Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

1. Etapy tworzenia oprogramowania. Algorytm

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

1.1. Wprowadzenie

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

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

Na pewno poważnie myślisz o karierze matematyka (być może w tzw. sektorze komercyjnym lub nawet naukowo-badawczym). Pamiętaj o tym, że opanowanie umiejętności programowania komputerów nie będzie wcale atutem w Twoim życiu zawodowym. Będzie warunkiem koniecznym powodzenia! Komputery bowiem towarzyszą nam na (prawie) każdym kroku, są nieodłącznym elementem współczesnego świata.

Zapamiętaj

Przedmiot AiPP łączy teorię (algorytmy) z praktyką (programowanie). Pamiętaj, że nie da się nabrać biegłości w żadnej dziedzinie życia bez intensywnych, systematycznych ćwiczeń.

Nie szczędź więc swego czasu na własne próby zmierzenia się z przedstawionymi problemami, eksperymentuj. Przecież to praktyka mistrza czyni.

W tym semestrze poznasz najbardziej podstawowe zagadnienia informatyki. Skrypt, którego celem jest systematyzacja Twej wiedzy, podzielony jest na 8 części:

1. Etapy tworzenia oprogramowania. Algorytm
2. Podstawy organizacji i działania komputerów. Reprezentacja liczb całkowitych i zmiennopozycyjnych
3. Deklaracja zmiennych w języku C++. Operatory arytmetyczne, logiczne i relacyjne.
4. Instrukcja warunkowa i pętle
5. Funkcje. Przekazywanie parametrów przez wartość i przez referencję. Rekurencja
6. Wskaźniki. Dynamiczna alokacja pamięci. Tablice jednowymiarowe. Sortowanie. Łańcuchy znaków
7. Macierze
8. Struktury w języku C++. Abstrakcyjne typy danych

W niniejszej części zastanowimy się, w jaki sposób — w wyidealizowanym przypadku — powstają programy komputerowe. Co ważne, niebawem i Tym będziesz postępował(a) mniej więcej w przedstawiony niżej sposób rozwiązując różne problemy z użyciem komputera.

1.2. Etapy tworzenia oprogramowania

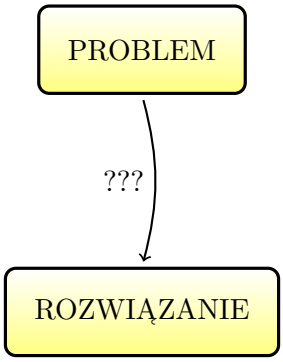
1.2.1. Sformułowanie i analiza problemu

1.2.1.1. Sformułowanie problemu

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

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

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



Rys. 1.1. Punkt wyjścia naszych rozważań

1.2.1.2. Analiza problemu

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

Jednakże niektóre zagadnienia mogą przytłaczać swoją złożonością. Na przykład „zaliczenie przedmiotu AiPP” wymaga m.in. określenia:

- jaki stan końcowy jest pożądany (ocena bardzo dobra?),
- jakie czynności są kluczowe do osiągnięcia rozwiązania (udział w ćwiczeniach i laboratoriach, słuchanie wykładu, zadawanie pytań, dyskusje na konsultacjach),
- jakie czynniki mogą wpływać na powodzenie na poszczególnych etapach nauki (czytanie książek, wspólna nauka?), a jakie je wręcz uniemożliwiać (codzienne imprezy? brak prądu w akademiku? popsuty komputer?).

Komputery. Istotną cechą wielu zagadnień jest to, że nadają się do rozwiązania za pomocą *komputera*. Jak pokazuje rozwój współczesnej informatyki, tego typu problemów jest wcale niemało. W innych przypadkach często mamy do czynienia z sytuacją, w której komputery mogą pomóc uzyskać *rozwiązanie częściowe* lub zgrubne przybliżenie rozwiązania, które przecież 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 samochodem po zatłoczonych drogach (osiągnięcie ekipy *Google*), wirtualną obsługą petenta w Zakładzie Ubezpieczeń Społecznych itp. Wyobraźnię o ich potędze podsycają opowiadania *science-fiction*, których autorzy przepowiadają, że te maszyny pewnego dnia będą potrafiły zrobić prawie wszystko, o czym tylko jesteśmy w stanie pomyśleć.

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

1	Komputer ma *ograniczoną moc obliczeniową*. Każda instrukcja wykonuje się przez pewien (niezerowy!) czas. Im bardziej złożone zadanie, tym jego rozwiązywanie trwa dłużej. Choć stan rzeczy poprawia się wraz z rozwojem technologii, zawsze będziemy ograniczeni zasadami fizyki.	Samochody mają np. ograniczoną prędkość, ograniczone przyspieszenie (także zasadami fizyki).
2	Komputer „rozumie" określony *język* (języki), do którego *syntaktyki* (składni) trzeba się dostosować, którego konstrukcję trzeba poznać, by móc się z nim „dogadać". Języki są zdefiniowane formalnie za pomocą ściśle określonej gramatyki. Nie toleruje on najczęściej żadnych odstępstw lub toleruje tylko nieliczne.	Aby zwiększyć liczbę obrotów silnika, należy wykonać jedną ściśle określoną czynność — wcisnąć mocniej pedał gazu. Nic nie da uśmiechnięcie się bądź próba sympatycznej konwersacji z deską rozdzielczą na temat zalet jazdy z inną prędkością.
3	Komputer ograniczony jest ponadto przez tzw. *czynnik ludzki* — mówi się, że jest tak mądry, jak jego programista. Potrafi zrobić tylko to (i aż tyle), co sami mu dokładnie powiemy, co ma zrobić, krok po kroku, instrukcja po instrukcji. Nie domyśli się, co tak naprawdę nam chodzi po głowie. Każdy wydawany rozkaz ma bowiem określoną *semantykę* (znaczenie). Komputer jedynie potrafi go posłusznie wykonać.	Samochód zawiezie nas, gdzie chcemy, jeśli będziemy odpowiednio posługiwać się kierownicą i innymi przyrządami. Nie będzie protestował, gdy skręcimy nie na tym skrzyżowaniu, co trzeba. Albo gdy wjedziemy na drzewo. Bądź dwa.

Zapamiętaj

Celem przewodnim naszych rozważań w tym semestrze jest więc takie poznanie natury i *języka* komputerów, by mogły zrobić *dokładnie* to, co *my* chcemy. Oczywiście skupimy się tutaj tylko (i aż) na dość prostych problemach.

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

Informacja

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

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

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

Dalej, przez jednostkę *informacji* rozumiemy dowolny *ciąg liczb lub symboli* (być może jednoelementowy). Oto kilka przykładów:

- (6) (liczba naturalna),
- (*PRAWDA*) (wartość logiczna),
- („STUDENT MiNI") (napis, czyli ciąg znaków drukowanych),
- (3,14159) (liczba rzeczywista),
- (4,-3,-6, 34) (ciąg liczb całkowitych),
- (011100) (liczba w systemie binarnym),
- ("WARSZAWA", 52°13'56"N, 21°00'30"E) (współrzędne GPS pewnego miejsca na mapie).

Ciekawostka

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 wszystkich cech opisywanego obiektu lecz tylko te, które są potrzebne w danym zagadnieniu. Ciąg („Maciek Pryk", „Matematyka II rok", 4,92, 21142) może być wystarczającą informacją dla pani z dziekanatu, by przyznać pewnemu studentowi stypendium za wyniki w nauce. W tym przypadku kolor oczu Maciusia czy imię dziewczyny, do której wzdycha on w nocy, to zbędne szczegóły (chyba, że ta ostatnia jest córką Pani z dziekanatu, ale...).

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

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

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

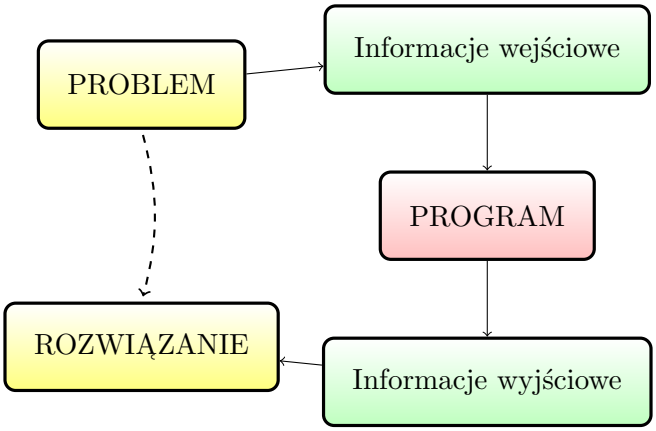
Nietrudno odgadnąć ich znaczenie. Co ważne: właśnie tak działa komputer!

Zapamiętaj

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

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

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



Rys. 1.2. Rozwiązanie problemu za pomocą programu komputerowego

1.2.2. Projektowanie

Po etapie analizy problemu następuje etap *projektowania algorytmów*.

Zapamiętaj

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 pierniczki,[1] przedstawiony w tab. 1.1.

Tab. 1.1. Przepis na pierniczki

Dane wejściowe: 2 szklanki mąki; 2 łyżki miodu; 3/4 szklanki cukru; 1,5 łyżeczki sody oczyszczonej; 1/2 torebki przyprawy piernikowej; 1 łyżka masła; 1 jajko (+ dodatkowo 1 jajko do posmarowania); około 1/3 szklanki lekko ciepłego mleka.

Algorytm:

1. Mąkę przesiać na stolnicę, wlać rozpuszczony gorący miód i wymieszać (najlepiej nożem). Ciągle siekając, dodawać kolejno cukier, sodę, przyprawy, a gdy masa lekko przestygnie – masło i jedno jajko.
2. Dolewając stopniowo (po 1 łyżce) mleka zagniatać ręką ciasto aż będzie średnio twarde i gęste (nie musimy wykorzystać całego mleka, bo masa może być za rzadka). Dokładnie wyrabiać ręką, aż będzie gładkie, przez około 10 minut.
3. Na posypanej mąką stolnicy rozwałkować ciasto na placek o grubości maksymalnie 1 cm. Foremkami wykrajać z ciasta pierniczki, smarować rozmąconym jajkiem i układać na blasze wyłożonej papierem do pieczenia w odstępach około 2–3 cm od siebie (pierniczki troszkę podrosną).
4. Piec w piekarniku nagrzanym do 180 stopni przez około 15 minut. Przechowywać w szczelnie zamkniętym pojemniku, do 4 tygodni lub jeszcze dłużej [*oj tam! — przyp. MG*]. Pierniczki im są starsze tym lepsze. Z czasem też stają się bardziej miękkie.
5. Dekorować przed podaniem, najlepiej jak już będą miękkie. Do dekoracji można użyć samego lukru lub lukru wymieszanego z barwnikiem spożywczym. Zamiast barwnika spożywczego można użyć soku z granatu lub z buraka. Pierniczki można dekorować roztopioną czekoladą i maczać w posypce cukrowej lub w wiórkach kokosowych.

Dane wyjściowe: Pyszne pierniczki.

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!

[1] Przepis ten pochodzi z serwisu Kwestia Smaku:
www.kwestiasmaku.com/desery/ciasteczka/pierniczki/przepis.html

Informacja

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

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

Wygląda na to, że nasz przepis na pierniczki posiada wszystkie wyżej wymienione cechy. Wykonując powyższe czynności ciasto to uda nam się kiedyś zjeść (skończoność). Wszystkie czynności są zrozumiałe nawet dla mało wprawionej gospodyni (aczkolwiek w *dobre zdefiniowanie* może tutaj trochę matematyk powątpiewać — co to znaczy *około* 1/3 szklanki ciepłego mleka?). Sam proces przygotowania jest efektywny (nie każe kucharce np. umalować się w trakcie mieszania składników bądź pojechać w międzyczasie na zagraniczną wycieczkę). Dane wejściowe i wyjściowe są dobrze określone.

Zapamiętaj

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”, „chili con carne z plackami tortilli” oraz „pierniczki” i skupić się najpierw na projektowaniu podzadań.

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

Dla przykładu, w przytoczonym wyżej przepisie, nie jest dokładnie wytłumaczone — krok po kroku — co oznacza „aż będzie średnio twarde i gęste”. Jednakże czynność tę da się pod pewnymi warunkami doprecyzować.

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

1.2.2.1. Przykład: Problem młodego Gaussa

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

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 wyznaczył wartość rozwiązania bardzo szybko 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*. Mimo to 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`...].

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

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

Ciekawostka

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

1.2.3. Implementacja

Na kolejnym etapie abstrakcyjne algorytmy, zapisane często w postaci pseudokodu (np. za pomocą języka polskiego), należy przepisać w formie, która jest *zrozumiała* nie tylko przez nas, ale i *przez komputer*. Jest to tzw. *implementacja* algorytmów.

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

Ponadto, jeśli zachodzi taka potrzeba, na tym etapie należy dokonać scalenia czy też powiązania podzadań („rosół", „chili con carne" oraz „pierniczki") w jeden spójny projekt („obiad").

Zapamiętaj

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

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

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

Ciekawostka

Język C++ powstał jako rozwinięcie języka C. Wśród innych, podobnych do niego pod względem składni, języków można wymienić takie popularne narzędzia jak Java, C#, PHP czy JavaScript.

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

Kod źródłowy programu zapisujemy zwykle w postaci kilku plików tekstowych (tzw. plików źródłowych, ang. *source files*). Pliki te można edytować za pomocą dowolnego edytora tekstowego, np. *Notatnik* systemu Windows. Jednak do tego celu lepiej przydają się środowiska programistyczne. Na przykład, my podczas laboratoriów będziemy używać *Microsoft Visual C++*[2].

Zapamiętaj

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

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

[2]Dostępna jest bezpłatna wersja tego środowiska, zob. instrukcję instalacji opisaną w samouczku nr 1.

1.2.4. Testowanie

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

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

1.2.5. Eksploatacja

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

Nauka programowania komputerów może nam pomóc rozwiązać wiele problemów. Problemów, których rozwiązanie w inny sposób wcale nie byłyby dla nas dostępne. A ponadto zobaczymy, że jest wspaniałą rozrywką!

1.2.6. Podsumowanie

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

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

Efekty pracy po każdym z etapów podsumowuje rys. 1.3.

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

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

> Programowanie przynosi wiele radości; *Pierwsza to zwyczajna rozkosz tworzenia czegoś. Jak dziecko lubi stawiać domki z piasku, tak dorosły lubi budować coś, zwłaszcza według własnego projektu. [...] Druga to przyjemność robienia czegoś przydatnego dla innych ludzi. W głębi duszy chcemy, żeby inni korzystali z naszej pracy i uznawali ją za użyteczną. [...] Trzecia przyjemność to fascynacja tworzenia złożonych przedmiotów, przypominających łamigłówki składające się z zazębiających się, ruchomych części, i obserwowanie ich działania [...]. Czwarta przyjemność wypływa z ciągłego uczenia się i wiąże się z niepowtarzalnością istoty zadania. W taki czy inny sposób problem jest wciąż nowy, a ten, kto go rozwiązuje, czegoś się uczy [...]*

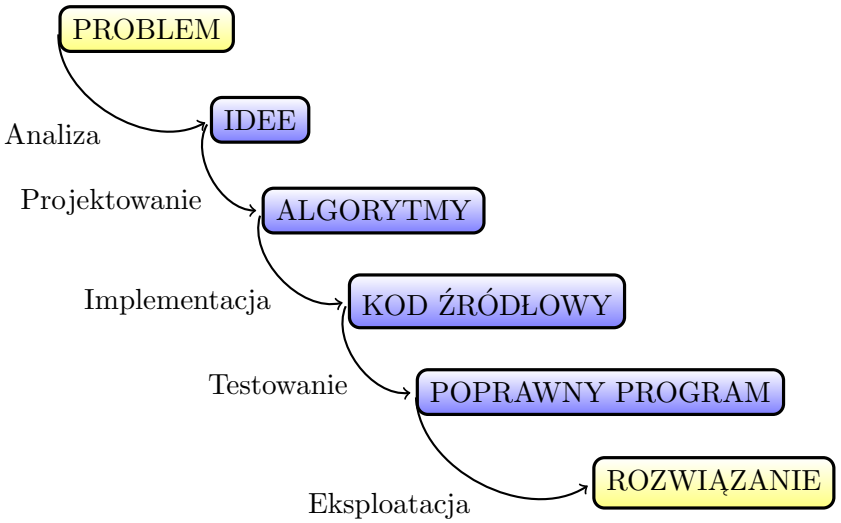
Brooks twierdzi także, że to, co przynosi radość, czasem musi powodować ból:

- należy działać perfekcyjnie,

– poprawianie i testowanie programu jest uciążliwe,

a także, w przypadku aplikacji komercyjnych:

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



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

1.3. Ćwiczenia

Zadanie 1.1. Rozważ poniższą implementację (w języku C++) algorytmu Euklidesa znajdywania największego wspólnego dzielnika (NWD) dwóch liczb naturalnych $0 \leqslant a < b$.

```
//  Wejście: 0 ⩽ a < b
//  Wyjście: NWD(a,b)
int NWD(int a, int b) // tj. NWD : ℕ × ℕ → ℕ
{
    assert(a >= 0 && a < b); // warunki poprawn. danych wej.
    int c;                   // niech c ∈ ℕ – zmienna pomocnicza
    while (a != 0)           // tzn. dopóki (a ≠ 0)
    {
        c = b % a;  // c staje się resztą z dzielenia b przez a
        b = a;
        a = c;
    }
    return b; // zwróć wartość b jako wynik;
}
```

Prześledź działanie algorytmu Euklidesa, prezentując w postaci tabelki, jakie wartości przyjmują zmienne a, b, c w każdym kroku. Rozpatrz: a) NWD(42, 56), b) NWD(192, 348), c) NWD(199, 544), d) NWD(2166, 6099).

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

Zadanie 1.3. Napisz kod, który przestawi wartości (a, b, c) na (c, a, b).

Zadanie 1.4. Napisz kod, który przestawi wartości (a, b, c, d) na (c, d, b, a).

Zadanie 1.5. Dokonaj permutacji ciągu (a, b, c) tak, by otrzymać ciąg (a', b', c') taki, że $a' \leqslant b' \leqslant c'$.

Zadanie 1.6. Pokaż, w jaki sposób za pomocą ciągu przypisań i podstawowych operacji arytmetycznych można mając na wejściu ciąg zmiennych (a, b, c) otrzymać a) $(b, c + 10, a/b)$, b) $(a+b, c^2, a+b+c)$, c) $(a-b, -a, c-a+b)$. Postaraj się zminimalizować liczbę użytych instrukcji i zmiennych pomocniczych.

Zadanie 1.7. [MD] Prześledź działanie poniższej funkcji w języku C++ wywołanej jako a) $f(2, 4)$ i b) $f(3, -1)$.

```
double f(double x, double y) // tj. f : ℝ × ℝ → ℝ
{
    double c;  // niech c ∈ ℝ – zmienna pomocnicza
    c = x + y + 1;
    if (y > 1) // jeśli y > 1
        c = c / y;
    return c;  // zwróć wartość c jako wynik
}
```

Zadanie 1.8. [MD] Wzorując się na kodzie z zadania 1.7 oraz wykorzystując funkcję f tam zdefiniowaną, napisz kod funkcji $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, która jest określona wzorem:

$$g(a, b) = \begin{cases} f(a + b, a - b) & \text{dla } a > b, \\ b - a & \text{dla } b \geqslant a. \end{cases}$$

Zadanie 1.9. Dany jest ciąg n liczb rzeczywistych $\mathbf{x} = (x[0], \ldots, x[n-1])$ (umawiamy się, że elementy ciągów numerujemy od 0). Dla ciągów $(1, -1, 2, 0, -2)$ oraz $(34, 2, -3, 4, 3.5)$

prześledź działanie następującego algorytmu służącego do wyznaczania średniej arytmetycznej, która jest określona wzorem:

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

```
// Wejście: n > 0 oraz x[0], x[1], ..., x[n-1] ∈ ℝ
// Wyjście: średnia arytmetyczna z (x[0], x[1], ..., x[n-1])
niech s ∈ ℝ;
s = 0;
dla (i=0,1,...,n-1)
    s = s + x[i];
zwróc s/n jako wynik;
```

Zadanie 1.10. Dany jest ciąg n dodatnich liczb rzeczywistych $\mathbf{x} = (x[0], \ldots, x[n-1])$ (różnych od 0). Napisz algorytm, który wyznaczy ich średnią harmoniczną:

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

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

Zadanie 1.11. Napisz kod, który dla danych dodatnich liczb rzeczywistych a, b, c sprawdzi, czy może istnieć trójkąt prostokątny o bokach podanych długości.

Zadanie 1.12. Napisz kod, który dla danych dodatnich liczb rzeczywistych a, b, c sprawdzi, czy może istnieć trójkąt (dowolny) o bokach podanych długości.

Zadanie 1.13. Napisz kod, który dla danych liczb rzeczywistych a, b, c wyznaczy rozwiązanie równania $ax^2 + bx + c = 0$ (względem x).

⋆ **Zadanie 1.14.** [PS] „Szklana pułapka”. Masz do dyspozycji pojemniki na wodę o objętości x i y litrów oraz dowolną ilość wody w basenie. Czy przy ich pomocy (pojemniki wypełnione do pełna) można wybrać z litrów wody? Napisz pseudokod algorytmu, który to sprawdza i dokonaj obliczeń dla a) $x = 13, y = 31, z = 1111$, b) $x = 12, y = 21, z = 111$.

1.4. 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 1.9.
Wynik dla $(1, -1, 2, 0, -2)$: 0.
Wynik dla $(34, 2, -3, 4, 3.5)$: 8,1. □

Odpowiedź do zadania 1.10.
Wynik dla $(1, 4, 2, 3, 1)$: $\frac{60}{37}$.
Wynik dla $(10, 2, 3, 4)$: $\frac{240}{71}$. □

Odpowiedź do zadania ??. 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 $(+, -, *, /)$. □

Marek Gągolewski
Instytut Badań Systemowych PAN
Wydział Matematyki i Nauk Informacyjnych Politechniki Warszawskiej

Algorytmy i podstawy programowania

2. Podstawy organizacji i działania komputerów. Zmienne w języku C++ i ich typy. Operatory

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

2.1. 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 najistotniejszych wydarzeń z dziejów tej dziedziny, które pomogą nam rzucić światło na przedmiot naszych zainteresowań.

- Ok. 2400 r. p.n.e. w Babilonii używany jest już kamienny pierwowzór liczydła, na którym rysowano piaskiem (specjaliści umiejący korzystać z tego urządzenia nie byli liczni). Podobne przyrządy w Grecji i Rzymie pojawiły się dopiero ok. V-IV w. p.n.e. Tzw. *abaki*, żłobione w drewnie, pozwalały dokonywać obliczeń w systemie dziesiętnym.
- Ok. V w. p.n.e. indyjski uczony Pãnini sformalizował teoretyczne reguły gramatyki sanskrytu. Można ten fakt uważać za pierwsze badanie teoretyczne w dziedzinie lingwistyki.
- Pierwszy algorytm przypisywany jest Euklidesowi (ok. 400-300 r. p.n.e.). Ten starożytny uczony opisał operacje, których wykonanie krok po kroku pozwala wyznaczyć największy wspólny dzielnik dwóch liczb (por. zestaw zadań nr 1).
- Matematyk arabski Al Kwarizmi (IX w.) określa reguły podstawowych operacji arytmetycznych dla liczb dziesiętnych. Od jego nazwiska pochodzi ponoć pojęcie *algorytmu*.
- W epoce baroku dokonuje się „obliczeniowy przełom". W 1614 r. John Napier, szkocki teolog i matematyk, znalazł zastosowanie logarytmów do wykonywania szybkich operacji mnożenia (zastąpił je dodawaniem). W roku 1622 William Oughtred stworzył *suwak logarytmiczny*, który jeszcze bardziej ułatwił wykonywanie obliczeń.
- W. Schickard (1623 r.) oraz B. Pascal (1645 r.) tworzą pierwsze *mechaniczne sumatory*. Ciekawe, czy korzystał z nich Isaac Newton (1643–1727)?

Mamy bowiem $xy = e^{\log x + \log y}$.

Zadanie

Zastanów się przez chwilę, do czego może być przydatna umiejętność szybkiego wykonywania operacji arytmetycznych. W jakich dziedzinach życia czy nauki może się przydać?

- Jacques Jacquard (ok. 1801 r.) skonstruował krosno tkackie sterowane dziurkowanymi kartami. Było to pierwsze *programowalne* urządzenie w dziejach techniki. Podobny ideowo sposób działania miały *pianole* (poł. XIX w.), czyli automatycznie sterowane pianina.
- Ok. 1837–1839 r. Charles Babbage opisał wymyśloną przez siebie „maszynę analityczną" (parową!), programowalne urządzenie do wykonywania obliczeń. Była to jednak tylko koncepcja teoretyczna: nigdy nie udało się jej zbudować. Pomagająca mu Ada Lovelace może być uznana za *pierwszego programistę*.
- H. Hollerith konstruuje w 1890 r. maszynę zorientowaną na przetwarzanie dużej ilości danych, która wspomaga podsumowywanie wyników spisu powszechnego w USA.
- Niemiecki inżynier w 1918 r. patentuje maszynę szyfrującą Enigma. Na marginesie, jej kod łamie polski matematyk Marian Rejewski w 1932 r., co przyczyni się później do sukcesu aliantów w drugiej Wojnie Światowej.
- 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*.

Pierwszym programistą była kobieta!

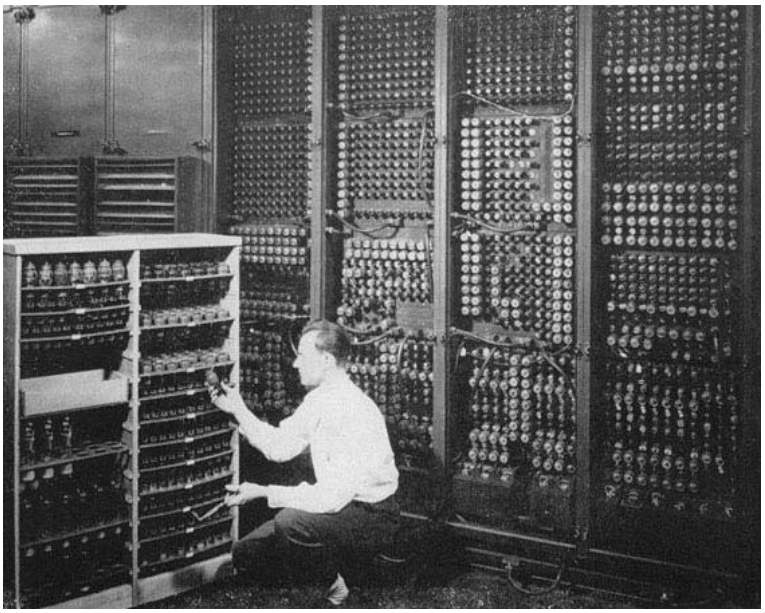
Informacja

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

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

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

- Claude E. Shannon w swojej pracy magisterskiej w 1937 r. opisuje możliwość użycia *elektronicznych przełączników* do wykonywania operacji logicznych (implementacja algebry Boole'a). Jego praca teoretyczna staje się podstawą konstrukcji wszystkich współczesnych komputerów elektronicznych.
- W latach 40' XX w. w Wielkiej Brytanii, USA i Niemczech *powstają pierwsze komputery elektroniczne*, np. Z1, Z3, Colossus, Mark I, ENIAC (zob. rys. 2.1), EDVAC, EDSAC. Pobierają bardzo duże ilości energii elektrycznej i potrzebują dużych przestrzeni. Np. ENIAC miał masę ponad 27 ton, zawierał około 18 000 lamp elektronowych i zajmował powierzchnię ok. 140 m^2. Były bardzo wolne (jak na dzisiejsze standardy), np. Z3 wykonywał jedno mnożenie w 3 sekundy. Pierwsze zastosowania były stricte militarne: łamanie szyfrów i 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 (np. słynny test Turinga sprawdzający „inteligencję" maszyny).
- Pierwszy *kompilator* języka Fortran zostaje stworzony w 1957 r. Kompilator języka LISP powstaje rok później. Wcześniej programowano komputery w tzw. języku maszynowym, bardzo trudnym do zrozumienia dla człowieka.
- Informatyka staje się *dyscypliną akademicką* dopiero w latach sześćdziesiątych XX w. (wytyczne programowe ACM). Pierwszy wydział informatyki powstał na Uniwersytecie Purdue w 1962 r. Pierwszy doktorat w tej dziedzinie został obroniony już w 1965 r.
- W 1969 r. następuje pierwsze *połączenie sieciowe* pomiędzy komputerami w ramach projektu ARPAnet (prekursora Internetu). Początkowo ma ono mieć głównie zastosowanie (znowu!) wojskowe.
- Dalej rozwój następuje bardzo szybko: powstają bardzo ważne algorytmy (np. wyszukiwanie najkrótszych ścieżek w grafie Dijkstry, sortowanie szybkie Hoara itp.), teoria relacyjnych baz danych, okienkowe wielozadaniowe systemy operacyjne, nowe języki programowania, systemy rozproszone i wiele, wiele innych...
- Współcześnie komputer osobisty (o mocy kilka miliardów razy większej niż ENIAC) podłączony do ogólnoświatowej sieci Internet znajdziemy w prawie każdym domu, a elementy informatyki wykładane są już w szkołach podstawowych.



Rys. 2.1. Komputer ENIAC

Zadanie

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

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

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

Zapamiętaj

Wielu matematyków zajmuje się informatyką, a wielu informatyków — matematyką! Bardzo płodnym z punktu widzenia nauki jest obszar na styku obydwu dziedzin. Jeśli planujesz w przyszłości karierę naukowca czy analityka danych może to będzie i Twoja droga. Jeśli nie, tym bardziej dostrzeżesz za kilka lat, że bez komputera po prostu nie będziesz mógł (mogła) wykonywać swojej pracy.

2.1.1. Główne kierunki badań w informatyce współczesnej

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

1. Matematyczne podstawy informatyki
   (a) Kryptografia (kodowanie informacji niejawnej)

[1] Por. *www.newworldencyclopedia.org/entry/Computer_science*.

   (b) Teoria grafów (np. algorytmy znajdywania najkrótszych ścieżek, algorytmy kolorowania grafów)
   (c) Logika (w tym logika wielowartościowa, logika rozmyta)
   (d) Teoria typów (analiza formalna typów danych, m.in. badane są efekty związane z bezpieczeństwem programów)

2. Teoria obliczeń

   (a) Teoria automatów (analiza abstrakcyjnych maszyn i problemów, które można z ich pomocą rozwiązać)
   (b) Teoria obliczalności (co jest obliczalne?)
   (c) Teoria złożoności obliczeniowej (które problemy są „łatwo" rozwiązywalne?)

3. Algorytmy i struktury danych

   (a) Analiza algorytmów (ze względu na czas działania i potrzebne zasoby)
   (b) Projektowanie algorytmów
   (c) Struktury danych (organizacja informacji)
   (d) Algorytmy genetyczne (znajdywanie rozwiązań przybliżonych problemów optymalizacyjnych)

4. Języki programowania i kompilatory

   (a) Kompilatory (budowa i optymalizacja programów przetwarzających kod w danym języku na kod maszynowy)
   (b) Języki programowania (formalne paradygmaty języków, własności języków)

5. Bazy danych

   (a) Teoria baz danych (m.in. systemy relacyjne, obiektowe)
   (b) Data mining (wydobywanie wiedzy z baz danych)

6. Systemy równoległe i rozproszone

   (a) Systemy równoległe (bada wykonywanie jednoczesnych obliczeń przez wiele procesorów)
   (b) Systemy rozproszone (rozwiązywanie tego samego problemu z użyciem wielu komputerów połączonych w sieć)
   (c) Sieci komputerowe (algorytmy i protokoły zapewniające niezawodny przesył danych pomiędzy komputerami)

7. Architektura komputerów

   (a) Architektura komputerów (projektowanie, organizacja komputerów, także z punktu widzenia elektroniki)
   (b) Systemy operacyjne (systemy zarządzające zasobami komputera i pozwalające uruchamiać inne programy)

8. Inżynieria oprogramowania

   (a) Metody formalne (np. automatyczne testowanie programów)
   (b) Inżynieria (zasady tworzenia dobrych programów, zasady organizacji procesu analizy, projektowania, implementacji i testowania programów)

9. Sztuczna inteligencja

   (a) Sztuczna inteligencja (systemy, które wydają się cechować inteligencją)
   (b) Automatyczne wnioskowanie (imitacja zdolności samodzielnego rozumowania)
   (c) Robotyka (projektowanie i konstrukcja robotów i algorytmów nimi sterujących)

   (d) Uczenie się maszynowe (tworzenie zestawów reguł w oparciu o informacje wejściowe)

10. Grafika komputerowa

   (a) Podstawy grafiki komputerowej (algorytmy generowania i filtrowania obrazów)
   (b) Przetwarzanie obrazów (wydobywanie informacji z obrazów)
   (c) Interakcja człowiek-komputer (zagadnienia tzw. interfejsów służących do komunikacji z komputerem)

Do tej listy można też dopisać wiele dziedzin z tzw. pogranicza matematyki i informatyki, np. bioinformatykę, statystykę obliczeniową, analizę danych, uczenie się maszynowe, zbiory rozmyte itd.

Zapamiętaj

Zauważ, że w trakcie studiów mniej lub bardziej dokładnie zapoznasz się z zagadnieniami z p. 1–5. Pamiętaj o tym.

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

Informacja

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

2.2. Organizacja współczesnych komputerów

Jak zdążyliśmy nadmienić, podstawy teoretyczne działania współczesnych komputerów elektronicznych zawarte zostały w pracy C. Shannona (1937 r.). Komputery, na na najniższym możliwym poziomie abstrakcji, mogą być pojmowane jako zestaw odpowiednio sterowanych *przełączników*, tzw. *bitów* (ang. *binary digits*).

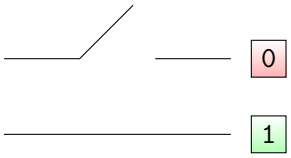
Zapamiętaj

Każdy przełącznik może znajdować się w jednym z dwóch stanów (rys. 2.2):

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

Co ważne, informacje, które przetwarza komputer (por. rozdz. 1 — określiliśmy je jako ciągi liczb lub symboli) zawsze reprezentowane są przez ciągi zer i jedynek. Tylko tyle

i aż tyle. W niniejszym rozdziale dowiemy się m.in. w jaki sposób za ich pomocą możemy zapisywać liczby całkowite, rzeczywiste, a kiedy indziej — znaki drukowane (np. litery alfabetu, tzw. kod ASCII).

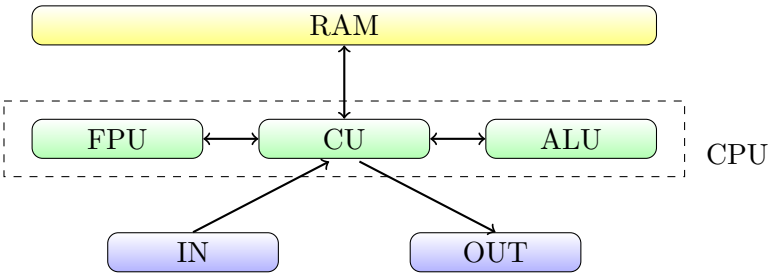


Rys. 2.2. Dwa stany przełączników

Na współczesny komputer osobisty (PC, od ang. *personal computer*) składają się następujące elementy:

1. 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;
2. pamięć *RAM* (od ang. *random access memory*) — zawiera dane i program,
3. urządzenia wejściowe (ang. *input devices*),
4. urządzenia wyjściowe (ang. *output devices*).

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



Rys. 2.3. Architektura współczesnych komputerów osobistych

2.3. Zmienne w języku C++ i ich typy

2.3.1. Pojęcie zmiennej

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

```
niech x ∈ ℝ;
```

Tym samym mieliśmy na myśli: „niech *od tej pory* `x` będzie *zmienną* ze zbioru liczb rzeczywistych". Formalnie, dokonaliśmy tym samym *deklaracji* zmiennej (ang. *variable declaration*) typu rzeczywistego. Dzięki temu mogliśmy użyć `x` np. do obliczenia wartości pewnego podwyrażenia w skomplikowanym wzorze tak, by rozwiązanie uczynić czytelniejszym.

Zapamiętaj

Zmienna służy do przechowywania w pamięci RAM komputera (lub, w przypadku pseudokodu, czymś, co pamięć komputera reprezentuje) dowolnych wartości z pewnego ustalonego zbioru (dziedziny), zwanego *typem* zmiennej.

2.3.2. Typy liczbowe

Jako że każda zmienna musi mieć określoną dziedzinę, omówmy wpierw typy zmiennych liczbowych dostępne w języku C++. Będą to:

1. typy całkowite, reprezentujące odpowiednie podzbiory $\mathbb{Z}$,
2. typy zmiennoprzecinkowe, reprezentujące pewne podzbiory $\mathbb{R}$ oraz
3. typ logiczny.

2.3.2.1. Typy całkowite

Do *typów całkowitych* zaliczamy:

Nazwa typu	Zakres	Liczba bitów[1]
`char`	$\{-128, -127, \ldots, 127\}$	8
`short`	$\{-32\,768, -32\,767, \ldots, 32\,767\}$	16
`int`	$\{-2\,147\,483\,648, \ldots, 2\,147\,483\,647\}$	32
`long long`	$\{\sim -9 \times 10^{18}, \ldots, \sim 9 \times 10^{18}\}$	64

Do reprezentacji liczb całkowitych będziemy korzystać z typu `int`. W 32-bitowych systemach operacyjnych typ ten pozwala zapisywać liczby rzędu $\pm$ 2 miliardów. W systemach 64-bitowych (zależy to ponadto od wersji kompilatora) typ ten być może pozwoli na reprezentację liczb z zakresu $\pm$ 9 trylionów.

Ciekawostka

Typ `int` reprezentuje najczęściej co najmniej 32-bitową (64-bitową w niektórych systemach operacyjnych i wersjach kompilatora) liczbę całkowitą w systemie U2 (ze znakiem). Czytelników zainteresowanych reprezentacją liczb odsyłamy do rozdz. 2.5.

Zapamiętaj

Zauważmy, że typy zmiennych nie reprezentują całego zbioru liczb całkowitych $\mathbb{Z}$, a jedynie ich podzbiór. Ich zakres jest jednak na tyle duży, że wystarcza do większości zastosowań praktycznych.

[1] Podana liczba bitów dotyczy *Microsoft Visual C++* działającym w środowisku 32 bitowym, zob. rozdz. 2.5.4.

Ciekawostka

Do każdego z wyżej wymienionych typów możemy zastosować modyfikator `unsigned` — otrzymujemy liczbę nieujemną (bez znaku), np.

$$\texttt{unsigned int} = \{0, 1, \ldots, 4{,}294{,}967{,}295\} \subset \mathbb{N}_0.$$

W codziennej praktyce jednak nie używa się go zbyt często.

Informacja

Stałe całkowite, np. `0`, `-5`, `15210` są reprezentantami typu `int` danymi w postaci dziesiętnej, czyli takiej, do której jesteśmy przyzwyczajeni.

