.1. Napisz funkcję parzysta, która sprawdza czy dany argument typu **int** jest liczbą parzystą czy nieparzystą. Zwróć wynik typu **bool**.

Zadanie 6.2. Napisz funkcję silnia, która dla danego $n \in \mathbb{N}$ zwraca wartość $1 \times 2 \times \cdots \times n$.

Zadanie 6.3. Napisz funkcję max, która dla danych $a, b, c \in \mathbb{Z}$ zwraca ich maksimum.

Zadanie 6.4. Napisz funkcję med, która znajduje medianę (wartość środkową) trzech liczb rzeczywistych, np. $\mathrm{med}(4, 2, 7) = 4$ i $\mathrm{med}(1, 2, 3) = 2$.

Zadanie 6.5. Napisz funkcję nwd zwracającą największy wspólny dzielnik dwóch liczb naturalnych.

Zadanie 6.6. Napisz funkcję nww zwracającą najmniejszą wspólną wielokrotność dwóch liczb naturalnych.

Zadanie 6.7. 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 $\mathrm{BMI} = \mathrm{waga}/\mathrm{wzrost}^2$. (Ciekawostka: wg WHO BMI od $18{,}5$ do $25{,}0$ jest uznawana za wartość prawidłową.)

Zadanie 6.8. Napisz funkcję odl, która przyjmuje współrzędne rzeczywiste dwóch punktów *x1*, *y1*, *x2*, *y2* i zwraca ich odległość euklidesową daną wzorem $|\mathbf{x}-\mathbf{y}| = \sqrt{(x1 - y1)^2 + (x2 - y2)^2}$.

Zadanie 6.9. Napisz funkcję odlsup, która przyjmuje współrzędne rzeczywiste dwóch punktów *x1*,*y1*,*x2*,*y2* i zwraca ich odległość w metryce supremum, tj.

$$d_m(\mathbf{x}, \mathbf{y}) = \max\{|x1 - y1|, |x2 - y2|\}.$$

Zadanie 6.10. 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

$$||\mathbf{x} - \mathbf{y}||_p = \sqrt[p]{|x1 - y1|^p + |x2 - y2|^p}.$$

Zadanie 6.11. $\sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{i} x^i$ jest rozwinięciem funkcji $\ln(x+1)$ dla $(-1, 1]$ w szereg Taylora. Napisz funkcję lognat02, która dla danego $x \in (0, 2]$ zwraca przybliżenie wartości $\ln x$, a dla $x \notin (0, 2]$ wartość NaN.

Zadanie 6.12. Wiemy, że szereg $\sum_{i=0}^{\infty} \frac{(-1)^i\ (2i)!\ x^i}{(1-2i)\ (i!)^2\ (4^i)}$ dla $|x| < 1$ jest zbieżny i jego suma równa jest $\sqrt{1+x}$. Napisz funkcję pierw02, która dla danej liczby rzeczywistej $x \in [0, 2]$ znajduje przybliżenie jej pierwiastka na podstawie podanego wzoru, a dla liczb $x \notin [0, 2]$ zwraca NaN.

Zadanie 6.13. Dane są rozwinięcia następujących funkcji w szereg Taylora.

a) $e^x = \sum_{i=0}^{\infty} \frac{x^n}{n!}$ dla pewnego $x \in \mathbb{R}$,

b) $\sin x = \sum_{\infty}^{100} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$ dla pewnego $x \in \mathbb{R}$,

c) $\cos x = \sum_{\infty}^{100} \frac{(-1)^n}{(2n)!} x^{2n}$ dla pewnego $x \in \mathbb{R}$,

d) $\arcsin x = \sum_{\infty}^{100} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1}$ dla pewnego $x \in (-1, 1)$.

Napisz funkcje w C++, które przybliżają wartości powyższych sum bądź zwracają NaN dla argumentów poza obszarem zbieżności.

Zadanie 6.14. 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 6.15. Niech dane będą a,b,c typu **double**. Zmienne te definiują równanie względem niewiadomej $x \in \mathbb{R}$ postaci

$$ax^2 + bx + c = 0.$$

Zaproponuj funkcję w języku C++, która wyznaczy jego rozwiązanie i wypisze wynik na ekran. Poprawnie identyfikuj przypadki, gdy dane równanie nie ma rozwiązań w $\mathbb{R}$, a także, gdy nie jest ono równaniem kwadratowym.

Zadanie 6.16. Napisz funkcję implementującą grę w „Zgadulę". Losuje ona liczbę całkowitą z zakresu od 1 do 100. Użytkownik próbuje odgadnąć liczbę wprowadzając swe typy z klawiatury, póki jej nie zgadnie. Za każdym razem otrzymuje komunikat zwrotny, np. „za mało" bądź „za dużo".

Zadanie 6.17. Napisz funkcję implementującą grę w „Zgadulę" zawierającą elementy „sztucznej inteligencji". Losuje ona liczbę całkowitą z zakresu od 1 do 100. Następnie komputer sam próbuje odgadnąć tę liczbę, przy okazji wypisując swe typy na ekranie. Zaproponuj prosty algorytm, który (nie oszukując!) znajdzie poprawne rozwiązanie w średnio jak najmniejszej liczbie kroków.

4 Wskazówki do ćwiczeń

Wskazówka do zadania 6.11. By znaleźć przybliżenie wartości funkcji albo rozpatrz tylko n początkowych wyrazów szeregu, np. $n = 30$, albo przerwij obliczenia dopiero, gdy moduł kolejnego dodawanego wyrazu jest mniejszy niż założone δ, np. $\delta = 10^{-6}$.

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

VII. Funkcje cz. II. Rekurencja

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Rozszerzenie wiadomości o funkcjach

W niniejszym paragrafie rozszerzymy naszą wiedzę związaną z funkcjami o bardziej zaawansowane zagadnienia.

1.1 Zasięg zmiennych

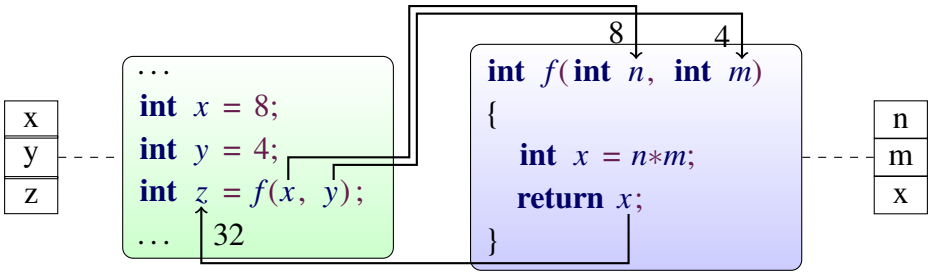
Przyjrzyjmy się **zasięgowi** definiowanych przez nas zmiennych. Zasięg określa, jak długo dany obiekt istnieje i w jaki sposób dane zmienne są widoczne z innych miejsc programu.

Wszystkie 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. I tak, zmienna x w funkcji $f()$ to zmienna inna niż x w funkcji $g()$. Funkcja nie ma możliwości bezpośredniego odwołania się do zmiennej lokalnej innej funkcji, nawet przez się wywoływanej. Zatem jedynym sposobem na wymianę 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. 1.



Rysunek 1: 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.

x i y są przekazane do $f()$ **przez wartość**. To znaczy, że parametry są obliczane przed wywołaniem (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** 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.

Uwaga

Warto zapamiętać, że zmienne przekazywane **przez wartość** są **kopiowane**, co w przypadku wielu wywołań funkcji na dużych obiektach (należących do złożonych struktur danych, zob. wykład X) może być czasochłonne.

Ponadto, 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 siebie samą. Więcej szczegółów podamy w kolejnej części niniejszego wykładu.

Uwaga

W języku C++ dostępne są także zmienne globalne. Jednakże stosowanie ich nie jest zalecane. Nie będziemy ich zatem omawiać.

1.2 Przekazywanie parametrów przez referencję

Standardowo parametry wejściowe funkcji zachowują się jak zmienne lokalne — ich zmiana nie jest odzwierciedlona „na zewnątrz”.

Rozważmy następujący przykład.

```
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 parametrów przez wartość
12                   // (skopiowanie wartości)
13     cout << n << ", " << m << endl;
14     return 0;
15 }
```

Wynikiem tej operacji 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 poprzez dodanie znaku & po nazwie typu zmiennej na liście parametrów funkcji, wg składni:

```
typ& identyfikator
```

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 takie, jakie są (lecz być może pod inną nazwą).

Co ważne, w taki sposób można tylko przekazać zmienną! Przekazanie wartości zwracanej przez jakąś inną funkcję, stałej albo złożonego wyrażenia arytmetycznego zakończyłoby się niepowodzeniem.

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 parametrów przez referencję
    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 prawidłowy 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.

Popatrzmy na 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;   // wejście
    int i, r;           /* zmienne, które wykorzystamy
```

```
11                                  do przekazania wyniku */
12      iloreszta(n, m, i, r);
13      cout << n << "=" << i << "*" << m << "+" << r;
14      return 0;
15  }
```

Wynikiem tego programu będzie więc `7=3*2+1`.

1.3 Parametry domyślne funkcji

Parametry domyślne to argumenty, których jawne pominięcie przy wywołaniu funkcji powoduje, że zostaje im przypisana pewna z góry ustalona wartość.

Składnia deklaracji parametru domyślnego:

```
typ identyfikator = wartość
```

Parametry z wartościami domyślnymi mogą być tylko przekazywane przez wartość. Mogą się one pojawić tylko na końcu listy parametrów (może być ich wiele).

Jeśli rozdziela się definicję i deklarację funkcji, parametry domyślne powinny się pojawić tylko w jednym miejscu w kodzie programu. Jednakże dla czytelności lepiej jest, gdy pojawiają się w deklaracji.

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);`

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);`

Dla ilustracji rozważmy funkcję wyznaczającą pierwiastek liczby rzeczywistej, domyślnie o podstawie 2.