Oprócz tego można używać stałych w innych systemach liczbowych (zob. rozdz. 2.5) np. 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.

Zapamiętaj

`010` oraz `10` oznaczają dwie różne liczby! Pierwsza z nich to liczba równa *osiem* (notacja ósemkowa), a druga — *dziesięć* (notacja dziesiętna).

2.3.2.2. Typy zmiennoprzecinkowe

Do *typów zmiennoprzecinkowych* zaliczamy:

Nazwa typu	Zakres (przybliżony)	Liczba bitów
`float`	$\subseteq [-10^{38}, 10^{38}]$	32
`double`	$\subseteq [-10^{308}, 10^{308}]$	64

Do reprezentacji liczb rzeczywistych będziemy używać najczęściej typu `double`.

Informacja

Stałe zmiennoprzecinkowe wprowadzamy podając zawsze część dziesiętną i część ułamkową rozdzieloną kropką (nawet gdy część ułamkowa jest równa 0), np. `3.14159`, `1.0` albo `-0.000001`. Domyślnie stałe te należą 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 $-2{,}32 \times 10^{-4}$.

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

Zainteresowanym osobom można polecić następujący angielskojęzyczny artykuł (o bardzo sugestywnym tytule): D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys* 21(1), 1991, 5–48; zob. *www.ibspan.waw.pl/~gagolews/Teaching/aipp/priv/Czytelnia/Goldberg1991.pdf*.

Zapamiętaj

Warto mieć na uwadze zawsze problemy związane z użyciem liczb zmiennoprzecinkowych.

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

Ich konsekwencje będziemy badać wielokrotnie podczas zajęć laboratoryjnych.

Informacja

W standardzie IEEE-754, który jest stosowany w komputerach typu PC, określone zostały wartości specjalne, które mogą być przyjmowane przez zmienne typu zmiennoprzecinkowego:

- ∞ (`Inf`, ang. *infinity*),
- $-\infty$ (`-Inf`),
- nie-liczba (`NaN`, ang. *not a number*).

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

2.3.2.3. Typ logiczny

Dostępny jest także typ `bool`, służący do reprezentowania *zbioru wartości logicznych*.

Informacja

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

2.3.3. Identyfikatory

Mamy dwie możliwości użycia zmiennej: możemy *przypisać* jej pewną wartość (jednocześnie „kasując” wartość poprzednią) bądź przechowywaną weń wartość *odczytać*. Każda

zmienna musi mieć swój *identyfikator*, to jest jednoznaczną nazwę używaną po to, by można się było do niej odwołać. Identyfikatory mogą składać się z następujących znaków:
`a b c d e f g h i j k l m n o p q r s t u v w x y z`
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _`
oraz z poniższych, pod warunkiem, że nie są one pierwszym znakiem identyfikatora:
`0 1 2 3 4 5 6 7 8 9`

Wobec tego przykładami poprawnych identyfikatorów są: `i`, `suma`, `wyrazenieZeWzoru3`, `_2m1a` oraz `zmienna_pomocnicza1`. Zgodnie jednak z podaną regułą np. `3523aaa`, `ala ma kota` oraz `:-)` nie mogą być identyfikatorami.

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

Zapamiętaj

W języku C++ wielkość liter ma znaczenie!

Dla przykładu, `T` oraz `t` to identyfikatory dwóch różnych zmiennych. Dalej, np. identyfikator `Int` (poprawny!) nie jest tożsamy z `int` (słowo kluczowe).

Informacja

Dobrą praktyką jest nadawanie zmiennym takich nazw, by „same się objaśniały". Dzięki temu czytając kod programu będzie nam łatwiej dowiedzieć się, do czego dana zmienna jest używana.

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

2.3.4. Deklaracja zmiennych

Dzięki zmiennym pisane przez nas programy (ale także np. twierdzenia matematyczne) nie opisują jednego, szczególnego przypadku pewnego interesującego nas problemu (np. rozwiązania konkretnego układu dwóch równań liniowych), lecz wszystkie możliwe w danym kontekście (np. rozwiązania dowolnego układu dwóch równań liniowych). Taki mechanizm daje więc nam możliwość *uogólniania* (abstrakcji) algorytmów.

Oto prosty przykład. Ciąg twierdzeń:
Twierdzenie 1. *Pole koła o promieniu* 1 *wynosi* π.
Twierdzenie 2. *Pole koła o promieniu* 2 *wynosi* 4π.
Twierdzenie 3. ...
przez każdego matematyka (a nawet przyszłego matematyka) byłby uznany, łagodnie rzecz ujmując, za nieudany żart. Jednakże uogólnienie powyższych wyników przez odwołanie się do pewnej, wcześniej 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 właśnie formie. Większość naukowców 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 (czyli założenie, że r jest liczbą rzeczywistą dodatnią). Wiemy już jednak z pierwszego wykładu, że komputery nie są tak domyślne jak my.

Zapamiętaj

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

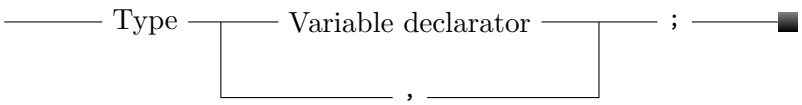
Aby zadeklarować zmienną, należy napisać w kodzie programu odpowiednią instrukcję o ściśle określonej składni.

Informacja

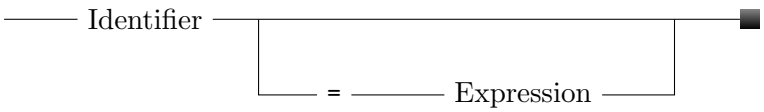
Instrukcja (ang. *statement*) jest podstawowym składnikiem pisanego programu. Dzięki niej możemy nakazać komputerowi wykonać pewną ustaloną czynność (być może bardzo złożoną), której znaczenie jest ściśle określone przez reguły semantyczne języka programowania.

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

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



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



Tutaj Identifier oznacza identyfikator zmiennej, a Expression jest pewnym wyrażeniem. Więcej o wyrażeniach dowiemy się za chwilę. Na razie wystarczy nam wiedzieć, że wyrażeniem może być po prostu stała całkowita bądź zmiennoprzecinkowa albo też pewna operacja arytmetyczna.

Zapamiętaj

Prawie wszystkie instrukcje w języku C++ muszą być zakończone *średnikiem*!

Oto kilka przykładów instrukcji deklarujących zmienne:

```
int x;              // niech x będzie zmienną całkowitą
bool p = false;     /* niech p będzie wartością logiczną
                       ustawioną (na początku) na PRAWDA */
int y=10;
double a, b;        // niech a i b będą zmiennymi typu double
int u = y+10, v = -y;
```

Informacja

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

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

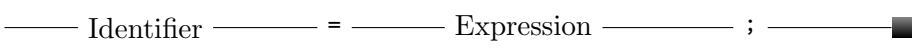
2.3.5. Operator przypisania

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

Informacja

Zmienna nie jest inicjowana automatycznie. Dopóki nie przypiszemy jej jawnie jakiejś wartości, będzie przechowywać „śmieci”. Próba odczytania wartości niezainicjowanej zmiennej może (lecz nie musi, jest to zależne od kompilatora) skutkować nagłym zakończeniem uruchomionego programu.

Operator przypisania (ang. *assignment operator*), służy do nadania zmiennej określonej wartości. Składnię instrukcji, która go wykorzystuje, obrazuje poniższy diagram:



Co ważne, najpierw obliczana jest wartość wyrażenia Expression. Dopiero po tym wynik wpisywany jest do zmiennej o nazwie Identifier.

Prześledźmy poniższy kod.

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

Informacja

Obiekt o nazwie `cout` umożliwia wypisywanie wartości zmiennych różnych typów na ekran monitora (a dokładnie, na tzw. standardowe wyjście). Wartość wyrażenia (powyżej: wartość przechowywaną w zmiennej `j`) „wysyła" się na ekran za pomocą operatora <<.

Inny sposób przypisywania wartości zmiennym, dzięki któremu wartości nie muszą być z góry ustalane podczas pisania programu, zapewnia obiekt `cin` i operator >>. Korzystając z nich możemy poprosić użytkownika już podczas *działania* napisanego przez nas programu o podanie za pomocą klawiatury konkretnych wartości, które mają być przypisane określonym zmiennym.

Przykład:

```
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;
```

Ciekawostka

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

— `const` — Type — Identifier — = — Expr. — ; —■

Przy tworzeniu stałej symbolicznej należy od razu przypisać jej wartość. Cechą zasadniczą stałych jest to, że po ich zainicjowaniu nie można wcale zmieniać ich wartości. Ich stosowanie może zmniejszać prawdopodobieństwo omyłek 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"
```

2.3.6. Rzutowanie (konwersja) typów. Hierarchia typów

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

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

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

x = 3.14159;

```

```
11 j = x;       // błąd! (typ docelowy jest "mniej pojemny")
12 j = (int)x; // jawna konwersja (tzw. rzutowanie) — teraz OK
13
14 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 wyniku działania takiej instrukcji mogłaby wystąpić utrata informacji. W języku C++ obowiązuje następująca *hierarchia typów*:

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

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

Z drugiej strony, rzutowanie do słabszego typu musi być zawsze jawne. Należy *explicité* oznajmić kompilatorowi, że postępujemy świadomie. Służy do tego tzw. *operator rzutowania*. Taki operator umieszczamy przed rzutowanym wyrażeniem. Oprócz zastosowanej w powyższym przykładzie składni `(typ)wyrażenie`, dostępne są dodatkowo dwie równoważne: `typ(wyrażenie)` oraz `static_cast<typ>(wyrażenie)`.

Zapamiętaj

W przypadku rzutowania zmiennej typu zmiennoprzecinkowego 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`.

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

2.4. Operatory

2.4.1. Operatory arytmetyczne

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

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

Informacja

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

Zastosowanie operatorów do argumentów dwóch różnych typów powoduje konwersję operandu o typie słabszym do typu silniejszego.

Przykłady:

```
cout << 2+2 << endl;
cout << 3.0/2.0 << endl;
cout << 3/2 << endl;
cout << 7%4 << endl;
cout << 1/0.0 << endl; // uzgodnienie typów: to samo, co 1.0/0.0
```

Wynikiem wykonania powyższych instrukcji będzie:

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

Oprócz operatorów arytmetycznych binarnych mamy też dostęp do kilku *operatorów unarnych*, czyli jednoargumentowych.

Operator	Znaczenie
+	unarny plus (nic nie robi)
−	unarny minus (zmiana znaku na przeciwny)
++	inkrementacja (przedrostkowy i przyrostkowy)
−−	dekrementacja (przedrostkowy i przyrostkowy)

Przykład:

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

Operator *inkrementacji* (++) służy do zwiększania wartości zmiennej o 1, a operator *dekrementacji* (−−) zmniejszania o 1. Stosuje się je najczęściej na *zmiennych* typu całkowitego. Operatory te występują w dwóch wariantach: przedrostkowym (ang. *prefix*) oraz przyrostkowym (ang. *suffix*).

W przypadku operatora inkrementacji/dekrementacji *przedrostkowego*, wartością całego wyrażenia jest wartość zmiennej po wykonaniu operacji (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).

Zadanie

Przeanalizuj następujący 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
```

Podsumowując, `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. Nie polecamy zatem stosować ich w przypadkach takich jak te oznaczone gwiazdką w powyższym przykładzie.

2.4.2. Operatory relacyjne

Operatory relacyjne służą do porównywania dwóch wartości. Są to więc operatory binarne. Wynikiem porównania jest zawsze wartość typu `bool`.

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

Zapamiętaj

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

Przykłady:

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

2.4.3. Operatory logiczne

Operatory logiczne służą do przeprowadzania działań na wyrażeniach typu `bool` (lub takich, które są sprowadzalne do `bool`). W wyniku ich działania otrzymujemy wartość logiczną.

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

Mamy, co następuje:

!	
false	true
true	false

||	false	true
false	false	true
true	true	true

&&	false	true
false	false	false
true	false	true

Kilka przykładów:

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

Ciekawostka

Okazuje się, że komputery są czasem leniwe. W przypadku wyrażeń składających się z wielu podwyrażeń logicznych, obliczane jest tylko to, co jest potrzebne do ustalenia wyniku. I tak w przypadku koniunkcji, jeśli pierwszy argument ma wartość `false`, to wyznaczanie wartości drugiego jest niepotrzebne. Podobnie jest dla alternatywy i pierwszego argumentu o wartości `true`. 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
```

Zwróćmy jednak uwagę, że czytelność takiego kodu pozostawia wiele do życzenia. Mimo to niektórzy programiści lubią stosować różne „sztuczki języka C++" w swoich programach. Dopóki dobrze nie opanujemy języka, starajmy się powstrzymać od tego typu skrótowców.

2.4.4. Operatory bitowe ⋆

Operatory bitowe działają na poszczególnych bitach liczb całkowitych (por. rozdz. 2.5). 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

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

Zapamiętaj

Nie należy mylić operatorów `~` (który w notacji matematycznej oznacza czasem logiczną negację) i `!` (który oznacza logiczną negację w C++).

Operatory przesunięcia zmieniają pozycje bitów w liczbie. Bity, które nie „mieszczą się" w danym typie są tracone. W przypadku operatora `<<`, na zwalnianych pozycjach

wstawiane są zera. Operator >> wstawia zaś na zwalnianych pozycjach bit znaku (por. rozdz. 2.5.4). Dla przykładu, dokonajmy przesunięcia bitów liczby 8 bitowej o 3 pozycje w lewo.

$$\underset{\texttt{0xab}}{10101011} \quad \texttt{<< 3} \quad == \quad \overbrace{101\,01011\mathbf{000}}^{\text{wynik}}\ \underset{\texttt{0x58}}{}$$

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
```

Ciekawostka

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

Informacja

Zauważmy, że operatory << i >> mają inne znaczenie w przypadku użycia ich na obiektach, odpowiednio, cout i cin. Jest to praktyczny przykład zastosowania tzw. *przeciążania operatorów*, czyli różnicowania ich znaczenia w zależności od kontekstu. Więcej na ten temat dowiemy się podczas wykładu z *Programowania obiektowego* na II semestrze.

2.4.5. Operatory łączone

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

$$\texttt{+=},\ \texttt{-=},\ \texttt{*=},\ \texttt{/=},\ \texttt{\%=}.$$

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

Ciekawostka

W analogiczny sposób można też łączyć operatory bitowe.

2.4.6. Priorytety operatorów

Zaobserwowaliśmy, że w jednym wyrażeniu można używać wielu operatorów. Kolejność wykonywania działań jest jednak ściśle określona. Domyślnie wyrażenia obliczane są według priorytetów, zestawionych w kolejności od największego do najmniejszego w tab. 2.1.

Dla przykładu, zapis `1+2*0` test tożsamy z `1+(2*0)`, ponieważ operacja mnożenia ma wyższy priorytet niż dodawanie. Co ważne, w przypadku operatorów o tym samym priorytecie, operacje wykonywane są w kolejności od lewej do prawej, np. `4*3/2` oznacza to samo, co `(4*3)/2`.

Zapamiętaj

W przypadku jakiejkolwiek wątpliwości co do priorytetu stosowanych operacji, określoną kolejność działań możemy zawsze wymusić za pomocą nawiasów okrągłych (...).

Tab. 2.1. Priorytety operatorów

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

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++;
                      doskonały przykład na to, jak nie pisać! */
cout << p1;           // 9
```

2.5. Reprezentacja liczb całkowitych ⋆

Powiedzieliśmy wcześniej, że wszelkie informacje w komputerze reprezentowane są przez ciągi bitów. Zainteresowany Czytelnik z pewnością będzie chciał się dowiedzieć, w jaki sposób zapisywane są *liczby całkowite*.

Zacznijmy od liczb nieujemnych.

Informacja

Systemy liczbowe można podzielić na pozycyjne i addytywne. Oto przykłady każdego z nich.

Pozycyjne systemy liczbowe:

1. system dziesiętny (decymalny, indyjsko-arabski) — o podstawie 10,
2. system dwójkowy (binarny) — o podstawie 2,
3. system szesnastkowy (heksadecymalny) — o podstawie 16,
4. ...

Addytywne systemy liczbowe:

1. system rzymski,
2. system sześćdziesiątkowy (Mezopotamia).

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

2.5.1. System dziesiętny ⋆

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

Jako przykład rozważmy liczbę 1945_{10} (przyrostek $_{10}$ wskazuje, że chodzi nam 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**	

Równoważne:

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

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**				$\times 10^3$
$+$	$\vdots$	**9**			$\times 10^2$
$+$	$\vdots$	$\vdots$	**4**		$\times 10^1$
$+$	$\vdots$	$\vdots$	$\vdots$	**5**	$\times 10^0$
	3	2	1	0	

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

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

Liczbę 10 występującą we wzorze nazywamy *podstawą systemu*.

Jak łatwo się domyślić, można określić systemy pozycyjne o innych podstawach. Systemy o podstawie 2 i 16 są najbardziej przydatne z punktu widzenia informatyki.

2.5.2. System dwójkowy ⋆

Dwójkowy pozycyjny system liczbowy (binarny, ang. *binary*) to system pozycyjny o podstawie 2. Używanymi tu symbolami (cyframi) są tylko 0 oraz 1. Zauważmy, że cyfry te mają właśnie „przełącznikową" interpretację: prąd nie płynie/płynie czy też 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.

Konwersja liczb do systemu dziesiętnego jest stosunkowo prosta. Przyjrzyjmy się jak przekształcić liczbę dziesiętną na dwójkową. Na listingu 2.1 podajemy algorytm, który może być wykorzystany w tym celu.

Listing 2.1. Konwersja liczby dziesiętnej na dwójkową

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

    i = i – 1;
}
```

Tab. 2.2. Przykład: konwersja liczby 1945_{10} na dwójkową

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

Dla przykładu, rozważmy liczbę 1945_{10}. Tablica 2.2 opisuje kolejno wszystkie kroki potrzebne do uzyskania wyniku w postaci dwójkowej. Możemy z niej odczytać, iż $1945_{10} = 11110011001_2$.

2.5.3. System szesnastkowy ⋆

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

Po co taki system, skoro komputer właściwie opiera się na liczbach w systemie binarnym? Jak można zauważyć, liczba w postaci dwójkowej prezentuje się na kartce lub na ekranie dosyć... okazale. Korzystając z faktu, że $16 = 2^4$, możemy użyć jednego symbolu w zapisie szesnastkowym zamiast czterech w systemie dwójkowym. Fakt ten sprawia również, że konwersja z systemu binarnego na heksadecymalny i odwrotnie jest bardzo prosta.

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

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

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

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

Zatem na przykład $1011110_2 = 5E_{16}$, 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 rozdz. 2.5.4!).

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

2.5.4. 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).

Zapamiętaj

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

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

Jeśli dana jest n cyfrowa liczba w systemie U2 postaci $b_{n-1}b_{n-2}\ldots b_1b_0$, gdzie $b_i \in \{0,1\}$ dla $i = 0, 1, \ldots, n-1$, 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, najstarszego bitu równego 0 nie można dowolnie pomijać w zapisie! Mamy bowiem $0b_{n-2}\ldots b_{0\mathrm{U}2} \neq b_{n-2}\ldots b_{0\mathrm{U}2}$, gdy $b_{n-2} = 1$. 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 (zob. ćwiczenia), że również

$$1b_{n-2}\ldots b_{0\mathrm{U}2} = 11\ldots 1b_{n-2}\ldots b_{0\mathrm{U}2}.$$

Informacja

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.

2.6. Ćwiczenia

Zadanie 2.1. Jaką wartość mają następujące liczby dane w notacji naukowej: a) `4.21e6`, b) `1.95323e2`, c) `2.314e-4`, d) `4.235532e-2`?

Zadanie 2.2. Wyznacz wartość następujących wyrażeń. Ponadto, określ jakiego są one typu (`int` czy `double`).

1. `10.0+15.0/2+4.3`,
2. `10.0+15/2+4.3`,
3. `3.0*4/6+6`,
4. `20.0-2/6+3`,
5. `10+17*3+4`,
6. `10+17/3.0+4`,
7. `3*4%6+6`,
8. `3.0*4%6+6`,
9. `10+17%3+4`.

Zadanie 2.3. Dane są trzy zmienne zadeklarowane i zainicjowane w następujący sposób.

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

Oblicz wartość poniższych wyrażeń.

1. `int(a)`,
2. `int(c)`,
3. `int(a+b)`,
4. `int(a)+b+c`,
5. `int(a+b)*c`,
6. `double(int(a))/c`.

Zadanie 2.4. Korzystając z przekształceń logicznych (np. praw De Morgana, praw rozdzielności) uprość następujące wyrażenia.

1. `!(!p)`,
2. `!p && !q` ,
3. `!(!p || !q || !r)`,
4. `!(b>a && b<c)`,
5. `!(a>=b && b>=c && a >=c)`,
6. `(a>b && a<c) || (a<c && a>d)`,
7. `p || !p` .

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

Zadanie 2.5. [MD] Napisz w języku C++ funkcję `abs()`, która obliczy wartość bezwzględną danej liczby rzeczywistej.

Zadanie 2.6. [MD] Napisz (w pseudokodzie, w którym korzysta się z poznanych już operatorów) funkcję `pierwiastek()`, która będzie wyznaczać pierwiastek kwadratowy danej liczby rzeczywistej dodatniej x według podanego algorytmu. (Uwaga: zakładamy, że `epsilon` jest ustaloną, małą wartością, a nie parametrem funkcji).

```
przyblizenie=1
dopóki różnica dwóch ostatnich przybliżeń jest większa niż epsilon
    przyblizenie = srednia z przyblizenie oraz x/przyblizenie
zwróć przybliżenie
```

Prześledź działanie algorytmu np. dla `x=9` oraz `epsilon=1`.

Zadanie 2.7. [MD] Wyznacz typ i wartość następujących wyrażeń:

1. `1+0.5`,
2. `1.0==1`,
3. `5/2 - 2.5`,
4. `pierwiastek(4)==2`,
5. `pierwiastek(9)-int(3.5)`,
6. `6/2 + 7/2`,
7. `abs(pierwiastek(9)-3)<1`.

Reprezentacje liczb całkowitych ⋆

⋆ **Zadanie 2.8.** 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 2.9.** Dana jest n cyfrowa liczba w systemie U2 zapisana jako ciąg cyfr $b_{n-1}b_{n-2}\ldots\ldots b_1b_0$, gdzie $b_i \in \{0,1\}$ dla $i = 0, 1, \ldots, n-1$. Pokaż, że powielenie pierwszej cyfry dowolną liczbę razy nie zmienia wartości danej liczby, tzn. $b_{n-1}b_{n-2}\ldots b_1b_0 = b_{n-1}b_{n-1}\ldots\ldots b_{n-1}b_{n-2}\ldots b_1b_0$.

⋆ **Zadanie 2.10.** Przedstaw następujące liczby dane w notacji U2 jako liczby dziesiętne: 10111100, 00111001, 1000000111001011.

⋆ **Zadanie 2.11.** Przedstaw następujące liczby dziesiętne w notacji U2 (do zapisu użyj 8, 16 lub 32 bitów): $-12, 54, -128, -129, 53263, -32000, -56321, -3263411$.

⋆ **Zadanie 2.12.** Dana jest wartość $x \in \mathbb{N}$. Napisz wzór matematyczny, który opisze najmniejszą liczbę cyfr potrzebnych do zapisania wartości x w systemie a) binarnym, b) dziesiętnym, c) szesnastkowym.

⋆⋆ **Zadanie 2.13.** 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. $1_2 + 1_2 = 10_2$ (tzw. przeniesienie).

1. $1000_2 + 111_2$,
2. $1000_2 + 1111_2$,
3. $1110110_2 + 11100111_2$,
4. $11111_2 + 11111111_2$,
5. $1111_2 - 0001_2$,
6. $1000_2 - 0001_2$,
7. $10101010_2 - 11101_2$,
8. $11001101_2 - 10010111_2$.

⋆⋆ **Zadanie 2.14.** 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.

1. $1000_{\mathrm{U2}} + 0111_{\mathrm{U2}}$,
2. $10110110_{\mathrm{U2}} + 11100111_{\mathrm{U2}}$,
3. $10101010_{\mathrm{U2}} - 00011101_{\mathrm{U2}}$,
4. $11001101_{\mathrm{U2}} - 10010111_{\mathrm{U2}}$.
5. $10001101_{\mathrm{U2}} - 01010111_{\mathrm{U2}}$.

⋆⋆ **Zadanie 2.15.** Dana jest liczba w postaci U2. Pokaż, że aby uzyskać liczbę do niej przeciwną, należy odwrócić wartości jej bitów (dokonać ich negacji, tzn. zamienić zera na jedynki i odwrotnie) i dodać do wyniku wartość 1. Ile wynoszą wartości $0101_{\mathrm{U2}}, 1001_{\mathrm{U2}}$ i 0111_{U2} po dokonaniu tych operacji? Sprawdź uzyskane rezultaty za pomocą konwersji tych liczb do systemu dziesiętnego.

⋆⋆ **Zadanie 2.16.** [SK] Dana jest liczba w postaci binarnej. W jaki sposób za pomocą jednej operacji logicznej, jednej bitowej oraz jednej arytmetycznej sprawdzić, czy dana liczba jest potęgą liczby 2.

⋆ **Zadanie 2.17.** [MD] Napisz algorytm, który znajdzie binarną reprezentację dodatniej liczby naturalnej. Wskazówka: jako wynik wypisuj na ekran kolejne cyfry dwójkowe, od najmłodszej do najstarszej.

⋆ **Zadanie 2.18.** [MD] Dany jest ciąg n liczb o elementach ze zbioru $\{0,1\}$, który reprezentuje pewną liczbę w postaci U2. Wartości te będą wczytywane z klawiatury, w kolejności tym razem od najstarszego bitu do najmłodszego. Napisz algorytm, który wypisze wartość tej liczby w systemie dziesiętnym.

⋆ **Zadanie 2.19.** Jaki wynik dadzą poniższe operacje bitowe dla zmiennych typu `short`?

1. `0x0FCD | 0xFFFF`,
2. `364 & 0x323`,
3. `~163`,
4. `0xFC93 ^ 0x201D`,
5. `14 << 4`,
6. `0xf5a3 >> 8`.

2.7. Wskazówki i odpowiedzi do ćwiczeń

Odpowiedź do zadania 2.2.

1. `10.0+15.0/2+4.3 == 21.8` (`double`).
2. `10.0+15/2+4.3 == 21.3` (`double`).
3. `3.0*4/6+6 == 8.0` (`double`).
4. `20.0-2/6+3 == 23.0` (`double`).
5. `10+17*3+4 == 65` (`int`).
6. `10+17/3.0+4` $\simeq$ `19.6667` (`double`).
7. `3*4%6+6 == 6` (`int`).
8. `3.0*4%6+6` — błąd kompilacji.
9. `10+17%3+4 == 16` (`int`).

□

Odpowiedź do zadania 2.8.

1. $10_{10} = 1010_2 = A_{16}$.
2. $18_{10} = 10010_2 = 12_{16}$.
3. $18_{16} = 11000_2 = 24_{10}$.
4. $101_2 = 5_{10} = 5_{16}$.
5. $101_{10} = 1100101_2 = 65_{16}$.
6. $101_{16} = 100000001_2 = 257_{10}$.
7. $\mathrm{ABCDEF}_{16} = 101010111100110111101111_2 = 11259375_{10}$.
8. $64135312_{10} = 11111010010101000010010000_2 = 3D2A090_{16}$.
9. $110101111001_2 = D79_{16} = 3449_{10}$.
10. $\mathrm{FFFFF0C2}_{16} = 11111111111111111111000011000010_2 = 4294963394_{10}$.

□

Odpowiedź do zadania 2.10.

1. $10111100_{\mathrm{U2}} = -68$.
2. $00111001_{\mathrm{U2}} = 57_{10}$.
3. $1000000111001011_{\mathrm{U2}} = -32309$.

□

Odpowiedź do zadania 2.11.

1. $-12_{10} = 11110100_{\mathrm{U2}}$.
2. $54_{10} = 00110110_{\mathrm{U2}}$.
3. $-128_{10} = 10000000_{\mathrm{U2}}$.
4. $-129_{10} = 1111111101111111_{\mathrm{U2}}$.
5. $53263_{10} = 00000000000000001101000000001111_{\mathrm{U2}}$.
6. $-32000_{10} = 11111111111111111000001100000000_{\mathrm{U2}}$.
7. $-56321_{10} = 11111111111111110010001111111111_{\mathrm{U2}}$.
8. $-3263411_{10} = 11111111100111000110100010011101_{\mathrm{U2}}$.

□

Odpowiedź do zadania 2.13.

1. $1000_2 + 111_2 = 1111_2$.
2. $1000_2 + 1111_2 = 10111_2$.
3. $1110110_2 + 11100111_2 = 101011101_2$, 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

4. $11111_2 + 11111111_2 = 100011110_2$.
5. $1111_2 - 0001_2 = 1110_2$.
6. $1000_2 - 0001_2 = 111_2$.
7. $10101010_2 - 11101_2 = 10001101_2$, 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	

8. $11001101_2 - 10010111_2 = 110110_2$, 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 2.14.

1. $1000_{U2} + 0111_{U2} = 1111_{U2} = -1_{10} = -8_{10} + 7_{10}$.
2. $10110110_{U2}+11100111_{U2} = 110011101_{U2} = 10011101_{U2} = -99_{10} = -74_{10}+(-25)_{10}$.
3. $10101010_{U2} - 00011101_{U2} = 10001101_{U2} = -115_{10} = -86_{10} - 29_{10}$.
4. $11001101_{U2} - 10010111_{U2} = 00110110_{U2} = 54_{10} = -51_{10} - (-105_{10})$.
5. $10001101_{U2} - 01010111_{U2} = 00110110_{U2} = 54_{10} \neq -115_{10} - 87_{10} = -202_{10}$! Wystąpiło tzw. *przepełnienie*, czyli przekroczenie zakresu wartości dla liczby 8-bitowej (ale mamy $2^8 - 202_{10} = 54_{10}$). Można zapisać operandy na większej liczbie bitów, np. szesnastu:

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

□

Wskazówka do zadania 2.16.

```
1 niech x ∈ ℕ;
2 zwróć (x & (x − 1)) == 0 jako wynik;
```

Dlaczego tak jest? Wykaż. □

Odpowiedź do zadania 2.18.

```
// Wejście: n > 0 oraz x[0], x[1], ..., x[n − 1] ∈ {0, 1}
// Wyjście: wartość x w systemie dziesiętnym
niech ret, pot ∈ ℕ;

ret = 0;
pot = 1;
dla (i=1,2,...,n−1)
{
  ret = ret + pot*x[n−i];
  pot = pot * 2;
}
ret = ret − pot * x[0];
zwróć ret jako wynik;
```

□

Odpowiedź do zadania 2.19.

1. `0x0FCD | 0xFFFF == −1 (0xFFFF)`.
2. `364 & 0x323 == 288 (0x0120)`.
3. `~163 == −164 (0xFF5C)` (dlaczego -164?).
4. `0xFC93 ^ 0x201D == −9074 (0xDC8E)`.
5. `14 << 4 == 224 (0x00E0)`.
6. `0xf5a3 >> 8 == 0xFFF5` (bit znaku == 1).

□

Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

3. Instrukcja warunkowa i pętle. Struktury. Funkcje – podstawy

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

3.1. Instrukcje sterujące

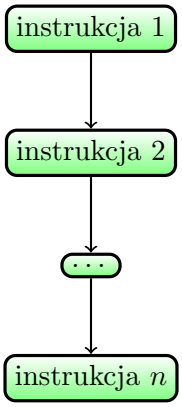
3.1.1. Bezpośrednie następstwo instrukcji

Znamy już następujące typy instrukcji:

– instrukcje deklarujące zmienne i stałe lokalne,
– instrukcję przypisania.

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

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



Rys. 3.1. Schemat blokowy bezpośredniego następstwa instrukcji

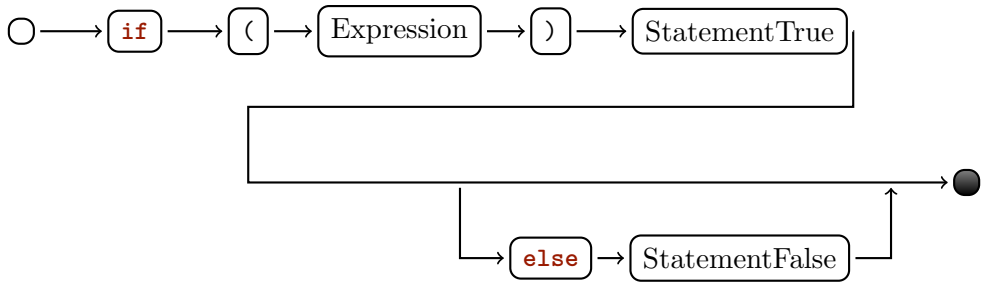
W kolejnych podrozdziałach poznamy konstrukcje, które pozwolą nam sterować przebiegiem programu. Dzięki nim będziemy mogli dostosowywać działanie programu do różnych szczególnych przypadków — w zależności od wartości przetwarzanych danych.

3.1.2. Instrukcja warunkowa if

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

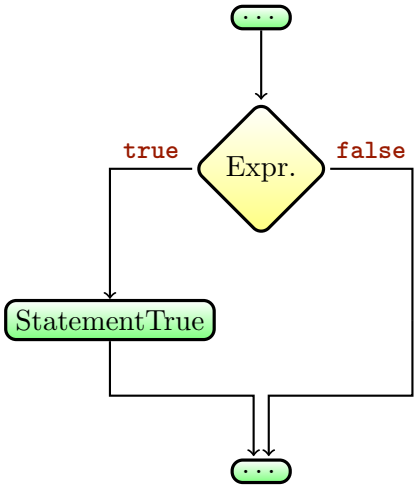
W języku C++ dokonywanie tego typu wyborów umożliwia *instrukcja warunkowa* `if` (od ang. *jeśli*). Wykonuje ona pewien fragment kodu *wtedy i tylko wtedy*, gdy pewien dany warunek logiczny jest spełniony. Może też, opcjonalnie, zadziałać w inny sposób w przeciwnym przypadku (`else`).

Jej składnię przedstawia poniższy diagram.



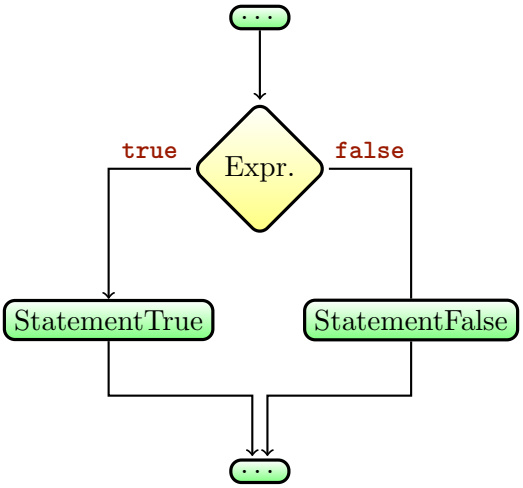
Expression jest wyrażeniem (koniecznie ujętym w nawiasy!) sprowadzalnym do typu `bool`, reprezentującym sprawdzany warunek logiczny. StatementTrue jest instrukcją wykonywaną wtedy i tylko wtedy, gdy wyrażenie Expression ma wartość `true`, Opcjonalne StatementFalse zaś — w przeciwnym przypadku.

Schemat blokowy instrukcji `if` bez części `else` przedstawia rys. 3.2.



Rys. 3.2. Schemat blokowy instrukcji warunkowej `if`

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



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

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;
```

Zapamiętaj

Sam średnik (;) oznacza instrukcję pustą (która nie robi nic). Dlatego następujący fragment kodu:

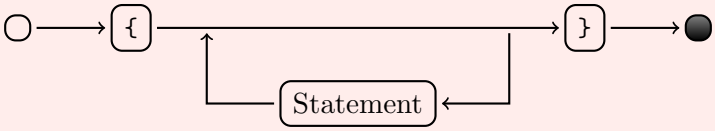
```
int x;
cin >> x;
if (x < 0);      // jeśli x < 0, nie rób nic specjalnego teraz
   cout << x;    // zawsze wypisuj x
```

wypisze wartość `x` na ekran *zawsze*.

Zapamiętaj

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

Składnia instrukcji złożonej przedstawiona jest na poniższym diagramie.



Zauważmy, że po nawiasie zamykającym blok nie ma potrzeby stawiania średnika.

Informacja

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

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

Informacja

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

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

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

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

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

nawiasami klamrowymi.

Warto pamiętać, że jako StatementFalse może się pojawić kolejna instrukcja `if... else`. Dzięki temu można obsłużyć w danym fragmencie programu więcej niż 2 rozłączne przypadki rozpatrywanego problemu na raz.

```
int x;
cout << "Określ swój nastrój w skali 1-3:";
cin >> x;

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

Na marginesie, czasem przydać nam się może instrukcja `switch`. I tak:

```
int x = ...;
switch(x) { // tylko typ calkowitoliczbowy
    case 1:
        // ... gdy x == 1 ...
        break;
    case 2:
    case 3:
        // ... gdy x == 2 || x == 3 ...
        break;
    default:
        // ... w przeciwnych przypadkach ...
        break;
}
```

odpowiada:

```
int x = ...;
if (x == 1) {
        // ... gdy x == 1 ...
}
else if (x == 2 || x == 3) {
    // ... gdy x == 2 || x == 3 ...
}
else {
    // ... w przeciwnych przypadkach ...
}
```

Jak widzimy, `switch` właściwie nie wprowadza niczego nowego do arsenału naszych możliwości. W niektórych przypadkach jednak może prowadzić do bardziej czytelnego kodu.

3.1.3. Pętle

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

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

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

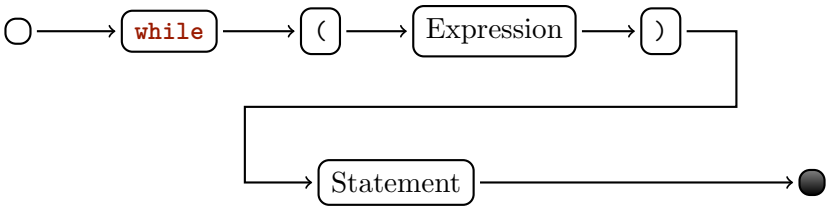
W języku C++ zdefiniowane zostały trzy instrukcje realizujące tego typu ideę.

- `while`,
- `for` oraz
- `do ... while`.

Przyjrzyjmy się im dokładniej.

3.1.3.1. Pętla while

Koncepcyjnie najprostszą konstrukcją realizującą pętlę jest instrukcja `while`. Jej składnię przedstawia poniższy diagram:

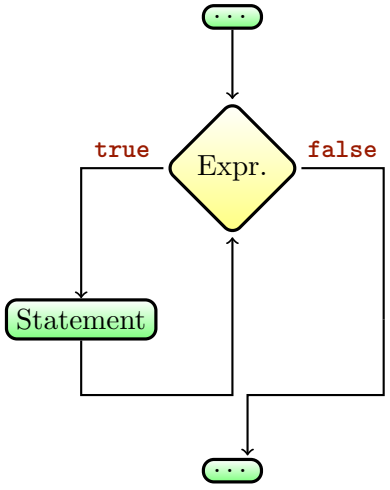


Expression jest wyrażeniem sprowadzalnym do typu `bool`. Najlepiej będzie, gdy istnieje możliwość jego zmiany na skutek wykonywania podanej w części Statement instrukcji. W przeciwnym wypadku stworzymy program, który nigdy się nie zatrzyma.

Schemat blokowy przedstawionej pętli zobrazowany jest na rys. 3.4.

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

```
int i = 1;
int suma = 0;
while (i <= 100) { // ten warunek 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;
```



Rys. 3.4. Schemat blokowy pętli `while`

Zapamiętaj

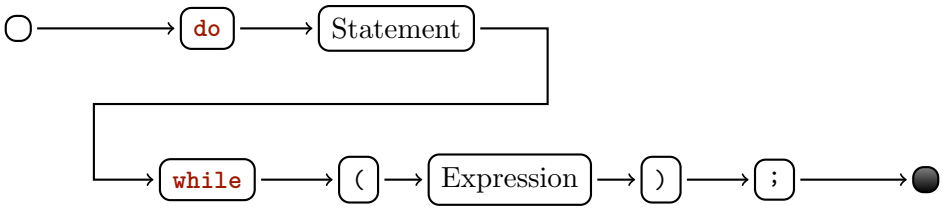
Powtórzmy, sam średnik (;) oznacza instrukcję pustą, dlatego pętla

```
while (true) ;
```

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

3.1.3.2. Pętla do...while

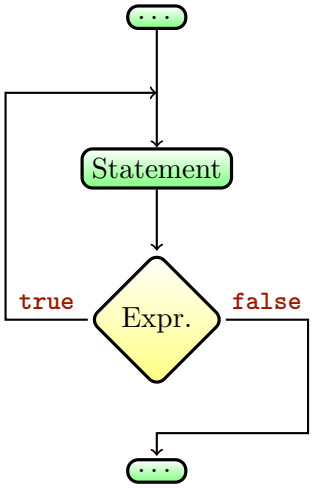
Inną odmianą pętli jest instrukcja `do ... while`, określona według składni:



Jak widać na schemacie blokowym na rys. 3.5, używamy jej, gdy chcemy zapewnić, by instrukcja Statement została wykonana *co najmniej* jeden raz.
Zapiszmy dla ćwiczenia powyższy przykład za pomocą pętli `do ... while`, urozmaicając go jednak nieco.

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

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



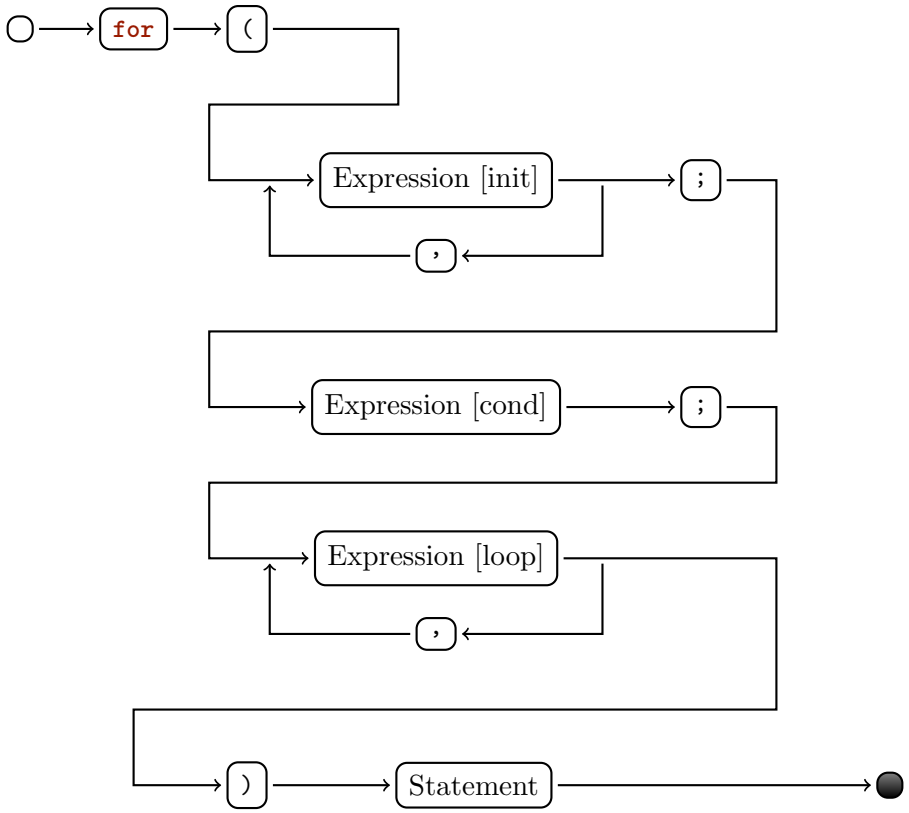
Rys. 3.5. Schemat blokowy pętli `do... while`

3.1.3.3. Pętla for

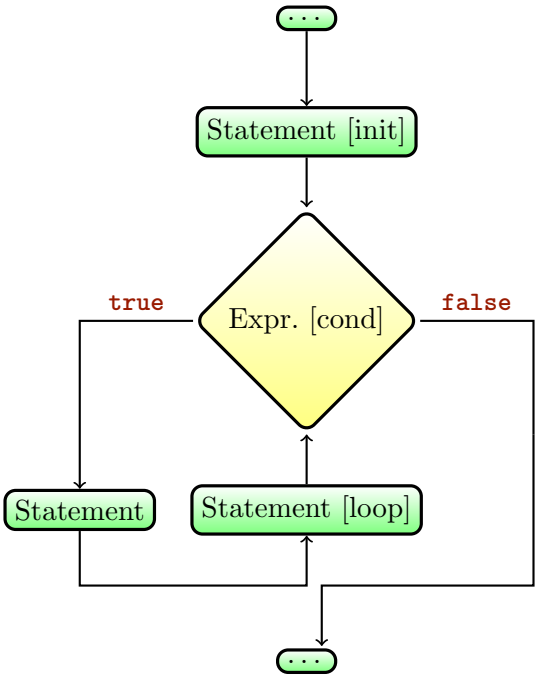
Dość często zachodzi potrzeba ustalenia przebiegu programu tak jak na schemacie blokowym przedstawionym na rys. 3.6.

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

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



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



Rys. 3.6. Schemat blokowy pętli `for`

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

```
int suma = 0;
for (int i=1; i<=100; ++i) {
    cout << i << endl;
    suma += i;
}
cout << "Suma=" << suma << endl;
```

Informacja

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

Zatem powyższy przykład można zapisać i tak:

```
int suma = 0;
for (int i=1; i<=100; suma += i, ++i) // albo suma += (i++)
    cout << i << endl;
cout << "Suma=" << suma << endl;
```

3.1.3.4. break i continue

Niekiedy zachodzi konieczność zmiany domyślnego przebiegu wykonywania pętli.

Informacja

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

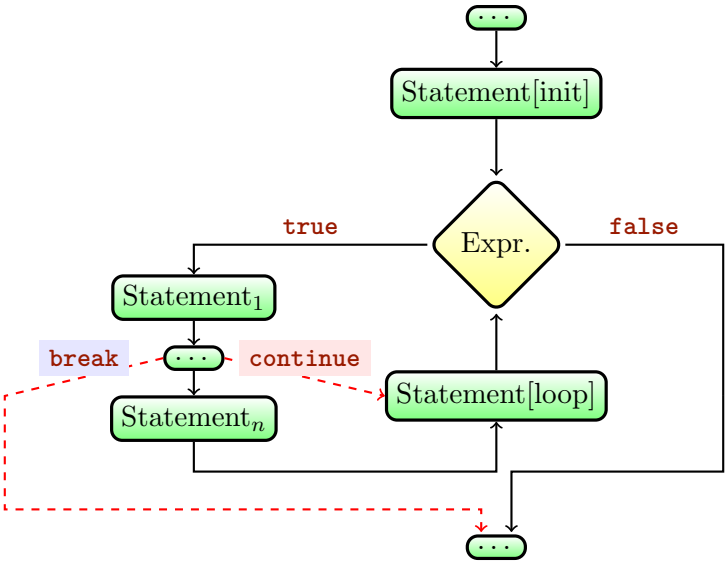
Z kolei instrukcja `continue` służy do przejścia do kolejnej iteracji pętli (ignorowane są instrukcje następujące po `continue`). W przypadku pętli `for` instrukcja aktualizująca jest jednak wykonywana.

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

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

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

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



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

3.2. Struktury, czyli typy złożone

W poprzednim rozdziale poznaliśmy tak zwane skalarne typy podstawowe, w tym m.in. typ całkowity `int`, zmiennoprzecinkowy `double` oraz logiczny `bool`.

W języku C++ możemy tworzyć własne *typy złożone* będące reprezentacją *iloczynu kartezjańskiego* różnych innych zbiorów (typów). Są to tzw. *struktury*.

Struktury definiujemy w następujący sposób.

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

Zapamiętaj

Zauważmy, że po nawiasie klamrowym w definicji struktury występuje średnik. Jego pominięcie jest częstą przyczyną występowania różnych (dziwnych na pierwszy rzut oka) błędów kompilacji.

Zdefiniowanie struktury powoduje utworzenie nowego typu. *Zmienne* typu złożonego deklarujemy w standardowy sposób, czyli np.

```
nazwaStruktury identyfikatorZmiennej;
```

Do poszczególnych pól (składników) struktury możemy odwołać się za pomocą *kropki*, np. `identyfikatorZmiennej.nazwaPolaX`. Pola zmiennej typu złożonego traktujemy jak zwykłe zmienne odpowiednich typów. Intuicyjnie, zmienną typu złożonego można wyobrażać sobie jako swego rodzaju „paczkę”, która wewnątrz skrywa różne ciekawe „drobiazgi”.

Dla przykładu rozważmy definicję struktury reprezentującej punkt w $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$. Łatwo się domyślić, że taka „paczka” powinna składać się z dwóch pól typu `double`.

```
struct Punkt
{
    double x;
    double y;
};
```

Oto przykładowe użycie tak zdefiniowanej struktury:

```
1  int main() {
2      Punkt p;          // tzn. niech p∈ℝ×ℝ
3      p.x = 0.0;
4      p.y = 0.5;
5      // wypisz na ekran:
6      cout << "(" << p.x << ", " << p.y << ")" << endl;
7
8      Punkt r;
9      cin >> r.x;
10     cin >> r.y;
11     cout << "(" << p.x+r.x << ", " << p.y+r.y << ")" << endl;
12
13     return 0;
14 }
```

Zapamiętaj

Zwróćmy uwagę, że instrukcja

```
cout << p;          // :-(
```

spowoduje wystąpienie błędu kompilacji. Komputer bowiem nie wie, jak miałby wykonać polecenie „wypisz coś złożonego” — musiałby zgadnąć, czy chodzi nam np. o wypisanie najpierw pola `x`, czy może pola `y`, czy chcemy je oddzielić spacją, a może znakiem nowej linii, a może... itp.

3.3. Funkcje — informacje podstawowe

3.3.1. Funkcje w matematyce

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

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

Przyjrzyjmy się dwom przykładom.

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

Przykład 1. Niech

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

będzie funkcją taką, że

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

Powyższa funkcja oblicza po prostu kwadrat danej liczby rzeczywistej. Zauważmy, że definicja każdej funkcji składa się z dwóch części. Pierwsza, „niech...", to tzw. *deklarator* określający, że od tej pory identyfikator kwadrat oznacza funkcję o danej dziedzinie i danej przeciwdziedzinie. Druga, „taka, że", to tzw. *definicja właściwa*, która prezentuje abstrakcyjny *algorytm*, przepis, za pomocą którego dla konkretnego elementu dziedziny (powyżej ten element oznaczyliśmy symbolem x, choć równie dobrze mogłoby to być t, ζ, bądź $\widehat{\alpha}^*$) możemy uzyskać wartość wynikową, tj. $\mathsf{kwadrat}(x)$. ■

Przykład 2. Niech

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

będzie funkcją taką, że

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

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

W niniejszym rozdziale przyjrzymy się sposobom definicji funkcji w języku C++ oraz różnym przykładom ich użycia. Dowiemy się, czym różnią się funkcje w C++ od ich matematycznych odpowiedników. Poznamy także bogatą bibliotekę gotowych do użycia funkcji wbudowanych, które możemy w każdej chwili wykorzystać do własnych potrzeb.

3.3.2. Definiowanie funkcji w języku C++

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

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

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

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

```
1 int main() {
2     ustawFotel();
3     ustawLusterka();
4     zapnijPasy();
5     przekręćKluczyk();
```

```
6     sprawdźKontrolki();
7     uruchomRozrusznik();
8     return 0;
9  }
```

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

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

```
1  int main() {
2     // ...
3     do {
4        if (dzieńTygodnia == niedziela)
5           continue;
6
7        pouczSięTrochę();
8
9        if (zmęczony) {
10          if (godzinNaukiDziś > 5)
11             możeszWreszcieIśćNaRandkę();
12          else
13             odpocznijChwilę();
14       }
15    } while (!nauczony);
16    // ...
17 }
```

■

Najprostszym sposobem podziału programu w języku C++ na (pod)moduły jest użycie funkcji[2].

Zapamiętaj

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

Informacja

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

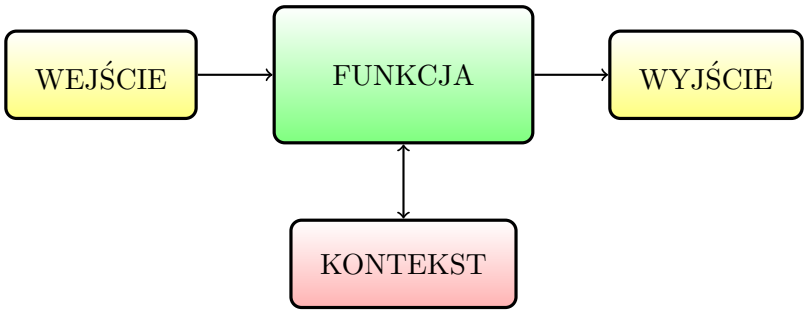
Dodatkowo, dzięki funkcjom uzyskujemy m.in. możliwość wykorzystania tego samego kodu wielokrotnie.

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

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

Zapamiętaj

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



Rys. 3.8. Schemat przepływu danych w funkcjach

Powyższa idea została zobrazowana na rys. 3.8.

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

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

2. *Dziedzina i przeciwdziedzina funkcji matematycznej nie może być zbiorem pustym*. W języku C++ został określony specjalny typ `void` ($\emptyset$, od ang. *pusty*), który pozwala na określenie funkcji „robiącej coś z niczego" (korzystającej tylko z kontekstu, np. danych pobranych z klawiatury), „niby-nic z czegoś" (tylko np. zmieniającej kontekst), bądź nawet „niby-nic z niczego".

3. *Przeciwdziedziną funkcji w języku C++ może być tylko jeden typ*, choć może to być typ złożony (struktura)[3].

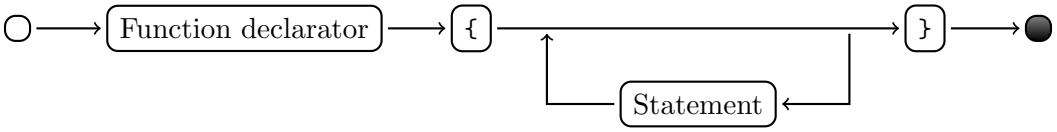
Ciekawostka

De facto więc, funkcje w języku programowania mogłyby być postrzegane jako funkcje matematyczne, jeśliby ich dziedziną było zawsze $X \times \mathbb{K}$, a przeciwdziedziną $Y \times \mathbb{K}$, gdzie $\mathbb{K}$ oznacza

[3] W pewnym sensie da się ominąć to ograniczenie korzystając z argumentów przekazanych przez referencję (o czym w innym rozdziale).

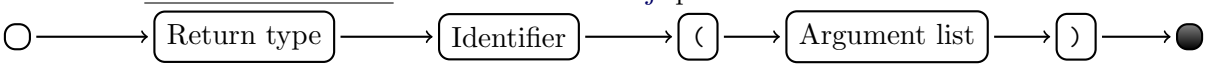
kontekst działania komputera (bardzo trudny w opisie, na niego bowiem składają się wszystkie urządzenia wejściowe i wyjściowe komputera, w tym urządzenia wideo, audio, sieciowe, pamięć masowa itp.).

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

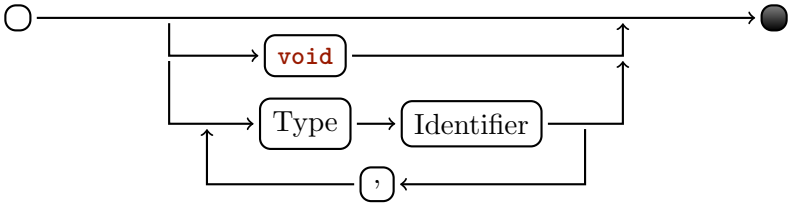


Blok instrukcji zawarty między nawiasami klamrowymi to tzw. *ciało funkcji* (definicja właściwa) — tutaj zawarta jest implementacja algorytmu realizowanego przez funkcję.

Z kolei Function declarator to *deklarator funkcji* postaci:



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



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

Informacja

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

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

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

Zapamiętaj

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

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

Przykład 5. Najprostszy, pełny kod źródłowy działającego programu w języku C++ można zapisać w sposób następujący.

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

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

Ciekawostka

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

Informacja

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

3.3.3. Wywołanie funkcji

Rozpatrzmy jeszcze raz definicję funkcji kwadrat.

Przykład 1 (cd.). Niech

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

będzie funkcją taką, że

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

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
```

```
12 {
13     double y = 0.5;
14     cout << kwadrat(y)    << endl; // wywołanie funkcji
15     cout << kwadrat(2.0) << endl; // można i tak
16     return 0;
17 }
```

Zapamiętaj

Zauważmy, że definicja funkcji `kwadrat()` została zamieszczona *przed* definicją funkcji `main()`, która z `kwadrat()` korzysta.

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

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

Zapamiętaj

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

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

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

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

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

Co więcej, należy pamiętać, że instrukcja

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

zostanie wykonana w sposób podobny do następującego:

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

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

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

3.3.4. Zasięg zmiennych

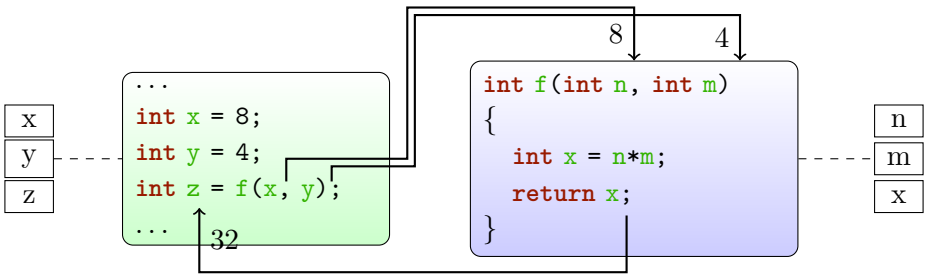
Przyjrzyjmy się *zasięgowi* definiowanych przez nas zmiennych. Zasięg określa, jak długo dany obiekt istnieje i w jaki sposób dane zmienne są widoczne z innych miejsc programu.

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

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

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

Przyjrzyjmy się rys. 3.9. 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 jak *czarną skrzynkę*, gdyż to, jakie procesy wewnątrz niej zachodzą, nie ma bezpośredniego wpływu na obiekt, który posiłkuje się `f()` do osiągnięcia swoich celów.



Rys. 3.9. Zasięg zmiennych

Spójrzmy na blok kodu po lewej stronie. Zmienne `x` i `y` są przekazywane do `f()` *przez wartość*. Znaczy to, że wartości parametrów są obliczane przed wywołaniem funkcji (w tym przypadku pobierane są po prostu wartości przechowywane w tych zmiennych), a wyniki tych operacji są *kopiowane* do argumentów wejściowych. Widoczne są one w `f()` jako zmienne lokalne, odpowiednio, `n` i `m`.

Z wartością zwracaną za pomocą instrukcji `return` (prawa strona rysunku) kompilator postępuje w sposób analogiczny, tzn. nie przekazuje obiektu `x` jako takiego, lecz wartość, którą on przechowuje. Jeśli w tym miejscu stałoby np. złożone wyrażenie arytmetyczne albo stała, reguła ta byłaby oczywiście zachowana.

Zapamiętaj

Powtórzmy, zmienne przekazywane *przez wartość* są *kopiowane*.

Co więcej, zmienne lokalne nie są przypisane na stałe samym funkcjom, tylko ich wywołaniom. Obiekty utworzone dla jednego wywołania funkcji są *niezależne* od zmiennych dla innych wywołań. Jest to bardzo istotne w przypadku techniki zwanej rekurencją, która polega na tym, że funkcja wywołuje samą siebie (zob. kolejne rozdziały). Innymi słowy, wywołując tę samą funkcję `f()` raz po raz, za każdym razem tworzony jest nowy, niezależny zestaw zmiennych lokalnych.

Ciekawostka

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

3.3.5. Przekazywanie parametrów przez referencję

Wiemy już, że standardowo parametry wejściowe funkcji zachowują się jak zmienne lokalne — ich zmiana nie jest odzwierciedlona „na zewnątrz".

Przykład 6. Rozważmy następujący przykład — zamianę (ang. *swap*) dwóch zmiennych.

```
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 wartość
                  // (skopiowanie wartości)
    cout << n << ", " << m << endl;
    return 0;
}
```

Wynikiem działania tego fragmentu programu jest, rzecz jasna, napis `1, 2` na ekranie. Funkcja `zamien()` z punktu widzenia `main()` nie robi zupełnie nic. ■

W pewnych szczególnych przypadkach uzasadnione jest zatem przekazywanie parametrów w inny sposób — *przez referencję* (odniesienie). Dokonuje się tego poprzez dodanie znaku `&` bezpośrednio po nazwie typu zmiennej na liście parametrów funkcji.

Przekazanie parametrów przez referencję umożliwia nadanie bezpośredniego dostępu (również do zapisu) do zmiennych lokalnych funkcji wywołującej (być może pod inną nazwą). Obiekty te *nie są kopiowane*; udostępniane są takie, jakie są.

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

Przykład 6 (cd.). Tym samym dopiero teraz możemy przedstawić prawidłową wersję funkcji `zamien()`.

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

int main() {
    int n = 1, m = 2;
    zamien(n, m); // przekazanie 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 wreszcie oczekiwany wynik: `2, 1`. ■

Istnieje jeszcze jedno ważne zastosowanie zmiennych przekazywanych przez referencję. Może ono służyć do obejścia ograniczenia związanego z tym, że funkcja za pomocą instrukcji `return` może zwrócić co najwyżej wartość jednego, ustalonego typu.

Przykład 7. 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
                           do przekazania wyniku */
    iloreszta(n, m, i, r);
    cout << n << "=" << i << "*" << m << "+" << r;
    return 0;
}
```

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

3.3.6. Argumenty domyślne funkcji

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

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

Oto kilka przykładów prawidłowych deklaracji funkcji:

- `void f(int x=3);`
- `void f(int x=3, int y=2, int z=5);`
- `void f(int x, int y=3, int z=2);`
- `void f(int x, int y, int z=2);`
- `void f(int& x, int y=2);`

A oto nieprawidłowe deklaracje funkcji:

- `void f(int x, int y=3, int z);`
- `void f(int x=3, int y);`
- `void f(int x, int& y=2);`

Przykład 8. Dla ilustracji rozważmy funkcję wyznaczającą pierwiastek liczby rzeczywistej, domyślnie o podstawie 2.

```
#include <cmath>

double pierwiastek(double x, double p=2)
{ // pierwiastek, domyślnie kwadratowy
    assert(p >= 1);
    return pow(x, 1.0/p);
}
```

```
8
9  int main(void)
10 {
11     cout << pierwiastek(10) << endl;       // 3.162278
12     cout << pierwiastek(10, 2.0) << endl; // 3.162278
13     cout << pierwiastek(10, 3.0) << endl; // 2.154435
14     return 0;
15 }
```

■

3.3.7. Przeciążanie funkcji

W języku C++ można nadawać te same nazwy (identyfikatory) wielu funkcjom pod warunkiem, że różnią się one co najmniej typem lub liczbą parametrów. Jest to tzw. *przeciążanie* funkcji (ang. *function overloading*). Funkcje takie mogą mieć różne przeciwdziedziny — nie jest to jednak warunek dostateczny pozwalający na rozróżnienie funkcji przeciążonych.

Przeciążanie ma sens, gdy funkcje wykonują podobne (w sensie funkcjonalności) czynności, jednakże na danych różnego typu.

Przykład poprawnych deklaracji:

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

Przykłady niepoprawnych deklaracji:

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

oraz:

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

3.4. Przegląd funkcji z biblioteki języka C

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

3.4.1. Funkcje matematyczne

Biblioteka `<cmath>` udostępnia wybrane funkcje matematyczne[4]. Ich przegląd zamieszczamy w tab. 3.1, 3.2 i 3.3.

Dla przypomnienia, funkcja „sufit” określona jest jako

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

a funkcja „podłoga” zaś jako

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

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

Tab. 3.1. 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.)

Tab. 3.2. 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

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

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

będzie funkcją taką, że

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

Oto przykładowy program wyznaczający wartość tej funkcji w różnych punktach.

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

double sindod(double);

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


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

■

Tab. 3.3. Funkcje dodatkowe w bibliotece <cmath>

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

3.4.2. Liczby pseudolosowe

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

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

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

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

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

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

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

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

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

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

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

Przykład 9. Korzystając z własnoręcznie napisanej funkcji generującej liczbę rzeczywistą z przedziału $[0, 1)$ łatwo napisać funkcję „losującą” liczbę ze zbioru $\{a, a+1, \ldots, b\}$, gdzie $a, b \in \mathbb{Z}$.

Oto przykładowy program.

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

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

```

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

```
12 int losAB(int a, int b)
13 {
14     double ab = los01()*(b-a+1)+a; // liczba rzeczywista
15                                    // z przedziału [a,b+1)
16     return (int)(floor(ab));       // ,,podłoga'' z ab
17 }
18
19 int main()
20 {
21     srand(time(0)); // zainicjuj generator
22     for (int i=0; i<5; i++) // 5 razy...
23        cout << losAB(1,10) << " ";
24     cout << endl;
25     return 0;
26 }
```

Przykładowy wynik na ekranie:
`4 3 8 2 1`
a po chwili:
`4 6 10 4 2`

■

Informacja

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

3.4.3. Asercje

Biblioteka <cassert> udostępnia funkcję o następującej deklaracji:

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

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

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

Przykład 10. Dla przykładu rozpatrzmy „bezpieczną” funkcję wyznaczającą pierwiastek z nieujemnej liczby rzeczywistej.

```
1  #include <cassert>
2  #include <cmath>
3
4  double pierwiastek(double x)
5  {
6      // Dane: x ⩾ 0
7      // Wynik: √x
8
9      assert(x>=0); // jeśli nie, to błąd – zakończenie programu
10     return sqrt(x);
11 }
```

■

Ciekawostka

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

```
#define NDEBUG
```

umieszczonej na początku pliku źródłowego bądź nagłówkowego.

3.5. Ćwiczenia

Zadanie 3.1. Wyraź następujące instrukcje dane w pseudokodzie za pomocą instrukcji języka C++ korzystających z pętli `for`.

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

Zadanie 3.2. Napisz kod w języku C++, który spośród liczb $1, 2, \ldots, 100$ wypisze:

1. wszystkie podzielne przez 7, tzn. $7, 14, 21, \ldots$,
2. wszystkie podzielne przez 2 lecz niepodzielne przez 5, tzn. $2, 4, 6, 8, 12, \ldots$,
3. co drugą podzielną przez 5 lub podzielną przez 7, tzn. $5, 10, 15, 21, 28, \ldots$

Zadanie 3.3. Napisz funkcję `max()`, która zwraca maksimum danych $a, b, c \in \mathbb{N}$.

Zadanie 3.4. Napisz funkcję `med()`, która znajduje medianę (wartość środkową) trzech liczb rzeczywistych, np. $\texttt{med}(4, 2, 7) = 4$ i $\texttt{med}(1, 2, 3) = 2$.

Zadanie 3.5. [MD] Napisz funkcję wyznaczającą zadaną (całkowitą) potęgę danej liczby rzeczywistej. Przykład wywołania: `potega(0.5,2)` zwróci 0.25

Zadanie 3.6. Napisz funkcję, która dla danego $n \in \mathbb{N}$ oblicza $\max\{k \in \mathbb{N}_0 : 2^k \leqslant n\}$.

Zadanie 3.7. Napisz funkcję, która zwraca przybliżenie wartości liczby π za pomocą wzoru $\pi \simeq 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right)$ na podstawie n (parametr funkcji) pierwszych elementów tego szeregu liczbowego.

Zadanie 3.8. Napisz funkcję, która zwraca przybliżenie wartości liczby e za pomocą wzoru $e \simeq 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$, gdzie $k! = 1 \times 2 \times \cdots \times k$. Sumuj kolejne wyrazy tego szeregu, dopóki różnica pomiędzy kolejnymi składnikami będzie mniejsza niż `eps` (parametr funkcji, wartość domyślna: 10^{-9}).

Zadanie 3.9. Napisz funkcje, które posłużą do wyznaczenia wartości następujących wyrażeń.

1. 2^n dla danego $n \in \mathbb{N}$,
2. $\sum_{i=1}^{n} i$ dla danego $n \in \mathbb{N}$,
3. $\sum_{i=1}^{n} \frac{1}{i!}$ dla danego $n \in \mathbb{N}$,
4. $\prod_{i=1}^{n} \frac{i}{i+1}$ dla danego $n \in \mathbb{N}$,
5. $e^x \simeq \sum_{n=0}^{m} \frac{x^n}{n!}$ dla danego $x \in \mathbb{R}$ i $m \in \mathbb{N}$,
6. $\ln(1+x) \simeq \sum_{n=1}^{m} \frac{(-1)^{n+1}}{n} x^n$ dla danego $x \in [-1, 1]$ i $m \in \mathbb{N}$,
7. $\sin x \simeq \sum_{n=0}^{m} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$ dla danego $x \in \mathbb{R}$ i $m \in \mathbb{N}$,
8. $\cos x \simeq \sum_{n=0}^{m} \frac{(-1)^n}{(2n)!} x^{2n}$ dla danego $x \in \mathbb{R}$ i $m \in \mathbb{N}$,
9. $\arcsin x \simeq \sum_{n=0}^{m} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1}$ dla danego $x \in (-1, 1)$ i $m \in \mathbb{N}$.

Zadanie 3.10. Napisz funkcję `odle()`, która przyjmuje jako parametr współrzędne rzeczywiste dwóch punktów w $\mathbb{R}^2$ (struktura `Punkt` składająca się z dwóch wartości typu `double`) $\mathbf{p}, \mathbf{r}$ i zwraca ich odległość euklidesową równą $\sqrt{(p_x - r_x)^2 + (p_y - r_y)^2}$.

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

Zadanie 3.11. Napisz funkcję `odlsup()`, która przyjmuje jako parametr współrzędne rzeczywiste dwóch punktów w $\mathbb{R}^2$ $\mathbf{p}, \mathbf{r}$ i zwraca ich odległość w metryce supremum: $\max\{|p_x - r_x|, |p_y - r_y|\}$.

Zadanie 3.12. Dane trzy punkty w $\mathbb{R}^2$: $\mathbf{a} = (a_x, a_y)$, $\mathbf{b} = (b_x, b_y)$, $\mathbf{c} = (c_x, c_y)$. Napisz funkcję, która wyznaczy kwadrat promienia okręgu przechodzącego przez $\mathbf{a}, \mathbf{b}$ i $\mathbf{c}$ (3 parametry będące strukturami o nazwie `Punkt`), określony wzorem

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

gdzie $|\mathbf{a}-\mathbf{b}| = \sqrt{(a_x-b_x)^2 + (a_y-b_y)^2}$ oraz $\mathbf{a} \times \mathbf{b} = a_x b_y - a_y b_x$.

Zadanie 3.13. Napisz funkcję, która dla danych $a, b, c, d, e, f \in \mathbb{R}$ rozwiąże układ dwóch równań liniowych względem niewiadomych $x, y \in \mathbb{R}$:

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

Wynik zwróć w postaci struktury o dwóch polach typu `double`. Jeśli układ jest sprzeczny bądź nieoznaczony, ustaw wartość pól na `NaN` (funkcja `nan()` w bibliotece `<cmath>`).

⋆ **Zadanie 3.14.** Dane są liczby rzeczywiste $x_1, \ldots, x_4, y_1, \ldots, y_4 \in \mathbb{R}$ (dwie czteroelementowe struktury). Napisz funkcję, która sprawdzi, korzystając z jak najmniejszej liczby warunków logicznych, czy prostokąty $[x_1, x_2] \times [y_1, y_2]$ oraz $[x_3, x_4] \times [y_3, y_4]$ mają część wspólną (tj. przecinają się).

Zadanie 3.15. Napisz funkcję implementującą algorytm Euklidesa do wyznaczania największego wspólnego dzielnika dla danych dwóch liczb całkowitych nieujemnych a, b. Poprawnie identyfikuj wszystkie możliwe przypadki danych wejściowych, w tym m.in. $a < b$, $b < a$, $a < 0$, $b < 0$.

Zadanie 3.16. Napisz funkcję, która sprawdza, czy dana liczba naturalna jest liczbą pierwszą (`true`), czy też liczbą złożoną (`false`).

Zadanie 3.17. [MD] Napisz funkcję wyznaczającą liczbę dzielników zadanej liczby naturalnej. Przykład wywołania: `ile_dzielnikow(10)` zwróci wartość 4.

Zadanie 3.18. [MD] Napisz funkcję, która wyznaczy liczbę liczb pierwszych w zadanym zbiorze $\{a, a+1, \ldots, b\}$, gdzie $a < b$ — parametry funkcji. Użyj funkcji napisanej w zad. 3.17. Przykład wywołania: `ile_pierwszych(1,5)` zwróci wartość 3.

Zadanie 3.19. Korzystając ze wzoru na przybliżoną wartość funkcji sin podanego w zad. 3.9, utwórz program, który wydrukuje tablice przybliżonych wartości $\sin x$ dla $x = \frac{k}{n}\pi$, $k = 0, 1, \ldots, n$ i pewnego n, np. $n = 10$. Wynik niech będzie postaci podobnej do poniższej.

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

Zadanie 3.20. [MŚN] Napisz funkcję, która przyjmuje jako parametr liczbę całkowitą i zwraca liczbę powstałą z zadanej liczby poprzez odwrócenie kolejności jej cyfr dziesiętnych. Zakładamy, że liczba jest maksymalnie siedmiocyfrowa. Na przykład dla 1475 wynikiem powinno być 5741.

⋆ **Zadanie 3.21.** [Euler#1] Napisz funkcję, która wyznaczy sumę wszystkich wielokrotności liczb 3 i 5 ze zbioru $\{1, 2, \ldots, u\}$, gdzie $u \in \mathbb{N}$ jest parametrem funkcji. Dla przykładu, dla $u = 10$ wielokrotności to 3,5,6,9, a ich suma jest równa 23.

⋆ **Zadanie 3.22.** [Euler#25] Wyrazy ciągu Fibonacciego $(F_n)_{n=0,1,\ldots}$ określone są wzorem $F_0 = F_1 = 1$ oraz $F_n = F_{n-1} + F_{n-2}$ dla $n \geqslant 2$. Napisz funkcję, która dla danego k zwróci takie i, że F_i jest najmniejszym elementem, który ma dokładnie k cyfr w zapisie dziesiętnym. Na przykład dla $k = 3$ jest to 12, bo $F_{12} = 144$, a $F_{11} = 89$. Uwaga: funkcja ma działać także np. dla $k = 1000$.

3.6. Wskazówki i odpowiedzi do ćwiczeń

Wskazówka do zadania 3.14. Spróbuj zrobić to zadanie niewprost, czyli sprawdzić, czy prostokąty nie mają części wspólnej. □

Wskazówka do zadania 3.16. Dla liczby k co najwyżej należy sprawdzić, czy dzieli się ona bez reszty przez każdą z liczb $2, 3, \ldots, \sqrt{k}$. □

Wskazówka do zadania 3.19. Do estetycznego sformatowania „tabelki" użyj narzędzi z biblioteki <iomanip>. □

Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

4. Wskaźniki i dynamiczna alokacja pamięci. Proste algorytmy sortowania tablic

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

4.1. Dynamiczna alokacja pamięci

4.1.1. Organizacja pamięci komputera

W drugim rozdziale skryptu dowiedzieliśmy się, że w *pamięci operacyjnej* komputera przechowywane są nie tylko dane, ale i kod maszynowy programów. Podstawową jednostką pamięci jest *komórka* o rozmiarze jednego bajta. Każda komórka pamięci posiada swój *adres*, który jest reprezentowany we współczesnych komputerach za pomocą 32- lub 64-bitowej liczby całkowitej.

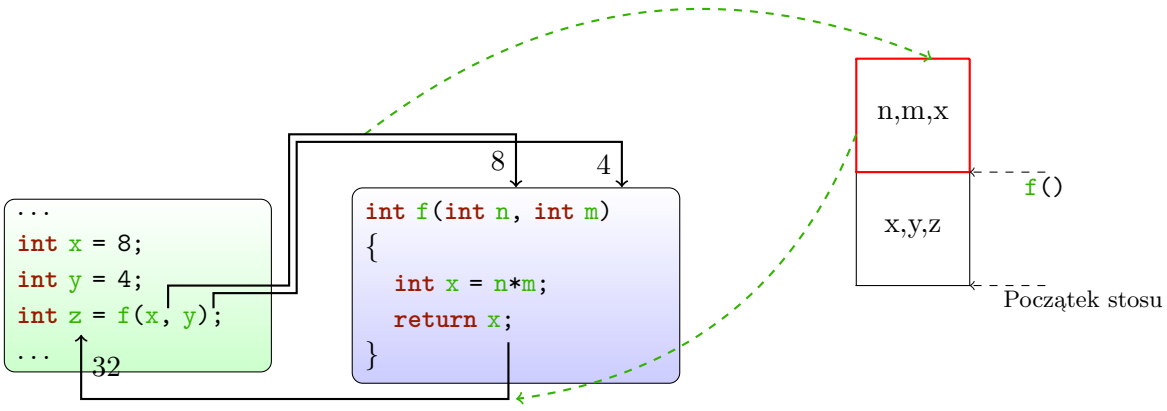
Z punktu widzenia każdego programu można wyróżnić następujący *podział puli adresowej* pamięci (w tzw. architekturze von Neumanna):

- kod programu — informacje interpretowane są tutaj jako instrukcje procesora,
- stos (ang. *stack*) — gdzie przechowywane są wartości zmiennych lokalnych funkcji,
- sterta (ang. *heap*) — gdzie znajdują się dane dynamicznie przydzielane (alokowane) na prośbę programu (zob. dalej),
- część niedostępna — zarządzana przez system operacyjny (m.in. dane innych programów).

Zatem każdy program przechowuje dane potrzebne do wykonywania swych czynności na *stosie* i *stercie*.

Stos jest częścią pamięci operacyjnej, na której dane umieszczane i kasowane są w porządku „ostatni na wejściu, pierwszy na wyjściu” (LIFO, ang. *last-in-first-out*). Umieszczanie i kasowanie danych na stosie odbywa się automatycznie. Każda wywoływana funkcja tworzy na stosie miejsce dla swoich zmiennych lokalnych. Gdy funkcja kończy działanie, usuwa z niego te informacje (to dlatego zmienne lokalne przestają wtedy istnieć).

Przyjrzyjmy się rozszerzonej wersji ilustracji z poprzedniego rozdziału (rys. 4.1). Po lewej stronie widzimy fragment funkcji `main()`, w której zostały zadeklarowane zmienne `x`, `y`, i `z`. Umieszczone są one na „dole” stosu (jako pierwsze w programie). Gdy funkcja ta wywołuje `f()`, na stosie tworzone jest miejsce dla zmiennych `n`, `m` i `x`. Gdy `f()` kończy swe działanie, są one ze stosu automatycznie usuwane.



Rys. 4.1. Zasięg zmiennych

4.1.2. Wskaźniki

Każda zmienna ma przyporządkowaną komórkę (bądź komórki) pamięci, w której przechowuje swoje dane, np. zmienna typu `int` zajmuje najczęściej 4 takie komórki (4 bajty). Fizyczny adres zmiennej (czyli numer komórki) można odczytać za pomocą operatora `&`.

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

Przypomnijmy, że 0xe3d30dbc oznacza liczbę całkowitą zapisaną w systemie szesnastkowym. W systemie dziesiętnym jest ona równa 3822259644. Co ważne, przy kolejnym uruchomieniu programu może to być inna wartość. Nas jednak interesuje tutaj fakt, że jest to „zwyczajna" liczba.

Każda zmienna ma zatem swoje „miejsce na mapie" (tzn. w pamięci komputera), znajdujące się pod pewnym adresem (np. na ul. Koszykowej 75 w Warszawie). Operator `&` pozwala więc uzyskać informację o pozycji danej zmiennej. Jeszcze inaczej: zmienna to „budynek magazynu" w którym można przechowywać towar określonego rodzaju, np. cukierki. Adres zmiennej to „współrzędne GPS" tegoż magazynu.

Specjalny typ danych do przechowywania informacji o adresach innych zmiennych („współrzędnych GPS") określonego typu zwany jest *typem wskaźnikowym*. Oznacza się go przez dodanie symbolu `*` (gwiazdka) bezpośrednio po nazwie typu.

Na przykład, zadeklarowanie zmiennej typu `int*` oznacza stworzenie zmiennej przechowującej fizyczny adres w pamięci komputera pewnej liczby całkowitej (współrzędne GPS pewnego magazynu do przechowywania cukierków), czyli *wskaźnika* na zmienną typu `int`.

Zapamiętaj

Istnieje specjalne miejsce w pamięci o adresie 0 (`NULL`, „czarna dziura"), do którego odwołanie się podczas działania programu powoduje wystąpienie błędu. Często używa się tego adresu np. do zainicjowania wskaźników celem oznaczenia, że początkowo nie wskazują one na „żadne konkretne miejsce".

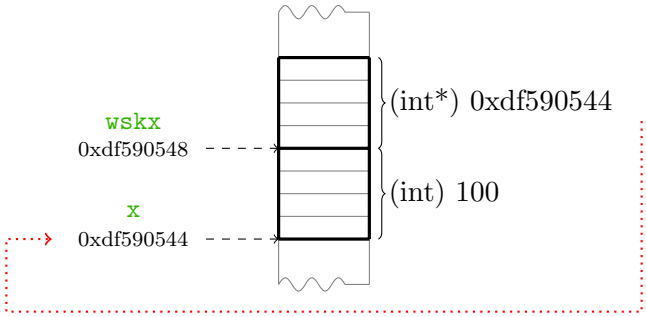
Na zmiennych wskaźnikowych został określony tzw. *operator wyłuskania*, `*` (nie mylić z „gwiazdką" modyfikującą znaczenie typu!), dzięki któremu możemy odczytać, co się znajduje pod danym adresem pamięci (co znajduje się w jakimś magazynie, którego znamy tylko współrzędne GPS).

Przyjrzyjmy się poniższemu przykładowi. Tworzone są dwie zmienne: jedna typu całkowitego, a druga wskaźnikowa. Ich rozmieszczenie w pamięci (a dokładniej na stosie, są to bowiem zmienne lokalne jakiejś funkcji) przedstawia rys. 4.2. Każda z tych zmiennych umieszczona jest pod jakimś adresem w pamięci RAM — można go odczytać za pomocą operatora `&`.

Listing 4.1. „Wyłuskanie" danych spod danego adresu