```
1  #include <cmath>
2
3  double pierwiastek(double x, double p=2)
4  { // pierwiastek, domyślnie kwadratowy
5      assert(p >= 1);
6      return pow(x, 1.0/p);
7  }
8
```

```
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;
}
```

1.4 Przeciążanie funkcji

W języku C++ można nadawać te same nazwy (identyfikatory) różnym 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 rozróżniania funkcji przeciążonych od siebie.

Ma to sens, gdy funkcje wykonują podobne (w sensie interpretacyjnym) czynności, jednakże na danych różnego typu.

Oto kilka przykładów:

```
int    modul(int x);
double modul(double x);
```

```
void f();
int  f(int x, int y=2);
int  f(int x); // Błąd! – szczególny przypadek powyższego
```

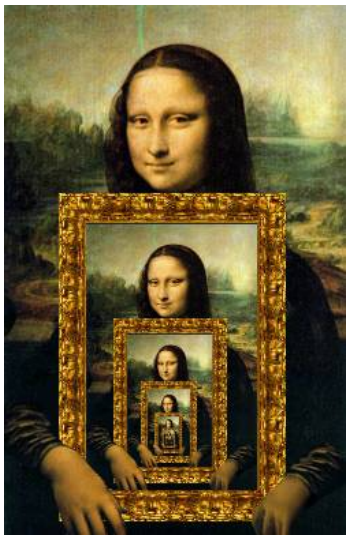
```
bool g(int x, int y);
char g(int x, int y);   // Błąd! – nie różnią się argumentami
```

2 Rekurencja

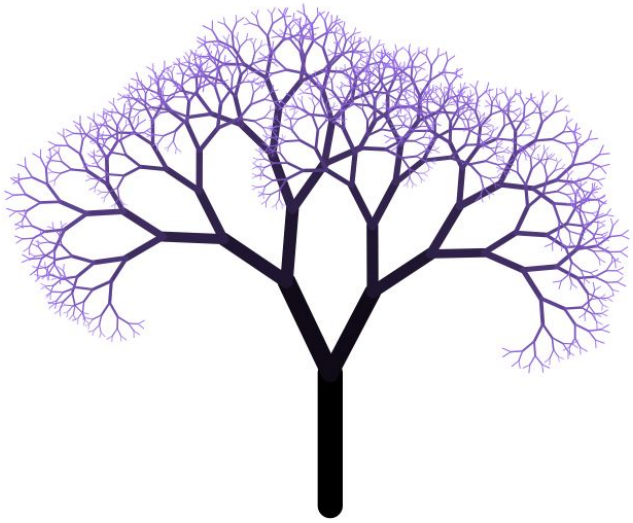
Z **rekurencją** (bądź rekursją, ang. *recursion*) mamy do czynienia wtedy, gdy w definicji pewnego obiektu pojawia się odniesienie do niego samego.

Rozważmy najpierw rys. 2. Przedstawia on pewną znaną, zacną damę, trzymającą obraz, który przedstawia ją samą trzymającą obraz, na którym znajduje się ona sama trzymająca obraz... Podobny efekt moglibyśmy uzyskać filmując kamerą telewizor pokazujący to, co właśnie nagrywa kamera.

A teraz popatrzmy na rys. 3 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.



Rysunek 2: Rekurencyjna Mona Lisa. Źródło: http://www.dominiek.eu/blog/wp-content/uploads/2007/11/megamonalisa_recursion.jpg.



Rysunek 3: Fraktalne drzewo. Źródło: http://diffusedproductions.net/images/fractal-tree2.jpg.

W przypadku języków programowania mówimy o rekurencji, **gdy funkcja wywołuje samą siebie**. Jednakże takie postępowanie spowodowałoby być może zawieszenie się komputera. Istotną cechą rekurencji jest więc **warunek stopu**, czyli zdefiniowanie przypadku, w którym zagłębianie się rekurencyjne zostaje przerwane.

Uwaga

Każde wywołanie rekurencyjne powoduje utworzenie nowego zestawu zmiennych lokalnych!

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 ciekawych przykładów takich zagadnień.

2.1 Przykład: silnia

W poniższej definicji silni pojawia się odniesienie do... silni. Zauważmy jednak, że obiekt ten jest dobrze określony, gdyż podany został warunek stopu.

$$\begin{aligned} 0! &= 1, \\ n! &= n\,(n-1)! \text{ dla } n > 0. \end{aligned}$$

Funkcję w języku C++ służącą do obliczenia silni można napisać, bazując wprost na definicji.

```
int silnia(int n)
{
    assert(n >= 0);
    if (n == 0) return 1;
    else return n*silnia(n-1);
}
```

Dla porównania przyjrzyjmy się, jak by mogła wyglądać analogiczna funkcja nierekurencyjna.

```
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.

2.2 Przykład: NWD

Poznaliśmy już algorytm wyznaczania największego wspólnego dzielnika dwóch liczb naturalnych. Okazuje się, że NWD można także określić poniższym równaniem rekurencyjnym. Niech $1 \leqslant n \leqslant m$.

$$\mathsf{NWD}(n,m) = \begin{cases} m & \text{dla } n = 0, \\ \mathsf{NWD}(m \bmod n, n) & \text{dla } n > 0. \end{cases}$$

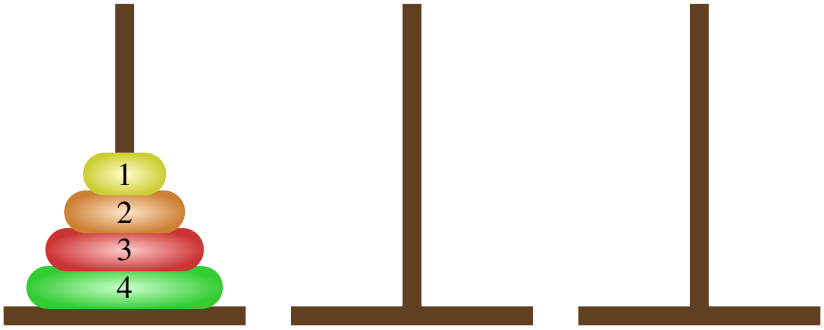
Przekłada się ona bezpośrednio na następujący kod w C++.

```
int nwd(int n, int m)
{
    assert(n >= m && n >= 0);
    if (n == 0) return m;
```

```
    else return nwd(m % n, n);
}
```

2.3 Przykład: wieże z Hanoi

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. 4.



Rysunek 4: 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 się znaleźć na mniejszym.

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).

Kod 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);
    }
}
```

Ciekawym zagadnieniem jest wyznaczenie liczby przestawień potrzebnych do rozwiązania łamigłówki. W zaproponowanym algorytmie jest to:

$$\begin{aligned} L(1) &= 1, \\ L(n) &= L(n-1) + 1 + L(n-1) \text{ dla } n > 0. \end{aligned}$$

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. Podobno, gdy skończą oni swe zmagania, nastąpi koniec świata. Zakładając jednak, że nawet gdyby wykonanie jednego ruchu zajmowało 1 sekundę, to na pełne rozwiązanie potrzeba by wtedy aż $2^{64} - 1 = 18446744073709551615$ sekund, czyli około 584542 miliardów lat. To ponad 400 razy dłużej niż szacowany wiek Wszechświata!

2.4 Przykład: 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. Para osiąga płodność po upłynięciu jednej jednostki czasu od narodzenia. Z każdej płodnej pary rodzi się w kolejnej jednostce jedna para potomstwa.

Liczbę par królików w populacji w chwili n można opisać za pomocą tzw. liczb Fibonacciego.

$$\begin{aligned} F(0) &= 1, \\ F(1) &= 1, \\ F(n) &= F(n-1) + F(n-2) \text{ dla } n > 1. \end{aligned}$$

W wyniku otrzymujemy zatem ciąg $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$.

Bezpośrednie przełożenie powyższego równania na kod w C++ może wyglądać jak niżej.

```
int fibrek(int n)
{
    if (n > 1) return fibrek(n-1)+fibrek(n-2);
    else return 1;
}
```

Bezpośredni algorytm rekurencyjny jest jednak nieefektywny! Przypatrzmy się, w jaki sposób przebiega wywołanie dla *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 jest potrzebnych operacji arytmetycznych (dodawanie) potrzebnych do znalezienia F_n za pomocą powyższej funkcji. Liczbę tę można opisać równaniem:

$$\begin{aligned} L(0) &= 0, \\ L(1) &= 0, \\ L(n) &= L(n-1) + L(n-2) + 1 \text{ dla } n > 1. \end{aligned}$$

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 szybko. Dla przykładu czas potrzebny do policzenia F_{40} na 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 jednak, że rozwiązanie rekurencyjne jest nieefektywne. Zatem pozostaje jedynie użycie wersji iteracyjnej, które wykorzystuje tylko co najwyżej n operacji arytmetycznych!

3 Ćwiczenia

Zadanie 7.1. Można pokazać, że $\sum_{i=0}^{\infty} \frac{x^i}{i!}$ jest rozwinięciem funkcji e^x w szereg Taylora. Napisz dwie wersje funkcji Exp, które dla danego $x \in \mathbb{R}$ znajdują przybliżenie jego eksponensu na podstawie podanego wzoru.

a) W jednej rozpatrz tylko n początkowych wyrazów szeregu, np. $n = 30$,
b) W drugiej przerwij obliczenia dopiero, gdy moduł kolejnego dodawanego wyrazu jest mniejszy niż założone δ, np. $\delta = 10^{-6}$.

Rada: Niech n i δ będą parametrami funkcji z wartościami domyślnymi.

Zadanie 7.2. Napisz kilka przeciążonych wersji funkcji swap, które przestawiają wartości dwóch argumentów wejściowych o typach **int**, **double** i **bool**.

Zadanie 7.3. Napisz nierekurencyjną funkcję służącą do znalezienie n-tej liczby Fibonacciego.

Zadanie 7.4. Napisz funkcję swap, która za pomocą ciągu przypisań przestawia wartości czterech zmiennych rzeczywistych (a, b, c, d), tak by na wyjściu otrzymać (c, b, d, a).

Zadanie 7.5. Napisz funkcję sort, która porządkuje niemalejąco wartości trzech argumentów wejściowych (liczby całkowite).

Zadanie 7.6. Zaproponuj funkcję wyznaczającej wartość tzw. funkcji 91 McCarthy'ego.

$$M(n) = \begin{cases} n - 10 & \text{dla } n > 100, \\ M(M(n + 11)) & \text{dla } n \leqslant 100. \end{cases}$$

Ciekawostka: okazuje się, że $M(n) = 91$ dla każdego $n \leqslant 101$ oraz $M(n) = M(n) - 10$ dla $n > 101$.

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

VIII. Wskaźniki. Dynamiczna alokacja pamięci

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Dynamiczna alokacja pamięci

1.1 Organizacja pamięci komputera

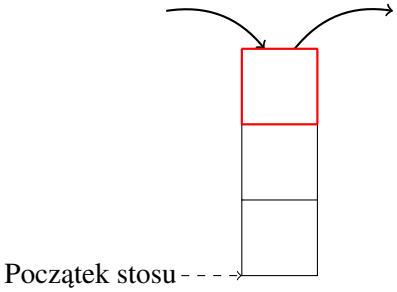
Jak już wiemy, w **pamięci operacyjnej** komputera przechowywane są zarówno programy jak i dane. Jej jednostką podstawową jest **komórka** o rozmiarze jednego bajta. Każda komórka pamięci posiada swój **adres**. Reprezentowany jest on we współczesnych komputerach jako 32- lub 64-bitowa liczba całkowita (bez znaku).

Z punktu widzenia każdego programu można wyróżnić następujący **podział puli adresowej** pamięci (w architekturze von Neumanna):

- kod programu — dane interpretowane są jako instrukcje procesora,
- stos — przechowywane są wartości zmiennych lokalnych funkcji,
- sterta — 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** (ang. *stack*) i **stercie** (ang. *heap*).

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*, zob. rys. 1). 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).



Rysunek 1: Umieszczanie i kasowanie danych na stosie.

1.2 Wskaźniki

Każda zmienna ma przyporządkowaną komórkę (bądź komórki) pamięci, w której przechowuje swoje dane, np. zmienna typu **int** zajmuje 4 takie komórki (4 bajty).

Fizyczny adres zmiennej (czyli numer komórki) można sprawdzić za pomocą operatora &.