```
1 int  x = 100;
2 int* wskx = &x;
3
4 cout <<  wskx << endl; // np. 0xdf590544
5 cout << *wskx << endl; // 100
6 cout <<     x << endl; // 100
```

Wypisanie wartości wskaźnika oznacza wypisanie adresu, na który wskazuje. Wypisanie zaś „wyłuskanego" wskaźnika powoduje wydrukowanie wartości komórki pamięci, na którą pokazuje wskaźnik. Jako że zmienna typu `int` u nas ma rozmiar 4 bajtów, adres następnej zmiennej (`wskx`) jest o 4 jednostki większy od adresu `x`.



Rys. 4.2. Zawartość pamięci komputera w programie z listingu 4.1

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

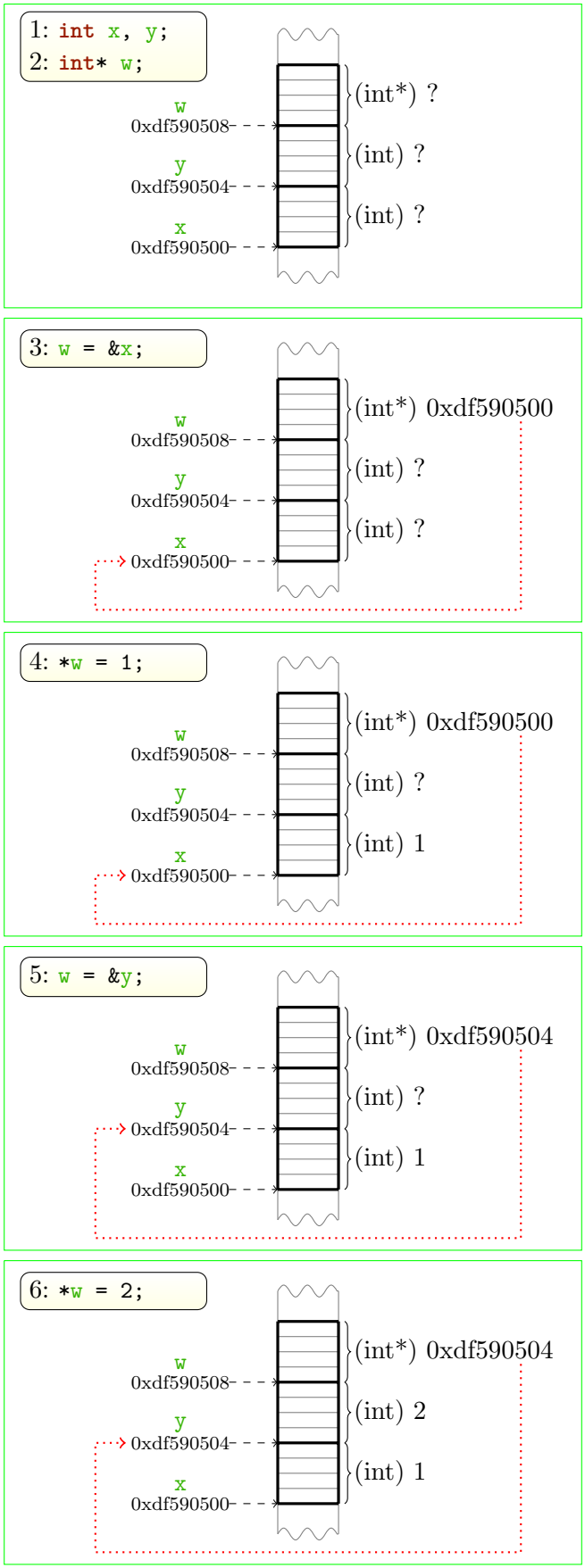
Listing 4.2. Proste operacje z użyciem wskaźników

```
1 int  x, y;
2 int* w;
3 w  = &x; // w = adres x (niech w wskazuje na zmienną x)
4 *w = 1;  // wstaw 1 tam, gdzie wskazuje teraz w
5 w  = &y; // w = adres y (niech w wskazuje na zmienną y)
6 *w = 2;  // wstaw 2 tam, gdzie wskazuje teraz w
```

Zawartość pamięci po wykonaniu kolejnych linii kodu przedstawia rys. 4.3. Tym razem za pomocą operatora wyłuskania zapisujemy dane do komórek pamięci, na które pokazuje wskaźnik `w`.

Zadanie

Prześledź pokazane rysunki bardzo uważnie. Wskaźniki są niezmiernie istotnym elementem języka C++.



Rys. 4.3. Zawartość pamięci po wykonaniu kolejnych instrukcji z listingu 4.2

Przykład z rozdz. 3 (cd.). W poprzednim rozdziale rozważaliśmy funkcję, która służyła do zamiany wartości dwóch zmiennych całkowitych. Przypomnijmy, że prawidłowe rozwiązanie tego problemu wymagało użycia nie argumentów przekazanych przez wartość, ale przez referencję. Równie skutecznym, acz w tym wypadku może nieco mniej eleganckim, jest użycie w tym przypadku wskaźników.

```
1  void zamien(int* x, int* y) {
2      int t = *x;
3      *x = *y;
4      *y = t;
5  }
6
7  int main() {
8      int n = 1, m = 2;
9      zamien(&n, &m); // przekazanie argumentów przez wskaźnik
10     cout << n << ", " << m << endl;
11     return 0;
12 }
```

Dzięki temu, że przekazaliśmy funkcji `zamien()` *adresy* zmiennych zadeklarowanych w funkcji `main()`, możemy nie tylko odczytywać, ale i nadpisywać tutaj ich wartości. Ominęliśmy tym samym ograniczenia przekazywania argumentów przez wartość (kopiowanie) — dostęp do „oryginalnych" zmiennych mamy tutaj w sposób *pośredni*, tj. za pomocą operatora wyłuskania `*`. ■

Ciekawostka

Pamiętamy, że aby dostać się do konkretnego pola struktury, należy użyć operatora „." (kropki). Jeśli mamy dostęp do wskaźnika na strukturę, możemy użyć do tego celu operatora „->".

```
struct Punkt {
    double x;
    double y;
};

// ... (np. main()) ...
Punkt p;
Punkt* wp = &p; // wskaźnik na p
wp->x = 1.0; // to samo, co (*wp).x = 1.0 - drugie mniej wygodne
wp->y = 2.0; // to samo, co (*wp).y = 2.0
// ...
```

4.1.3. Przydział i zwalnianie pamięci ze sterty

Oprócz ściśle określonej na etapie pisania programu ilości danych na stosie, można również dysponować pamięcią na *stercie*, por. s. 1. Miejsce na nasze dane może być przydzielane (alokowane) *dynamicznie* podczas działania programu za pomocą operatora `new`. Po użyciu należy je zwolnić za pomocą operatora `delete`.

Zapamiętaj

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, w jaki sposób możemy dokonać alokacji i dealokacji pamięci dla jednego obiektu.

```
typ* obiekt = new typ; // alokacja (new zwraca wskaźnik na
    przydzielone miejsce w pamięci)
// ...
delete obiekt; // zwolnienie pamięci
```

Co bardzo ważne, możemy w ten sposób również przydzielić pamięć na wiele obiektów następujących kolejno po sobie.

```
int n = 4;
typ* obiekt = new typ[n]; // alokacja (zwraca wskaźnik na pierwszy
    z obiektów)
// ...
delete [] obiekt; // uwaga na ,,[]'' - wiele obiektów!
```

Za pomocą powyższego kodu utworzyliśmy *ciąg* obiektów określonego typu, czyli inaczej *tablicę*.

4.1.4. Tablice

Do tej pory przechowywaliśmy dane używając pojedynczych zmiennych. Były to tzw. *zmienne skalarne* (atomowe). Pojedyncza zmienna odpowiadała jednej „jednostce informacji" (liczbie, wartości logicznej, złożonej strukturze, wskaźnikowi).

Często jednak w naszych programach będzie zachodzić potrzeba rozważenia ciągu n zmiennych tego samego typu, gdzie n niekoniecznie musi być znane z góry. Dla przykładu, rozważmy fragment programu dokonujący podsumowania rocznych zarobków pewnego dość obrotnego studenta.

```
1  double zarobki1, zarobki2, /*...*/, zarobki12;
2                                    // deklaracja 12 zmiennych
3  zarobki1  = 1399.0; // styczeń
4  zarobki2  = 1493.0; // luty
5  // ...
6  zarobki12 = 999.99; // grudzień
7
8  double suma = 0.0;
9  suma += zarobki1;
10 suma += zarobki2;
11 //...
12 suma += zarobki12;
13
14 cout << "Zarobiłem w 2012 r. " << suma << " zł.";
```

Dochód z każdego miesiąca przechowywany jest w oddzielnej zmiennej. Nietrudno zauważyć, że operowanie na nich nie jest zbyt wygodne. Mało tego, dość żmudne byłoby rozszerzanie funkcjonalności takiego programu na przypadek obejmujący podsumowanie np. zarobków z 2,3,... lat.

Rozwiązanie tego problemu może być jednak bardzo czytelnie zapisane z użyciem dynamicznie alokowanych tablic, które są reprezentacją znanych nam obiektów matematycznych: *ciągów skończonych* bądź *wektorów*.

Wiemy, że za pomocą operatora `new` możemy przydzielić pamięć dla $n \geqslant 1$ obiektów określonego typu. Operator ten zwraca wskaźnik na pierwszy element takiego ciągu. Pozostaje tylko odpowiedzieć sobie na pytanie, w jaki sposób możemy się dostać do kolejnych elementów.

```
double* zarobki = new double[12];
// zarobki - wskaznik na pierwszy element ciągu
*zarobki = 1399.0; // wprowadź zarobki w pierwszym miesiącu
// co dalej?
delete [] zarobki; // zwolnienie pamięci
```

Przypomnijmy, wskaźnik jest po prawdzie liczbą całkowitą. Okazuje się, że została określona na nim operacja dodawania. I tak `zarobki+i`, gdzie `i` jest liczbą całkowitą nieujemną, oznacza „podaj adres `i`-tego obiektu z ciągu”. Tym samym `zarobki+0` jest tym samym, co po prostu `zarobki` (adresem pierwszego elementu), a `zarobki+n-1`, adresem ostatniego elementu z `n`-elementowego ciągu.

Wobec powyższego, fragment pierwotnej wersji programu związany z wprowadzeniem zarobków możemy zapisać w następujący sposób:

```
double* zarobki = new double[12];
// zarobki - wskaznik na pierwszy element ciągu
*(zarobki+0)  = 1399.0; // wprowadź zarobki w pierwszym miesiącu
*(zarobki+1)  = 1493.0; // wprowadź zarobki w drugim miesiącu
// ...
*(zarobki+11) = 999.99; // wprowadź zarobki w ostatnim miesiącu
// ...
delete [] zarobki; // zwolnienie pamięci
```

Zapamiętaj

Wygodniejszy dostęp do poszczególnych elementów tablicy możemy uzyskać za pomocą *operatora indeksowania*, „$[\cdot]$”.

Jeśli `t` jest tablicą (a ściślej: wskaźnikiem na pierwszy element ciągu obiektów przydzielonych dynamicznie), to `*(t+i)` możemy zapisać równoważnie przez `t[i]`.

Elementy tablicy są numerowane od 0 do $n-1$, gdzie n to rozmiar tablicy.

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

Informacja

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

```
1 int* t = new int[5];
2 t[-100]  = 15123;  // :-(
3 t[10000] = 25326;  // :-(
4 delete [] t;
```

Powyższe instrukcje jednak zmieniają wartości komórek pamięci reprezentujących dane innych obiektów. Skutki tego działania mogą się objawić w innym miejscu programu, powodując nieprzewidywalne i trudne do wykrycia błędy.

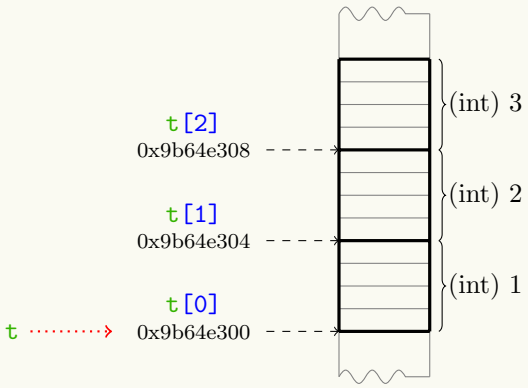
Ciekawostka

Niech `t` będzie wskaźnikiem na pewien `typ`. Zauważmy, że zapis `t+i` nie oznacza koniecznie, że chodzi nam o adres przechowywany w zmiennej wskaźnikowej `t` *plus* jeden bajt. Operacja dodawania bierze pod uwagę `typ` zmiennej wskaźnikowej i dokonuje „przesunięcia" adresu o wielokrotność liczby bajtów, które zajmuje w pamięci jedna zmienna typu `typ`.

Możemy to sprawdzić np. w następujący sposób.

```
int* t = new int[3]; // u nas int to 4 bajty
cout << t;    // np. 0x9b64e300 - to samo, co cout << &t[0];
cout << t+1;  // np. 0x9b64e304 - to samo, co cout << &t[1];
cout << t+2;  // np. 0x9b64e308 - to samo, co cout << &t[2];
t[0] = 1;
t[1] = 2;
t[2] = 3;
delete [] t;
```

Kolejne elementy tablicy w pamięci zawsze następują po sobie, co ilustruje poniższy rysunek.



Jesteśmy już gotowi, by napisać fragment programu do sumowania zarobków naszego kolegi. Tym razem nie będziemy zakładać, że liczba miesięcy jest określona z góry. Tutaj będziemy ją wprowadzać z klawiatury.

```
int n;
cout << "Ile miesięcy? ";
cin >> n;
assert(n>0); // wymaga <cassert>

double* zarobki = new double[n];

for (int i=0; i<n; ++i) {
   cout << "Podaj zarobki w miesiącu nr " << i << ": ";
   cin >> zarobki[i]; // to samo, co cin >> *(zarobki+i);
}

// tutaj możemy np. wygenerować ładne zestawienie
```

```
14 // wprowadzonych danych itp.
15 for (int i=0; i<n; ++i)
16    cout << i << ": " << zarobki[i] << endl;
17
18 // policzmy sumę:
19 double suma = 0.0;
20 for (int i=0; i<n; ++i)
21    suma += zarobki[i];
22
23 cout << "Zarobiłem w 2012 r. " << suma << " zł.";
24
25 delete [] zarobki;
```

Ciekawostka

W języku C++ można także deklarować tablice o ustalonym z góry, stałym rozmiarze (na stosie). Ich zaletą jest możliwość ustalenia wartości ich elementów podczas deklaracji, co może poprawiać czytelność programów demonstrujących implementacje pewnych szczególnych algorytmów.

```
1 int t[3] = {1, 2, 3}; // deklaracja tablicy i inicjowanie
      wartości
2 // t wciąż jest wskaźnikiem na pierwszy element:
3 // używamy go tak, jak w przypadku tablicy dynamicznie
      alokowanej
4 // ...
5 // nie stosujemy delete! (bo nie było new)
```

Jednak w przypadku dużej klasy problemów ich implementacja z użyciem tablic o zadanym rozmiarze wydaje się mało naturalna i słabo „skalowalna” względem rozmiaru problemu.

4.1.5. Przekazywanie tablic funkcjom

Podsumujmy: tablica to ciąg obiektów tego samego typu położonych w pamięci kolejno, jeden po drugim. Dostęp do elementów tablicy mamy zapewniony przez wskaźnik na pierwszy z tych obiektów. Obiekty typu wskaźnikowego możemy przekazywać funkcjom tak, jak zwykłe zmienne skalarne. Wobec tego, aby można było napisać funkcję, która w jakiś sposób przetwarza dane zawarte w tablicy, należy dodatkowo przekazać jej informację o tym, ile jest elementów w tablicy, czyli jej *rozmiar*.

Ciekawostka

W rozdziale 5 poznamy inny sposób informowania funkcji, gdzie znajduje się ostatni element tablicy. Założymy wtedy, że ostatnim elementem ciągu jest pewna specjalna wyróżniona wartość, (tzw. *wartownik*) która nie pojawia się jako „zwykły” element gdzie indziej. Wówczas wystarczy nam tylko informacja na temat położenia pierwszego elementu tablicy. Jej koniec bowiem będziemy mogli sami sobie znaleźć.

Przykład. *Funkcja wyznaczająca sumę wartości elementów z podanej tablicy.*

```
1  double suma(double const* t, int n)
2  {  // dostęp do elementów t tylko do odczytu
3     assert(n > 0);
4     double s = 0.0;
5     for (int i=0; i<n; ++i)
6        s += t[i];
7     return s;
8  }
9
10 int main(void)
11 {
12    int ile = 10;
13    double* punkty = new double[ile];
14    for (int i=0; i<ile; ++i)
15       cin >> punkty[i];
16    cout << suma(punkty, ile); // przekazanie tablicy do funkcji
17    delete [] punkty;
18    return 0;
19 }
```

Na marginesie, zauważmy, że funkcja `suma()` nie ma żadnych efektów ubocznych. Nie wypisuje nic na ekran ani o nic nie pyta się użytkownika. Wyznacza tylko wartość sumy elementów zawartych w tablicy — czyli tylko to, czego użytkownik (funkcja `main()`) może się po niej spodziewać.

Ponadto zapis `double const*` zapewnia, że elementów danej tablicy nie można zmieniać. Jest to tzw. wskaźnik na wartości stałe. ■

4.2. Proste algorytmy sortowania tablic

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

Zadanie

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

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

Zadanie

Przykładowo, rozważmy w jaki sposób wyszukujemy interesujące nas strony internetowe. Większość wyszukiwarek działa w następujący sposób.

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

Warto przypomnieć, że o rynkowym sukcesie wyszukiwarki Google zadecydowało to, że — w przeciwieństwie do ówczesnych konkurencyjnych serwisów — zwracała ona wyniki w dość „pożytecznej” dla większości osób kolejności.

Problem sortowania, w swej najprostszej postaci, można sformalizować w następujący sposób.

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

Ciekawostka

Zauważmy, że rozwiązanie takiego problemu wcale nie musi być jednoznaczne.

Dla tablic zawierających elementy `t[i]` i `t[j]` takie, że dla `i≠j` zachodzi `t[i] <= t[j]` oraz `t[j] <= t[i]`, tj. `t[i] == t[j]`, może istnieje więcej niż jedna permutacja spełniająca powyższe warunki.

Algorytm sortowania nazwiemy *stabilnym*, jeśli względna kolejność elementów o tej samej wartości zostaje zachowana po posortowaniu. Własność ta jest przydatna w przypadku sortowania obiektów złożonych za pomocą więcej niż jednego kryterium na raz.

W niniejszym paragrafie omówimy trzy algorytmy sortowania:

1. sortowanie przez wybór,
2. sortowanie przez wstawianie,
3. sortowanie bąbelkowe.

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

4.2.1. Sortowanie przez wybór

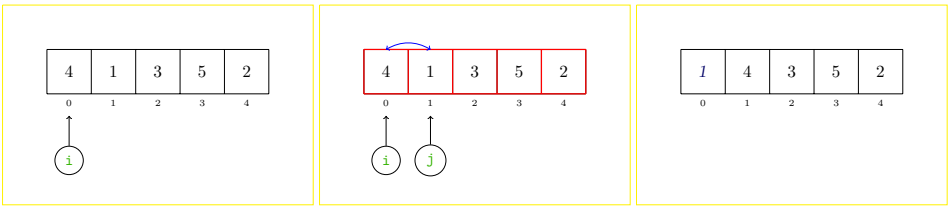
W algorytmie *sortowania przez wybór* (ang. *selection sort*) dokonujemy za każdym razem wyboru elementu najmniejszego spośród do tej pory nieposortowanych, póki cała tablica nie zostanie uporządkowana.

Ideę tę przedstawia następujący pseudokod:

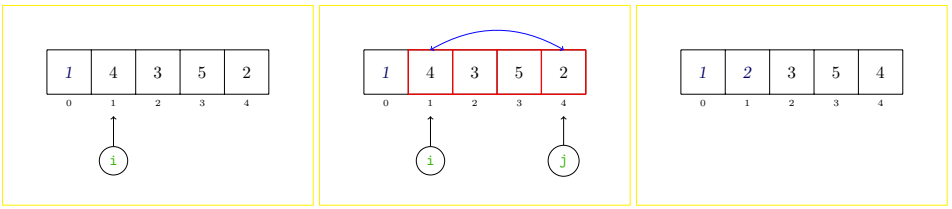
```
dla i=0,1,...,n-2
{
   // t[0], ..., t[i-1] są już uporządkowane względem relacji <=
   // (nadto, są już na swoich ostatecznych miejscach)
   j = indeks najmniejszego elementu
         spośród t[i], ..., t[n-1];
   zamień elementy t[i] i t[j];
}
```

Jako przykład rozważmy, krok po kroku, przebieg sortowania ciągu liczb naturalnych `(4,1,3,5,2)`. Kolejne iteracje działania tego algorytmu ilustrują rys. 4.4–4.8.

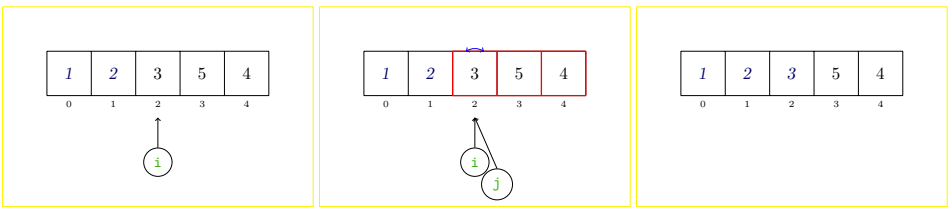
1. W kroku I (rys. 4.4) mamy `i==0`. Dokonując wyboru elementu najmniejszego spośród `t[0]`, ..., `t[4]` otrzymujemy `j==1`. Zamieniamy więc elementy `t[0]` i `t[1]` miejscami.
2. W kroku II (rys. 4.5) mamy `i==1`. Wybór najmniejszego elementu wśród `t[1]`, ..., `t[4]` daje `j==4` Zamieniamy miejscami zatem `t[1]` i `t[4]`.
3. Dalej (rys. 4.6), `i==2`. Elementem najmniejszym spośród `t[2]`, ..., `t[4]` jest `t[j]` dla `j==2` Zamieniamy miejscami zatem niezbyt sensownie `t[2]` i `t[2]`. Komputer, na szczęście, zrobi to bez grymasu.
4. W ostatnim kroku (rys. 4.7) `i==3` i `j==4`, dzięki czemu możemy uzyskać ostateczne rozwiązanie (rys. 4.8).



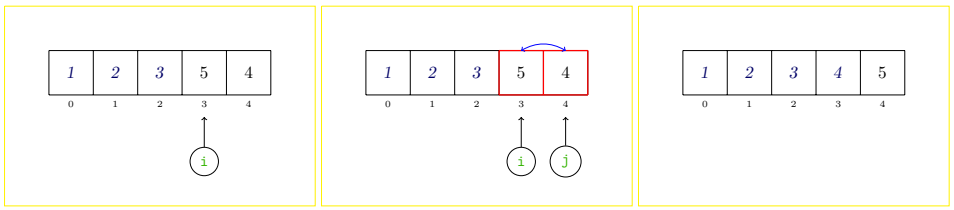
Rys. 4.4. Sortowanie przez wybór — przykład — iteracja I



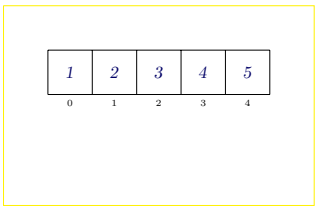
Rys. 4.5. Sortowanie przez wybór — przykład — iteracja II



Rys. 4.6. Sortowanie przez wybór — przykład — iteracja III



Rys. 4.7. Sortowanie przez wybór — przykład — iteracja IV



Rys. 4.8. Sortowanie przez wybór — przykład — rozwiązanie

4.2.2. Sortowanie przez wstawianie

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

W niniejszej metodzie w `i`-tym kroku elementy `t[0], ..., t[i-1]` są już wstępnie uporządkowane względem relacji `<=`. 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 wstępnie uporządkowane względem <=
   // (ale niekoniecznie jest to ich ostateczne miejsce)
   j = indeks takiego elementu spośród t[0],...,t[i], że
      t[u] <= t[i] dla każdego u < j oraz
      t[i] <  t[v] dla każdego v ⩾ j;
   jeśli (j < i) wstaw t[i] przed t[j];
}
```

gdzie przez operację „`wstaw t[i] przed t[j]`”, dla $0 \leqslant j < i$ 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]`

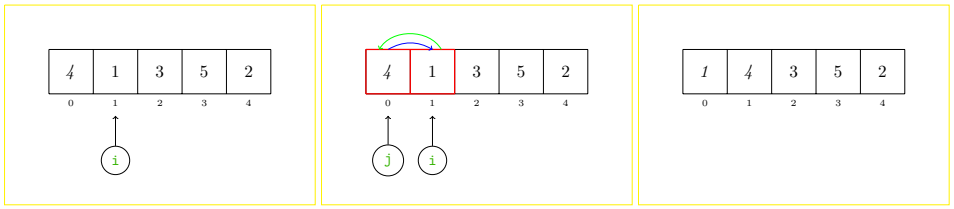
Powyższy pseudokod może być wyrażony w następującej równoważnej formie:

```
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];
   jeśli (j < i) wstaw t[i] przed t[j];
}
```

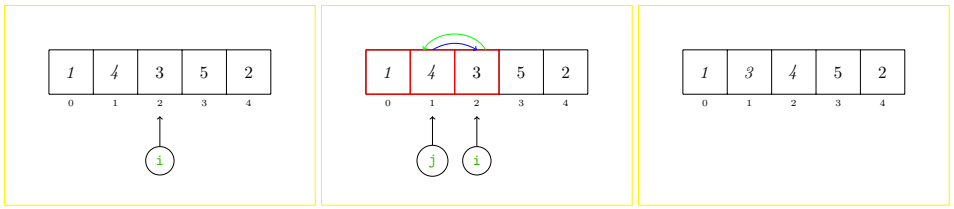
Jako przykład rozpatrzmy znów ciąg liczb naturalnych `(4,1,3,5,2)`.
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 4.9–4.12.

Ciekawostka

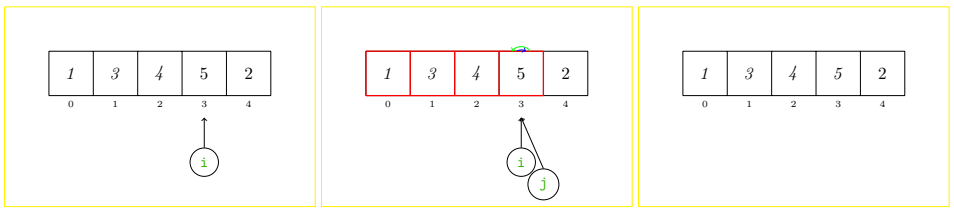
Można pokazać, że tak sformułowany algorytm jest stabilny. Prześledź jego działanie np. dla ciągu $(2', 4, 2'', 5, 1, 3)$ (dla czytelności te same elementy wyróżniliśmy, by wskazać ich pierwotny porządek). Porównaj uzyskany wynik z tym, który można uzyskać za pomocą algorytmu sortowania przez wybór.



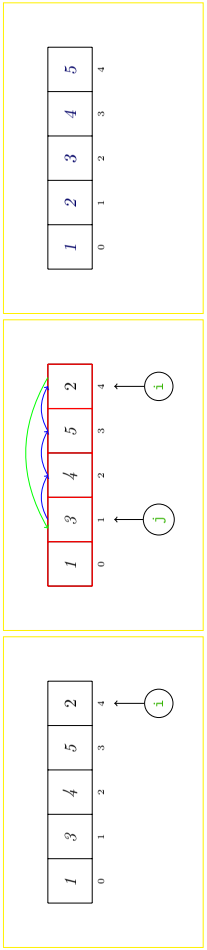
Rys. 4.9. Sortowanie przez wstawianie — przykład — iteracja I



Rys. 4.10. Sortowanie przez wstawianie — przykład — iteracja II



Rys. 4.11. Sortowanie przez wstawianie — przykład — iteracja III



Rys. 4.12. Sortowanie przez wstawianie — przykład — iteracja IV

4.2.3. Sortowanie bąbelkowe

Sortowanie bąbelkowe (ang. *bubble sort*) jest interesującym przykładem algorytmu pojawiającego się w większości podręczników akademickich dotyczących podstawowych sposobów sortowania tablic, którego prawie wcale nie stosuje się 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 niniejszej książeczki. Tak umotywowani, przystąpmy więc do zapoznania się z nim.

W tym algorytmie porównywane są tylko elementy 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.

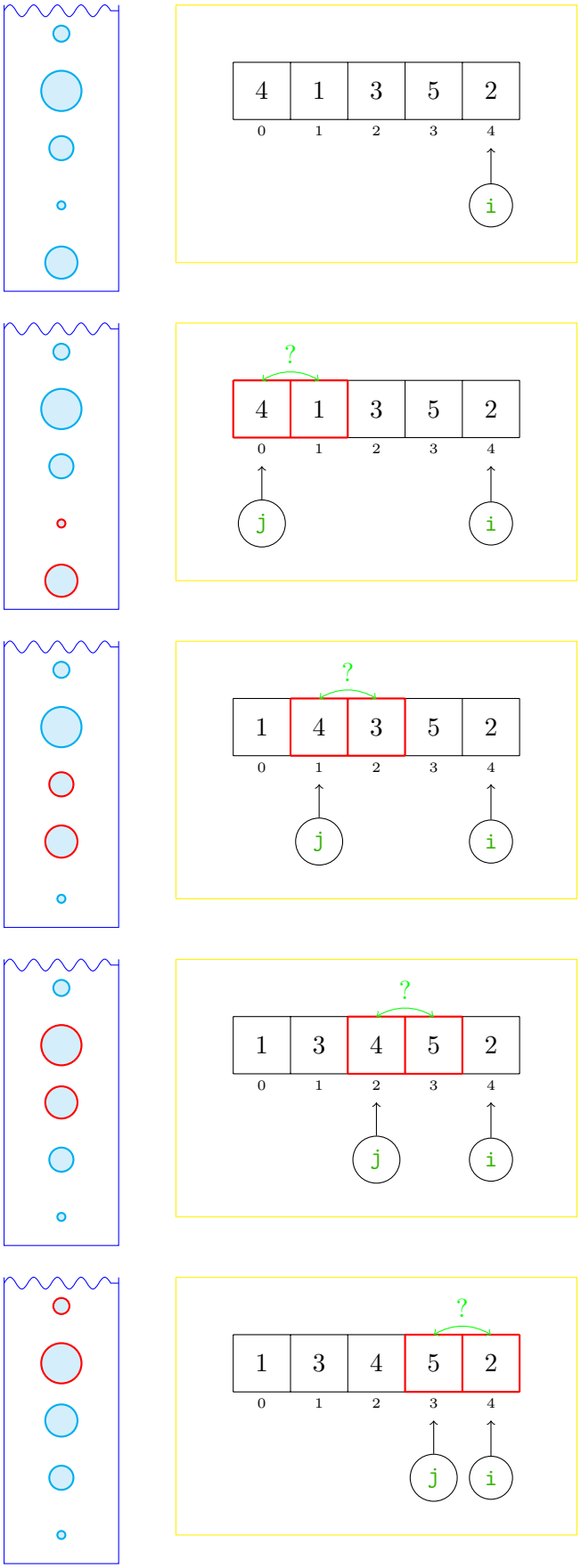
A oto pseudokod:

```
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
}
```

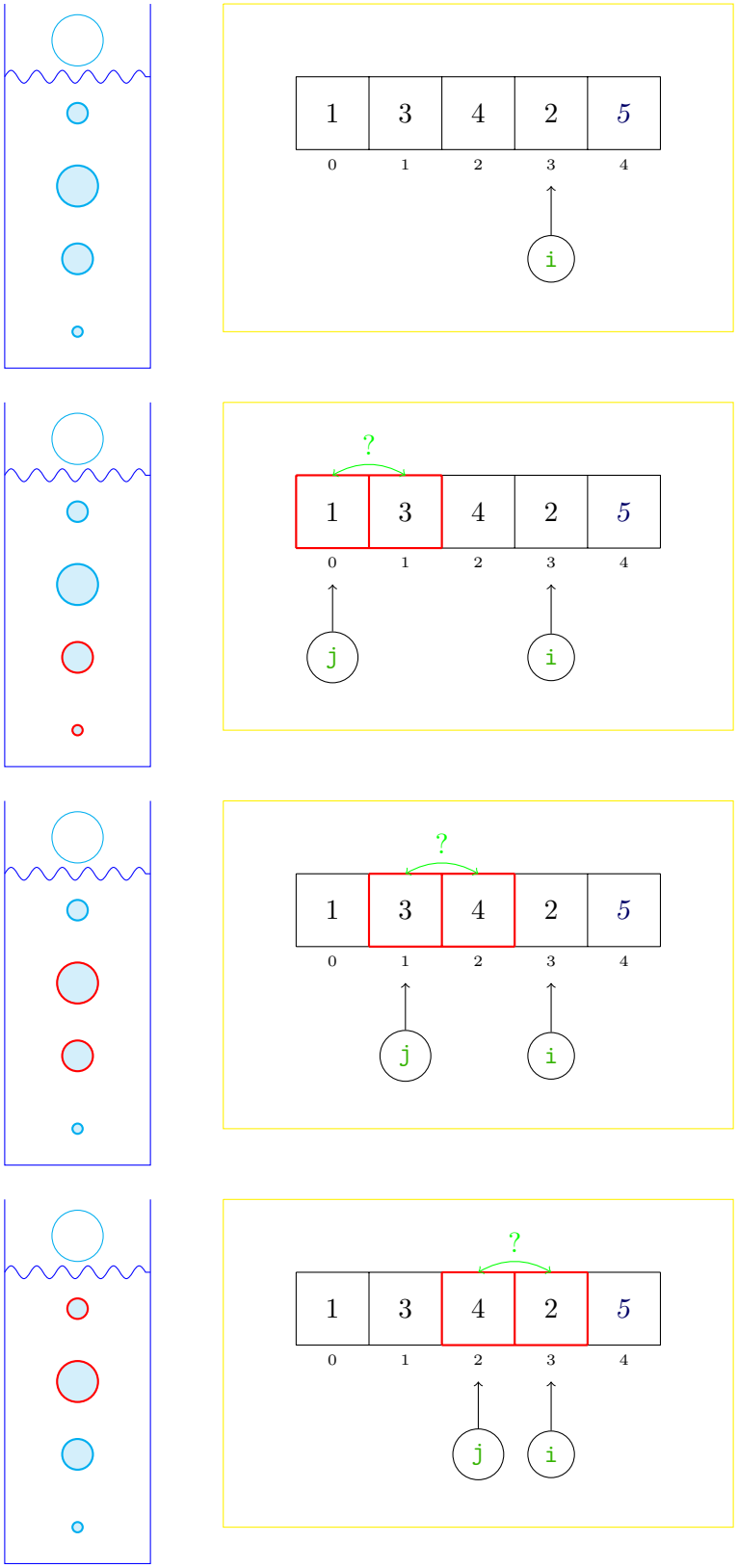
Dla przykładu rozpatrzmy ponownie tablicę `(4,1,3,5,2)`.
Przebieg kolejnych wykonywanych kroków przedstawiają rys. 4.13–4.17.

Ciekawostka

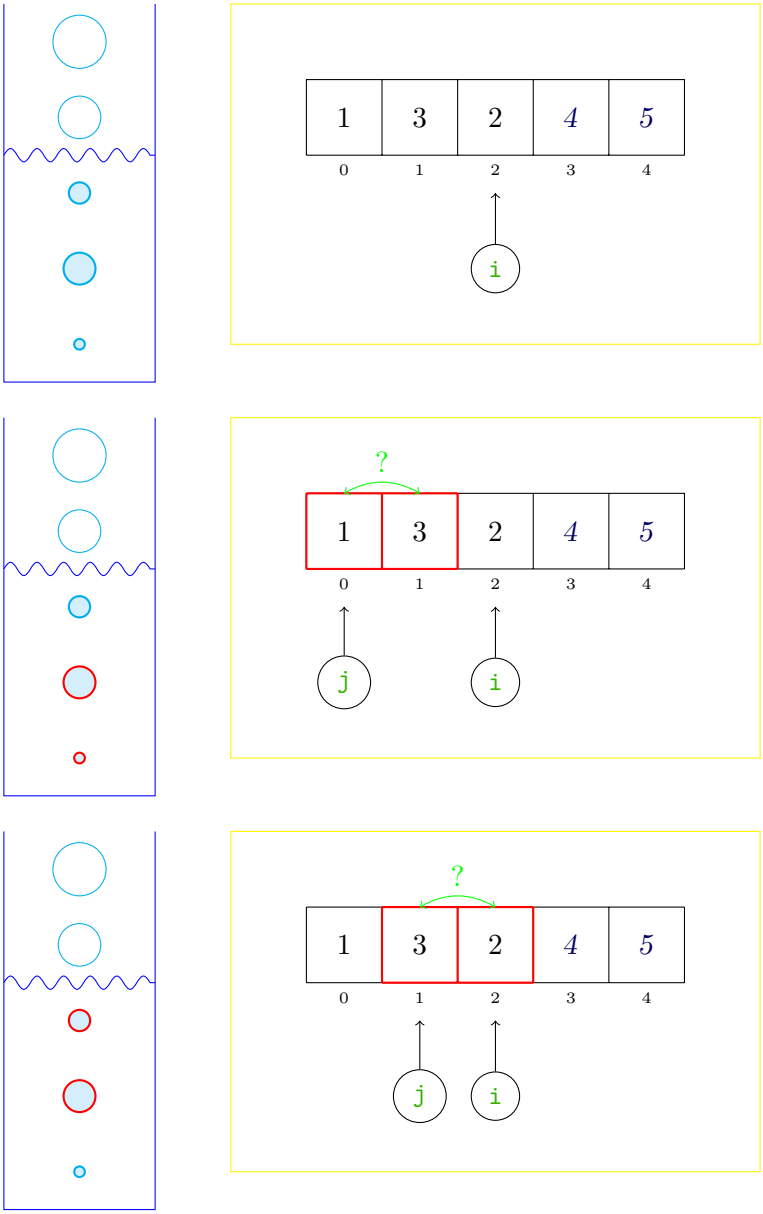
Ten algorytm również jest stabilny.



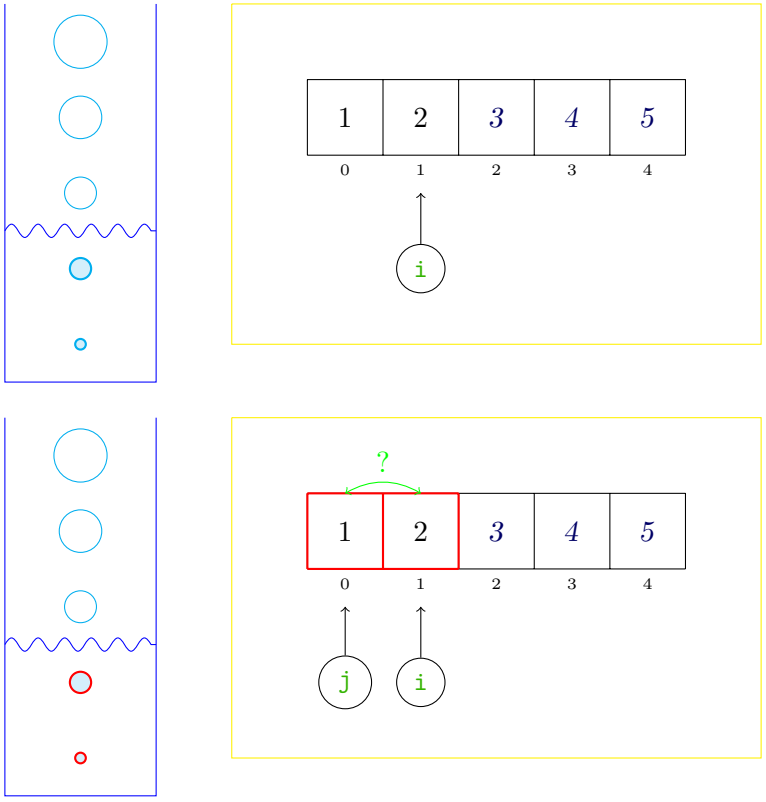
Rys. 4.13. Sortowanie bąbelkowe — przykład — krok I



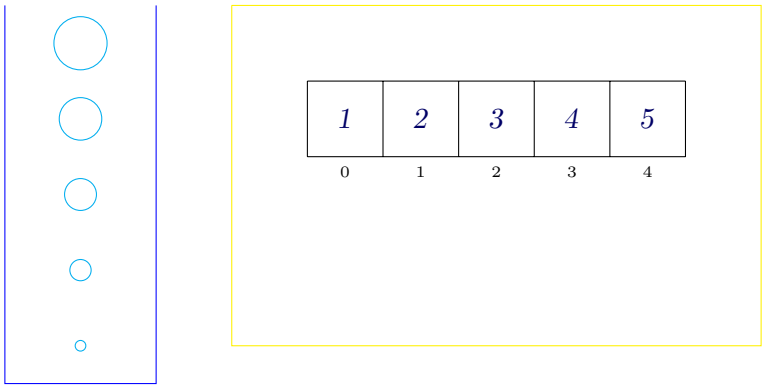
Rys. 4.14. Sortowanie bąbelkowe — przykład — krok II



Rys. 4.15. Sortowanie bąbelkowe — przykład — krok III



Rys. 4.16. Sortowanie bąbelkowe — przykład — krok IV



Rys. 4.17. Sortowanie bąbelkowe — przykład — rozwiązanie

4.2.4. Efektywność obliczeniowa

Rozważając algorytmy sortowania w sposób oczywisty obserwujemy wspomniane w pierwszym rozdziale zjawisko „jeden problem — wiele algorytmów". Rzecz jasna, trzy podane wyżej algorytmy (co zresztą zostało zapowiedziane) nie wyczerpują wszystkich możliwości rozwiązania zagadnienia sortowania tablic jednowymiarowych.

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

„Leniuszki" z pewnością wybrałyby algorytm najprostszy w implementacji. Domyślamy się jednak, że nie jest to zbyt roztropne kryterium.

W tzw. *analizie algorytmów* wyróżnia się dwie najważniejsze miary efektywności obliczeniowej:

1. *złożoność czasowa* — czyli liczbę tzw. *operacji znaczących* (zależną od danego problemu; mogą to być przestawienia, porównania, operacje arytmetyczne itp.) potrzebnych do rozwiązania danego zagadnienia,

2. *złożoność pamięciowa* — czyli ilość dodatkowej pamięci potrzebnej do rozwiązania problemu.

Zauważmy, że w przypadku przedstawionych wyżej algorytmów, oprócz danej tablicy którą należy posortować i kilku zmiennych pomocniczych nie jest potrzebna żadna dodatkowa liczba komórek pamięci komputera (w przypadku bardziej złożonych metod omawianych w sem. III będzie jednak inaczej, np. konieczne będzie użycie jeszcze jednej tablicy). Skupimy się więc teraz tylko na omówieniu *złożoności czasowej*.

Przyjrzyjmy się zatem przykładowym implementacjom dwóch algorytmów dla danej tablicy `t` rozmiaru `n`. Oto kod stosujący sortowanie przez wybór:

```
// Sortowanie przez wybór:
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;

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

A oto kod sortujący tablicę „bąbelkowo":

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

```
10              t[j+1] = p;
11          }
12      }
13  }
```

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

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

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

Ciekawostka

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

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

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

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

1. Analiza liczby porównań elementów.

1. Sortowanie przez wybór.

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

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

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

2. Analiza liczby przestawień elementów.

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

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

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

Zauważamy, że użycie powyższych algorytmów już dla $n > 100000$ nie jest dobrym pomysłem (na szczęście, istnieją lepsze, naprawdę szybkie sposoby rozwiązania problemu sortowania). Ponadto, sama liczba przestawień nie tłumaczy czasu wykonania obliczeń (dlatego konieczne było także zbadanie liczby potrzebnych porównań).

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

	Liczba przestawień		Czas [s]	
n	Selection	Bubble	Selection	Bubble
100	99	4950	∼0.01	∼0.01
1000	999	499500	∼0.01	∼0.01
10000	9999	49995000	0.05	0.10
100000	99999	4999950000	6.02	10.31
1000000	999999	499999500000	631.38	1003.60

4.3. Ćwiczenia

Zadanie 4.1. Napisz funkcję, która za pomocą tylko jednej pętli `for` znajduje i wypisuje na ekran element najmniejszy i element największy danej tablicy liczb całkowitych.

Zadanie 4.2. Napisz funkcję, która za pomocą tylko jednej pętli `for` wyznaczy i zwróci trzeci największy element z danej $3 \leqslant n$-elementowej tablicy liczb całkowitych, np. dla ciągu $(3.0, 5.0, 2.0, 4.0, 1.0, 6.0)$ będzie to 4.0.

Zadanie 4.3. Niech dany będzie n-elementowy ciąg liczb rzeczywistych $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Średnią ważoną (ang. *weighted mean*) względem nieujemnego wektora wag $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ takiego, że $\sum_{i=1}^{n} w_i = 1$, nazywamy wartość $\mathsf{WM}_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i$. Napisz funkcję, który wyznaczy wartość WM danego ciągu względem danego nieujemnego wektora, który przed obliczeniami należy unormować tak, by jego elementy sumowały się do 1.

Zadanie 4.4. Dana jest tablica `t` składająca się z liczb całkowitych od 0 do 19. Napisz funkcję, która wyznaczy jej dominantę (modę), czyli najczęściej pojawiającą się wartość. *Wskazówka. Skorzystaj z pomocniczej, 20-elementowej tablicy, za pomocą której zliczysz, ile razy w tablicy* `t` *występuje każda możliwa wartość. Tablica powinna być zainicjowana zerami. Następnie należy znaleźć jej maksimum.*

Zadanie 4.5. Uogólnij funkcję z zad. 4.4 tak, by działała nie tylko dla tablic o elementach ze zbioru $\{0, 1, \ldots, 19\}$, ale dla dowolnego $\{a, a+1, \ldots, b\}$.

Zadanie 4.6. *Sortowanie kubełkowe.* Dana jest n-elementowa tablica `t` wypełniona liczbami naturalnymi ze zbioru $\{1, 2, \ldots, k\}$ dla pewnego k. Napisz funkcję, która za pomocą tzw. sortowania kubełkowego uporządkuje w kolejności niemalejącej elementy z `t`. W algorytmie sortowania kubełkowego korzystamy z k-elementowej tablicy pomocniczej, która służy do zliczania liczby wystąpień każdej z k wartości elementów z `t`. Na początku tablica pomocnicza jest wypełniona zerami. Należy rozpatrywać kolejno każdy element tablicy `t`, za każdym razem zwiększając o 1 wartość odpowiedniej komórki tablicy pomocniczej.

Dla przykładu, załóżmy $n = 6$, $k = 4$ oraz że dana jest tablica `t` o elementach $(4, 3, 2, 4, 4, 2)$. Tablica pomocnicza powinna posłużyć do uzyskania informacji, że w `t` jest 0 elementów równych 1, 2 elementy równe 2, 1 element równy 3 oraz 3 elementy równe 4. Na podstawie tej wiedzy można zastąpić elementy tablicy `t` kolejno wartościami $(2, 2, 3, 4, 4, 4)$, tym samym otrzymując w wyniku posortowaną tablicę.

Zadanie 4.7. Dane są dwie liczby całkowite w systemie dziesiętnym: n-cyfrowa liczba reprezentowana za pomocą tablicy `a` oraz m-cyfrowa reprezentowana za pomocą tablicy `b`, gdzie $n > m$. Każda cyfra zajmuje osobną komórkę tablicy. Cyfry zapisane są w kolejności od najmłodszej do najstarszej, tzn. element o indeksie 0 oznacza jedności, 1 – dziesiątki itd., przy czym najstarsza cyfra jest różna od 0. Napisz funkcję, która utworzy nową, dynamicznie alokowaną tablicę reprezentującą wynik odejmowania liczb `a` i `b`.

Zadanie 4.8. Dane są dwie liczby całkowite w systemie dziesiętnym reprezentowane za pomocą dwóch tablic `a` i `b` o rozmiarze $n > 1$. Każda cyfra zajmuje osobną komórkę tablicy. Cyfry zapisane są w kolejności od najmłodszej do najstarszej, tzn. element o indeksie 0 oznacza jedności, 1 – dziesiątki itd., przy czym najstarsza cyfra jest różna od 0. Napisz funkcję, która zwróci tablicę reprezentującą wynik dodawania liczb `a` i `b`.

Zadanie 4.9. Utwórz strukturę o nazwie `DzienMiesiac` zawierającą dwa pola typu całkowitego: `dzien` i `miesiac`. Następnie stwórz funkcje `poprawneDaty()`, która dla danej tablicy o elementach typu `DzienMiesiac` zwróci wartość logiczną `true` wtedy i tylko wtedy, gdy każdy jej element określa prawidłową datę w roku nieprzestępnym.

Zadanie 4.10. Zmodyfikuj funkcję z zad. 4.9 tak, by przyjmowała jako argument także rok. Sprawdzanie daty tym razem ma uwzględniać, czy dany rok jest przestępny.

Zadanie 4.11. Dany jest wielomian rzeczywisty stopnia $n > 0$ którego współczynniki przechowywane są w $(n+1$-elementowej tablicy `w` tak, że zachodzi $w(x) = \texttt{w[0]}x^0 + \texttt{w[1]}x^1 + \ldots \texttt{w[n]}x^n$, $\texttt{w[n]} \neq 0$. Napisz funkcję, która zwróci wartość $w(x)$ dla danego x.

Zadanie 4.12. Zmodyfikuj funkcję z zad. 4.11 tak, by korzystała z bardziej efektywnego obliczeniowo wzoru na wartość $w(x)$, zwanego schematem Hornera: $w(x) = (\cdots(((\texttt{w[n]}x + \texttt{w[n-1]})\,x + \texttt{w[n-2]})\,x + \texttt{w[n-3]})\cdots)\,x + \texttt{w[0]}$.

Zadanie 4.13. Niech dane będą wielomiany $w(x)$ i $v(x)$ stopnia, odpowiednio, n i m. Napisz funkcję, która wyznaczy wartości współczynników wielomianu $u(x)$ stopnia $n+m$, będącego iloczynem wielomianów w i v.

Zadanie 4.14. Dane są dwie uporządkowane niemalejąco tablice liczb całkowitych `x` i `y` rozmiarów, odpowiednio, n i m. Napisz funkcję, który zwróci uporządkowaną niemalejąco tablicę rozmiaru $n+m$ powstałą ze scalenia tablic `x` i `y`. Algorytm powinien mieć liniową (rzędu $n+m$) pesymistyczną złożoność czasową. Np. dla $\texttt{x} = (1,4,5)$ i $\texttt{y} = (0,2,3)$ powinniśmy otrzymać $(0,1,2,3,4,5)$.

Zadanie 4.15. Dana jest n-elementowa, już posortowana tablica liczb całkowitych oraz liczba całkowita x. Napisz funkcję, która za pomocą wyszukiwania binarnego (połówkowego) sprawdzi, czy wartość x jest elementem tablicy i jeśli tak, to poda pod którym indeksem tablicy się znajduje bądź zwróci -1 w przeciwnym przypadku.

Zadanie 4.16. Zaimplementuj w postaci funkcji algorytm sortowania bąbelkowego danej tablicy o n elementach typu `int`, w którym wykorzystywany jest tzw. wartownik. Wartownik to dodatkowa zmienna w, która zapamiętuje indeks tablicy, pod którym wystąpiło ostatnie przestawienie elementów w pętli wewnętrznej. Dzięki niemu wystarczy, że w kolejnej iteracji pętla wewnętrzna zatrzyma się na indeksie w. W przypadku pomyślnego ułożenia danych w tablicy wejściowej może to bardzo przyspieszyć obliczenia.

Porównaj swoją wersję algorytmu z podaną w skrypcie pod względem liczby potrzebnych operacji porównań oraz przestawień elementów w zależności od n dla tablicy już posortowanej oraz dla tablicy posortowanej w kolejności odwrotnej.

⋆⋆ **Zadanie 4.17.** Dana jest tablica o $n > 1$ elementach typu `double`. Napisz funkcję wykorzystującą tylko jedną pętlę `for`, która obliczy wariancję jej elementów. Wariancja ciągu $\mathbf{x} = (x_1, \ldots, x_n)$ dana jest wzorem $s^2(\mathbf{x}) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{\mathbf{x}})^2$, gdzie $\overline{\mathbf{x}}$ jest średnią arytmetyczną ciągu $\mathbf{x}$.

⋆ **Zadanie 4.18.** [PN i PS] Dana jest tablica o elementach typu `double`. Napisz funkcję, która sprawdza, czy każde 3 liczby z tablicy mogą być długościami boków jakichś trójkątów.

⋆ **Zadanie 4.19.** [MD] Tak zwany ciąg EKG $(a_1, a_2, \ldots)$ o elementach naturalnych jest zdefiniowany następująco: $a_1 = 1$, $a_2 = 2$ oraz a_{n+1} jest najmniejszą liczbą naturalną nie występującą wcześniej w ciągu taką, że $\mathrm{NWD}(a_n, a_{n+1}) > 1$, tzn. mającą nietrywialny wspólny dzielnik z a_n. Napisz funkcję, która wyznaczy a_n dla danego n.

⋆ **Zadanie 4.20.** [MD] Dana jest n-elementowa tablica t o elementach całkowitych, reprezentująca permutację zbioru $\{0,1,\ldots,n-1\}$. Napisz fragment kodu, który wyznaczy liczbę cykli w tej permutacji. Dla przykładu, tablica $\begin{pmatrix} 2, & 1, & 5, & 4, & 3, & 0 \\ t[0] & t[1] & t[2] & t[3] & t[4] & t[5] \end{pmatrix}$ reprezentuje permutację o 3 cyklach. Zawiera bowiem cykl o długości 3 ($t[0] = 2 \to t[2] = 5 \to t[5] = 0$), cykl o długości 2 ($t[3] = 4 \to t[4] = 3$) oraz cykl o długości 1 ($t[1] = 1$).

⋆ **Zadanie 4.21.** [MD] Ciąg Golomba $(g_1, g_2, \ldots)$ to jedyny niemalejący ciąg liczb naturalnych, w którym każda liczba i występuje dokładnie g_i razy, przy założeniu $g_1 = 1$. Napisz funkcję, która wyznaczy n-ty wyraz tego ciągu dla danego n.

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

4.4. Wskazówki i odpowiedzi do ćwiczeń

Wskazówka do zadania 4.13. Na przykład dla $w(x) = x^4 + 4x^2 - x + 2$ oraz $v(x) = x^4 + x^3 + 10$ zachodzi $u(x) = x^8 + x^7 + 4x^6 + 3x^5 + 11x^4 + 2x^3 + 40x^2 - 10x + 20$. □

Wskazówka do zadania 4.17. Należy wyprowadzić wzory (rekurencyjne) na średnią arytmetyczną i wariancję k początkowych elementów ciągu, zależne od elementu x_k oraz średniej arytmetycznej i wariancji elementów $(x_1, \ldots, x_{k-1})$. □

Wskazówka do zadania 4.18. Jeśli mamy $a \leqslant b \leqslant c$, to warunkiem koniecznym i dostatecznym tego, by dało się skonstruować trójkąt o bokach długości a, b, c, jest $c < a + b$. Uwaga. Nie trzeba sprawdzać wszystkich możliwych trójek! □

Wskazówka do zadania 4.21. Skorzystaj z pomocniczej n-elementowej tablicy liczb całkowitych. □

Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

5. Rekurencja. Abstrakcyjne typy danych

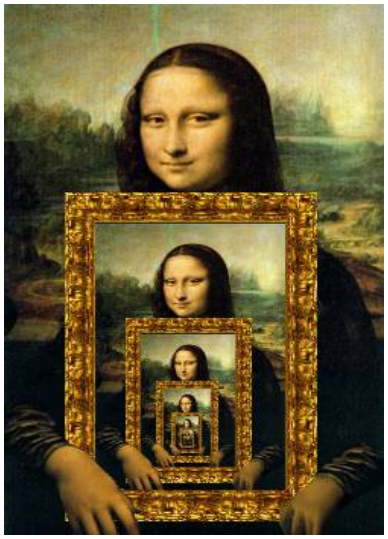
Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

5.1. Rekurencja

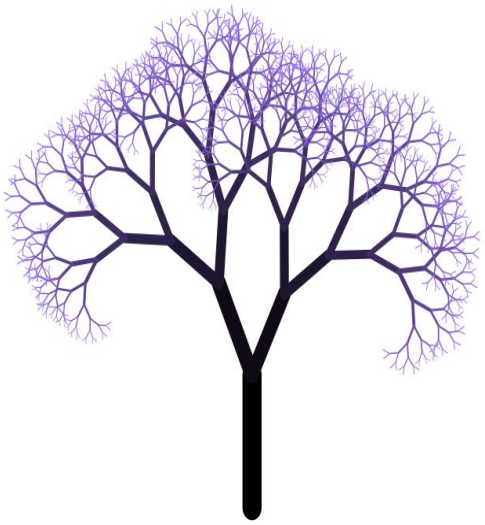
Z *rekurencją* (bądź rekursją, ang. *recursion*) mamy do czynienia wtedy, gdy w definicji pewnego obiektu pojawia się odniesienie do niego samego.

Rozważmy najpierw rys. 5.1. 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.



Rys. 5.1. Rekurencyjna Mona Lisa

A teraz popatrzmy na rys. 5.2 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.



Rys. 5.2. Fraktalne drzewo

W przypadku języków programowania mówimy o rekurencji, *gdy funkcja wywołuje samą siebie*. Czyniąc tak jednak we „frywolny” sposób spowodowalibyśmy zawieszenie kom-

putera. Istotną cechą rekurencji jest więc dodatkowo *warunek stopu*, czyli zdefiniowanie przypadku, w którym rekurencyjne wywołanie zostaje przerwane.

Zapamiętaj

Każde wywołanie funkcji (także 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.

5.1.1. Przykład: silnia

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

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

Funkcję służącą do obliczenia silni można napisać opierając się wprost na powyższym wzorze.

```
1 int silnia(int n)
2 {
3     assert(n >= 0); // upewnijmy się ...
4     if (n == 0) return 1;
5     else return n*silnia(n-1);
6 }
```

Dla porównania przyjrzyjmy się, jakby mogła wyglądać analogiczna funkcja niekorzystająca z dobrodziejstw rekurencji.

```
1 int silnia2(int n)
2 {
3     assert(n >= 0);
4     int w = 1;
5     for (int i=1; i<=n; ++i)
6         w *= i;
7     return w;
8 }
```

To, którą wersję uważamy za czytelniejszą, zależy oczywiście od nas samych.

5.1.2. Przykład: NWD

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

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

Definicja ta przekłada się bezpośrednio na następujący kod w języku C++.

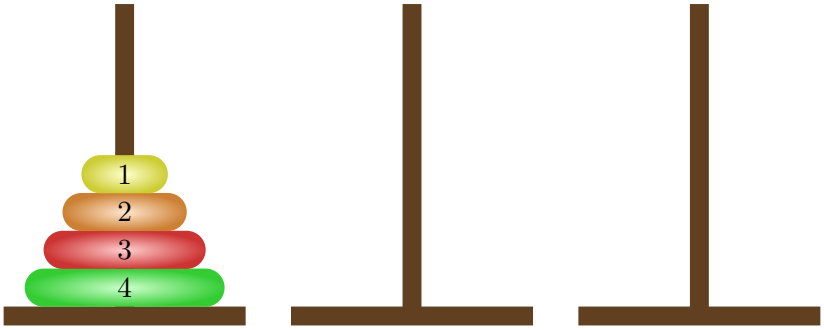
```
int nwd(int n, int m)
{
    assert(0 <= n && n <= m);
    if (n == 0) return m;
    else return nwd(m % n, n);
}
```

I w tym przypadku łatwo jest napisać równoważną wersję nierekurencyjną:

```
int nwd(int n, int m)
{
    assert(0 <= n && n <= m);
    while (n != 0) {
        int c = n;
        n = m % n;
        m = c;
    }
    return m;
}
```

5.1.3. Przykład: wieże z Hanoi

A teraz zabawna łamigłówka. Załóżmy, że danych jest n *krążków* o różnych średnicach oraz trzy *słupki*. Początkowo wszystkie krążki znajdują się na słupku nr 1, w kolejności od największego (dół) do najmniejszego (góra). Sytuację wyjściową dla $n = 4$ przedstawia rys. 5.3.



Rys. 5.3. Sytuacja wyjściowa dla 4 krążków w problemie wież z Hanoi.

Cel: przeniesienie wszystkich krążków na słupek nr 3.

Zasada #1: krążki przenosimy pojedynczo.

Zasada #2: krążek o większej średnicy nie może znaleźć się na krążku o mniejszej średnicy.

Zastanówmy się, jak by mógł wyglądać algorytm służący do rozwiązywania tego problemu. Niech będzie to funkcja `Hanoi(k, A, B, C)`, która przekłada `k` krążków ze słupka `A` na słupek `C` z wykorzystaniem słupka pomocniczego `B`, gdzie `A,B,C` $\in \{1, 2, 3\}$. Wykonywany ruch będzie wypisywany na ekranie.

Aby przenieść n klocków ze słupka `A` na `C`, należy przestawić $n-1$ mniejszych elementów na słupek pomocniczy `B`, przenieść n-ty krążek na `C` i potem pozostałe $n-1$ krążków przestawić z `B` na `C`. Jest to, rzecz jasna, sformułowanie zawierające elementy rekurencji.

Wywołanie początkowe: `Hanoi(n, 1, 2, 3)`.

Rozwiązanie w języku C++:

```
void Hanoi(int k, int A, int B, int C)
{
    if (k>0) // warunek stopu
    {
        Hanoi(k-1, A, C, B);
        cout << A << "->" << C << endl;
        Hanoi(k-1, B, A, C);
    }
}
```

Zauważmy, że stworzenie wersji nierekurencyjnej powyższej funkcji byłoby dla nas bardzo trudne – w przeciwieństwie do przykładów z silnią i NWD, gdzie rekurencja łatwo się „rozplątywała". Podczas bardziej zaawansowanego kursu algorytmiki dowiemy się, że każdą funkcję stosującą rekurencję możemy zapisać w postaci iteracyjnej przy użyciu pomocniczej struktury danych typu stos. Często jest to jednak zadanie nietrywialne i wynik pozostawia wiele do życzenia pod względem czytelności.

Ciekawostka

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

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

Można pokazać, że jest to optymalna liczba przestawień.

Rozwiążmy powyższe równanie rekurencyjne, aby uzyskać postać jawną rozwiązania:

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

Zauważmy, że $L(n)+1$ tworzy ciąg geometryczny o ilorazie 2. Zatem

$$\begin{aligned} L(1)+1 &= 2, \\ L(2)+1 &= 4, \\ L(3)+1 &= 8, \\ &\vdots \\ L(n)+1 &= 2^n. \end{aligned}$$

Więc

$$L(n) = 2^n - 1.$$

Zagadka Wież z Hanoi stała się znana w XIX wieku dzięki matematykowi E. Lucasowi. Jak głosi tybetańska legenda, mnisi w świątyni Brahmy rozwiązują tę łamigłówkę przesuwając 64 złote krążki. Podobno, gdy skończą oni swe zmagania, nastąpi koniec świata. Zakładając jednak, że nawet gdyby wykonanie jednego ruchu zajmowało tylko 1 sekundę, to na pełne rozwiązanie i tak potrzeba $2^{64}-1 = 18446744073709551615$ sekund, czyli około 584542 miliardów lat. To ponad 400 razy dłużej niż szacowany wiek Wszechświata!

5.1.4. Przykład (niemądry): liczby Fibonacciego

Jako ostatni problem rozpatrzmy równanie, do którego doszedł Leonard z Pizy (1202), rozważając rozmnażanie się królików.

Sformułował on następujący uproszczony model szacowania liczby par w populacji. Na początku mamy daną jedną parę królików. Każda para osiąga płodność po upłynięciu jednej jednostki czasu od narodzenia. Z każdej płodnej pary rodzi się w kolejnej chwili jedna para potomstwa.

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

$$\begin{cases} F_0 &= 1, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \text{ dla } n > 1. \end{cases}$$

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

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

```
1 int fibrek(int n)
2 {
3     if (n > 1) return fibrek(n-1)+fibrek(n-2);
4     else return 1;
5 }
```

Powyższy algorytm rekurencyjny jest jednak bardzo nieefektywny! Przypatrzmy się, jak przebiegają obliczenia dla wywołania `fibrek(5)`:

```
fibrek(5);
   fibrek(4);
      fibrek(3);
         fibrek(2);
            fibrek(1); #
            fibrek(0); #
         fibrek(1); #
      fibrek(2);
         fibrek(1); #
         fibrek(0); #
   fibrek(3);
      fibrek(2);
         fibrek(1); #
         fibrek(0); #
      fibrek(1); #
```

Większość wartości jest niepotrzebnie liczona wielokrotnie.

Rozważmy dla przykładu, ile potrzebnych jest operacji arytmetycznych (dodawanie) do znalezienia wartości F_n za pomocą powyższej funkcji. Liczbę tę można opisać równaniem:

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

Rozpatrując kilka kolejnych wyrazów, zauważamy, że $L(n) = F_{n-1} - 1$ dla $n > 0$.

Okazuje się, że dla dużych n, $L(n)$ jest proporcjonalne do c^n dla pewnej stałej c. Zatem liczba kroków potrzebnych do uzyskania n-tej liczby Fibonacciego algorytmem rekurencyjnym rośnie wykładniczo, tj. bardzo (naprawdę bardzo) szybko! 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, że implementacja stosująca rekurencję wprost jest wysoce nieefektywne. Przykład ten jest jednak bardzo pouczający dla matematyków – nie zawsze „przepisanie" definicji interesującego nas obiektu będzie prowadziło do dobrej w praktyce metody obliczeniowej[1]

Zatem w tym przypadku powinniśmy użyć raczej „zwykłego" rozwiązania iteracyjnego, które – co istotne – wykorzystuje tylko $O(n)$ operacji arytmetycznych.

Przykładowy pełny program wyznaczający n-tą liczbę Fibonacciego (dla danego z klawiatury n) mógłby wyglądać następująco.

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

int pobierz_n()
{
    int n;
    do
    {
        cout << "Podaj n: ";
        cin >> n;
        if (n < 0) {
            cout << "Niepoprawne dane! Spróbuj ponownie." << endl;
        }
    while (n < 0);

    return n; // tutaj na pewno n >= 0
}

int fibiter(int n)
{
    if (n <= 1) return 1; // tu wynik jest znany

    int k = 1;
    int Fk_poprz = 1;   // F_{k-1}
    int Fk_teraz = 1;   // F_k

    do {
        int nast = Fk_teraz+Fk_poprz; // policz kolejny wyraz
        k++;

        Fk_poprz = Fk_teraz;   // F_{k-1} (już dla nowego k)
        Fk_teraz = nast;       // F_k
    }
    while (k <= n);

    return Fk_teraz;
}


int main()
{
```

[1]Inny przykład: mimo że rozwiązanie układu równań postaci $Ax = b$ możemy poprawnie wyrazić za pomocą $x = A^{-1}b$, nigdy implementacja komputerowa nie powinna odwracać macierzy wprost. Podyktowane jest to tzw. stabilnością numeryczną metody – innymi słowy, z powodu niedoskonałości arytmetyki zmiennoprzecinkowej komputera, taki sposób obliczeń generuje często zbyt niedokładny wynik.

```
43      int n = pobierz_n();
44      cout << "F(" << n << ") = " << fibiter(n) << endl;
45      return 0;
46  }
```

Zwróćmy uwagę na podział programu na czytelne moduły (tj. funkcje), z których każdy służy do przeprowadzenia ściśle określonej czynności.

5.2. Podstawowe abstrakcyjne typy danych

W tym podrozdziale omówimy cztery następujące abstrakcyjne typy danych:

1. stos (rozdz. 5.2.1),
2. kolejkę (rozdz. 5.2.2),
3. kolejkę priorytetową (rozdz. 5.2.3),
4. słownik (rozdz. 5.2.4).

Zapamiętaj

Abstrakcyjny typ danych (ATD) umożliwia przechowywanie i organizowanie w specyficzny dla siebie sposób informacji danego, ustalonego typu.

Każdy abstrakcyjny typ danych określony jest za pomocą:

1. dziedziny, tj. typu zmiennych, które można w nim przechowywać,
2. właściwego sobie dopuszczalnego zbioru operacji, które można na nim wykonać.

Co ważne, definicje wszystkich abstrakcyjnych typów danych *abstrahują od sposobu ich implementacji*. Każdy zaprezentowany przez nas ATD może zostać zaprogramowany na wiele — z punktu widzenia realizowanej funkcjonalności — równoważnych sposobów. Jednakże owe implementacje będą różnić się m.in. efektywnością obliczeniową i pamięciową, które — jak wiemy — mają kluczowe znaczenie w praktyce. W trakcie kolejnych zajęć będziemy rzecz jasna starać się poszukiwać rozwiązań optymalnych. Najprostsze, tablicowe implementacje, pozostawiamy jako ćwiczenie.

Informacja

Dla uproszczenia umawiamy się, że każdy prezentowany abstrakcyjny typ danych będzie służył do przechowywania pojedynczych wartości całkowitych, tj. danych typu `int`.

Jako jeden z najbardziej przyjaznych matematykowi przykładów ATD można wymienić znany z teorii mnogości *zbiór* (u nas zwany *słownikiem*, zob. rozdz. 5.2.4). Otóż zbiór może przechowywać różne (odmienne) wartości (zgodnie z umową — całkowite). Dopuszczalne operacje wykonywane na nim to m.in. sprawdzenie, czy dany element znajduje się w zbiorze, dodanie elementu do zbioru i usunięcie elementu ze zbioru. Zauważmy, że wszystkie

inne operacje teoriomnogościowe (np. suma, iloczyn, różnica zbiorów, czy dopełnienie) mogą zostać zrealizowane za pomocą trzech wymienionych (aczkolwiek w sposób daleki od optymalnego pod względem szybkości działania[2]).

Zadanie

Zastanów się, w jaki sposób można najprościej i najefektywniej zaimplementować ATD typu zbiór (słownik) o elementach z $\{0, 1, \ldots, 100\}$, a w jaki o dowolnych elementach z $\mathbb{Z}$ z użyciem tablicy. Jaką pesymistyczną złożoność obliczeniową mają w każdym przypadku Twoje implementacje trzech wyżej wymienionych operacji? Czy da się je zrealizować w lepszy sposób?

5.2.1. Stos

Stos (ang. *stack*) jest abstrakcyjnym typem danych typu LIFO (ang. *last-in-first-out*), który udostępnia przechowywane weń elementy w kolejności odwrotnej niż ta, w której były one zamieszczane.

Stos udostępnia trzy operacje:

- *push* (umieść) — wstawia element na stos,
- *pop* (zdejmij) — usuwa i zwraca element ze stosu (LIFO!),
- *empty* (czy pusty) — sprawdza, czy na stosie nie znajdują się żadne elementy.

Przykład użycia stosu.

```
int main()
{
    Stos s = tworzStos();

    push(s, 3);
    push(s, 7);
    push(s, 5);

    while (!empty(s))
       cout << pop(s);
    // wypisze na ekran 573

    kasujStos(s);

    return 0;
}
```

[2] Więcej na ten temat dowiemy się z wykładu z *Algorytmów i struktur danych*. Zob. także książkę N. Wirtha „Algorytmy + struktury danych = programy", Wyd. WNT, Warszawa, 2001.

Zadanie

Zaimplementuj operacje wykorzystywane w powyższej funkcji `main()`. Stos reprezentuj w postaci struktury zawierającej dynamicznie alokowaną tablicę, na przykład:

```
struct Stos {
   int* wartosci;
   int n; // aktualny rozmiar tablicy
   int g; // indeks ,,wierzchołka'' stosu
};
```

Zaproponuj różne implementacje, m.in. przypadek, n zmienia się za każdym razem, gdy wykonujemy operacje *push* i *pop* oraz gdy n podwaja się, gdy dopiero zachodzi taka potrzeba. Jaką złożoność obliczeniową mają ww. operacje w pesymistycznym, a jaką w optymistycznym przypadku?

5.2.2. Kolejka

Kolejka (ang. *queue*) to ATD typu FIFO (ang. *first-in-first-out*). Podstawową własnością kolejki jest to, że pobieramy z niej elementy w takiej samej kolejności (a nie w odwrotnej jak w przypadku stosu), w jakiej były one umieszczane.

Kolejka udostępnia następujące trzy operacje:

- *enqueue* (umieść) — wstawia element do kolejki,
- *dequeue* (wyjmij) — usuwa i zwraca element z kolejki (FIFO!),
- *empty* (czy pusty).

Przykład użycia kolejki.

```
int main()
{
    Kolejka k = tworzKolejke();

    enqueue(k, 3);
    enqueue(k, 7);
    enqueue(k, 5);

    while (!empty(k))
       cout << dequeue(k);
    // wypisze na ekran tym razem 375

    kasujKolejke(k);

    return 0;
}
```

Zadanie

Zaimplementuj operacje wykorzystywane w powyższej funkcji `main()`. Kolejkę reprezentuj w postaci struktury zawierającej dynamicznie alokowaną tablicę. Jaką złożoność obliczeniową mają ww. operacje w pesymistycznym, a jaką w optymistycznym przypadku?