```
int x;
cout << "x znajduje się pod adresem " << &x;
// np. 0xe3d30dbc
```

Istnieje specjalny typ danych do przechowywania informacji o adresach innych zmiennych, zwany **wskaźnikami**. Składnia deklaracji wskaźnika na zmienną typu **typ** (czyli deklaracji zmiennej przechowującej adres w pamięci jakiejś jednostki danych typu **typ**) jest następująca:

```
typ* zmienna; // * oznacza: wskaźnik
```

Uwaga

Istnieje specjalne miejsce w pamięci o adresie 0 (*NULL*), do którego odwołanie się powoduje wystąpienie błędu. Często używa się tego adresu np. dla niezainicjowanych wskaźników w celu oznaczenia, że nie wskazują one „nigdzie".

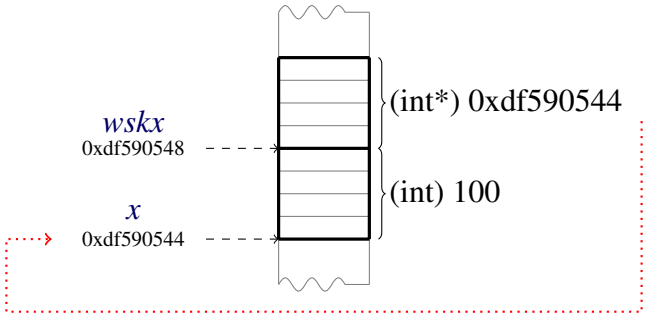
Na wskaźnikach został określony tzw. **operator wyłuskania**, *, dzięki któremu możemy odczytać, co się znajduje pod danym adresem pamięci.

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. 2. Każda z tych zmiennych umieszczona jest pod pewnym adresem w pamięci RAM. Początkowy numer komórki można oddczytać za pomocą operatora &.

Listing 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
```

Wypisanie wartości wskaźnika oznacza wypisanie adresu, na który wskazuje. Wypisanie „wyłuskanego" wskaźnika zaś 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*.



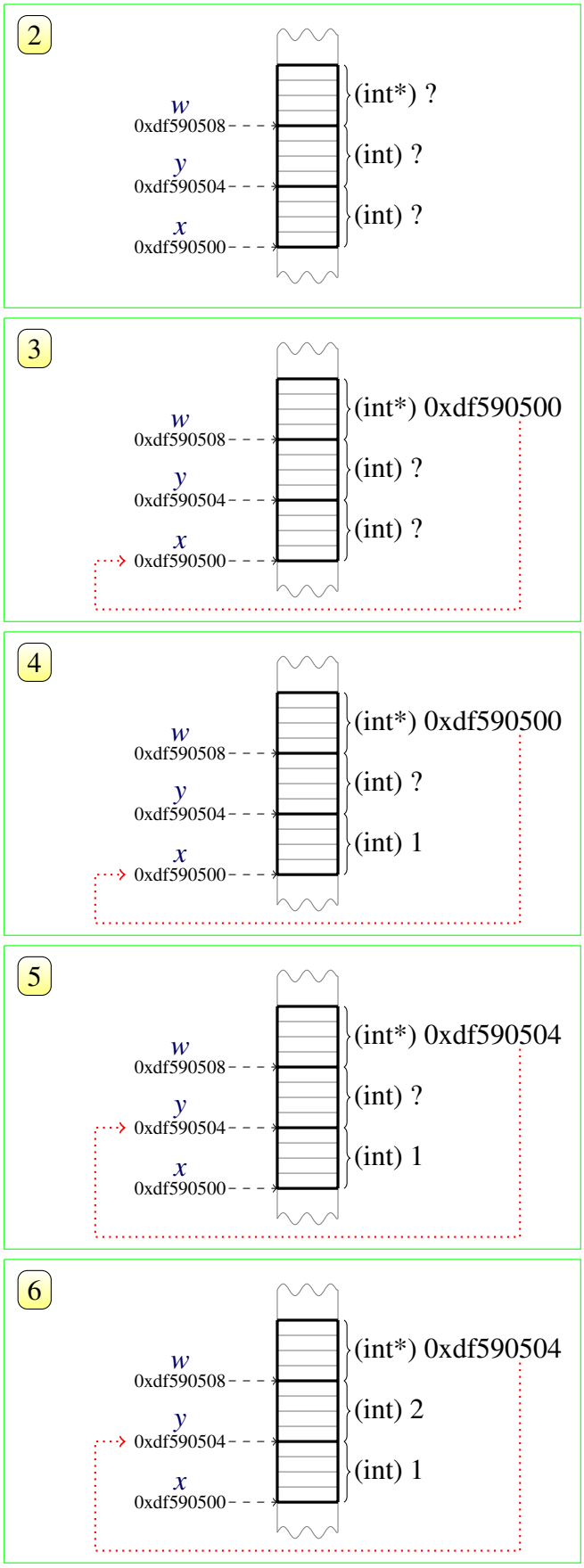
Rysunek 2: Zawartość pamięci w programie 1.

Aby jeszcze lepiej zrozumieć omawiane zagadnienie, rozważmy fragment kolejnego programu.

Listing 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. 3. Tym razem za pomocą operatora wyłuskania zapisujemy dane do komórek pamięci, na które pokazuje wskaźnik *w*.



Rysunek 3: Zawartość pamięci po wykonaniu kolejnych linii kodu z przykładu 2.

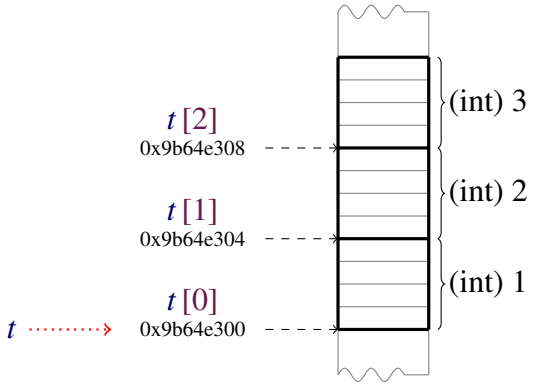
1.3 Tablice a wskaźniki

Zapewne ciekawi nas, w jaki sposób zorganizowana jest pamięć na stosie, gdy deklarowane są tablice jednowymiarowe. Możemy to sprawdzić w następujący sposób.

Listing 3: Organizacja pamięci dla tablic.

```
int t[3] = {1,2,3};
cout << &t;      // np. 0x9b64e300
```

Kolejne elementy tablicy w pamięci zawsze następują po sobie (tablica tworzy spójny ciąg bajtów), zob. rys. 4.



Rysunek 4: Organizacja pamięci w przypadku tablic w przykładzie 3.

Ponadto, zwróćmy uwagę na to, co się dzieje, gdy wykonamy następujący kod.

```
cout << &t;      // np. 0x9b64e300 - znajduje sie tu
cout << t;       //      0x9b64e300 - wskazuje tu
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).

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;
}
```

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 SIĘ GORĄCO!!! :-)
    return s;
}
```

```
double suma(double* t, int n)
{
    double s = 0.0;
    for (int i=0; i<n; ++i)
        s += *(t++); // OLABOGA!!!
    return s;
}
```

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**.

Uwaga

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 obiekt; // przydział (zwraca wskaźnik)
// ...
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 już nie musi być stała
typ* obiekt = new obiekt[n]; // przydział (zwraca wskaźnik)
// ...
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 się uczyłeś do kolokwium?";
5  cin >> n;
6
7  godzinyNauki = new double[n]; // utwórz tablicę o n elementach
8
9  cout << "Ile godzin się uczyłeś każdego dnia?";
10 for (int i=0; i<n; i++)
11    cin >> godzinyNauki[i];
12
13 //....
14 cout << "I tak za mało :-)";
15
16 delete [] godzinyNauki; // tablica już nie jest potrzebna dalej
```

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.

2 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 bądź przetworzyć informacje o nienumerycznym charakterze.

2.1 Kod ASCII

Pojedyncze znaki drukowane przechowywane są najczęściej jako typ **char** (ang. *character*). Oczywiście pamiętamy, że tego typu używaliśmy do przechowywania bajtów, czyli 8 bitowych liczb 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 1–4.

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 przejście 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 zanotować następujące prawidłowości.

- Mamy następujący porządek leksykograficzny: `'A'` $<$ `'B'` $<\ldots<$ `'Z'` $<$ `'a'` $<$ `'b'` $<\ldots<$ `'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) bądź 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`.

Uwaga

Istnieją jeszcze 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.

Tablica 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

Tablica 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	?

Tablica 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	_

Tablica 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

2.2 Reprezentacja napisów

Wiemy już, w jaki sposób obsługiwać pojedyncze znaki. Najprostszym sposobem reprezentowania ciągów symboli drukowanych, czyli **napisów**, są tablice elementów typu **char zakończone** umownie bajtem 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 znaków + bajt 0
// *prawie* równoważnie:
char napis2[14] =
   { 'P', 'e', 'w', 'i', 'e', 'n', ' ',
     'n', 'a', 'p', 'i', 's', '.', '\0' };

// Znaki w zmiennej napis1 są tylko do odczytu!
napis1[1] = 'k'; // nie wiadomo co się stanie
napis2[1] = 'k'; // ok
```

2.3 Operacje na łańcuchach znaków

Jako że napisy są zwykłymi tablicami, 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 = "Jakiś napis."; // tablica znaków zakończona
   zerem

// Zatem:
cout << napis;

// Jest równoważne:
int i=0;
while (napis[i] != '\0') // dopóki nie koniec napisu
{
   cout << napis[i];
   ++i;
}
```

Długość napisu można sprawdzić w następujący sposób.

```
int dlug(char* napis)
{
   // Zwraca długość napisu (bez bajtu zerowego). Zgodnie
   // z umową, mimo że jest to tablica, potrafimy jednak
   // sprawdzić, gdzie się ona kończy
```

```
    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 więcej!

    // 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 pamięci
}
```

2.4 Biblioteka cstring

Biblioteka <cstring> definiuje wiele funkcji przetwarzających łańcuchy znaków [1]. Wybrane funkcje zestawia tab. 5.

[1] Zobacz angielskojęzyczną dokumentację dostępną pod adresem http://www.cplusplus.com/reference/clibrary/cstring/.

Tablica 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∗ *strncpy* (**char**∗ *cel*, **char**∗ *zrodlo*, **int** *n*);	Kopiuje co najwyżej *n* znaków z jednego napisu do drugiego.
char∗ *strcat* (**char**∗ *cel*, **char**∗ *zrodlo*);	Łączy napisy *cel* i *zrodlo*.
char∗ *strncat* (**char**∗ *cel*, **char**∗ *zrodlo*, **int** *n*);	Dołącza do *cel* co najwyżej *n* znaków z napisu *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*.
int *strncmp*(**char**∗ *nap1*, **char**∗ *nap2*, **int** *n*);	Porównuje co najwyżej *n* pierwszych znaków napisów.
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*.

3 Ćwiczenia

Zadanie 8.1. Zaimplementuj samodzielnie funkcje z biblioteki `<cstring>`: *strlen* (), *strcpy* (), *strncpy* (), *strcat* (), *strncat* (), *strcmp* (), *strstr* (), *strchr* (), *strrchr* ().

Zadanie 8.2. Napisz funkcję, która w danym łańcuchu znaków zamieni wszystkie małe litery alfabetu łacińskiego na wielkie.

Zadanie 8.3. Napisz funkcję, która usunie wszystkie znaki odstępów (spacje) z końca danego łańcucha znaków.

Zadanie 8.4. Napisz funkcję, która usunie wszystkie znaki odstępów (spacje) z początku danego łańcucha znaków.

Zadanie 8.5. Napisz funkcję, która usunie z danego napisu wszystkie znaki niebędące cyframi bądź kropką.

Zadanie 8.6. Napisz funkcję, która odwróci kolejność znaków w danym napisie.

Zadanie 8.7. Napisz funkcję, która jako parametr przyjmuje dwa łańcuchy znaków i zwraca nowy, dynamicznie alokowany napis będący ich połączeniem, np. dla "ala" i "ola" wynikiem powinno być "alaola".

Zadanie 8.8. Napisz funkcję, która oblicza, ile razy w danym napisie występuje dany znak.

Zadanie 8.9. Napisz funkcję, która oblicza, ile razy w danym napisie występuje dany inny łańcuch znaków, np. w "ababbababa" łańcuch "aba" występuje 3 razy.

Zadanie 8.10. Napisz funkcję, która dla danej liczby **int** zwróci dynamicznie alokowany łańcuch znaków, składający się z symboli 0 lub 1, przechowujący binarną reprezentację argumentu.

Zadanie 8.11. Napisz funkcję, która dla danego łańcucha znaków, składającego się z symboli 0 lub 1, reprezentującego pewną liczbę w postaci binarnej, zwróci jej wartość jako zmienną typu **int**.

Zadanie 8.12. Napisz funkcję, która dla danego łańcucha znaków, składającego się z symboli 0 lub 1, reprezentującego pewną liczbę w postaci binarnej, zwróci dynamicznie alokowany napis przechowujący jej szesnastkową reprezentację.

Zadanie 8.13. Palindrom[2] to ciąg liter, które są takie same niezależnie od tego, czy czytamy je od przodu czy od tyłu, np. *kobyłamamałybok*, *możejutrotadamasamadatortujeżom*, *ikarłapałraki*. Napisz funkcję sprawdzającą czy dany napis jest palindromem. Zwróć wartość typu **bool**.

[2] Zob. http://www.palindromy.pl.

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

IX. Macierze

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

1 Reprezentacja macierzy

Dana jest macierz określona nad pewnym zbiorem

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$$

gdzie n — liczba wierszy, m — liczba kolumn.

Macierze mogą być wygodnie reprezentowane w języku C++ na dwa sposoby:

– jako tablice tablic,

– jako tablice jednowymiarowe.

Preferujemy tutaj sposób pierwszy. Pomimo nieco zawiłego tworzenia i usuwania tego typu obiektów, zapewnia on bardzo wygodny dostęp do poszczególnych elementów.

```
int n = ...; // liczba wierszy
int m = ...; // liczba kolumn
typ** A; // tablica o elementach typu "typ*"

A = new typ*[n];
for (int i=0; i<n; ++i)
   A[i] = new typ[m];

// A to n-elementowa tablica tablic m-elementowych

// 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*n+3] to element w II wierszu i IV kolumnie....

delete [] A;
```

2 Przykładowe algorytmy z wykorzystaniem macierzy

Omówimy teraz następujące przykładowe algorytmy:

a) dodawanie macierzy,
b) mnożenie macierzy,
c) rozwiązywanie układów równań liniowych metodą eliminacji Gaussa.

Będziemy rozpatrywać macierze o wartościach rzeczywistych (reprezentowanych przez typ **double**).

2.1 Dodawanie macierzy

Dane są dwie macierze A, B typu $n \times m$. Wynikiem dodawania $A + B$ jest macierz:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1m}+b_{1m} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2m}+b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}+b_{n1} & a_{n2}+b_{n2} & \cdots & a_{nm}+b_{nm} \end{bmatrix}.$$

Oto kod funkcji służącej do dodawania macierzy.

```
1  void dodaj(double** A, double** B, int n, int m,
2             double** C)
3  {
4     /* A, B - macierze wejściowe typu n*m
5        C - macierz wynikowa typu n*m (pamięć już przydzielona)
6           */
6     assert(n > 0 && m > 0);
7
8     for (int i=0; i<n; ++i)
9        for (int j=0; j<m; ++j)
10          C[i][j] = A[i][j] + B[i][j];
11 }
```

Uwaga

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];
```

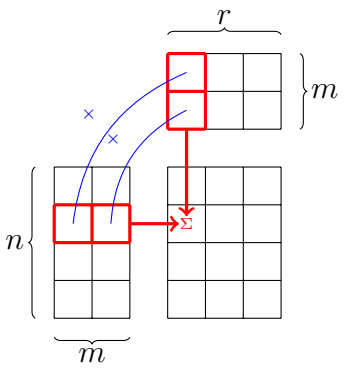
w przypadku macierzy dużych rozmiarów powoduje, że program może wykonywać się wolniej. Na komputerze skromnego autora niniejszego skryptu dla macierzy 10000×10000 czas wykonywania rośnie z 2.7 aż do 24.1 sekundy. Drugie rozwiązanie nie wykorzystuje bowiem w pełni szybkiej **pamięci podręcznej** komputera.

2.2 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

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj},$$

gdzie $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant r$ (por. rys. 1).



Rysunek 1: Ilustracja algorytmu mnożenia macierzy.

A oto kod funkcji służącej do wyznaczenia iloczynu dwóch macierzy.

```
void mnoz(double** A, double** B, int n, int m, int r,
          double** C)
{
   /* A - macierz wejściowa typu n*m
      B - macierz wejściowa typu m*r
      C - macierz wynikowa typu n*r (pamięć już przydzielona)
        */
   assert(n > 0 && m > 0 && r > 0);

   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];
      }
}
```

2.3 Rozwiązywanie układów równań liniowych

Dany jest oznaczony układ równań postaci

$$A\mathbf{x} = \mathbf{b},$$

gdzie A jest macierzą 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.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Uproszczona metoda **eliminacji Gaussa** polega na sprowadzeniu macierzy rozszerzonej $[A|\mathbf{b}]$ do postaci schodkowej $[A'|\mathbf{b}']$:

$$\begin{bmatrix} a'_{1,1} & a'_{1,2} & a'_{1,3} & \cdots & a'_{1,n-1} & a'_{1,n} & b'_1 \\ 0 & a'_{2,2} & a'_{2,3} & \cdots & a'_{2,n-1} & a'_{2,n} & b'_2 \\ 0 & 0 & a'_{3,3} & \cdots & a'_{3,n-1} & a'_{3,n} & b'_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a'_{n-1,n-1} & a'_{n-1,n} & b'_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & a'_{n,n} & b'_n \end{bmatrix}$$

Dokonuje się tego za pomocą następujących tzw. **operacji elementarnych**:

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 **eliminacji wstecznej**:

$$x_i = \frac{1}{a'_{i,i}} \left(b'_i - \sum_{k=i+1}^{n} a'_{i,k} x_j \right).$$

dla kolejnych $i = n, n-1, \ldots, 1$.

Rozpatrzmy przykładową implementację tej metody w języku C++.

```
void schodkowa(double** A, double* b, int n);
void eliminwst(double** A, double* b, int n, double* x);

void gauss(double** A, double* b, int n, double* x)
{
    assert(n > 0);
    schodkowa(A, b, n);
    eliminwst(A, b, n, x);
}
```

Funkcja sprowadzająca macierz do postaci schodkowej:

```
void schodkowa(double** A, double* b, int n)
{
    for (int i=0; i<n; ++i)
    { /* dla każdego wiersza */

        assert(A[i][i]!=0.0);//jeśli nie=>macierz osobliwa
        // assert(fabs(A[i][i]) > 1e-10); // lepiej

        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:

```
void eliminwst(double** A, double* b, int n, double* x)
{
    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];
    }
}
```

3 Ćwiczenia

Zadanie 9.1. 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.

Zadanie 9.2. 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 9.3. 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 9.4. Dana jest macierz A typu 2×2 o wartościach rzeczywistych. Napisz funkcję, która zwróci wyznacznik danej macierzy.

Zadanie 9.5. Dana jest macierz A typu 3×3 o wartościach rzeczywistych. Napisz funkcję, która zwróci wyznacznik danej macierzy.

$\star$ **Zadanie 9.6.** 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 9.7. Napisz funkcję, która rozwiązuje układ 2 równań liniowych korzystając z metody Cramera. Poprawnie identyfikuj przypadki, w których dany układ nie jest oznaczony.