5.2.3. Kolejka priorytetowa

Kolejka priorytetowa (ang. *priority queue*) to abstrakcyjny typ danych typu LPF (ang. *lowest-priority-first*). Może ona służyć do przechowywania elementów, na których została określona pewna relacja porządku liniowego <=[3]. Wydobywa się z niej elementy poczynając od tych, które mają najmniejszy priorytet (są minimalne względem relacji <=). Jeśli w kolejce priorytetowej znajdują się elementy o takich samych priorytetach, obowiązuje dla nich kolejność FIFO, tak jak w przypadku zwykłej kolejki.

Omawiany ATD udostępnia trzy operacje:

- *insert* (włóż) — wstawia element na odpowiednie miejsce kolejki priorytetowej.
- *pull* (pobierz) — usuwa i zwraca element minimalny,
- *empty* (czy pusty).

Przykład użycia kolejki priorytetowej.

```
int main()
{
    KolejkaPriorytetowa k = tworzKolejkePriorytetowa();

    insert(k, 3);
    insert(k, 7);
    insert(k, 5);

    while (!empty(k))
        cout << pull(k);
    // wypisze na ekran tym razem 357

    kasujKolejkePriorytetowa(k);

    return 0;
}
```

Zadanie

Zaimplementuj operacje wykorzystywane w powyższej funkcji `main()`. Kolejkę priorytetową reprezentuj w postaci struktury zawierającej dynamicznie alokowaną tablicę. Jaką złożoność obliczeniową mają ww. operacje w pesymistycznym, a jaką w optymistycznym przypadku?

5.2.4. Słownik

Słownik (ang. *dictionary*) to abstrakcyjny typ danych, który udostępnia trzy operacje:

- *search* (znajdź) — sprawdza, czy dany element znajduje się w słowniku,
- *insert* (dodaj) — dodaje element do słownika, o ile taki już się w nim nie znajduje,
- *remove* (usuń) — usuwa element ze słownika.

Dla celów pomocniczych rozważać możemy także następujące działania na słowniku:

- *print* (wypisz),
- *removeAll* (usuń wszystko).

[3] W naszym przypadku, jako że umówiliśmy się, iż omawiane ATD przechowują dane typu `int`, najczęściej będzie to po prostu zwykły porządek $\leqslant$.

Informacja

Czasem zakłada się, że na przechowywanych elementach została określona relacja porządku liniowego `<=`. Wtedy możemy tworzyć bardziej efektywne implementacje słownika, np. z użyciem drzew binarnych (zob. rozdz. 7) albo algorytmu wyszukiwania binarnego (połówkowego) w przypadku tablic.

Pesymistyczna złożoność obliczeniowa algorytmu wyszukiwania binarnego wynosi zaledwie $O(\log n)$ dla danej uporządkowanej tablicy `t` rozmiaru n. Metoda ta bazuje na założeniu, że poszukiwany element `x`, jeśli oczywiście znajduje się w tablicy, musi być ulokowany na pozycji (indeksie) ze zbioru `{l,l+1,...,p}`. Na początku ustalamy oczywiście `l=0` i `p=n-1`. W każdej kolejnej iteracji znajdujemy „środek" owego obszaru poszukiwań, tzn. `m=(p+l)/2` oraz sprawdzamy, jak ma się `t[m]` do `x`. Jeśli okaże się, że jeszcze nie trafiliśmy na właściwe miejsce, to wtedy zawężamy (o co najmniej połowę) nasz obszar modyfikując odpowiednio wartość `l` bądź `p`.

```
bool search(int x, int* t, int n)
{
   if (n <= 0) return false;

   // wyszukiwanie binarne (połówkowe)
   int l = 0;      // początkowy obszar poszukiwań
   int p = n-1;    //          - cała tablica
   while (l <= p)
   {
      int m = (p+l)/2; // "środek" obszaru poszukiwań

      if (x == t[m]) return true; // znaleziony - koniec
      else if (x > t[m]) l = m+1; // zawężamy obszar poszukiwań
      else               p = m-1; // jw.
   }

   return false; // nieznaleziony
}
```

Powyższą funkcję łatwo zmodyfikować, by zwracała indeks, pod którym znajduje się znaleziony element bądź jakąś wyróżnioną wartość (np. -1), gdy elementu nie ma w tablicy (potrafisz?). Spróbuj także samodzielnie zaimplementować rekurencyjną wersję tego algorytmu.

Przykład użycia słownika.

```
int main()
{
   Slownik s = tworzSlownik();

   int i, j;
   cout << "Podaj wartości, które maja się znaleźć w słowniku. ";
   cout << "Wpisz wartość ujemną, by zakończyć wstawianie. ";

   cin >> i;
   while (i >= 0) {
      insert(s, i);
      cin >> i;
   }

```

```
15      cout << "Podaj wartości, które chcesz znaleźć w słowniku. ";
16      cout << "Wpisz wartość ujemną, by zakończyć wyszukiwanie. ";
17
18      cin >> i;
19      while (i >= 0) {
20         if (search(s, i)) {
21            cout << "Jest!" << endl;
22            cout << "Usunąć? 0 == NIE." << endl;
23            cin >> j;
24            if (j != 0)
25               remove(s, i);
26         }
27         else
28            cout << "Nie ma!" << endl;
29         cin >> i;
30      }
31
32      kasujSlownik(s);
33
34      return 0;
35  }
```

5.3. Ćwiczenia

Zadanie 5.1. Zdefiniuj strukturę `Wektor2d` o dwóch składowych typu `double`. Utwórz funkcję `dlugosc()`, która dla danego obiektu typu `Wektor2d` zwraca jego normę euklidesową. Dalej, stwórz funkcję `inssort()`, która implementuje algorytm sortowania przez wstawianie danej tablicy o elementach typu `Wektor2d*` (zamianę elementów implementujemy wprost na wskaźnikach). Napisz też funkcję `main()`, w której wszystko zostanie dokładnie przetestowane.

Zadanie 5.2. Napisz funkcję `horners()`, która wyznacza wartości wszystkich k podanych wielomianów w podanym punkcie. Wielomian reprezentujemy przy użyciu struktury `Wielomian` (składającej się z tablicy `w` o elementach typu `double` oraz stopnia wielomianu `n`). Deklaracja funkcji do napisania: `double* horners(Wielomian* W, int k, double x)`.

Zadanie 5.3 (MD)**.** Zdefiniuj strukturę `Zbior`, która reprezentuje podzbiór zbioru liczb naturalnych (pola: tablica `e` o elementach typu `int` oraz liczba elementów `n`).

Napisz funkcję `void dodaj_element(Zbior& zb, int e)`, która doda do wskazanego zbioru element `e` (jeżeli element już znajduje się w zbiorze, funkcja nie powinna robić nic).

Ponadto napisz funkcję `Zbior przeciecie(Zbior* zbiory, int k)`, która wyznaczy przecięcie wszystkich zbiorów zadanych w postaci `k`-elementowej tablicy `zbiory`, a także `Zbior suma(Zbior* zbiory, int k)`, która oblicza sumę podanych zbiorów.

Zadanie 5.4 (MB)**.** Zdefiniuj strukturę `Ulamek` o dwóch składowych `licznik`, `mianownik` typu `int` – reprezentuje ona liczbę wymierną. Napisz funkcje służące do dodawania, odejmowania, mnożenia i dzielenia ułamków, każdą postaci `Ulamek nazwa_funkcji(Ulamek x, Ulamek y)`. Po wykonaniu każdej z operacji ułamki powinny być odpowiednio znormalizowane; patrz algorytm NWD (zalecana metoda: przez dodatkową funkcję `normalizuj()`).

Następnie napisz funkcje, które mając dane tablice ułamków wyznaczą ich skumulowane minimum, maksimum, sumę i iloczyn. Napisz też funkcje, które wyznaczają sumę, różnicę, iloczyn i iloraz ułamków z dwóch zadanych tablic (odpowiednio zwektoryzowane). Dodatkowo, możesz stworzyć funkcję, która sortuje ułamki względem ich wartości oraz wypisuje ułamki na ekranie.

Zadanie 5.5 (MB)**.** Dana jest struktura zdefiniowana jako `struct Naukowiec {int *cytowania; int n; };`. Zakładamy, że `cytowania` są zawsze posortowane nierosnąco.

Napisz funkcję, która dodaje nową publikację z podaną liczbą cytowań do tablicy `cytowania`, np. dla tablicy $(5, 4, 4, 1)$ po dodaniu 3 otrzymujemy $(5, 4, 4, 3, 1)$ (zwracamy uwagę na poprawną alokację nowej tablicy oraz dealokację starej).

Utwórz także funkcję `h()`, która wyznacza indeks Hirscha danego naukowca.

Dodatkowo, zaimplementuj następujące funkcje, które na wejściu otrzymują tablicę naukowców: (a) wyznaczanie naukowca o największym indeksie Hirscha, (b) sortowanie naukowców pod względem indeksu h (od najlepszego do najgorszego).

Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

6. Napisy i macierze

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

6.1. Własne biblioteki funkcji

Jako że program czytany jest (przez nas jak i przez kompilator) od góry do dołu, *definicja każdej funkcji musi się pojawić przed jej pierwszym użyciem*. W niektórych przypadkach może się jednak zdarzyć, że lepiej będzie (np. dla czytelności) umieścić definicję funkcji w jakimś innym miejscu. Można to zrobić pod warunkiem, że obiektom z niej korzystającym zostanie udostępniona jej *deklaracja*.

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

Informacja

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

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

Przykład:

```
1  #include <iostream>
2  using namespace std;
3
4  double kwadrat(double); // deklaracja - tu jest średnik!
5
6  int main()
7  {
8     double x = kwadrat(2.0);
9     cout << x                          << endl;
10    cout << kwadrat(x)+kwadrat(8.0)  << endl;
11    cout << kwadrat(kwadrat(0.5))    << endl;
12    return 0;
13 }
14
15 double kwadrat(double x) // definicja - tu nie ma średnika!
16 {
17    return x*x;
18 }
```

■

Zapamiętaj

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

Zbiór różnych funkcji, które warto mieć „pod ręką”, możemy umieścić w osobnych plikach, tworząc tzw. *bibliotekę* funkcji. Dzięki temu program stanie się bardziej czytelny,

łatwiej będziemy mogli zapanować nad jego złożonością, a ponadto otrzymamy sposobność wykorzystania pewnego kodu ponownie w przyszłości. Raz napisany i przetestowany zestaw funkcji może oszczędzić sporo pracy.

Aby stworzyć bibliotekę funkcji, tworzymy najczęściej dwa dodatkowe pliki.

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

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

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

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

Informacja

Najlepiej, gdy wszystkie pliki źródłowe i nagłówkowe będą znajdować się w tym samym katalogu.

Przykład 1. Dla ilustracji przyjrzymy się, jak wyglądałby program korzystający z biblioteki zawierającej dwie następujące funkcje. Niech

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

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

będą funkcjami takimi, że

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

oraz

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

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

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

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

Zapamiętaj

Dyrektywa

```
#pragma once
```

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

Ciekawostka

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

```
#ifndef __PODRECZNA_H
#define __PODRECZNA_H

// deklaracje....

#endif
```

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

```
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 }
```

3. Gdy biblioteka jest gotowa, można przystąpić do napisania funkcji głównej. 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)
7  {
8     int x, y;
```

```
9
10      cout << "Podaj dwie liczby. ";
11      cin  >> x >> y; // wczytaj z klawiatury
12
13      cout << "Max=" << max(x,y) << endl;
14      cout << "Min=" << min(x,y) << endl;
15
16      return 0;
17  }
```

■

6.2. Napisy (ciągi znaków)

Do tej pory nie zastanawialiśmy się, 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 nieliczbowym charakterze.

6.2.1. Kod ASCII

Pojedyncze znaki drukowane przechowywane są najczęściej w postaci obiektów typu `char` (ang. *character*). Oczywiście pamiętamy, że jest to po prostu typ, służący do przechowywania 8 bitowych *liczb* (sic!) całkowitych.

Istnieje ogólnie przyjęta umowa (standard), że liczbom z zakresu 0–127 *odpowiadają* ściśle określone znaki, tzw. kod ASCII (ang. *American Standard Code for Information Interchange*). Zestawiają je tablice 6.1–6.3.

Tab. 6.1. Kody 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 (\t)	25	19	End of Medium
10	0A	Line Feed (\n)	26	1A	Substitute
11	0B	Vertical Tab	27	1B	Escape
12	0C	Form Feed	28	1C	File Separator
13	0D	Carriage Return (\r)	29	1D	Group Separator
14	0E	Shift Out	30	1E	Record Separator
15	0F	Shift In	31	1F	Unit Separator

Tab. 6.2. Kody ASCII cz. II

DEC	HEX	Znaczenie	DEC	HEX	Znak	DEC	HEX	Znaczenie
32	20	Spacja	48	30	0	64	40	@
33	21	!	49	31	1	65	41	A
34	22	”	50	32	2	66	42	B
35	23	#	51	33	3	67	43	C
36	24	$	52	34	4	68	44	D
37	25	%	53	35	5	69	45	E
38	26	&	54	36	6	70	46	F
39	27	’	55	37	7	71	47	G
40	28	(	56	38	8	72	48	H
41	29	)	57	39	9	73	49	I
42	2A	*	58	3A	:	74	4A	J
43	2B	+	59	3B	;	75	4B	K
44	2C	,	60	3C	<	76	4C	L
45	2D	-	61	3D	=	77	4D	M
46	2E	.	62	3E	>	78	4E	N
47	2F	/	63	3F	?	79	4F	O

Tab. 6.3. Kody ASCII cz. III

DEC	HEX	Znak	DEC	HEX	Znaczenie	DEC	HEX	Znak
80	50	P	96	60	`	112	70	p
81	51	Q	97	61	a	113	71	q
82	52	R	98	62	b	114	72	r
83	53	S	99	63	c	115	73	s
84	54	T	100	64	d	116	74	t
85	55	U	101	65	e	117	75	u
86	56	V	102	66	f	118	76	v
87	57	W	103	67	g	119	77	w
88	58	X	104	68	h	120	78	x
89	59	Y	105	69	i	121	79	y
90	5A	Z	106	6A	j	122	7A	z
91	5B	[	107	6B	k	123	7B	{
92	5C	\	108	6C	l	124	7C	|
93	5D	]	109	6D	m	125	7D	}
94	5E	^	110	6E	n	126	7E	∼
95	5F	_	111	6F	o	127	7F	Delete

Na szczęście w języku C++ nie musimy pamiętać, która liczba odpowiada jakiemu symbolowi. Aby uzyskać wartość liczbową symbolu drukowanego, należy ująć go w *apostrofy*.

```
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 kodów ASCII, warto zanotować następujące prawidłowości.

- Mamy następujący porządek leksykograficzny:
  `'A'` $<$ `'B'` $<$ ... $<$ `'Z'` $<$ `'a'` $<$ `'b'` $<$ ... $<$ `'z'`.
- Symbol cyfry `c` $\in \{0, 1, \ldots, 9\}$ można uzyskać za pomocą wyrażenia `'0'+c`.
- Kod ASCII `n`-tej wielkiej litery alfabetu łacińskiego to `'A'+n-1`.
- Kod ASCII `n`-tej małej litery alfabetu łacińskiego to `'a'+n-1`.
- Zamiana litery `l` na małą literę następuje za pomocą operacji `l+32`.
- Zamiana litery `l` na wielką literę następuje za pomocą operacji `l-32`.

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`.

Ciekawostka

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 podczas tego kursu.

Informacja

Biblioteka `<cctype>` zawiera kilka funkcji przydatnych do sprawdzania, czy np. dany znak jest literą alfabetu łacińskiego, cyfrą „białym znakiem" (spacją, tabulacją itp.) itd. Zob. więcej: `http://www.cplusplus.com/reference/clibrary/cctype/`.

6.2.2. Reprezentacja napisów (ciągów znaków)

Wiemy już, w jaki sposób reprezentować pojedyncze znaki.

Zapamiętaj

Najprostszym sposobem reprezentowania ciągów symboli drukowanych, czyli *napisów*, są tablice elementów typu `char` *zakończone* umownie bajtem (liczbą całkowitą) o wartości *zero* (znak `'\0'`).

Napisy można utworzyć używając cudzysłowu (`"..."`). Takie napisy jednak są tablicami tylko do odczytu. Nie wiadomo bowiem, w jakim miejscu w pamięci zostaną one umieszczone i czy przypadkiem nie są one wykorzystywane w wielu miejscach tego samego programu.

```
char* napis1 = "Pewien napis."; // 13 znaków + bajt 0
// *prawie* równoważnie:
char napis2[14] = // tablica o ustalonym rozmiarze
   { '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
```

6.2.3. Operacje na napisach

Jako że napisy są zwykłymi tablicami liczb całkowitych, implementacja podstawowych operacji na nich jest dość prosta. W niniejszym paragrafie rozważymy kilka z nich, resztę pozostawiając jako ćwiczenie.

Najpierw przyjrzymy się wypisywaniu.

```
char* napis = "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;
}
```

Zapamiętaj

Jeśli mamy zaalokowaną tablicę typu char o identyfikatorze `napis` mogącą przechowywać maksymalnie $n+1$ znaków (dodatkowe miejsce na bajt zerowy), to wywołanie

```
cin >> napis;
```

wczytuje tylko jeden wyraz (uwaga, by nie wprowadzić więcej niż n znaków!). Jeśli chcemy wczytać cały wiersz, musimy wywołać:

```
cin.getline(napis, n);
```

Długość napisu można sprawdzić w następujący sposób.

```
int dlug(char* napis)
{
   // Zwraca 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
}
```

6.2.4. Biblioteka <cstring>

Biblioteka **<cstring>** definiuje wiele funkcji służących do przetwarzania ciągów znaków[1]. Wybrane narzędzia zestawia tab. 6.4.

Zadanie

Spróbuj napisać własną implementację ww. funkcji samodzielnie.

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

Tab. 6.4. Wybrane funkcje biblioteki `<cstring>`

Deklaracja	Opis
`int strlen(char* nap);`	Zwraca długość napisu.
`char* strcpy(char* cel, char* zrodlo);`	Kopiuje napisy. Uwaga: długość tablicy `cel` nie może być mniejsza niż długość napisu `zrodlo`+1.
`char* strcat(char* cel, char* zrodlo);`	Łączy napisy `cel` i `zrodlo`.
`int strcmp(char* nap1, char* nap2);`	Porównuje napisy. Zwraca 0, jeśli są identyczne. Zwraca wartość dodatnią, jeśli `nap1` jest większy (w porządku leksykograficznym) niż `nap2`.
`char* strchr(char* nap, char znak);`	Zwraca podnapis rozpoczynający się od pierwszego wystąpienia danego znaku.
`char* strrchr(char* nap, char znak);`	Zwraca podnapis rozpoczynający się od ostatniego wystąpienia danego znaku.
`char* strstr(char* nap1, char* nap2);`	Zwraca podnapis rozpoczynający się od pierwszego wystąpienia podnapisu `nap2` w napisie `nap1`.

6.3. Reprezentacja macierzy

Dana jest macierz określona nad pewnym zbiorem

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix}$$

gdzie n — liczba wierszy, m — liczba kolumn.

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

– jako tablice tablic,

– jako tablice jednowymiarowe.

My na zajęciach wybieramy sposób pierwszy – zapewnia on bardzo wygodny dostęp do poszczególnych elementów, jednak cechuje się nieco zawiłym sposobem tworzenia i usuwania interesujących nas obiektów:

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

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....

```

```
13 for (int i=0; i<n; ++i)
14    delete [] A[i];
15 delete [] A;
```

Z drugiej strony, macierze możemy reprezentować za pomocą jednowymiarowych tablic. Łatwo się je tworzy, jednak odwoływanie się do elementów jest nieco zawiłe.

```
1 int n = ...; // liczba wierszy
2 int m = ...; // liczba kolumn
3 typ* A; // jednowymiarowa tablica
4
5 A = new typ[n*m];
6
7 // teraz np. A[1+3*n] to element w II wierszu i IV kolumnie....
8
9 delete [] A;
```

Często (i tak jest w powyższym przypadku) zakłada się, że elementy macierzy przechowywane są kolumnowo (najpierw cała I kolumna, potem II itd.).

6.4. Przykładowe algorytmy z wykorzystaniem macierzy

Omówmy kilka przykładowych algorytmów:

1. mnożenie dowolnego wiersza przez stałą,
2. odejmowanie pewnego wiersza przez inny pomnożony przez stałą,
3. zamiana dwóch kolumn,
4. wyszukanie wiersza, który ma największy co do modułu element w danej kolumnie,
5. dodawanie macierzy,
6. mnożenie macierzy,
7. rozwiązywanie układów równań liniowych metodą eliminacji Gaussa.

Będziemy rozpatrywać macierze o wartościach rzeczywistych (reprezentowanych jako tablice tablic o elementach typu `double`). Zauważ, że żadna z funkcji (o ile nie zaznaczono inaczej) nie „psuje” macierzy wejściowej, w większości przypadków zwracana jest nowa, dynamicznie alokowana macierz (oczywiście wtedy, gdy tego się spodziewamy). Takie postępowanie zależy od naszych potrzeb i specyfiki implementowanego programu, czasem być może prościej i wydajniej będzie z niego po prostu zrezygnować.

Zapamiętaj

Powyższe algorytmy będziemy implementować w postaci funkcji. Za każdym razem jako argumenty przekazujemy:

1. macierz (wskaźnik),
2. rozmiar macierzy,
3. parametry wykonywanej operacji.

Przydadzą nam się funkcje, które wyręczą nas w tworzeniu, kasowaniu i (w celach testowych) wypisywaniu na ekran macierzy.

```
double** tworzmacierz(int n, int m)
{
   assert(n > 0 && m > 0);
   double** W = new double*[n]; // tablica tablic
   for (int i=0; i<n; ++i)
      W[i] = new double[m];
   return W;
}
```

```
void kasujmacierz(double** W)
{
   assert(n > 0);
   for (int i=0; i<n; ++i)
      delete [] W[i];
   delete [] W;
}
```

```
void wypiszmacierz(double** W, int n, int m)
{
   assert(n > 0 && m > 0);

   // nie będzie zbyt pięknie, ale...
   for (int i=0; i<n; ++i) // dla każdego wiersza
   {
      for (int j=0; j<m; ++j) // dla każdej kolumny
         cout << '\t' << W[i][j];
      cout << endl;
   }
}
```

6.4.1. Mnożenie wiersza macierzy przez stałą

Załóżmy, że dana jest macierz rzeczywista A typu $n \times m$. Chcemy pomnożyć k-ty wiersz przez stałą rzeczywistą c. Oto przykładowa funkcja, która realizuje tę operację.

```
double** mnozwiersz(double** A, int n, int m, int k, double c)
{
   assert(k >= 0 && k < n && m > 0);
   double** B = tworzmacierz(n, m);

   for (int j=0; j<m; ++j)
   {
      for (int i=0; i<n; ++i)
         if (i == k)
            B[i][j] = A[i][j]*c;
         else
            B[i][j] = A[i][j]; // tylko przepisanie
   }

   return B;
}
```

6.4.2. Odejmowanie wiersza macierzy od innego pomnożonego przez stałą

Dana jest macierz rzeczywista A typu $n \times m$. Załóżmy, że chcemy odjąć od k-tego wiersza wiersz l-ty pomnożony przez stałą rzeczywistą c. Oto stosowny kod.

```
double** odskalwiersz(double** A, int n, int m, int k, int l,
    double c)
{
    assert(k >= 0 && k < n && l >= 0 && l < n && m > 0);

    double** B = tworzmacierz(n, m);

    for (int j=0; j<m; ++j)
    {
       for (int i=0; i<n; ++i)
          if (i == k)
             B[i][j] = A[i][j] - c*A[l][j];
          else
             B[i][j] = A[i][j];
    }

    return B;
}
```

6.4.3. Zamiana dwóch kolumn

Dana jest macierz A typu $n \times m$. Załóżmy, że chcemy zamienić k-tą kolumnę z l-tą.

```
double** zamienkol(double** A, int n, int m, int k, int l)
{
    assert(k >= 0 && k < m && l >= 0 && l < m && n > 0);

    double** B = tworzmacierz(n, m);

    for (int j=0; j<m; ++j)
    {
       for (int i=0; i<n; ++i)
       {
          if (j == k)
             B[i][j] = A[i][l];
          else if (j == l)
             B[i][j] = A[i][k];
          else
             B[i][j] = A[i][j];
       }
    }

    return B;
}
```

Ciekawostka — wersja alternatywna (modyfikująca macierz wejściową):

```
void zamienkol(double** A, int n, int m, int k, int l)
{
    assert(k >= 0 && k < m && l >= 0 && l < m && n > 0);

    for (int i=0; i<n; ++i)
    {
```

```
        double t = A[i][k];
        A[i][k] = A[i][l];
        A[i][l] = t;
    }
}
```

6.4.4. Wiersz z największym co do modułu elementem w kolumnie

Dana jest macierz rzeczywista A typu $n \times m$. Załóżmy, że chcemy znaleźć indeks wiersza, który ma największy co do modułu element w kolumnie k.

```
int wyszukajnajwkol(double** A, int n, int m, int k)
{
    assert(k >= 0 && k < m);

    int max = 0;
    for (int i=1; i<n; ++i)
    {
        if (fabs(A[i][k]) >= fabs(A[max][k]))
            max = i;
    }

    return max;
}
```

Przypomnijmy, że funkcja `double fabs(double x)` z biblioteki `<cmath>` zwraca wartość bezwzględną swego argumentu.

6.4.5. Dodawanie macierzy

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

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix} + \begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,m-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,m-1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}+b_{0,0} & a_{0,1}+b_{0,1} & \cdots & a_{0,m-1}+b_{0,m-1} \\ a_{1,0}+b_{1,0} & a_{1,1}+b_{1,1} & \cdots & a_{1,m-1}+b_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0}+b_{n-1,0} & a_{n-1,1}+b_{n-1,1} & \cdots & a_{n-1,m-1}+b_{n-1,m-1} \end{bmatrix}.$$

Oto kod funkcji służącej do dodawania macierzy.

```
double** dodaj(double** A, double** B, int n, int m)
{
    assert(n > 0 && m > 0);
    double** C = tworzmacierz(n, m);

    for (int j=0; j<m; ++j)
        for (int i=0; i<n; ++i)
            C[i][j] = A[i][j] + B[i][j];

    return C; /* nie zapomnij o zwolnieniu pamięci dalej! */
}
```

Ciekawostka

Zamiana kolejności pętli wewnętrznej i zewnętrznej

```
for (int i=0; i<n; ++i)
   for (int j=0; j<m; ++j)
      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* (ang. *cache*) komputera.

6.4.6. Mnożenie macierzy

Niech A — macierz typu $n \times m$ oraz B — macierz typu $m \times r$. Wynikiem mnożenia macierzy $A \cdot B$ jest macierz C typu $n \times r$, dla której

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj},$$

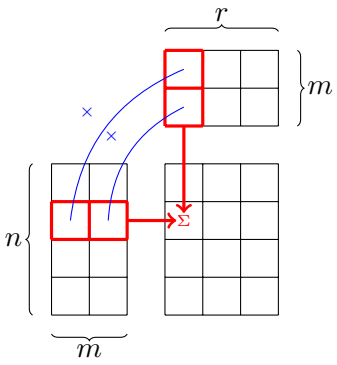
gdzie $0 \leqslant i < n$, $0 \leqslant j < r$ (por. rys. 6.1).

A oto kod funkcji służącej do wyznaczenia iloczynu dwóch macierzy.

```
1  double** mnoz(double** A, double** B, int n, int m, int r)
2  {
3     assert(n > 0 && m > 0 && r > 0);
4
5     double** C = tworzmacierz(n,r);
6
7     for (int j=0; j<r; ++j)
8     {
9        for (int i=0; i<n; ++i)
10       {
11          C[i][j] = 0.0;
12          for (int k=0; k<m; ++k)
13             C[i][j] += A[i][k] * B[k][j];
14       }
15    }
16
17   return C;  /* nie zapomnij o zwolnieniu pamięci dalej! */
18 }
```

Ciekawostka

Podany algorytm wykonuje nmr operacji mnożenia, co dla macierzy kwadratowych o rozmiarach $n \times n$ daje n^3 operacji. Lepszy (i o wiele bardziej skomplikowany) algorytm, autorstwa Coppersmitha i Winograda, ma złożoność rzędu $n^{2,376}$.



Rys. 6.1. Ilustracja algorytmu mnożenia macierzy

6.4.7. Rozwiązywanie układów równań liniowych

Dany jest oznaczony układ równań postaci

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

gdzie A jest macierzą nieosobliwą typu $n \times n$, $\mathbf{b}$ jest n-elementowym wektorem wyrazów wolnych oraz $\mathbf{x}$ jest n-elementowym wektorem niewiadomych.

Rozpatrywany układ równań można zapisać w następującej postaci.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

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

$$\left[\begin{array}{cccc|c} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} & b_0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-1} & b_{n-1} \end{array}\right]$$

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

1. pomnożenie dowolnego wiersza macierzy rozszerzonej przez niezerową stałą,
2. dodanie do dowolnego wiersza kombinacji liniowej pozostałych wierszy.

Następnie uzyskuje się wartości wektora wynikowego $\mathbf{x}$ korzystając z tzw. *eliminacji wstecznej*:

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

dla kolejnych $i = n-1, n-2, \ldots, 0$.

Rozpatrzmy przykładową implementację tej metody w języku C++.

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

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

Funkcja sprowadzająca macierz do postaci schodkowej (która modyfikuje wejściową macierz):

```
void schodkowa(double** A, double* b, int n)
{
    for (int i=0; i<n; ++i)
    { /* dla każdego wiersza */
       assert(fabs(A[i][i]) > 1e-12);//jeśli nie=>macierz osobliwa (
           dlaczego tak?)

       for (int j=i+1; j<n; ++j)
       { // wiersz j:=(wiersz j)-A[j,i]/A[i,i]*(wiersz i)

          for (int k=i; k<n; ++k)
             A[j][k] -= A[j][i]/A[i][i]*[i][k];

          b[j] -= A[j][i]/A[i][i]*b[i];
       }
    }
}
```

Eliminacja wsteczna:

```
double* eliminwst(double** A, double* b, int n)
{
    double* x = new double[n];
    for (int i=n-1; i>=0; --i)
    {
       x[i] = b[i];
       for (int k=i+1; k<n; ++k)
       {
          x[i] -= A[i][k] * x[k];
       }
       x[i] /= A[i][i];
    }
    return x;
}
```

6.5. Ćwiczenia

Zadanie 6.1. Zaimplementuj samodzielnie funkcje z biblioteki `<cstring>`: `strlen()`, `strcpy()`, `strcat()`, `strcmp()`, `strstr()`, `strchr()`, `strrchr()`.

Zadanie 6.2. Napisz funkcję, która tworzy i zwraca nową, dynamicznie stworzoną kopię danego napisu z zamienionymi wszystkimi małymi literami alfabetu łacińskiego (bez polskich znaków diakrytycznych) na wielkie.

Zadanie 6.3. Napisz funkcję, która usuwa wszystkie znaki odstępów (spacje) z końca danego napisu. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.4. Napisz funkcję, która usuwa wszystkie znaki odstępów (spacje) z początku danego łańcucha znaków. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.5. Napisz funkcję, która usuwa z danego napisu wszystkie znaki niebędące cyframi bądź kropką. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.6. Napisz funkcję, która odwróci kolejność znaków w danym napisie. Napis przekazany na wejściu powinien pozostać bez zmian.

Zadanie 6.7. Napisz funkcję obliczającą, ile razy w danym napisie występuje dany znak.

Zadanie 6.8. Napisz funkcję, która oblicza, ile razy w danym napisie występuje dany inny napis, np. w *"ababbababa"* łańcuch *"aba"* występuje 3 razy.

Zadanie 6.9. Napisz funkcję, która dla danych dwóch napisów `n1` i `n2` oraz znaku `c` utworzy nowy, dynamicznie alokowany napis, będący złączeniem `n1` i `n2` z pominięciem wszystkich wystąpień znaku `c`. Dla przykładu, dla napisów `"ala"` i `"ma kota"` oraz znaku `'a'` powinien być zwrócony napis `"lm kot"`.

Zadanie 6.10. Napisz funkcję, która złączy dwa dane napisy `n1` i `n2` i podwoi wszystkie wystąpienia danego znaku `c`. Dla przykładu, dla napisów `"ulubione"`, `"turuburu"` i znaku `'u'` powinniśmy otrzymać nowy napis `"uuluubionetuuruubuuruu"`.

Zadanie 6.11. Palindrom to ciąg liter, które są takie same niezależnie od tego, czy czytamy je od przodu czy od tyłu, np. *kobyłamamałybok*, *możejutrotadamasamadatortujeżom*, *ikarłapałraki*. Napisz funkcję, która sprawdza czy dany napis jest palindromem.

Zadanie 6.12. Napisz funkcję `char* napscal(char* napis1, char* napis2)`, która przyjmuje na wejściu dwa napisy, których litery są posortowane (w porządku leksykograficznym). Funkcja zwraca nowy, dynamicznie alokowany napis będący scaleniem (z zachowaniem porządku) tych napisów, np. *"alz"* i *"lopxz"* wynikiem powinno być *"allopxzz"*.

Zadanie 6.13. Napisz funkcję `int napporownaj(char* napis1, char* napis2)`, która zwraca wartość równą 1, jeśli pierwszy napis występuje w porządku leksykograficznym przed napisem drugim, wartość -1, jeśli po napisie drugim, bądź 0 w przypadku, gdy napisy są identyczne. Dla przykładu, `napporownaj("ola", "ala")==-1` oraz `napporownaj("guziec", "guziec2")==1`.

Zadanie 6.14. Napisz funkcję `char* napsortlitery(char* napis)`, która zwraca nowy, dynamicznie alokowany napis składający się z posortowanych (dowolnym znanym Ci algorytmem) liter napisu wejściowego. Dla przykładu, `napsortlitery("abacdcd")` powinno dać w wyniku napis *"aabccdd"*.

Zadanie 6.15. Napisz funkcję `void sortuj(char* napisy[], int n)`, która wykorzysta dowolny algorytm sortowania tablic oraz funkcję `napporownaj(...)` do leksykograficznego uporządkowania danej tablicy napisów.

Zadanie 6.16. Napisz funkcję `char losuj()`, która wykorzystuje funkcję `rand()` z biblioteki `<cstdlib>`: do „wylosowania” wielkiej litery z alfabetu łacińskiego. Zakładamy, że funkcja `srand()` została wywołana np. w funkcji `main()`.

Zadanie 6.17. Napisz funkcję `char* permutuj(char* napis)`, która zwraca nowy, dynamicznie alokowany napis będący losową permutacją znaków występujących w napisie przekazanym jako argument wejściowy.

Zadanie 6.18. Dana jest macierz $A \in \mathbb{R}^{n\times m}$ oraz liczba $k \in \mathbb{R}$. Napisz funkcję, która wyznaczy wartość kA, czyli implementującą mnożenie macierzy przez skalar.

Zadanie 6.19. Dana jest macierz $A \in \mathbb{R}^{n\times m}$ 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 6.20. Dana jest macierz $A \in \mathbb{R}^{n\times m}$ 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 6.21. Dana jest macierz $A \in \mathbb{R}^{3\times 3}$. Napisz funkcję wyznaczającą $\det A$.

Zadanie 6.22. 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 6.23. Napisz funkcję, która rozwiązuje układ 3 równań liniowych korzystając z metody Cramera (dana jest macierz 3×3 i wektor). Poprawnie identyfikuj przypadek, w których dany układ nie jest oznaczony.

Zadanie 6.24. Dana jest macierz $A \in \mathbb{Z}^{n\times m}$. Napisz funkcję zwracającą jej transpozycję.

Zadanie 6.25. Dana jest macierz $A \in \mathbb{Z}^{n\times m}$. 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 6.26. Dana jest kwadratowa macierz A o wartościach całkowitych. Napisz funkcję, która sprawdzi, czy macierz jest symetryczna. Zwróć wynik typu `bool`.

Zadanie 6.27. Dana jest macierz kwadratowa A o wartościach rzeczywistych typu $n \times n$. Napisz funkcję, która zwróci jej ślad, określony jako $\operatorname{tr}(A) = \sum_{i=0}^{n-1} a_{i,i}$.

Zadanie 6.28. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej diagonalę w postaci tablicy jednowymiarowej.

Zadanie 6.29. Dla danej macierzy kwadratowej A napisz funkcję, która zwróci jej macierz diagonalną, czyli macierz z wyzerowanymi wszystkimi elementami poza przekątną.

Zadanie 6.30. 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 6.31. Kwadratem magicznym stopnia n nazywamy macierz kwadratową typu $n\times n$ o elementach ze zbioru liczb naturalnych taką, że sumy elementów w każdym wierszu, w każdej kolumnie i na każdej z dwóch przekątnych są takie same. Napisz funkcję, która sprawdza, czy dana macierz jest kwadratem magicznym. Zwróć wynik typu `bool`.

Zadanie 6.32. Napisz funkcję, która przyjmuje jako parametr macierz `A` o elementach typu `bool` i zwróci nową, dynamicznie alokowaną macierz, która zawiera te i tylko te wiersze z `A`, które zawierają nie mniej wartości `true` niż `false`.

Zadanie 6.33. Napisz funkcję, która przyjmuje jako parametr macierz `A` o elementach typu `char` i zwróci nową, dynamicznie alokowaną macierz, która zawiera te i tylko te kolumny z `A`, które zawierają parzystą lecz niezerową liczbę wielkich liter.

Zadanie 6.34. Stwórz funkcje `double* wstawW(double* M, int n, int m, int i, double c)` oraz `wstawK()`, która wstawi do danej macierzy M nowy wiersz (nową kolumnę) pomiędzy wierszami (kolumnami) i oraz $i + 1$, oraz wypełni go wartością c.

⋆ **Zadanie 6.35.** Napisz funkcję, która sprawdzi czy dana macierz może być otrzymana z macierzy 1×1 przez ciąg operacji `wstawW()` i `wstawK()`, zdefiniowanych w zadaniu 6.34.

⋆ **Zadanie 6.36.** Napisz nierekurencyjną funkcję wyznaczającą wartość wyrażenia:

- $f(n, m) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (f(i, j) - i - j)^2$,
- $f(n, m) = \begin{cases} 1 & \text{dla } n = 0 \text{ lub } m = 0, \\ \|\{(i, j) \in \mathbb{N}^2 : i < n, j < m, f(i, j) < (i - j)^2\}\| & \text{dla } n, m > 0, \end{cases}$

gdzie $n, m \in \mathbb{N}$.

6.6. Wskazówki i odpowiedzi do ćwiczeń

Odpowiedź do zadania 6.1.

```
int strlen(char *s)
{
  int i = 0;
  while (s[i]!=0) i++;
  return i;
}
```

```
char* strcpy(char *cel, char *zrodlo)
{
  // zalozenie - pamiec dla tablicy przydzielona przed wywolaniem
      funkcji (cel = new char[strlen(zrodlo)+1];)

  int i = 0;
  while (zrodlo[i]!=0)
  {
     cel[i] = zrodlo[i];
     i++;
  }
  cel[i] = '\0'; // wstaw "wartownika"

  return cel;
}
```

```
int strcmp(char *s1, char *s2)
{
  int i = 0;
  int j = 0;
  while (s1[i]!=0 && s1[i] == s2[j])
  {
     i++;
     j++;
  }

  return s1[i] - s2[j];
}
```

```
char* strcat(char* cel, char* zrodlo)
{
  // zalozenie - napis w tablicy cel ma n znakow,
  // napis w tablicy zrodlo ma m znakow,
  // pamiec przydzielona na tablice cel wynosi n+m+1 znakow
  // (bajt zerowy w cel nie musi wystepowac przeciez na koncu
      tablicy!)
  int n = strlen(cel); // cel[n] == '\0' (!)
  int j = 0;
  while (zrodlo[j]!=0)
  {
     cel[n] = zrodlo[j];
     n++;
     j++;
  }
  cel[n] = '\0';

```

```
17   return cel;
18 }
```

```
1  char* strchr(char* nap, char znak)
2  {
3    int i=0;
4    while (nap[i] != 0)
5    {
6       if (nap[i] == znak)
7          return &(nap[i]); // znaleziony - zwracamy podnapis
8
9       i++;
10   }
11
12   // tutaj doszedłszy, stwierdzamy, że znak wcale nie występuje w
         napisie
13   return NULL; // albo return 0; - adres zerowy to "nigdzie"
14 }
```

```
1  char* strrchr(char* nap, char znak)
2  {
3    int i=strlen(nap)-1; // zaczynamy od ostatniego znaku drukowanego
4    while (i>=0)
5    {
6       if (nap[i] == znak)
7          return &(nap[i]); // znaleziony - zwracamy podnapis
8
9       i--;
10   }
11
12   return NULL;
13 }
```

```
1  char* strstr(char* nap1, char* nap2)
2  {
3    int n = strlen(nap1);
4    int m = strlen(nap2);
5
6    assert(n>=m);
7
8    int i, j;
9    for (i=0; i<n-m+1; ++i)
10   {
11      for (j=0; j<m; ++j)
12      {
13         if (nap1[i+j] != nap2[j]) break; // przerywamy petle
14      }
15
16      if (j == m) // doszlismy do konca petli => znaleziony!
17         return &(nap1[i]);
18   }
19
20   return NULL; // on ne le trouve pas :-)
21 }
```

□

Wskazówka do zadania 6.22. Wyznacznik macierzy 4×4 można policzyć ze wzoru

$$\det A = \sum_{i=0}^{3} (-1)^{i+j} a_{ij} \det A_{i,j},$$

gdzie j jest dowolną liczbą ze zbioru $\{0, 1, 2, 3\}$, 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. □

Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

7. Dynamiczne struktury danych

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

7.1. Wprowadzenie

Wróćmy do zagadnienia implementacji podstawowych abstrakcyjnych typów danych wprowadzonych w rozdz. 5, tj.:

1. stosu (rozdz. 7.2),
2. kolejki (rozdz. 7.3),
3. kolejki priorytetowej (rozdz. 7.4),
4. słownika (rozdz. 7.5).

Przypomnijmy, że definicje wszystkich abstrakcyjnych typów danych *abstrahują od sposobu ich implementacji*. Do tej pory zastanawialiśmy się, w jaki sposób zaimplementować je z użyciem tablicy.

Informacja

Tak jak ostatnio, bez straty ogólności umawiamy się, że każdy prezentowany abstrakcyjny typ danych będzie służył do przechowywania pojedynczych wartości całkowitych, tj. danych typu `int`.

7.2. Stos

Stos (ang. *stack*) jest abstrakcyjnym typem danych typu LIFO (ang. *last-in-first-out*), który udostępnia przechowywane weń elementy w kolejności odwrotnej niż ta, w której były one zamieszczane.

Stos udostępnia trzy operacje:

– *push* (umieść) — wstawia element na początek stosu,
– *pop* (zdejmij) — usuwa i zwraca element z początku stosu,
– *empty* (czy pusty) — sprawdza, czy na stosie nie znajdują się żadne elementy.

Chciałoby się rzec — tylko tyle i aż tyle. W jaki sposób zaimplementować taki ATD? Dysponując wiedzą, którą posiadamy z poprzednich wykładów, najprawdopodobniej spróbowalibyśmy stworzyć stos oparty na zwykłej tablicy.

7.2.1. Implementacja I: tablica

Do stworzenia najprostszej implementacji stosu potrzebować będziemy trzech zmiennych (zadeklarowanych np. w funkcji `main()`[1]):

```
int  n = ...;          // maksymalna liczba elementów na stosie
int* s = new int[n]; // tu przechowujemy elementy
int  t = -1;           // indeks aktualnego ostatniego elementu
```

Zacznijmy od prostego przykładu użycia stosu.

[1] Lub umieszczonych w jednej strukturze `struct Stos{ ... };`

```
int main()
{
    int  n = 100;
    int* s = new int[n];
    int  t = -1;

    push(7, s, t, n);
    push(5, s, t, n);
    push(3, s, t, n);

    while (!empty(s, t, n))
       cout << pop(s, t, n);
    // wypisze na ekran 357

    return 0;
}
```

Rzecz jasna, najprostszą do zrealizowania operacją jest sprawdzenie, czy stos jest pusty:

```
bool empty(int* s, int t, int n)
{
    return (t < 0); // true, jeśli pusty
}
```

Operacja zdejmowania elementu ze stosu może wyglądać tak:

```
int pop(int* s, int& t, int n)
{
    assert(t >= 0); // zakładamy, że mamy co zdejmować
    return s[t--];  // zwróć s[t], a potem zmniejsz t o 1
}
```

Zauważmy, że zmienna `t` została przekazana przez referencję. Chcemy, by jej zmiana była „widoczne na zewnątrz”.

Na koniec operacja dodawania elementu do stosu:

```
void push(int x, int* s, int& t, int n)
{
    assert(t < n-1); // zakładamy, że mamy gdzie umieścić element
    s[++t] = x;      // najpierw zwiększ t o 1, a potem s[t] = x;
}
```

Widzimy, że wszystkie operacje potrzebują zawsze stałej liczby dostępów do tablicy, tzn. są rzędu $O(1)$. Jest to *rozwiązanie optymalne* (bardziej efektywne z obliczeniowego punktu widzenia istnieć nie może).

Napotykamy w tym miejscu jednak istotny problem — *w definicji naszego ATD nie pojawia się wcale założenie, że liczba przechowywanych elementów ma być z góry ograniczona*. Z tego powodu powyższa implementacja stosu jest tylko częściowo zgodna z definicją (formalista rzekłby, że to wcale nie jest implementacja stosu). Rzecz jasna, w przypadku niektórych programów takie ograniczenie wcale nie musi być niepoprawne. Co się stanie, jeśli naprawdę nie możemy z góry ustalić, ile elementów powinien móc przechowywać stos?

7.2.2. Implementacja II: tablica rozszerzalna

Spróbujmy zatem operacji dodawania elementów do stosu na powiększanie (w razie potrzeby) tablicy `s`. Będziemy więc musieli ją przekazywać przez referencję, gdyż może ona

(a właściwie adres jej pierwszego elementu) ulec zmianie. Co więcej, także `n` może wówczas się zmienić.

```
void push(int x, int*& s, int& t, int& n)
{
   if (t == n-1)
   {
      int* starys = s;  // zapamiętaj, co było do tej pory
      s = new int[n+1]; // nowa tablica - o jedno miejsce więcej (
         na przykład)

      for (int i=0; i<n; ++i)
         s[i] = starys[i];  // przepisz cały ,,stary stos''

      ++n;                  // rozmiar jest już większy

      delete [] starys;     // stare już niepotrzebne
   }

   s[++t] = x;
}
```

Niestety, *pesymistyczna złożoność obliczeniowa operacji push wynosi teraz* $O(n)$ (dlaczego?). Co więcej (powyższa implementacja dzieli tę własność z poprzednią), jeśli stos jest jedynie częściowo wypełniony, marnotrawimy wiele cennych komórek pamięci tylko po to, by były one w gotowości „na zaś". Jeśli zaprogramujemy dodatkowo operację *pop* tak, by zmniejszała tablicę `s`, to i ta operacja stanie się bardziej kosztowna obliczeniowo. Taki stan rzeczy nie może nam się podobać, spróbujemy więc poszukać lepszego rozwiązania.

7.2.3. Implementacja III: lista jednokierunkowa

Przedstawimy teraz implementację stosu, która wykorzystuje tzw. *listę jednokierunkową*. Istotną zaletą takie rozwiązania jest to, że wszystkie 3 omawiane operacje są realizowane w stałym czasie ($O(1)$) oraz że liczba przechowywanych elementów jest dowolna.

Lista jednokierunkowa to dynamiczna struktura danych, która składa się z *węzłów* reprezentowanych przez typ złożony o następującej definicji:

```
struct wezel
{
   int elem;     // element przechowywany w węźle
   wezel* nast; // wskaźnik na następny element
};
```

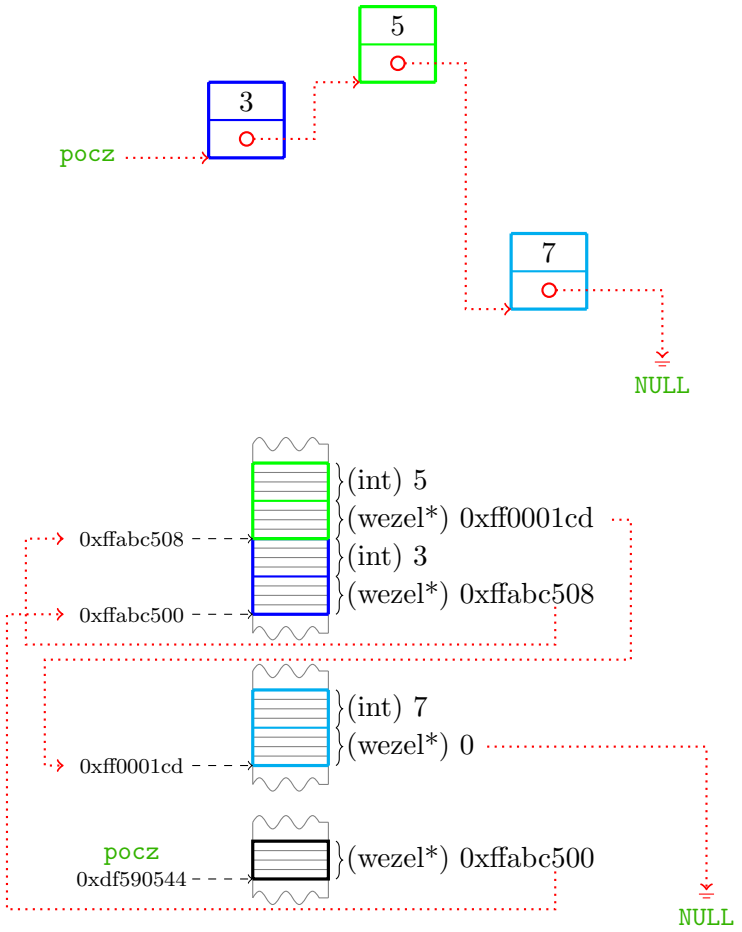
Zauważmy, że taka definicja ma w pewnym sensie charakter rekurencyjny. W definicji węzła pojawia się odwołanie do innego węzła (za pomocą wskaźnika). Węzły w liście są ze sobą połączone, jeden za drugim. Do stworzenia listy wystarczy zatem wskaźnik na jej pierwszy element (tzw. *głowa*):

```
wezel* pocz;
```

oraz informacja, po którym węźle lista się kończy. Do tego celu stosuje się najczęściej wartownika — adres `NULL` (0).

Schemat przykładowej listy jednokierunkowej przechowującej elementy 3, 5 oraz 7 przedstawia rys. 7.1. Zwróćmy szczególną uwagę na to, w jaki sposób węzły *mogą być* rozlokowane w pamięci.

Nim zaczniemy się zastanawiać, w jaki sposób zrealizować stos za pomocą listy, zacznijmy od stworzenia pliku nagłówkowego (np. `stos.h`).



Rys. 7.1. Przykładowa lista jednokierunkowa przechowująca elementy 3,5,7. Schemat graficzny (powyżej) i przykładowa organizacja pamięci (poniżej).

```
#pragma once

struct wezel
{
    int elem;     // element przechowywany w węźle
    wezel* nast; // wskaźnik na następny element
};

void push(int i, wezel*& pocz);
int  pop(wezel*& pocz);
bool empty(wezel* pocz);
```

Zakładamy tutaj, że operacje *push* i *pop* mogą spowodować zmianę pierwszego elementu listy.

Oto przykładowy plik `main.cpp`, zawierający funkcję `main()` analogiczną do tej, którą zaprezentowaliśmy dla implementacji tablicowej stosu.

```
#include ...

int main()
{
    wezel* pocz = NULL; // pusta lista (na początek)

    push(7, pocz);
```

```
8     push(5, pocz);
9     push(3, pocz);
10
11    while (!empty(pocz))
12       cout << pop(pocz);
13
14    // 357
15
16    return 0;
17 }
```

Znowu najprostszą funkcją będzie operacja sprawdzająca, czy stos jest pusty. Wystarczy sprawdzić, czy węzeł początkowy wskazuje bezpośrednio na wartownika.

```
bool empty(wezel* pocz)
{
   return (pocz == NULL); // true, jeśli lista pusta
}
```

Operacje *push* i *pop* można zrealizować jako, odpowiednio, *wstawianie elementu na początek oraz usuwanie elementu z początku listy*. Łatwo się domyślić, że będą one potrzebowały stałej liczby dostępów do węzłów listy, tzn. będą rzędu $O(1)$.

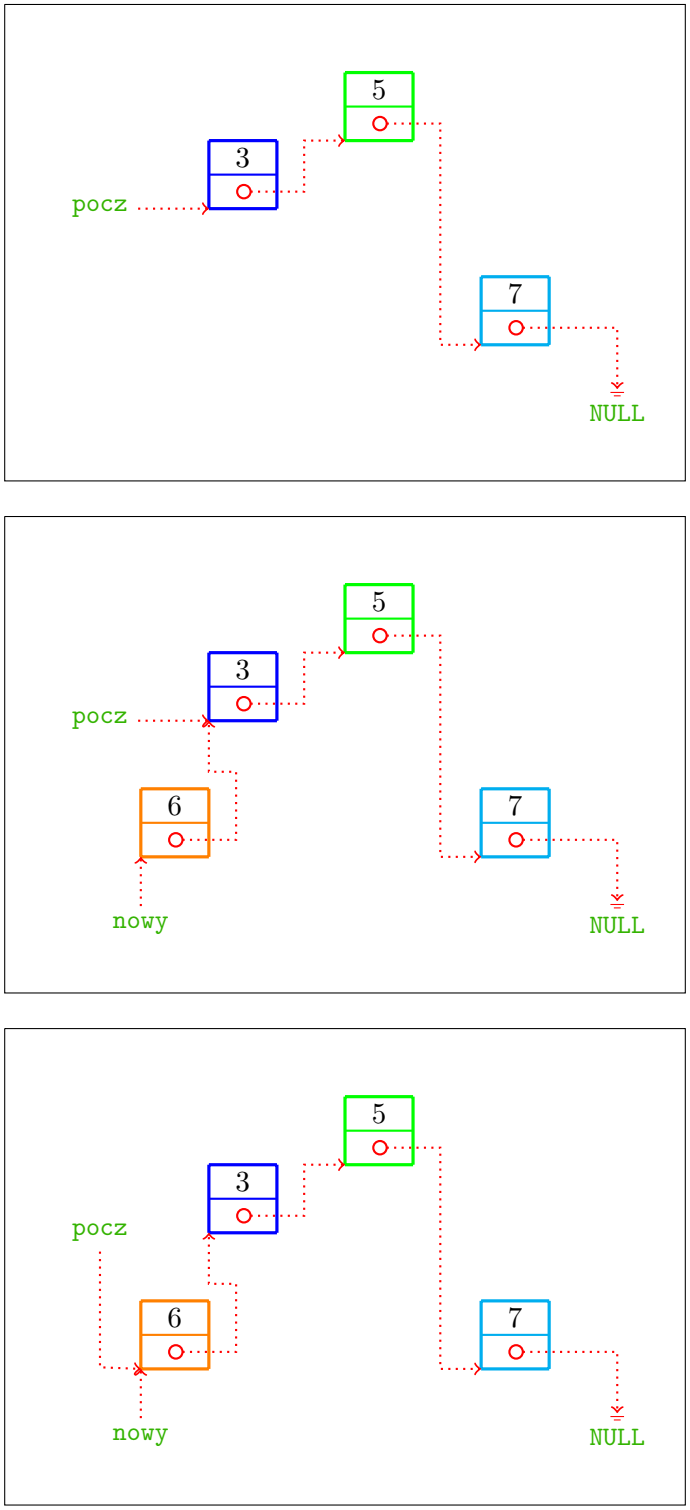
Najpierw zajmijmy się wstawianiem elementu na początek listy. Rys. 7.2 ilustruje kolejne kroki potrzebne do utworzenia listy (6,3,5,7) z listy (3,5,7). Zwróćmy uwagę, że po dokonaniu tej operacji zmienia się głowa listy.

Oto przykładowa implementacja operacji *push* stosująca wstawianie elementu na początek listy jednokierunkowej. Zauważmy raz jeszcze, że pierwszym parametrem jest referencja na zmienną wskaźnikową. Przekazujemy tutaj głowę listy, która będzie zmieniona przez tę funkcję.

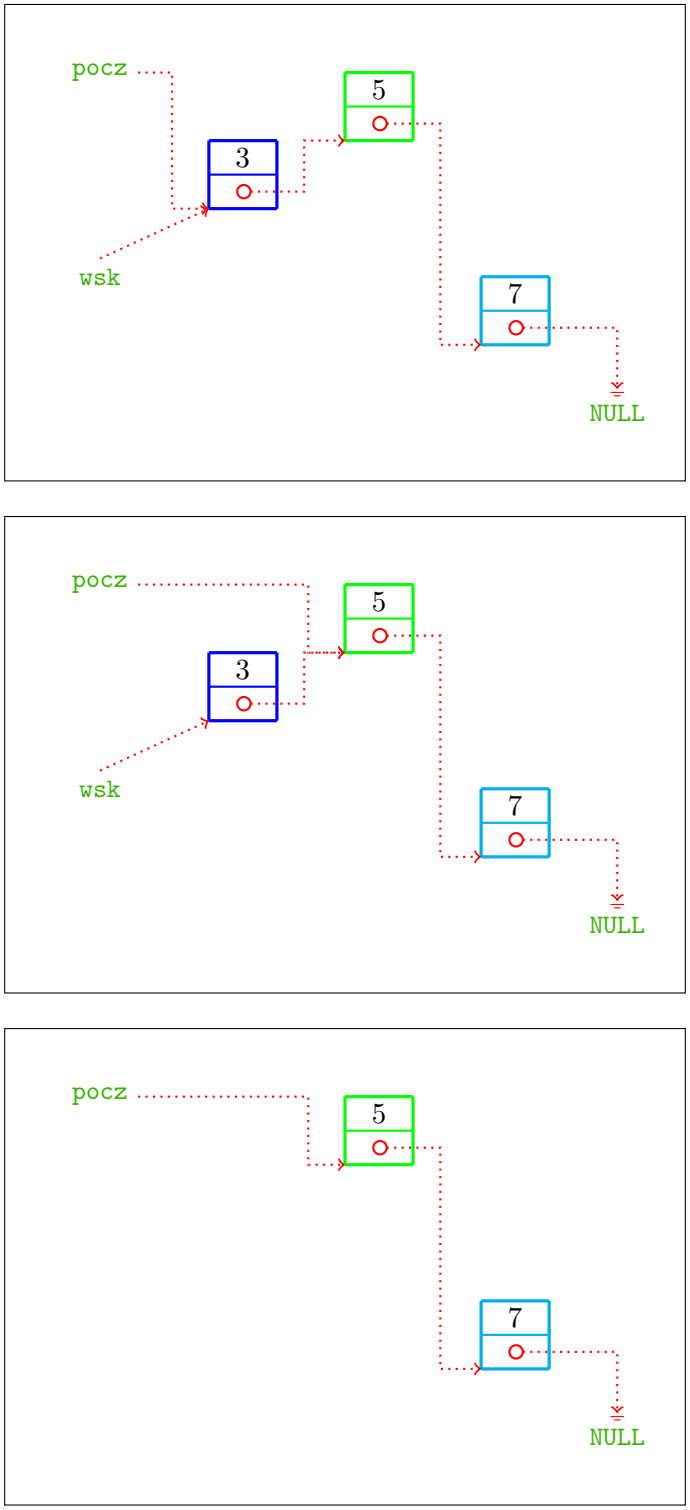
```
void push(int i, wezel*& pocz)
{
   wezel* nowy = new wezel;
   nowy->elem = i;

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

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

Rys. 7.3 przedstawia możliwy sposób usuwania elementów z początku listy jednokierunkowej. Implementację operacji *pop* przedstawia poniższy kod. Zauważmy, że po jej dokonaniu możliwe jest osiągnięcie stanu `glowa == NULL`, co oznacza, że usunięty element był jedynym.

```
int pop(wezel*& pocz)
{
   assert(pocz != NULL); // na wszelki wypadek...

   int i = pocz->elem;    // zapamiętaj przechowywany element

   wezel* wsk = pocz;     // zapamiętaj "stary" pocz
   pocz = pocz->nast;     // zmień pocz
```



Rys. 7.2. Wstawianie elementu 6 na początek listy jednokierunkowej.



Rys. 7.3. Usunięcie elementu z początku listy jednokierunkowej.

```
    delete wsk;             // skasuj "stary" pocz

    return i;
}
```

Jak widzimy, powyższe operacje wykonywane na liście jednokierunkowej wcale nie są (wbrew obiegowej opinii) skomplikowane. Ważne jest, byśmy już teraz dobrze zrozumieli zasadę ich działania.

W następnym podrozdziale omówimy kolejny abstrakcyjny typ danych — kolejkę. Okaże się, że jej efektywną implementację można także oprzeć na (odpowiednio zmodyfikowanej) liście jednokierunkowej.

7.3. Kolejka

Kolejka (ang. *queue*) to ATD typu FIFO (ang. *first-in-first-out*). Podstawową własnością kolejki jest to, że pobieramy z niej elementy w takiej samej kolejności (a nie w odwrotnej jak w przypadku stosu), w jakiej były one umieszczane.

Kolejka udostępnia następujące trzy operacje:

- *enqueue* (umieść) — wstawia element na koniec kolejki,
- *dequeue* (wyjmij) — usuwa i zwraca element z początku kolejki,
- *empty* (czy pusty).

Zadanie

Jak łatwo się domyślić, implementacje kolejki oparte na tablicy będą cechować się podobnymi wadami, jakimi cechowały się analogiczne rozwiązania z rozdz. 7.2.1 i 7.2.2. Ich wykonanie pozostawmy wobec tego zacnemu Czytelnikowi. Będzie to dlań dobrym ćwiczeniem przed kolokwium.

7.3.1. Implementacja I: lista jednokierunkowa

Zastanówmy się, w jaki sposób można zrealizować operacje kolejki z użyciem listy jednokierunkowej. Mamy tutaj dwie możliwości:

1. *enqueue1* — wstaw element na początek listy, *dequeue1* — usuń z końca, bądź
2. *enqueue2* — wstaw element na koniec listy, *dequeue2* — usuń z początku.

Niestety, choć (co już wiemy) *enqueue1* / *dequeue2* wymagają $O(1)$ operacji, to pozostałe działania na liście cechują się pesymistyczną złożonością obliczeniową rzędu $O(n)$. Mimo to spróbujmy je zaimplementować.

Rozważmy najpierw, w jaki sposób wstawić element na koniec listy (działanie potrzebne do zrealizowania operacji *enqueue2*). Rys. 7.4 i 7.5 ilustrują, jak 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 omawianą operację. Zauważmy, że drugim parametrem jest referencja do wskaźnika (a dokładniej — do głowy listy).

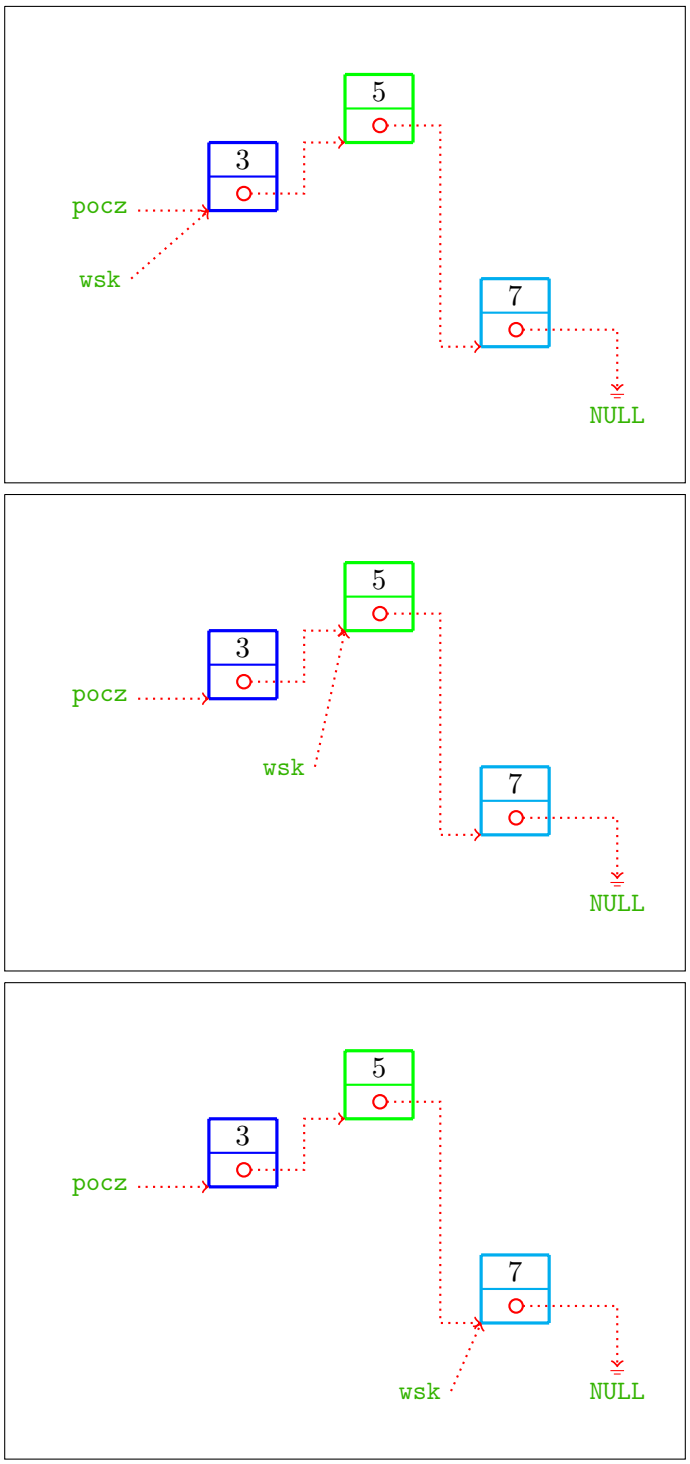
```
void enqueue2(int i, wezel*& pocz)
{
    wezel* nowy = new wezel;
    nowy->elem = i;
    nowy->nast = NULL; // to będzie ostatni element

    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.

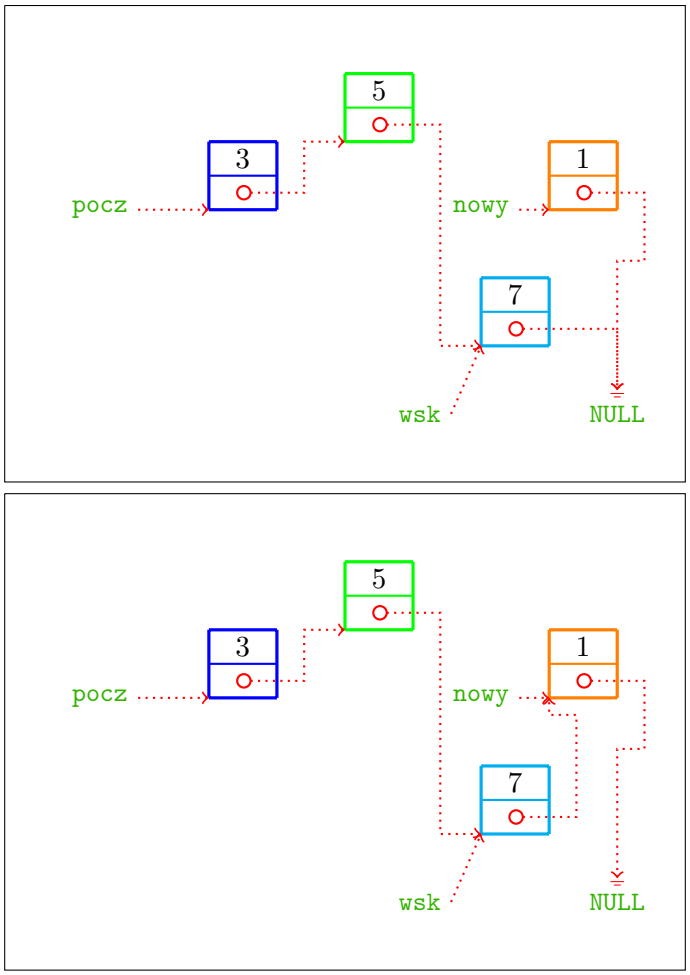
```
void enqueue2_rek(int i, wezel*& wsk) // referencja!
{
    if (wsk != NULL)                 // to nie jest koniec...
       enqueue2_rek(i, wsk->nast); // więc idź dalej
    else
    {                                // jesteś na końcu
       wsk = new wezel;
       wsk->elem = i;
       wsk->nast = NULL;
    }
}
```

Pora na operację *dequeue1*, realizowaną za pomocą usuwania elementu z końca listy. Rys. 7.6 ilustruje przykładowy przebieg takiego procesu. W wersji iteracyjnej tej funkcji musimy przejść na przedostatni element listy. Należy się jednak wcześniej upewnić, czy przypadkiem nie kasujemy jedynego elementu (oddzielny przypadek).

```
int dequeue1(wezel*& pocz)
{
    assert (pocz != NULL);

    int i;
    if (pocz->nast == NULL)    // tylko 1 element
    {
       i = pocz->elem;
       delete pocz;
       pocz = NULL;
    }
    else                       // więcej niz 1 element
    {
       wezel* wsk = pocz;
       while (wsk->nast->nast != NULL)
          wsk = wsk->nast;     // idź na przedost. el.
       i = wsk->nast->elem;
       delete wsk->nast;
       wsk->nast = NULL;
    }
    return i;
}
```



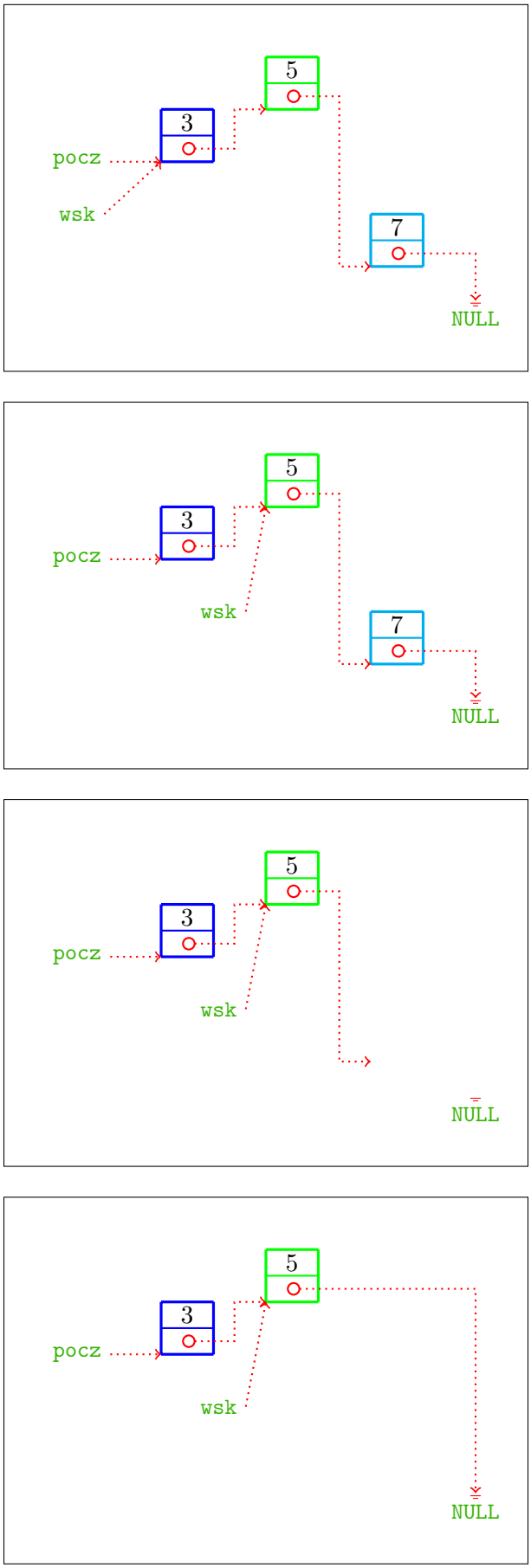
Rys. 7.4. Wstawianie elementu 1 na koniec listy jednokierunkowej cz. I.



Rys. 7.5. Wstawianie elementu 1 na koniec listy jednokierunkowej cz. II.

Omawianą procedurę również można sformułować znacznie prościej za pomocą rekurencji.

```
1  int dequeue1_rek(wezel*& wsk)
2  {
3      assert(wsk != NULL);
4
5      if (wsk->nast == NULL) // jesteśmy na końcu
6      {
7          int i = wsk->elem;
8          delete wsk;
9          wsk = NULL;
10         return i;
11     }
12     else // idziemy dalej
13         return dequeue1_rek(wsk->nast);
14 }
```



Rys. 7.6. Usunięcie elementu z końca listy jednokierunkowej.

7.3.2. Implementacja II: lista jednokierunkowa z „ogonem”

Okazuje się jednak, że operacje kolejki da się zaimplementować tak, by potrzebowały zawsze stałej liczby ($O(1)$) dostępów do elementów. W poprzednim rozwiązaniu krytyczną (tzn. najbardziej czasochłonną) czynnością było przejście na koniec listy. Gdybyśmy zapamiętali dodatkowo wskaźnik na ostatni element listy (ogon), wstawianie mogłoby się wykonywać bardzo szybko.

Zadanie

Ogon nie pozwala przyspieszyć operacji usuwania ostatniego elementu. Dlaczego?

Przyjrzyjmy się pełnej implementacji naszej kolejki. Deklaracje funkcji (w pliku `.h`) mogą wyglądać tak:

```
void enqueue(int i, wezel*& pocz, wezel*& ost);
int  dequeue(wezel*& pocz, wezel*& ost);
bool empty(wezel* pocz, wezel* ost);
```

W dwóch pierwszych przypadkach dopuszczamy zarówno możliwość zmiany głowy jak i ogona listy.

Przykładowa funkcja główna:

```
int main()
{
   wezel* pocz = NULL; // pusta lista (na początek)
   wezel* ost = NULL;  // wskaźnik na ostatni element — też NULL

   for (int i=0; i<10; ++i) // wstaw coś w celach testowych
      enqueue(i, pocz, ost);

   while (!empty(pocz, ost))
      cout << dequeue(pocz, ost);

   return 0;
}
```

Na koniec — definicje funkcji. Tak może wyglądać sprawdzenie, czy kolejka jest pusta:

```
bool empty(wezel* pocz, wezel* ost)
{
   return (pocz == NULL); // true, jeśli lista pusta
}
```

Umieszczenie elementu w kolejce:

```
void enqueue(int i, wezel*& pocz, wezel*& ost)
{
   // wstawianie na koniec listy

   wezel* nowy = new wezel; // stwórz nowy wezel
   nowy->elem = i;
   nowy->nast = NULL;

   if (pocz == NULL)          // gdy lista pusta...
   {
      pocz = ost = nowy;      // zmieniamy pocz i ost
   }
   else                       // gdy lista niepusta...
   {
      ost->nast = nowy;       // po ostatnim - nowy
      ost = nowy;             // teraz ostatni to nowy
   }

   // koszt operacji: O(1)
}
```

Wyjęcie elementu z kolejki:

```
int dequeue(wezel*& pocz, wezel*& ost)
{
   assert(pocz != NULL); // na wszelki wypadek...

   // usuwanie z początku listy

   int i = pocz->elem;    // zapamiętaj przechowywany element

   wezel* wsk = pocz;     // zapamiętaj "stary" pocz
   pocz = pocz->nast;     // zmień pocz
   delete wsk;            // skasuj "stary" pocz

   if (pocz == NULL) ost = NULL;

   return i;

   // koszt operacji: O(1)
}
```

Dzięki prostej modyfikacji listy (jednej dodatkowej informacji) udało nam się znacząco przyspieszyć działanie kolejki. W następnym podrozdziale wprowadzimy nowy ATD — bardziej zaawansowaną kolejkę, w której będą mogły występować obiekty w pewnym sensie „uprzywilejowane”.

7.4. Kolejka priorytetowa

Kolejka priorytetowa (ang. *priority queue*) to abstrakcyjny typ danych typu HPF (ang. *highest-priority-first*). Może ona służyć do przechowywania elementów, na których została określona pewna relacja porządku liniowego <=[2]. Elementy wydobywa się z niej

[2] W naszym przypadku, jako że umówiliśmy się, iż omawiane ATD przechowują dane typu `int`, najczęściej będzie to po prostu zwykły porządek $\leqslant$.

poczynając od tych, które mają największy priorytet (są maksymalne względem relacji <=). Jeśli w kolejce priorytetowej znajdują się elementy o takich samych priorytetach, obowiązuje dla nich kolejność FIFO, tak jak w przypadku zwykłej kolejki.

Omawiany ATD udostępnia trzy operacje:

– *insert* (włóż) — wstawia element na odpowiednie miejsce kolejki priorytetowej.

– *pull* (pobierz) — usuwa i zwraca element maksymalny,

– *empty* (czy pusty).

Zadanie

Podobnie jak wcześniej, implementacja tej struktury danych z użyciem tablicy wymaga $O(n)$ dostępów do obiektów w przypadku operacji *insert* oraz $O(1)$ dostępów przy wywołaniu *pull*. Pozostawmy jeszcze raz to pouczające zagadnienie jako wyzwanie dla zdolnego i pracowitego Czytelnika.

7.4.1. Implementacja I: lista jednokierunkowa

Spróbujmy zaimplementować kolejkę priorytetową używając listy jednokierunkowej, która przechowuje elementy posortowane względem relacji <= (w kolejności odwrotnej, tzn. największy element jest pierwszy). Operacja *pull* powinna więc po prostu zwracać i usuwać aktualną głowę listy (patrz wyżej, złożoność $O(1)$).