Zadanie 9.8. Napisz funkcję, która rozwiązuje układ 3 równań liniowych korzystając z metody Cramera. Poprawnie identyfikuj przypadki, w których dany układ nie jest oznaczony.

Zadanie 9.9. Dana jest macierz A o wartościach całkowitych. Napisz funkcję, która zwróci jej transpozycję.

Zadanie 9.10. 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 9.11. Dana jest kwadratowa macierz A o wartościach całkowitych. Napisz funkcję, która sprawdzi, czy macierz jest symetryczna. Zwróć wynik typu **bool**.

Zadanie 9.12. Dana jest macierz kwadratowa A o wartościach rzeczywistych typu $n \times n$. Napisz funkcję, która zwróci jej ślad, określony jako

$$\operatorname{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn} = \sum_{i=1}^{n} a_{ii}.$$

Zadanie 9.13. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej diagonalę w postaci tablicy jednowymiarowej.

Zadanie 9.14. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej macierz diagonalną, czyli macierz z wyzerowanymi wszystkimi elementami poza przekątną.

Zadanie 9.15. 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 9.16. 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**.

4 Wskazówki do ćwiczeń

Wskazówka do zadania 9.6. 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.

PROGRAM ROZWOJOWY
POLITECHNIKI WARSZAWSKIEJ

ALGORYTMY I PODSTAWY PROGRAMOWANIA

Marek Gągolewski

X. Podstawowe abstrakcyjne struktury danych

KAPITAŁ LUDZKI
CZŁOWIEK – NAJLEPSZA INWESTYCJA!

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

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 skalarnego** różnych innych zbiorów (typów). Są to tzw. **struktury**. Oto składnia ich definicji:

```
struct NazwaNowegoTypu
{
    typ1 nazwaPola1;
    typ2 nazwaPola2;
    //....
    typN nazwaPolaN;
}; // konieczny średnik!
```

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;
4      double poler;
5  };
6
7  int main()
8  {
9      NR x;          // tzn. x ∈ ℕ × ℝ
10     x.polen = 1;
11     x.poler = 3.14;
12
13     cout << x; // BŁĄD! operacja niezdefiniowana
14     cout << x.polen; // OK
15
16     return 0;
17 }
```

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 wartość - kopiowana
    cout << a.polen << " " << a.poler;
}

void f2(NR& a)
{
    // a przekazana przez referencję - 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 2 kolejne funkcje.

```
void f3(const NR& a)
{
    // a przekazana przez referencję - niekopiowana
    // + zabezpieczona przed zmianą
    cout << a.polen << " " << a.poler;
}

void f4(NR* a)
{
    // a przekazana przez wskaźnik - niekopiowana
    cout << (*a).polen << " " << (*a).poler;
    // równoważny 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.

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. Np. 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**.

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:

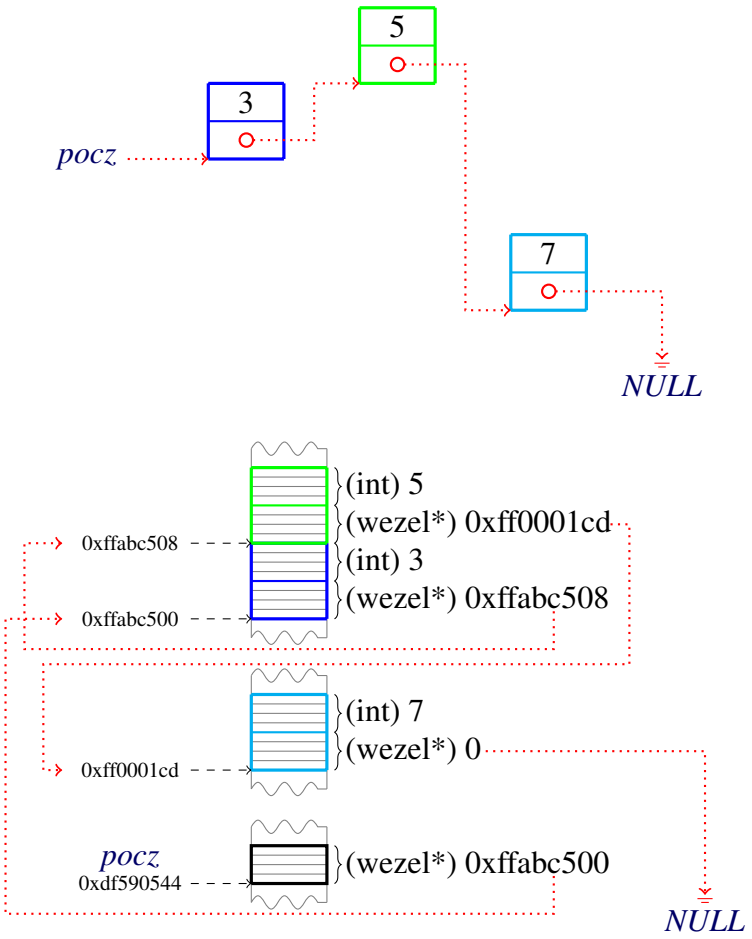
```
struct wezel
{
    int elem; // element(y), który przechowujemy w węźle
    wezel* nast; // wskaźnik na następny element
};
```

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. głowa listy
```

Schemat przykładowej listy jednokierunkowej przechowującej elementy 3, 5 oraz 7 przedstawia rys. 1. Zwróćmy uwagę na to, w jaki sposób węzły mogą być rozlokowane w pamięci.



Rysunek 1: Przykładowa lista jednokierunkowa przechowująca elementy 3,5,7. Schemat graficzny i organizacja pamięci.

Przyjrzymy się implementacji następujących operacji:

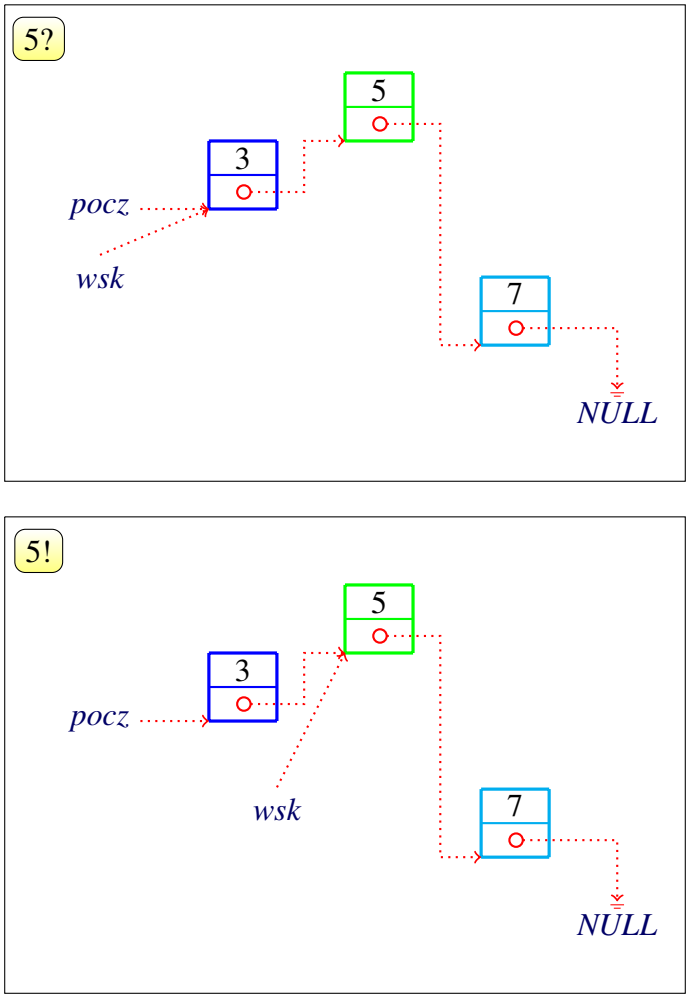
- sprawdzenie, czy dany element występuje na liście,
- wstawienie elementu na początek listy,
- wstawienie elementu na koniec listy,
- usunięcie elementu z początku listy,
- usunięcie elementu z końca listy.

Uwaga

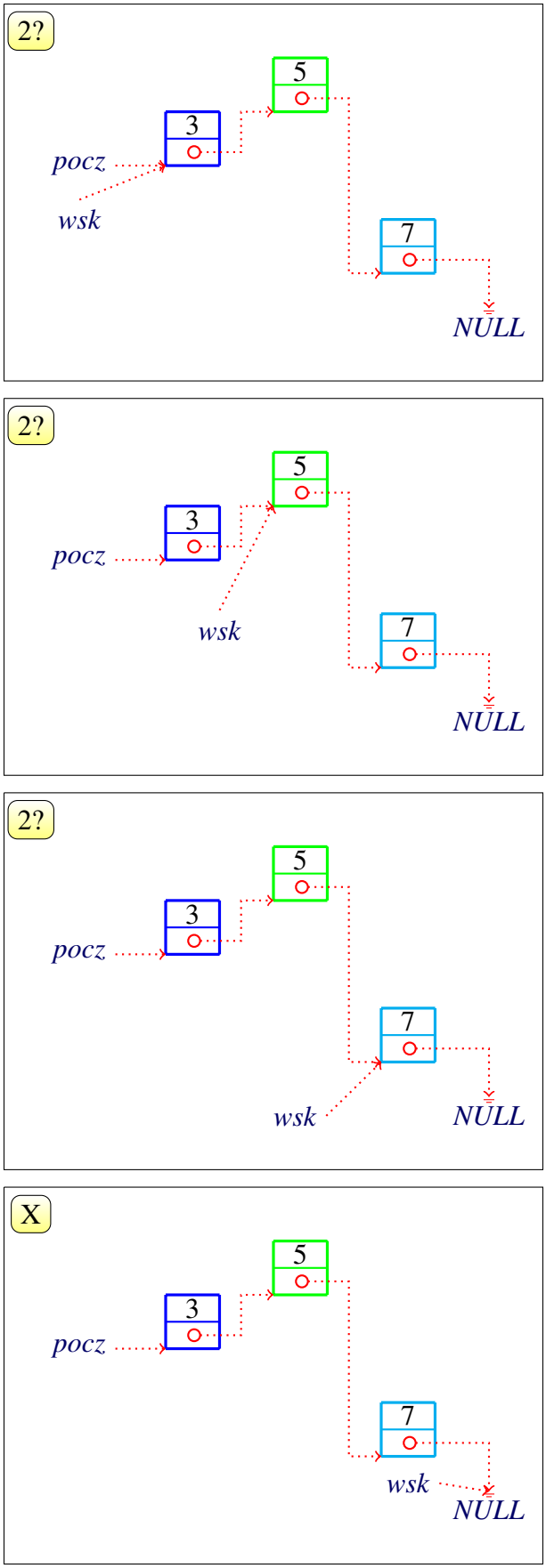
Należy zwrócić szczególną uwagę na przypadek, gdy lista początkowo jest pusta (gdy *pocz* == *NULL*)!

2.1.1 Wyszukiwanie elementu

Na rys. 2 i 3 znajdziemy ilustrację, krok po kroku, w jaki sposób przebiega wyszukiwanie elementów 5 oraz 2 na liście zawierającej (3,5,7).



Rysunek 2: Wyszukiwanie elementu 5 w liście jednokierunkowej.



Rysunek 3: Wyszukiwanie elementu 2 w liście jednokierunkowej.

W pierwszym przypadku element zostaje odnaleziony już w drugim kroku. W kolejnym stwierdzamy, że nie ma go wcale na danej liście.

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*.

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; // przejdź do następnego węzła
    }

    return false; // doszliśmy do końca listy
}
```

Wywołanie:

```
... szukaj(pocz, x);
```

Podany wyżej algorytm można również zapisać równoważnie w formie rekurencyjnej.

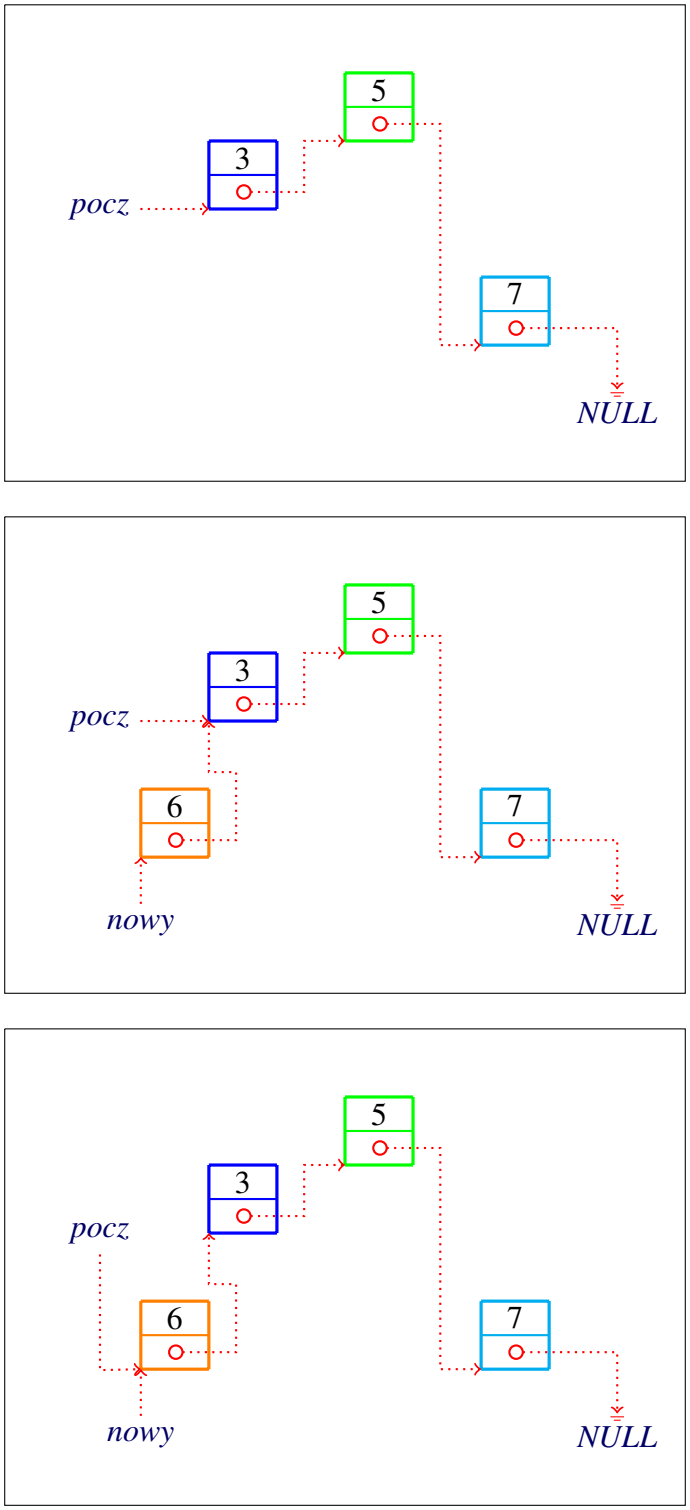
```
bool szukaj2(wezel* wsk, int x)
{
    if (wsk == NULL)
        return false; // doszliśmy do końca listy
    else if (wsk->elem == x)
        return true; // znaleziony
    else
        return szukaj2(wsk->nast, x); // szukaj dalej
}
```

Wywołanie:

```
... szukaj2(pocz, x);
```

2.1.2 Wstawianie elementu

Najpierw zajmijmy się wstawianiem elementu na początek listy. Rys. 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 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.

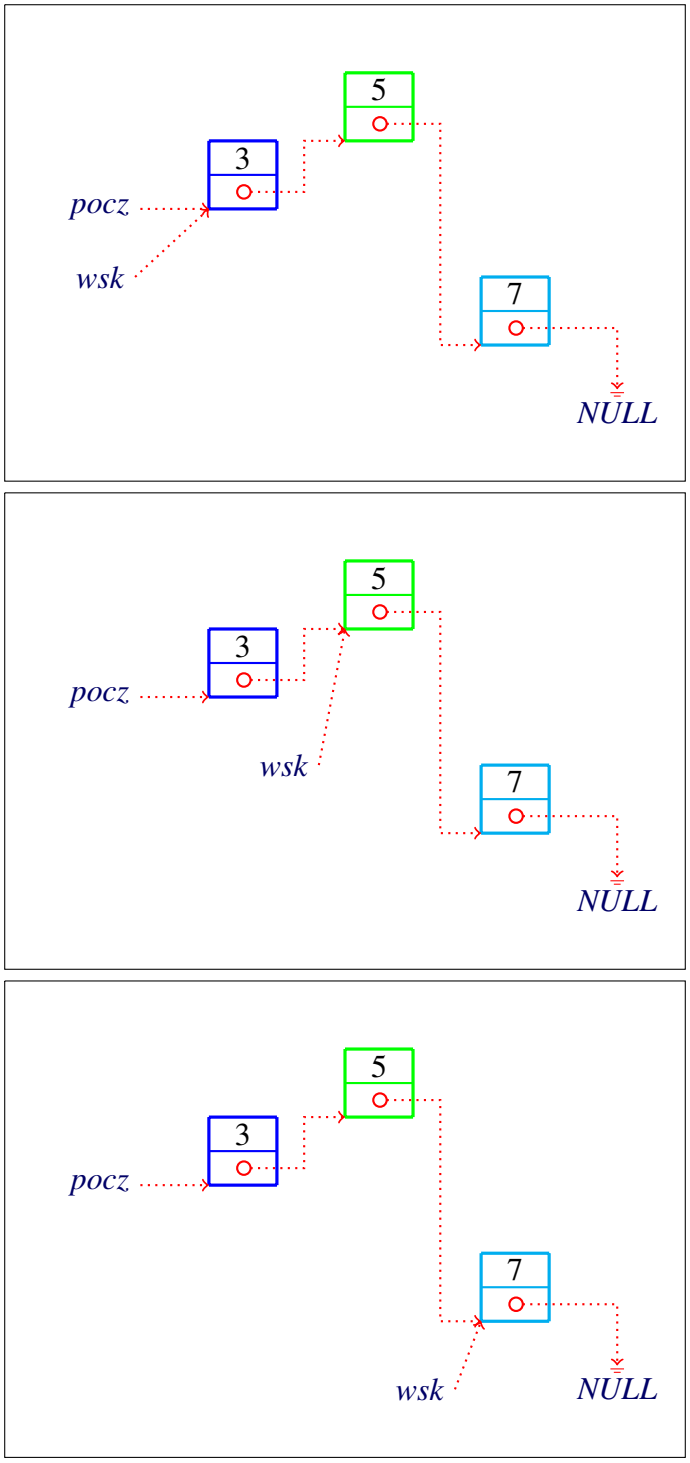
```
void wstawpocz(wezel*& pocz, int x)
{
    wezel* nowy = new wezel;
    nowy->elem = x;

    nowy->nast = pocz; /* wskazuj na to, na co wskazuje pocz,
       może być NULL */

    pocz = nowy;            /* teraz lista zaczyna się
                               od nowego węzła */
}
```

Rozważmy teraz, w jaki sposób wstawić element na koniec listy. Rys. 5 i 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.

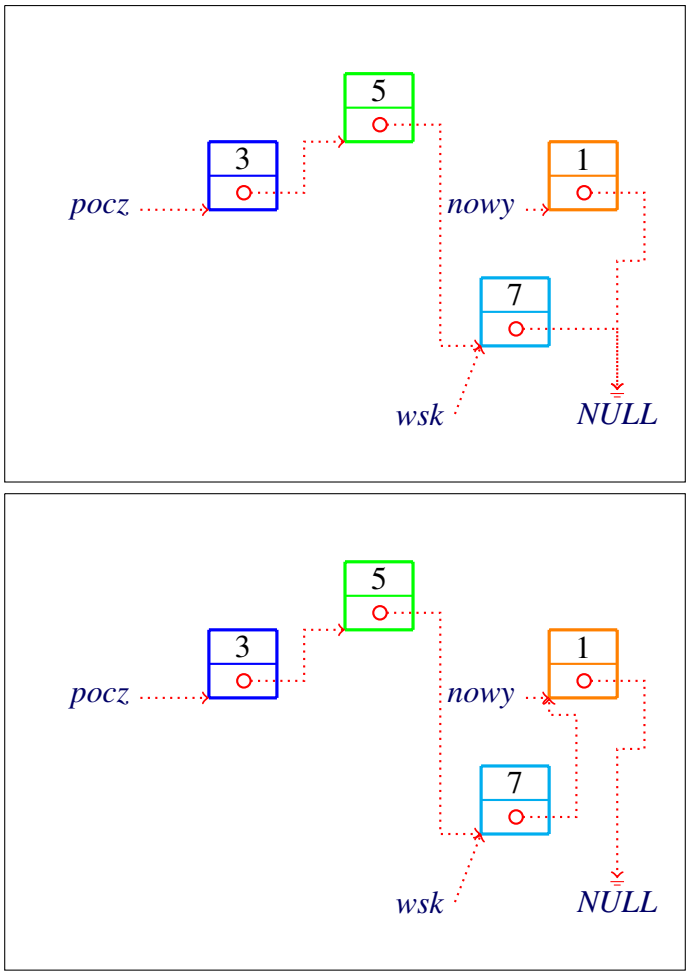
Poniższa funkcja implementuje interesującą nas operację. Zauważmy, że pierwszym parametrem jest referencja do wskaźnika. Jeżeli lista jest pusta, przekazana głowa może zostać zmieniona.

```
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; // przejdź na ostatni element

        wsk->nast = nowy; // nowy ostatni element
    }
}
```

A oto równoważna wersja rekurencyjna.