Oto wersja rekurencyjna operacji *insert*, która realizuje wstawianie elementu na odpowiednim miejscu listy (bez zaburzenia istniejącego porządku).

```
void insert(int i, wezel*& pocz)
{
   if (pocz == NULL || i > pocz->elem) // >, bo FIFO, gdy ==
   {     // wstawiamy tutaj
      wezel* starypocz = pocz; // może być NULL

      pocz = new wezel; // stwórz nowy wezel
      pocz->elem = i;
      pocz->nast = starypocz;
   }
   else // idziemy dalej
      insert(i, pocz->nast);
}
```

Niestety, powyższa operacja cechuje się liniową ($O(n)$) złożonością obliczeniową. Nie pomoże tu oczywiście uwzględnienie np. dodatkowego wskaźnika na ostatni element listy. Dostrzegamy wobec tego konieczność zastosowania w tym miejscu innej dynamicznej struktury danych.

7.4.2. Implementacja II: drzewo binarne

Drzewo binarne bądź binarne drzewo poszukiwań (ang. *binary search tree*, BST), to dynamiczna struktura danych, w której elementy są uporządkowane zgodnie z pewnym porządkiem liniowym <=. Dane przechowuje się tutaj w węzłach zdefiniowanych następująco:

```
struct wezel
{
    int elem;      // element przechowywany w węźle
    wezel* lewy;   // lewe podrzewo (lewy potomek)
    wezel* prawy; // prawe podrzewo (lewy potomek)
};
```

Widzimy więc, że każdy węzeł ma co najwyżej dwóch potomków (równoważnie: rodzic ma co najwyżej dwoje dzieci).

Drzewo binarne pozwala szybko (przeważnie[3], zob. dalej) wyszukiwać, wstawiać i usuwać elementy (średnio $O(\log_2 n)$[4], pesymistycznie $O(n)$).

Każde drzewo poszukiwań binarnych cechuje się następującymi własnościami (aksjomaty BST):

1. Drzewo ma strukturę *acykliczną*, tzn. wychodząc z dowolnego węzła za pomocą wskaźników nie da się do niego wrócić.
2. Niech v będzie dowolnym węzłem. Wówczas:
   - wszystkie elementy w lewym poddrzewie v są nie większe (`<=`) niż element przechowywany w v,
   - wszystkie elementy w prawym poddrzewie v są większe (`>`) niż element przechowywany w v.

Ciekawostka

Czasem rozpatruje się drzewa, w których zabronione jest przechowywanie duplikatów elementów.

Początek (pierwszy węzeł) drzewa określa element zwany *korzeniem* (ang. *root*), por. analogię do głowy listy. Korzeń nie ma, rzecz jasna, żadnych przodków. Co więcej, z korzenia można dojść do każdego innego węzła (warunek *spójności*).

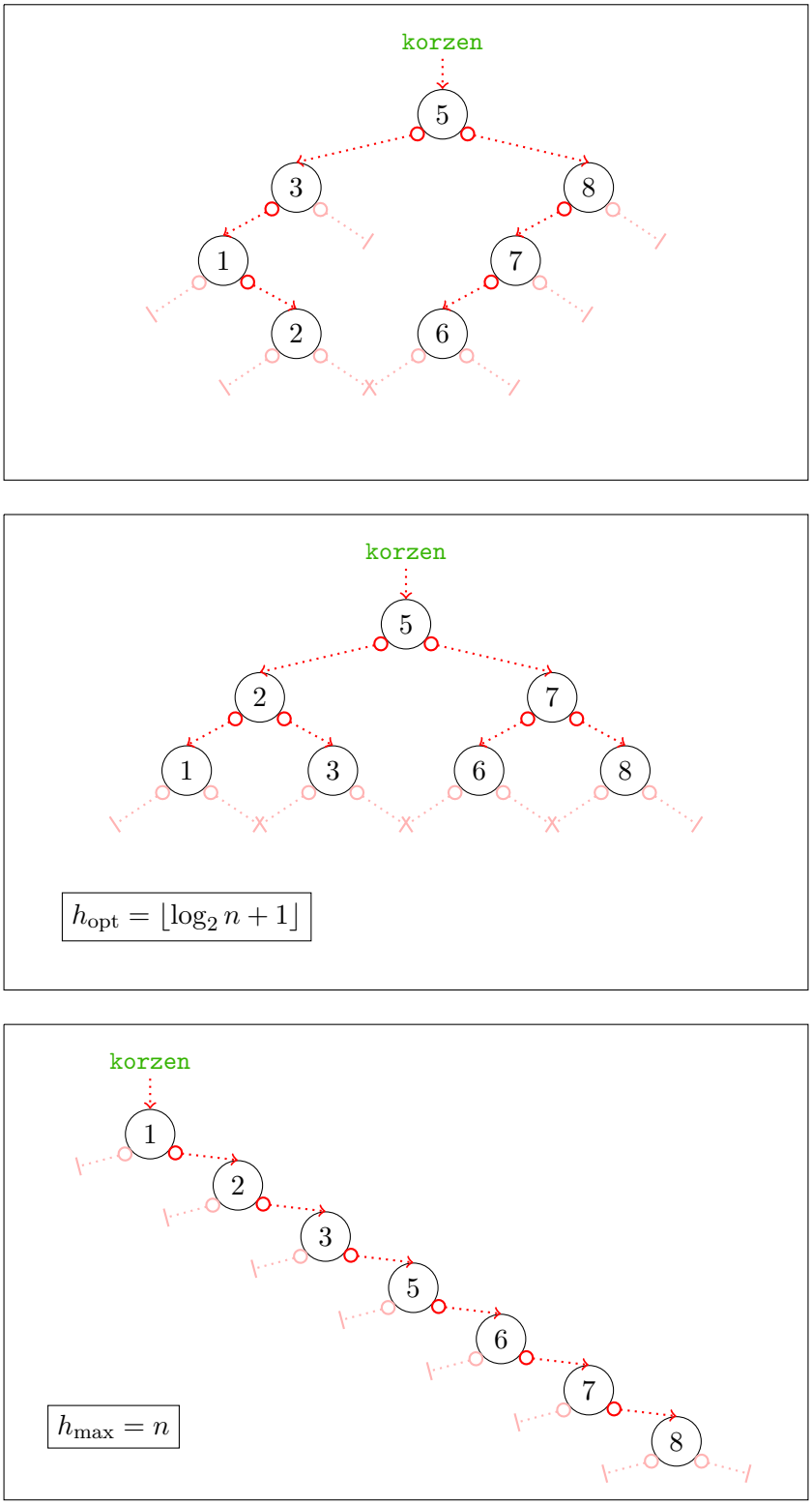
Węzły, które nie mają potomków (*węzły terminalne*, tzn. takie, których obydwaj potomkowie są ustaleni na `NULL`), zwane są inaczej *liśćmi* (ang. *leaves*).

Warto być świadomym faktu, iż drzewa *nie mają jednoznacznej postaci* — różne drzewa mogą przechowywać te same wartości. Rys. 7.7 przedstawia przykładowe BST, które zawierają elementy $1, 2, 3, 5, 6, 7, 8$. Drugie z nich jest *idealnie zrównoważone* — ma optymalną wysokość (liczbę poziomów). Liczba operacji koniecznych np. do znalezienia dowolnego elementu jest rzędu $O(\log_2 n)$. Z kolei trzecie drzewo jest *zdegenerowane do listy*. Wymaga $O(n)$ operacji w przypadku pesymistycznym.

Spróbujmy więc zaimplementować kolejkę priorytetową opartą na drzewie binarnym. Znowu najprostszą jest operacja *empty*:

[3] Studenci MiNI na przedmiocie Algorytmy i struktury danych (III sem.) poznają inne podobne struktury danych, min. drzewa AVL czy drzewa czerwono-czarne, które *gwarantują* wykonanie wymienionych operacji zawsze w czasie $O(\log_2 n)$. Dokonują one odpowiedniego równoważenia drzewa. Ciekawą alternatywą są także różne struktury danych typu kopiec.

[4] Logarytm jest funkcją, której wartości rosną dość wolno, np. $\log_2 1000 \simeq 10$, a $\log_2 10000 \simeq 13$ itd.



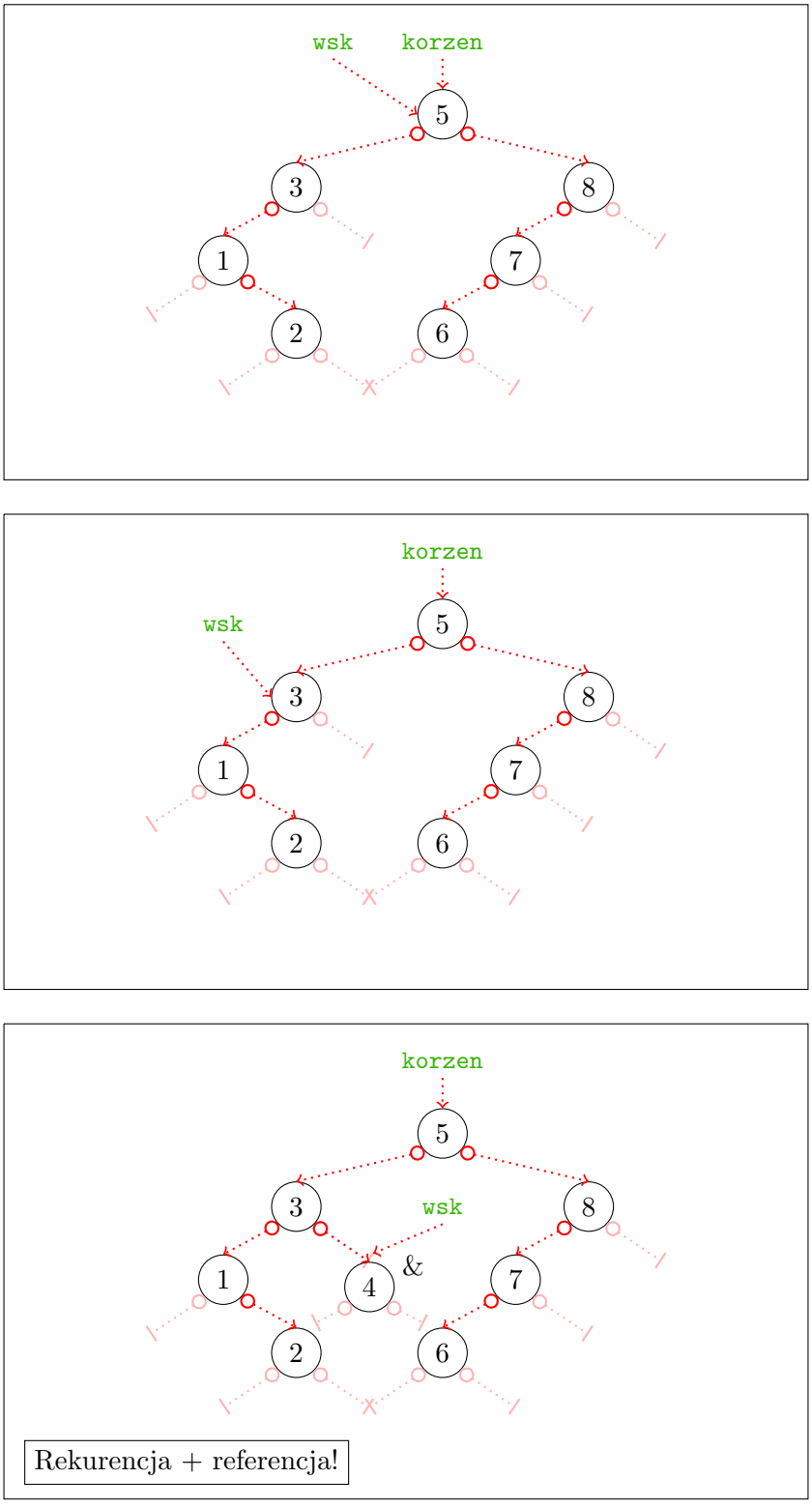
Rys. 7.7. Różne drzewa binarne przechowujące wartości $1, 2, 3, 5, 6, 7, 8$.

```
bool empty(wezel* pocz)
{
   return (pocz == NULL); // true, jeśli drzewo puste
}
```

Operacja *insert* powinna umieszczać nowy element jako liść „na swoim właściwym miejscu” (pamiętamy, że drzewo jest zawsze uporządkowane). Rys. 7.8 ilustruje wstawianie elementu o wartości 4, który powinien stać się lewym potomkiem węzła „5” ($4 \leqslant 5$). Nie będzie to jednak potomek bezpośredni, gdyż „honorowe” miejsce jest zajęte przez węzeł „3” (nie będziemy go „wydziedziczać”). „4” zostanie prawym ($4 > 3$) potomkiem „3” — bezpośrednim, bo „3” nie ma „większego” dziecka. Oto przykładowa implementacja rekurencyjna:

```
void insert(int i, wezel*& pocz)
{
   if (pocz == NULL)          // gdy poddrzewo puste...
   {
      pocz = new wezel;       // stwórz liść
      pocz->elem = i;
      pocz->lewy = NULL;
      pocz->prawy = NULL;
   }
   else if (i <= pocz->elem)
      insert(i, pocz->lewy);  // wstaw do lewego poddrzewa, będzie
         FIFO, gdy ==
   else
      insert(i, pocz->prawy); // wstaw do prawego poddrzewa
}
```

Pesymistyczny koszt wykonania tej operacji operacji jest rzędu $O(n)$ (dla drzewa zdegenerowanego), oczekujemy jednakże, że operacja wykonywać się będzie średnio w czasie $O(\log_2 n)$ (dla danych losowych).



Rys. 7.8. Wstawianie wartości 4 do drzewa binarnego.

Trzecia operacja, tzn. *pull*, wyszukiwać będzie największy element przechowywany w drzewie. Znajduje się on zawsze w najbardziej wysuniętym na prawo węźle (por. rys. 7.9). Przykładowa implementacja korzystająca z rekurencji:

```
int pull(wezel*& pocz)
{
   assert(pocz != NULL); // na wszelki wypadek...

   // usuwanie "skrajnie prawego" liścia
   if (pocz->prawy == NULL) // wtedy to, co w pocz >= od
       wszystkiego, co w całym poddrzewie
   {
      int i = pocz->elem;
      wezel* stary = pocz;
      pocz = pocz->lewy; // może być NULL
      delete stary;
      return i;
   }
   else
      return pull(pocz->prawy); // idź dalej
}
```

Koszt wykonania tej operacji jest taki sam jak w poprzednim przypadku.

7.5. Słownik

Słownik (ang. *dictionary*) to abstrakcyjny typ danych, który udostępnia trzy operacje:

- *search* (znajdź) — sprawdza, czy dany element znajduje się w słowniku,
- *insert* (dodaj) — dodaje element do słownika, o ile taki już się w nim nie znajduje,
- *remove* (usuń) — usuwa element ze słownika.

Dla celów pomocniczych rozważać będziemy także następujące działania na słowniku:

- *print* (wypisz),
- *removeAll* (usuń wszystko).

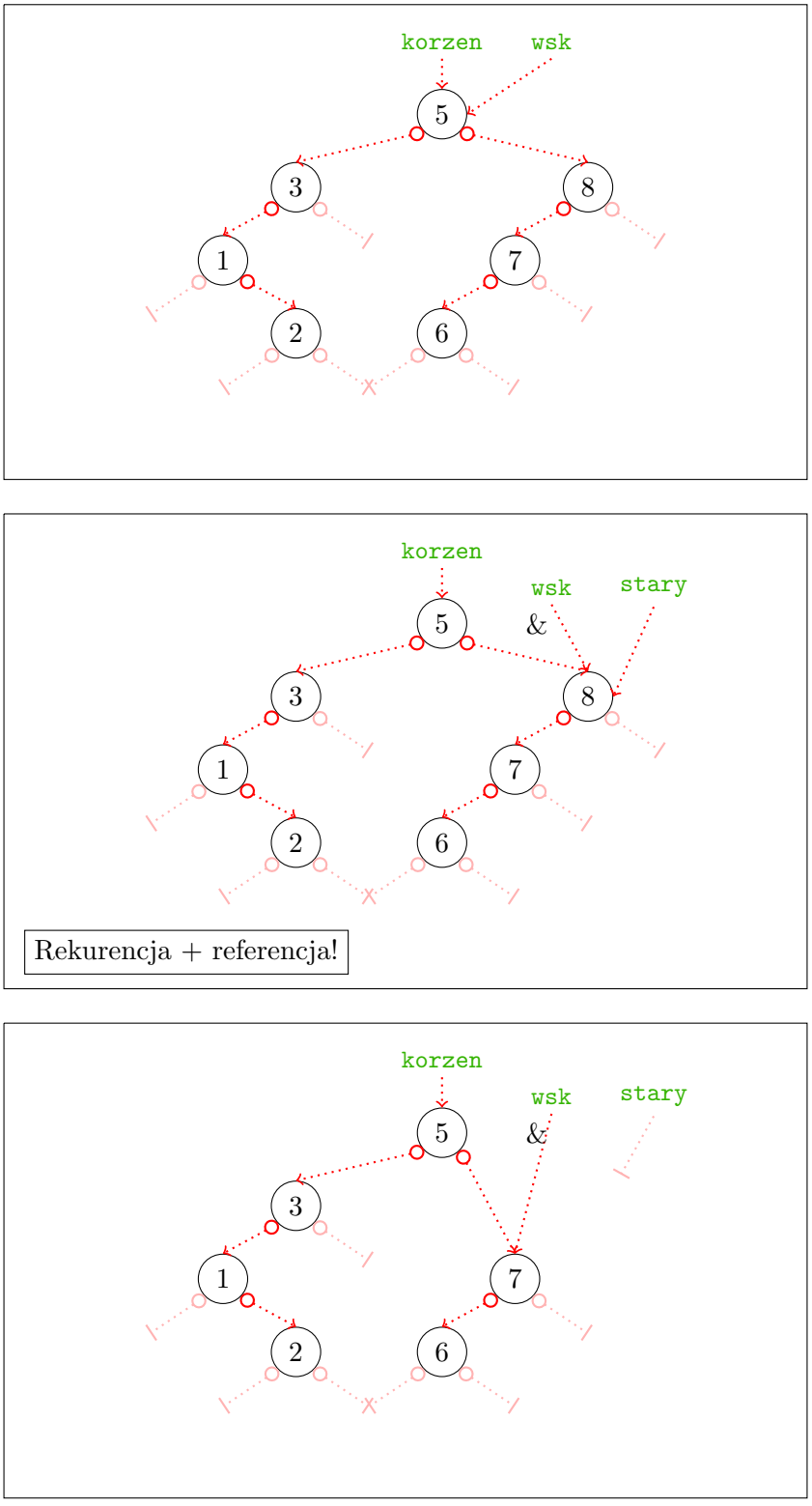
Informacja

Czasem zakłada się, że na przechowywanych elementach została określona relacja porządku liniowego <=. Wtedy możemy tworzyć bardziej efektywne implementacje słownika, np. z użyciem drzew binarnych (zob. dalej).

7.5.1. Implementacja I: tablica

Dość łatwo będzie nam zaimplementować słownik z użyciem zwykłej tablicy. Niestety, wszystkie operacje w swej najprostszej implementacji będą cechować się liniową ($O(n)$) pesymistyczną złożonością obliczeniową.

Jeśli jednak na elementach została określona relacja <= (co jest bardzo częstym przypadkiem), operację wyszukiwania można znacząco przyspieszyć przyjmując, że elementy są



Rys. 7.9. Usuwanie elementu największego w drzewie binarnym.

przechowywane zgodnie z powyższym porządkiem. Wymagane jest tu jedynie odpowiednie zaimplementowanie operacji *insert*).

W takiej sytuacji, w operacji *search* można skorzystać z algorytmu tzw. *wyszukiwania binarnego* (połówkowego), którego pesymistyczna złożoność wynosi zaledwie $O(\log n)$. Metoda ta bazuje na założeniu, że poszukiwany element `x`, jeśli oczywiście znajduje się w tablicy, musi być ulokowany na pozycji (indeksie) ze zbioru `{l,l+1,...,p}`. Na początku ustalamy oczywiście `l=0` i `p=n-1`. W każdej kolejnej iteracji znajdujemy „środek" owego obszaru poszukiwań, tzn. `m=(p+l)/2` oraz sprawdzamy, jak ma się `t[m]` do `x`. Jeśli okaże się, że jeszcze nie trafiliśmy na właściwe miejsce, to wtedy zawężamy (o co najmniej połowę) nasz obszar modyfikując odpowiednio wartość `l` bądź `p`.

```
bool search(int x, int* t, int n)
{
   if (n == 0) return false; // słownik pusty

   // wyszukiwanie binarne (połówkowe)
   int l = 0;     // początkowy obszar poszukiwań -
   int p = n-1;   //        - cała tablica
   while (l <= p)
   {
      int m = (p+l)/2; // "środek" obszaru poszukiwań

      if (x == t[m]) return true; // znaleziony - koniec
      else if (x > t[m]) l = m+1; // zawężamy obszar poszukiwań
      else               p = m-1; // jw.
   }

   return false; // nieznaleziony
}
```

Pozostałe operacje pozostawiamy drogiemu Czytelnikowi do samodzielnej implementacji.

7.5.2. Implementacja II: lista jednokierunkowa

Słownik także nietrudno jest zaimplementować na liście jednokierunkowej. Chociaż takie rozwiązanie jest mało efektywne, wszystkie operacje cechują się bowiem z konieczności złożonością $O(n)$, to jednak próba zmierzenia się z nim może być dla nas dobrym ćwiczeniem. Przedstawmy wobec tego gotowy kod wszystkich pięciu interesujących nas funkcji.

Zauważmy, że poza operacją *print* korzystamy tutaj w każdym przypadku z bardzo dla nas wygodnej rekurencji.

```
bool search(int x, wezel* w)
{
   if (w == NULL)
      return false; // koniec - nie ma
   else if (w->elem == x)
      return true;  // jest
   else
      return search(x, w->nast); // może jest dalej?
}
```

```
void insert(int x, wezel*& w)
{
   /* Gdyby w słowniku nie mogły występować duplikaty elementów,
      ta operacja cechowałaby się złożonością O(1)
      - można by było wstawić nowy węzeł na początek.
```

```
        Niestety my musimy iść być może na koniec listy... */

    if (w == NULL)
    {
        // wstaw tu
        w = new wezel;
        w->nast = NULL;
        w->elem = x;
    }
    else if (w->elem == x)
        return; // element już jest - nic nie rób
    else
        insert(x, w->nast); // wstaw gdzieś dalej
}
```

```
void remove(int x, wezel*& w)
{
    if (w == NULL) return; // nic nie rób
    else if (w->elem == x)
    {
        // skasuj się
        wezel* wstary = w;
        w = w->nast;
        delete wstary;

        // koniec - nie idziemy dalej
    }
    else remove(x, w->nast); // szukaj dalej
}
```

```
void print(wezel* w)
{
    cout << "{ ";
    while (w != NULL) {
        cout << w->elem << " ";
        w = w->nast;
    }
    cout << "}" << endl;
}
```

```
void removeAll(wezel*& w)
{
    if (w == NULL) return; // nie ma czego usuwać - koniec

    removeAll(w->nast); // usuń najpierw wszystkie następniki

    delete w; // usuń siebie samego
    w = NULL;
}
```

7.5.3. Implementacja III: drzewo binarne

Jeśli na elementach słownika została określona relacja porządku <=, możemy zaimplementować ten abstrakcyjny typ danych z użyciem drzew binarnych. Dzięki czemu wszystkie trzy podstawowe operacje będą miały *oczekiwaną* złożoność obliczeniową rzędu $O(\log n)$ (por. dyskusję w rozdz. 7.4.2).

Stosunkowo najprostszymi, a zatem i niewymagającymi szerszego komentarza, są operacje *search* i *removeAll*. Dla ilustracji, na rys. 7.10 pokazujemy przebieg wyszukiwania elementu 2 w przykładowym drzewie złożonym z elementów $1, 2, 3, 5, 6, 8$.

```
bool search(int x, wezel* w)
{
   if (w == NULL)          return false;
   else if (w->elem == x) return true;
   else if (w->elem < x)  return search(x, w->prawy);
   else                    return search(x, w->lewy);
}
```

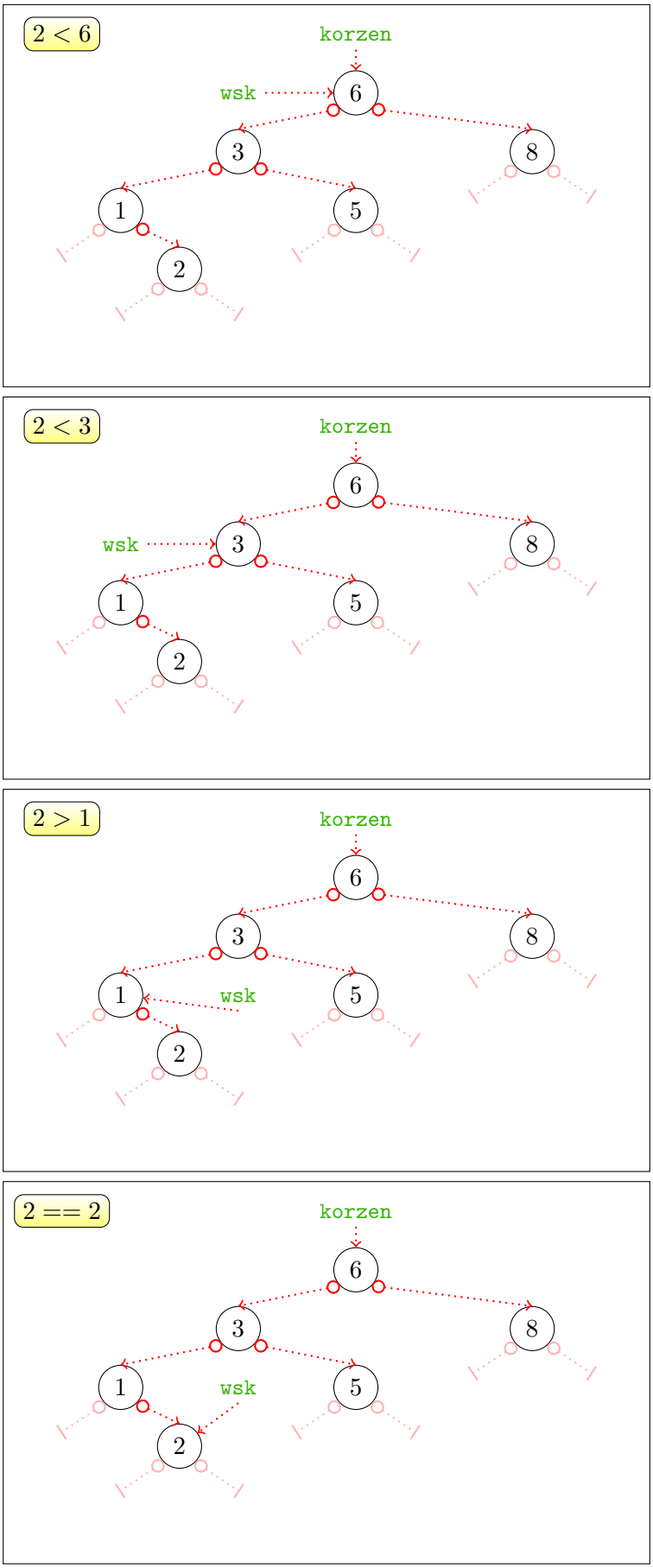
```
void removeAll(wezel*& w)
{
   if (w == NULL) return; // nie ma czego usuwać

   removeAll(w->lewy);    // usuń najpierw
   removeAll(w->prawy);   // wszystkich potomkow

   delete w;     // potem usuń samego siebie
   w = NULL;
}
```

Aby nieco „uatrakcyjnić” sposób wyświetlania drzewa, w operacji *print* skorzystamy z dodatkowej, pomocniczej funkcji rekurencyjnej.

```
void print_pomocnicza(wezel* w) // funkcja pomocnicza
{
   // realizuje ,,zwykłe'' wypisanie elementów
   if (w == NULL) return;
   print_pomocnicza(w->lewy);
   cout << w->elem << " ";
   print_pomocnicza(w->prawy);
}

void print(wezel* w)
{
   cout << "{ ";
   print_pomocnicza(w);
   cout << "}" << endl;
}
```

Operacja *insert* realizowana będzie za pomocą wstawiania nowego elementu jako potomka odpowiednio wybranego liścia.

```
void insert(int x, wezel*& w)
{
   if (w == NULL)
   {                        // wstaw tu
      w = new wezel;
      w->lewy = NULL;
      w->prawy = NULL;
      w->elem = x;
   }
   else if (w->elem == x)
      return;       // element już jest — nic nie rób
   else if (w->elem <  x)
```



Rys. 7.10. Wyszukiwanie elementu 2 w przykładowym drzewie binarnym.

```
      insert(x, w->prawy);
   else
      insert(x, w->lewy);
}
```

Nieco bardziej skomplikowaną będzie ostatnia operacja. Usuwając dany węzeł musimy bowiem rozważyć trzy przypadki:

1. jeśli węzeł jest liściem — po prostu go usuwamy,
2. jeśli jest węzłem z tylko jednym dzieckiem — zastępujemy go jego dzieckiem,
3. jeśli węzeł ma dwoje dzieci — należy go zastąpić wybranym z dzieci, a następnie doczepić drugie dziecko (wraz ze wszystkimi potomkami) za najmniejszym/największym potomkiem pierwszego.

```
void remove(int x, wezel*& w)
{
   if (w == NULL) return;        // nic nie rób
   else if (w->elem < x)
      remove(x, w->prawy);
   else if (w->elem > x)
      remove(x, w->lewy);
   else                          // element znaleziony
   {
      wezel* wstary = w;

      if (w->lewy != NULL)       // ma lewego potomka ---
      {
         w = w->lewy;            // zastąp węzeł lewym dzieckiem

         // A co z ewentualnym prawym potomkiem i jego dziećmi?
         // Musimy go doczepic jako "skrajnie" prawy liść
         wezel* wsk = w;
         while (wsk->prawy != NULL)
            wsk = wsk->prawy;
         wsk->prawy = wstary->prawy;
      }
      else if (w->prawy != NULL) // ma prawego potomka,
                                 // lecz nie ma lewego ---
         w = w->prawy;           // po prostu zastąp prawym
      else                       // jest lisciem ---
         w = NULL;               // po prostu usuń

      delete wstary;
   }
}
```

7.6. Podsumowanie

Przewodnim celem niniejszej części skryptu było przedstawienie czterech bardzo ważnych w algorytmice abstrakcyjnych typów danych. Omówiliśmy:

1. stos (LIFO),
2. kolejkę (FIFO),

3. kolejkę priorytetową (HPF),

4. słownik.

Wiemy już, iż każdy ATD można zaimplementować na wiele sposobów, używając np. tablic albo dynamicznych struktur danych (list, drzew itp.). W każdym przypadku interesowała nas najbardziej wydajna implementacja.

Tablica 7.1 zestawia złożoność obliczeniową najważniejszych operacji wykonywanych na tablicach, listach jednokierunkowych i drzewach binarnych. Każda z tych trzech struktur danych ma swoje „mocne" i „słabe" strony. Z dyskusji przeprowadzonej w poprzednich rozdziałach wiemy już, jak ważny jest świadomy wybór odpowiedniego narzędzia, którym zamierzamy się posłużyć w praktyce.

Tab. 7.1. Złożoność obliczeniowa wybranych operacji.

Operacja	**Tablica**	**Lista**	**Drzewo BST**
Dostęp do i-tego elementu	$O(1)$	$O(n)$	$O(n)$
Wyszukiwanie	$O(n)$ $^{(a)}$	$O(n)$	$O(n)$ $^{(b)}$
Wstawianie	$O(n)$	$O(n)$	$O(n)$ $^{(b)}$
— na początek		$O(1)$	
— na koniec		$O(n)$ $^{(c)}$	
Usuwanie	$O(n)$	$O(n)$	$O(n)$ $^{(b)}$
— z początku		$O(1)$	
— z końca		$O(n)$ $^{(d)}$	

(a) $O(\log n)$, jeśli tablica jest posortowana.
(b) Oczekiwana złożoność $O(\log n)$.
(c) $O(1)$, jeśli lista zawiera wskaźnik na ogon.
(d) Nieomawiana tutaj lista dwukierunkowa pozwala na realizację tej operacji w czasie $O(1)$.

Oczywiście w algorytmice znanych jest o wiele więcej ciekawych algorytmów i struktur danych, np. tablice mieszające (ang. *hash tables*) implementujące słownik. Takowe Czytelnik może poznać studiując bogatą literaturę przedmiotu bądź słuchając bardziej zaawansowanych wykładów.

7.7. Ćwiczenia

Zadanie 7.1. Zaimplementuj w postaci osobnych funkcji następujące operacje na liście jednokierunkowej przechowującej wartości typu `double`:

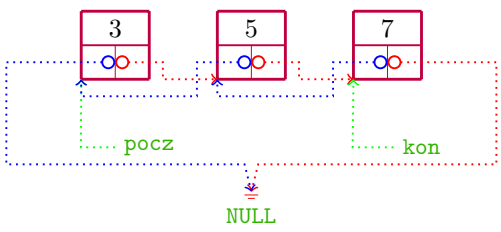
1. Wypisanie wszystkich elementów listy na ekran.
2. Wyznaczenie sumy wartości wszystkich elementów.
3. Wyznaczenie sumy wartości co drugiego elementu.
4. Sprawdzenie, czy dana wartość rzeczywista `y` występuje w liście.
5. Wstawienie elementu na początek listy.
6. Wstawienie elementu na koniec listy.
7. Wstawienie elementu na i-tą pozycję listy.
8. Wstawienie elementu `y` z zachowaniem porządku, tzn. jeśli pierwotna lista jest posortowana, to po wstawieniu porządek nie jest zaburzony.
9. Usuwanie elementu z początku listy. Usuwany element jest zwracany przez funkcję.
10. Usuwanie elementu z końca listy. Usuwany element jest zwracany przez funkcję.
11. Usuwanie i-tego w kolejności elementu.
12. Usuwanie wszystkich elementów o zadanej wartości. Zwracana jest wartość logiczna w zależności od tego, czy choć jeden element znajdował się na liście, czy nie.
13. ⋆ Odwrócenie kolejności elementów listy.
14. Usuwanie z uporządkowanej listy powtarzających się elementów.
15. ⋆ Usuwanie z listy (niekoniecznie uporządkowanej) powtarzających się elementów.
16. ⋆ Przesunięcie co drugiego w kolejności elementu na koniec listy, z zachowaniem ich pierwotnej względnej kolejności.
17. ⋆ Przesunięcie wszystkich elementów o wartości parzystej na koniec listy, z zachowaniem ich pierwotnej względnej kolejności.

Zadanie 7.2. 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 7.3. Dla danych dwóch *posortowanych* list jednokierunkowych napisz funkcję, która je połączy w taki sposób, że wartości będą nadal posortowane, np. dla (1,2,4,5) oraz (2,3,5) sprawi, że I lista będzie postaci (1,2,2,3,4,5,5), a II zostanie skasowana.

Zadanie 7.4. Rozwiąż zadanie 7.1 zakładając tym razem, że operacje dokonywane są na liście jednokierunkowej, dla której dodatkowo przechowujemy wskaźnik na ostatni element.

Zadanie 7.5. Rozwiąż zadanie 7.1 zakładając tym razem, że operacje dokonywane są na liście dwukierunkowej. Lista dwukierunkowa składa się z węzłów, w których znajduje się nie tylko wskaźnik na następny, ale i także na poprzedni element. Co więcej, prócz wskaźnika na pierwszy element, powinniśmy zapewnić bezpośredni dostęp (np. w funkcji `main()`) do ostatniego, por. zad. 7.4, tzn. dysponujemy jednocześnie „głową” oraz „ogonem” listy. Zaletą listy dwukierunkowej jest możliwość bardzo szybkiego wstawiania i kasowania elementów znajdujących się zarówno na początku jak i na końcu tej dynamicznej struktury danych. Poniższy rysunek przedstawia schemat przykładowej listy dwukierunkowej przechowującej elementy 3, 5, 7.



Zadanie 7.6. Napisz samodzielnie pełny program w języku C++, który implementuje z użyciem listy jednokierunkowej i testuje (w funkcji `main()`) stos (LIFO) zawierający dane typu `double`.

Zadanie 7.7. Napisz samodzielnie pełny program w języku C++, który implementuje z użyciem listy jednokierunkowej z dodatkowym wskaźnikiem na ostatni element i testuje (w funkcji `main()`) zwykłą kolejkę (FIFO) zawierającą dane typu `char*` (napisy).

⋆ **Zadanie 7.8.** Zaimplementuj operacje `enqueue()` i `dequeue()` zwykłej kolejki (FIFO) typu `int` korzystając tylko z dwóch gotowych stosów.

Zadanie 7.9. Napisz samodzielnie pełny program w języku C++, który implementuje i testuje (w funkcji `main()`) kolejkę priorytetową zawierającą dane typu `int`.

Zadanie 7.10. Napisz funkcję, która wykorzysta kolejkę priorytetową do posortowania danej tablicy o elementach typu `int`.

Zadanie 7.11. Zaimplementuj w postaci osobnych funkcji następujące operacje na drzewie binarnym przechowującym wartości typu `double`:

1. Wyszukiwanie danego elementu.
2. Zwrócenie elementu najmniejszego.
3. Zwrócenie elementu największego.
4. Wypisanie wszystkich elementów w kolejności od najmniejszego do największego.
5. Wypisanie wszystkich elementów znajdujących się wierzchołkach będących liśćmi.
6. Sprawdzenie, czy dane drzewo jest zdegenerowane (bezpośrednio redukowalne do listy), tzn. czy każdy element posiada co najwyżej jednego potomka).
7. Wstawianie elementu o zadanej wartości.
8. Wstawianie elementu o zadanej wartości. Jeśli wstawiany element znajduje się już w drzewie nie należy wstawiać jego duplikatu.
9. Usuwanie elementu największego.
10. Usuwanie elementu najmniejszego.
11. Usuwanie elementu o zadanej wartości.
12. Usuwanie co drugiego elementu.
13. Usuwanie wszystkich powtarzających się wartości.
14. ⋆ Odpowiednie przekształcenie drzewa (z zachowaniem własności drzewa binarnego) tak, by element największy znalazł się w korzeniu.
15. ⋆ Wypisanie wszystkich elementów, których wszyscy potomkowie (pośredni i bezpośredni) mają dokładnie dwóch potomków lub są liśćmi.
16. ⋆ Wypisanie wszystkich elementów, których wszyscy przodkowie (pośredni i bezpośredni) przechowują wartości nieparzyste.

Zadanie 7.12. Napisz funkcję, która jako parametr przyjmuje drzewo binarne i zwraca listę jednokierunkową (wskaźnik na pierwszy element) zawierającą wszystkie elementy przechowywane w drzewie uporządkowane niemalejąco.

⋆ **Zadanie 7.13.** Napisz funkcję, która jako parametr przyjmuje drzewo binarne i zwraca listę jednokierunkową (wskaźnik na pierwszy element) zawierającą wszystkie elementy przechowywane w drzewie uporządkowane poziomami (tzn. pierwszy jest korzeń, następnie dzieci korzenia, potem dzieci-dzieci korzenia itd).