Rysunek 5: Wstawianie elementu 1 na koniec listy jednokierunkowej cz. I.



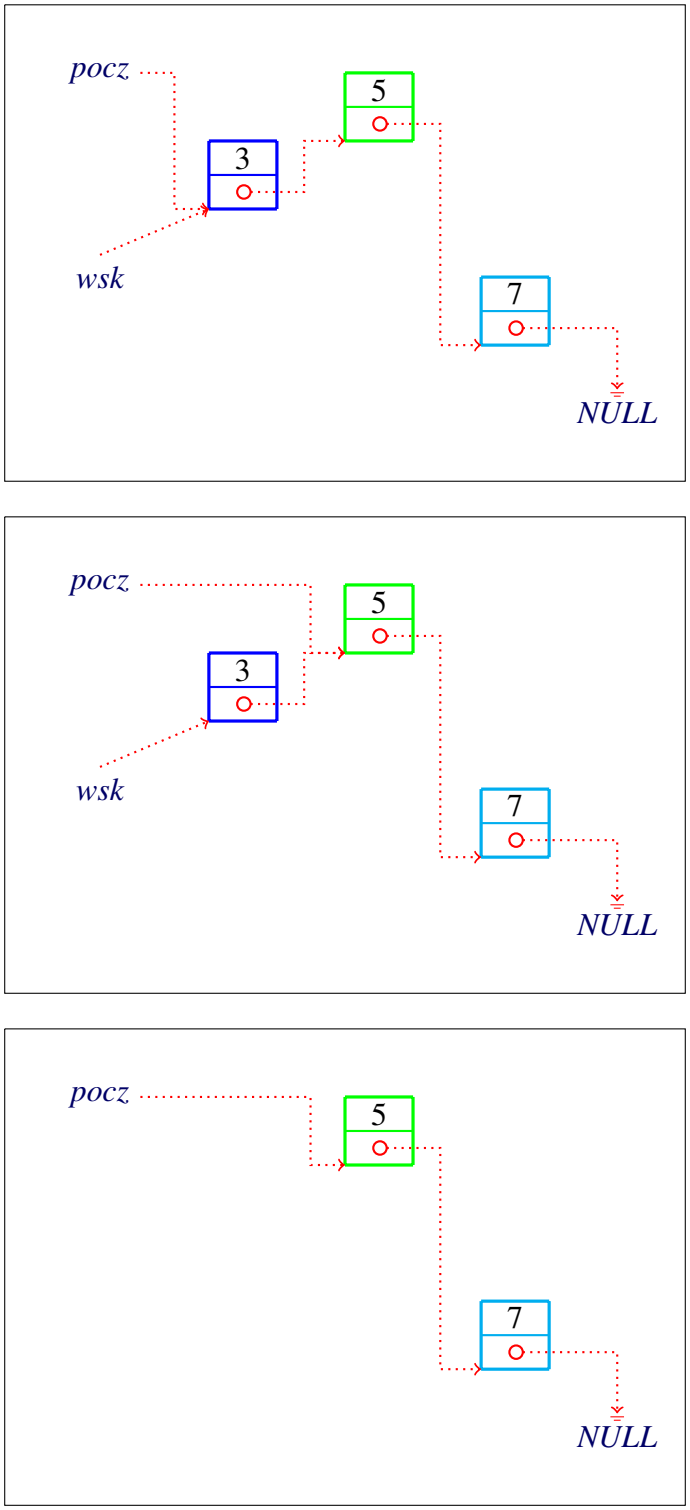
Rysunek 6: Wstawianie elementu 1 na koniec listy jednokierunkowej cz. II.

```
1  void wstawkon2(wezel*& wsk, int x)
2  {
3      if (wsk != NULL)
4          wstawkon2(wsk->nast, x); // idź dalej
5      else
6      {
7          wsk = new wezel;
8          wsk->elem = x;
9          wsk->nast = NULL;
10     }
11 }
```

2.1.3 Usuwanie elementu

Podobnie jak wyżej, i tutaj rozpatrzymy dwa przypadki, w których usuwany element znajduje się na początku bądź na końcu listy.

Rys. 7 przedstawia możliwy sposób usuwania elementów z początku listy jednokierunko-



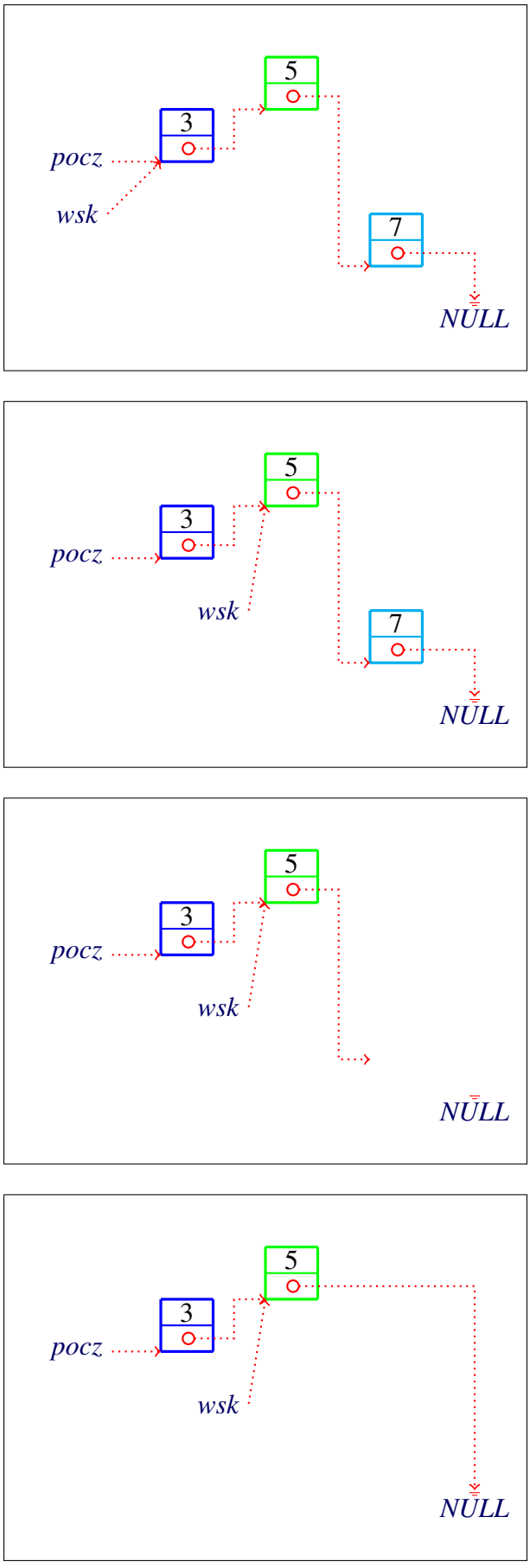
Rysunek 7: Usunięcie elementu z początku listy jednokierunkowej.

wej. 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 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;
        // przejdź na przedostatni element
        while (wsk->nast->nast != NULL)
            wsk = wsk->nast;

        delete wsk->nast;
        wsk->nast = NULL;
    }
}
```



Rysunek 8: Usunięcie elementu z końca listy jednokierunkowej.

Omawianą procedurę można zapisać również w postaci rekurencyjnej.

```
void usunkon2(wezel*& wsk)
{
    if (wsk == NULL) return; // lista była pusta
    else if (wsk->nast == NULL) // jesteśmy na końcu
    {
        delete wsk;
        wsk = NULL;
    }
    else // idziemy dalej
        usunkon2(wsk->nast);
}
```

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 bądź zmniejszać swój rozmiar. Nie wymaga ona bowiem kopiowania wszystkich elementów za każdym razem.

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.

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.

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.

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), który przechowujemy w węźle
    wezel* nast; // wskaźnik na następny element
    wezel* poprz; // wskaźnik na poprzedni element
};
```

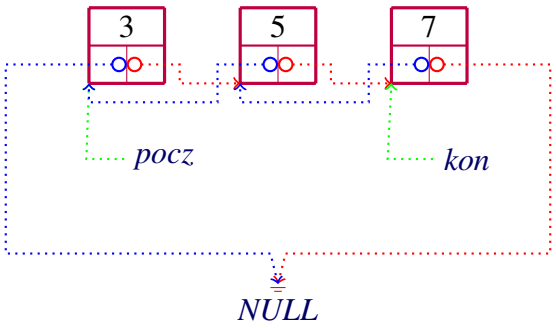
Należy rzecz jasna zapamiętać wskaźnik na pierwszy (*pocz*) i wskaźnik na ostatni (*kon*) element listy dwukierunkowej.

Przykładowa lista dwukierunkowa (3,5,7) przedstawiona jest na rys. 9.

Jako że jej implementacja jest dość podobna do omawianych już konstrukcji, pozostawiamy ją jako ćwiczenie.

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



Rysunek 9: Przykładowa lista dwukierunkowa.

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), który przechowujemy w węźle
    wezel* lewy; // wskaźnik na lewe poddrzewo (lewego potomka)
    wezel* prawy; // wskaźnik 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. Ponadto, jeśli istnieje ścieżka pomiędzy węzłami v i w, 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.

2.6.1 Wyszukiwanie elementu

Rys. 10 ilustruje krok po kroku wyszukiwanie elementu 2 w drzewie składającym się z elementów $\{1, 2, 3, 5, 6, 8\}$.

Uwaga

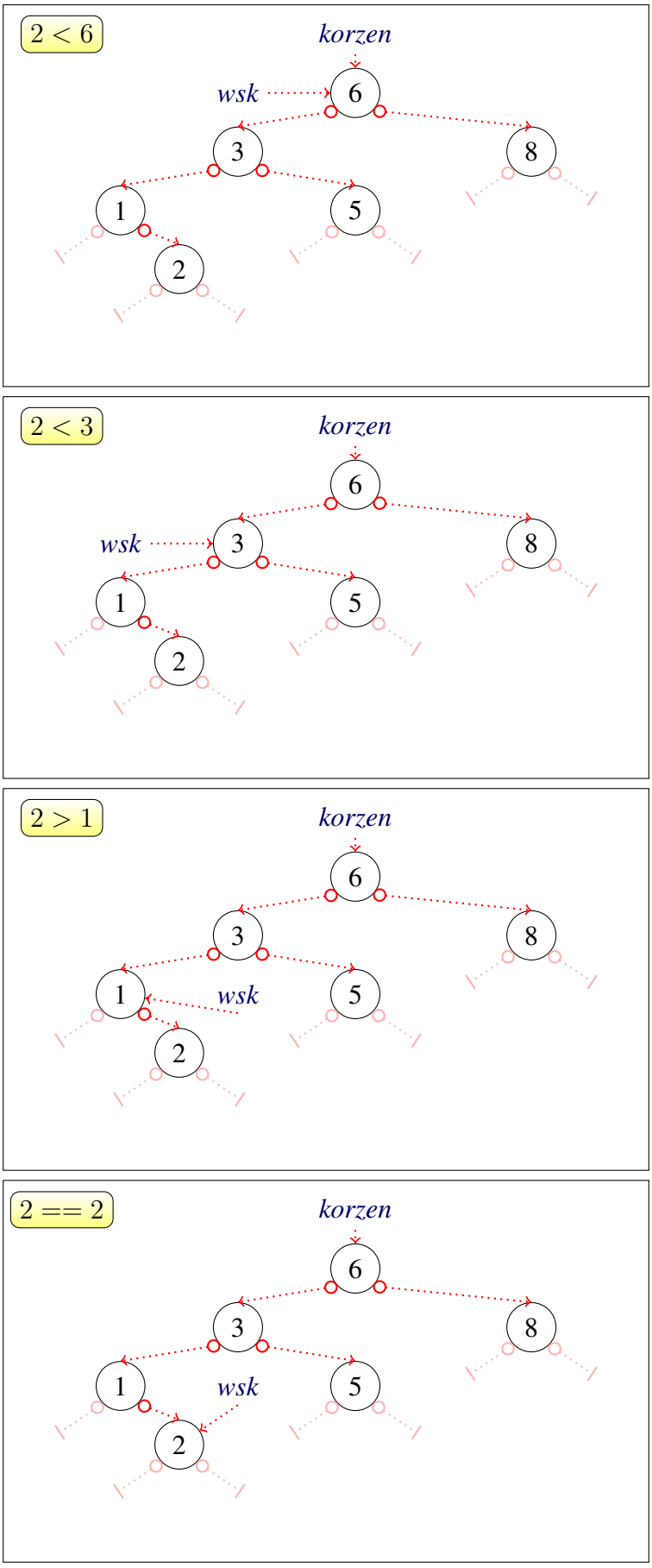
Zwróćmy uwagę, że istnieje wiele postaci drzew binarnych, które mogłyby służyć do przechowywania powyższego zbioru wartości. W szczególności drzewo takie mogłoby się zredukować do listy, np. jeśli żaden z węzłów nie miałby lewego bądź prawego poddrzewa. W tym przypadku wyszukiwanie może wymagać przejrzenia wszystkich elementów.

Oto wersja iteracyjna funkcji służącej do wyszukiwania elementów o zadanej wartości.

```
bool szukaj(wezel* korzen, int x)
{
    wezel* wsk = korzen;
    while (wsk != NULL)
    {
        if (x == wsk->elem) return true;
        else if (x < wsk->elem) wsk = wsk->lewy;
        else                    wsk = wsk->prawy;
    }

    return false; // nie znaleziono
}
```

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);
}
```



Rysunek 10: Wyszukiwanie elementu 2 w przykładowym drzewie binarnym.

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
}
```

2.6.3 Wstawianie elementu

Wstawianie elementu 4 do przykładowego drzewa binarnego ilustruje rys. 11.

Najprościej wstawić element jako liść. Zatem najpierw należy znaleźć węzeł, do którego nowy element będziemy doczepiać jako potomka. Oto wersja rekurencyjna stosownej funkcji:

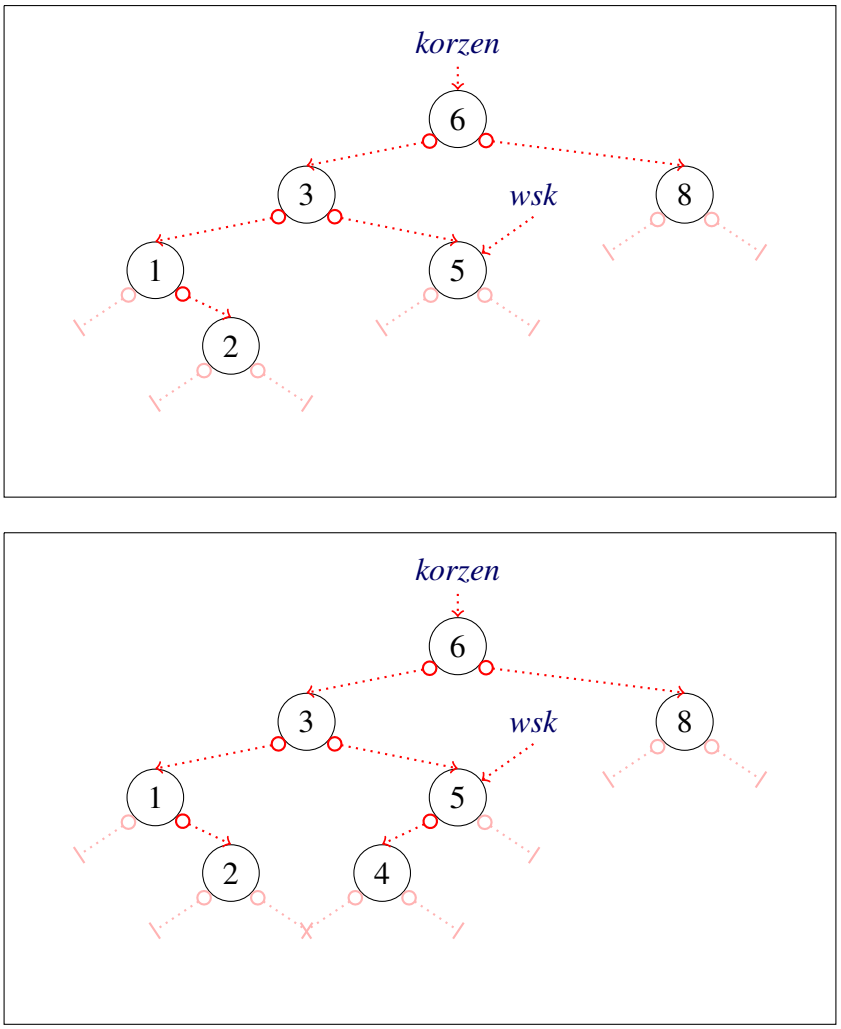
```
void wstaw(wezel*& wsk, int x)
{
    if (wsk == NULL)
    { // tutaj będzie nowy liść
        wsk = new wezel;
        wsk->elem = x;
        wsk->lewy = NULL;
        wsk->prawy = NULL;
    }
    else if (x == wsk->elem) return; // nic nie rób
    else if (x < wsk->elem)  return wstaw(wsk->lewy,  x);
    else                     return wstaw(wsk->prawy, x);
}
```

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,



Rysunek 11: Wstawianie elementu do przykładowego drzewa binarnego.

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; // zastąp prawym, nawet jeśli 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 potomków
19
20              // zastąpimy "sprytnie" usuwany węzeł jego prawym
                    potomkiem
21
22              // znajdź największy element mniejszy niż usuwany->
                    elem
23              wezel* wsk2 = wsk->lewy;
24              while (wsk2->prawy != NULL)
25                  wsk2 = wsk2->prawy;
26
27              // elementy większe od wsk2->elem, ale mniejsze niż
28              // wsk->prawy->elem będą "adoptowane" przez wsk2,
29              // które będzie się znajdować gdzieś w lewym podrzewie
                     wsk
30              wsk2->prawy = wsk->prawy->lewy;
31
32              // powiększone lewe wsk podrzewo przenosimy jako lewe
                    poddrzewo
33              // prawego potomka wsk, żeby własności drzewa
34              // binarnego zostały zachowane
35              wsk->prawy->lewy = wsk->lewy;
36
37
38              // zapamiętaj 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  }
```

Podziękowania. Serdecznie dziękuję Jowicie Hąci, Kasi Fokow i Michałowi Dębskiemu za pomoc w udoskonaleniu niniejszych materiałów.

3 Ćwiczenia

Zadanie 10.1. Napisz samodzielnie pełny program w języku C++, 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 10.2. Rozwiąż powyższe zadanie, implementując listę jednokierunkową, która dodatkowo przechowuje wskaźnik na ostatni element.

Zadanie 10.3. Rozwiąż powyższe zadanie, implementując listę dwukierunkową.

Zadanie 10.4. Dla danych dwóch list jednokierunkowych napisz funkcję, która je połączy, np. dla (1,2,5,4) oraz (3,2,5) sprawi, że I lista będzie postaci (1,2,5,4,3,2,5), a II zostanie skasowana.

Zadanie 10.5. Napisz samodzielnie pełny program w języku C++, który implementuje i testuje (w funkcji *main*()) stos (LIFO) zawierający dane typu **int**.

Zadanie 10.6. Napisz samodzielnie pełny program w języku C++, który implementuje i testuje (w funkcji *main*()) zwykłą kolejkę (FIFO) zawierającą dane typu **char**$*$ (napisy).

$\star$ **Zadanie 10.7.** Zaimplementuj operacje *enqueue*() i *dequeue*() zwykłej kolejki (FIFO) typu **int** korzystając tylko z dwóch gotowych stosów.

Zadanie 10.8. Napisz samodzielnie pełny program w języku C++, który implementuje i testuje (w funkcji *main*()) kolejkę priorytetową zawierającą dane typu **int**.

Zadanie 10.9. Napisz funkcję, która wykorzysta kolejkę priorytetową do posortowania danej tablicy o elementach typu **int**.

Zadanie 10.10. Napisz samodzielnie pełny program w języku C++, 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.

4 Wskazówki do ćwiczeń

Wskazówka do zadania 10.7. 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*().